



White Paper

Open platforms and the impact of Security Technologies, Initiatives, and Deployment Practices

*Bill Jacobs
Cisco Systems, Inc.*

*Vincent J. Zimmer
Intel Corporation*

November 2012

Executive Summary

This paper is an overview of security technologies as applies to current PC systems. The goal of this paper is to contrast and compare various security technologies, initiatives, and practices that may be applied to client or server x86 platforms. These technologies range from silicon-based to standards-based. Here we are not intending to prescribe certain technologies, rather our goal is to define all of these disparate technologies in one place, show how these various technologies complement each other, indicate where they may overlap or leave gaps, and illustrate how they can be applied to help secure your platform.

This paper is mainly intended for hardware, firmware, software, and BIOS engineers. But beyond this audience, some of the information in this paper will be valuable for IT decision makers, marketing, and other parties.

The reader should take away an understanding of the motivations behind trusted platform design, the terminology of trust, how to navigate the Trusted Computing Group specifications and technology that relate to the platform, impact on platform firmware and UEFI, instances of deployment in the market, and some future possible directions for hardware and firmware.

The views and opinions of the authors do not necessarily reflect the views of Cisco Systems, Inc. and Intel Corporation, or their respective affiliates.

Table of Contents

Overview	2
Problems to solve	3
Secure Platform	4
Secure Platform Life Cycle	4
Design Phase	5
Development and Test Phase	6
Maturity of Secure Development Practices	7
Manufacturing	7
Deployment	8
Production	8
End of Life	8
Summary	9
Application of Security Initiatives.....	10
Managed Pool State	11
Pre-OS State	11
Secure Boot.....	16
Late Launch State.....	18
Persistent Memory Resident Platform Information Spanning Boot	19
Summary.....	19
Conclusion	21
Glossary	22
References	24

Overview

This section will describe an overview of the ensuing chapters.

Description of each chapter

We start by describing the present state of computing as related to security issues, focused on the platform itself. This chapter should be of interest to BIOS developers interested in how their efforts relate to the larger class of business and market concerns, all the way to the IT staff making procurement decisions based upon their particular risk profile.

Next we explore solutions in a life-cycle perspective so that we may illustrate where specific technologies and initiatives apply to platform security. Here we seek to show that secure platform design can leverage many technologies from various sources, and that these technologies can be complementary when properly applied. This chapter should be of interest to IT decision makers who have to ensure that their investment is consistent with their overall strategy around capital deployment. This section is also of interest to the system architects at platform manufacturers who need to balance the security properties of their product with other competing requirements, such as manageability, cost, and performance. Within each of these chapters we add a secure design participant view of the platform, highlighting areas of specific interest to various parties ranging from the marketer seeking to establish a security requirements list to the BIOS developer desiring to know how to apply technologies, and we tie these views back to the life-cycle perspective given earlier. The hope is that these various approaches enable crisper takeaways for various reader perspectives.

The conclusion will provide a recap of some key points and summary of the items treated in the preceding chapters.

Summary

This chapter has provided a roadmap to reviewing the successive sections of this paper.

Problems to solve

In this initial chapter on “Problems to Solve,” some motivation for the development of platform trust is covered. Specifically, the many concerns found in the industry around security are discussed.

Problem background

In the past, proprietary security solutions, non-open standards based initiatives, and confusing terminology have meant that applying effective security to an x86 platform has been difficult and costly. And the proliferation of solutions has led customers to have to understand all of the competing solutions, determine how to make disparate products interplay, and in many cases, create custom middle-ware solutions and agents to manage them.

While it is true that standardizing interfaces may in fact invite attacks, taking for example UEFI firmware’s open standard BIOS replacement, such a move also brings with it opportunities to create effective counter measures that everyone can apply and in so doing, effectively strengthens the standard solution for all.

Further, Operating System vendors (OSVs), hypervisor and virtual machine monitor vendors (VMMV’s), original equipment manufacturers (OEM’s), and Information Technology (IT) staff all want to enjoy the open platform. This allows for amortizing development costs across a large class of systems and the ability to procure platforms from different vendors. As part of this openness and choice, though, the trustworthiness of the underlying platform must remain.

Key points

Security isn’t hype, but a real market need. Security can also be a moving target if open system design is your goal.

Summary

This chapter has provided a brief introduction to some of the business problems and threats that will motivate the development of trusted platforms.

Secure Platform

Fortunately, in addition to the numerous and ever increasing threats to platform security, there are several initiatives and applicable technologies specifically designed to counter these threats. Many, if properly applied, can anchor a strong trust chain to the hardware layer and make penetration into platform firmware and above more difficult.

This chapter moves into solution space by presenting a secure platform life-cycle and addressing the security initiatives, technologies, and techniques involved in comprehensive secure platform development and deployment. Here we will elaborate on many of the myriad of commonly, and less-commonly, referenced terms applied by various organizations for platform security.

Secure Platform Life Cycle

In order to adequately address the possible security gaps on these platforms, one has to take a holistic view of the entire life-cycle of the platform from concept to end of life decommissioning or repurposing. It is not sufficient to simply try to patch up a platform with “bolt-on security”. In fact, security counter-measures must be as deeply rooted as an attack might go. For example, protecting the firmware cannot be effective with only operating system-level measures. Additionally, in order to create a truly secure platform, design practices must also be employed in all facets of development – hardware design working in concert with sound coding practices, sane deployment and management schemes, as well as secure operating and maintenance environments must all be addressed. Finally, one must also think about the possibility of secure data (e.g. keys) persisting after a platform has been “decommissioned”.

Platform Life Cycle

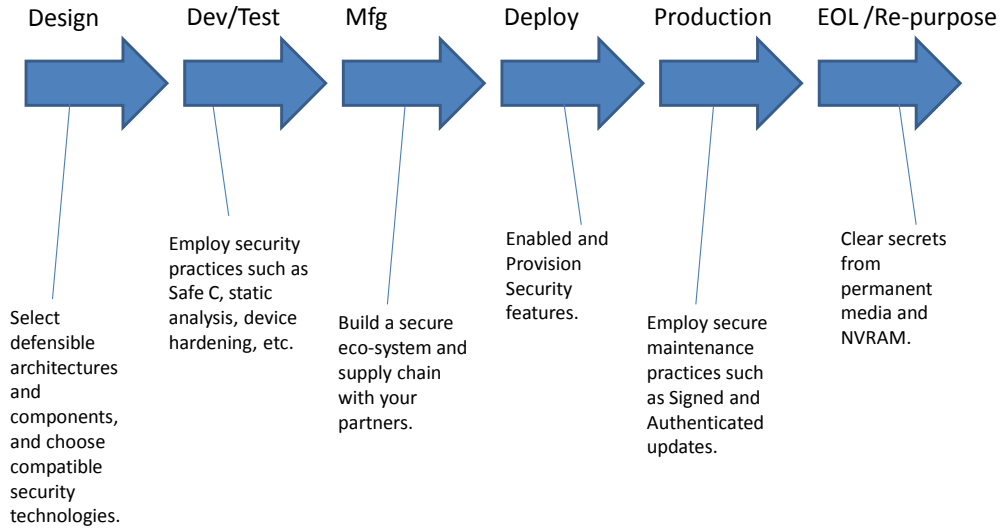


Figure 1 Platform Life-Cycle

The above timeline illustrates where security should be addressed in order to produce a life-cycle secure platform. From this diagram it is obvious that a platform's security must be addressed from inception to end of life. We believe there are no phases which can be ignored without imperiling the platform's secure use. Below we will expand on each phase of the Secure Platform Life-Cycle.

Design Phase

When conceptualizing a new PC platform, one must think of the mission or use-case for that platform. Included in that use-case analysis one must consider its security/threat environment and security requirements. If the platform is required to meet any security requirements, which may be market-driven or may be required by your own organization's security policies, then a review of the hardware-firmware-OS stack relative to the threat environment must be under taken in order to understand where potential issues lie.

Here we may even want to create a formal Threat Model of the platform and understand based on physical interfaces, software APIs, management control points, use-model, the user, and functionality where potential attacks may come from. This analysis should be detailed enough to consider protocols employed across the many potential interfaces to the secure platform. The results of this analysis will inform the designers of the platform of specific threats via specific channels.

With this information we want to choose the technologies and make the explicit design decisions that support the level of security we are after. This may involve platform architectural decisions, specific silicon or firmware choices, auxiliary device add-on selections, etc. Here the designer should be thinking in terms of Trust Boundaries; that is, thinking about areas that we can inherently trust and how to leverage these into building trust generally on the platform. Some of these decisions may impact assumptions or pre-conceived design ideals. The point is, Security must be approached as any other use-case feature decision on your secure platform, and should not be applied only as an after-thought once development has begun.

Here you will want to leverage the initiatives of industry groups like Trusted Computing Group ([TCG](#)) and government organizations like (US Government) [NIST](#). It is also highly advisable to consult with your silicon vendor, ODM partners, firmware providers, any add-in IO card or device providers, OS vendors and the rest of your eco-system partners to confirm that you have explored all options for platform security. Each of these components can expose attack surfaces on your platform so each must be addressed in a comprehensive way. Again, in most platforms based on off-the-shelf components, a holistic solution that is based on open standards has a better chance of playing well together to positively affect security on your platform than one-off, home-grown custom solutions.

Also, special attention should be placed on understanding the software as well as hardware requirements that build platform security. Many times hardware security is augmented by software changes, and vice versa.

Stakeholders with particular interest in the Design phase are planners and marketers as well as platform architects.

Development and Test Phase

The specific requirements called out in the Design Phase are to be addressed in this Development and Test Phase. This means that the initiatives, components and requirements of the Design phase are integrated into the platform and made to function, and validated as such.

Leveraging secure development practices in the platform Development and Test phase will go a long way toward comprehensive secure platform development. In fact, many hygiene practices such as employing Safe C libraries, static code analysis, ensuring 3rd party code is free of back doors, and so on should be part of any mature, secure development organization. If these practices are not part of your organization's DNA, then they should be explicitly called out as Requirements and tracked. In either case, testing should always seek to verify that safe development practices were adhered to.

The Development and Test phase is where you will want to establish practices, methods, and tools for entering keys, and provisioning security devices with platform manufacturer, and in some cases, platform owner data. Here you will want to work out how to sign code modules and images which support authenticated update and secure loading schemes.

Examples are signing of BIOS updates, programming launch control policies, and so on.

Other portions of the development phase include testing. This can be positive testing, such as the UEFI Self-Certification Tests ([SCT's](#)) or the Trusted Platform Module ([TPM](#)) compliance tests. Herein, the device under test is subjected to expected or specified input values and the system is observed to see if the expected output or return values occur. This is as opposed to negative testing, or [fuzzing](#), where unexpected inputs are injected and the system is observed to see if it survives and continues functioning or enters the appropriate remediation or recovery mode. Both positive and negative testing have a place in testing secure platforms.

Maturity of Secure Development Practices

Another key technique in the development of a truly secure platform is the development methodologies employed in the creation of the secure platform software. Mature Secure Development means that sound security practices are in place and rigorously followed and validated. These practices can, of course, be tweaked and advanced, but it does not mean that each time a platform is developed the secure development process is re-invented, and relearned.

In the Development and Test phase, major stakeholders include, hardware, firmware and software developers and validation engineers.

Manufacturing

The supply chain presents an inviting target for secure platform subversion. There are many known cases where viruses have been placed into platforms in manufacturing, and certainly many, many more such cases kept secret. Secure platform manufacturing requires that you know and are able to trust the manufacturer and their manufacturing practices. In order to do this, you should specify your secure manufacturing requirements and qualify your factory and their processes against those requirements. And this means you will probably want to audit your factory from time to time to help confirm compliance.

There are several potential areas where you and your manufacturer may work together on security. One opportunity includes the use of trusted computing technology. If you use the Trusted Platform Module (TPM) and other specifications from the Trusted Computing Group (TCG), you can provide expected values of the platform state, as recorded by the TPM and the TCG-aware firmware, which can be used to assure that the platform was not altered in certain ways. More on the use of TCG capabilities is provided elsewhere this paper.

The manufacturing phase may also be utilized to establish white and black lists for various technologies and may be the best place to enable UEFI Secure Boot User Mode, for example. Based on a given deployment strategy, this may preset your platform policy and prevent unauthorized changes in the following platform life-cycle phases.

Stakeholders to the Manufacturing phase are system engineers, manufacturing engineers, and hardware and firmware developers.

Deployment

In Deployment, too, one must be cognizant of security threats. Understanding the use environment and configuring the platform appropriately can raise its security posture.

There are aspects of end-user provisioning, “taking ownership”, etc. that are key in the Deployment phase. In the case of deploying a tightly managed server, for example, one may want to retain TPM ownership with the management entity.

The Deployment phase is largely the concern of the IT guy or security admin.

Production

The Production life-cycle phase is where the platform does its intended work. This is where the platform is being used to process data, crunch numbers, host or view a website, etc. Platform security in the production phase may appear to be completely determined by choices made in previous life-cycle phases, but this is no time to rest of your laurels. One has to be aware that a platform in production often requires updates to firmware, software, and sometimes to physical IO devices as represented by PCIe cards, for example.

A subset of the production phase is maintenance. The platform must be available to consume updates, patches, and other activities to evolve the system in the face of discovered flaws or emergent threats in the market. Here we should look to leverage initiatives that define Trusted Roots for Update (RTU), such as NIST SP 800-147, for example. This publication describes a secure update scheme that seeks to protect the platform from untrusted firmware updates by prescribing a single, non-bypassable update path, locked NVRAM firmware storage devices, and authenticated firmware updates only.

Production phase concerns the IT guy, security admin, IaaS or SaaS provider, and end user.

End of Life

After its useful Production life, a platform is either trashed or repurposed. Trashed platforms are typically picked apart and components are recycled or repurposed. Or, the entire platform may be repurposed – redeployed into another operational environment. In any case, the secure life-cycle of that platform has not ended until any data, secret or not, has been cleaned from that platform.

Stakeholders at end of life are the previous user or owner, as well as the receiving user or owner.

Summary

In this chapter we took a secure platform life-cycle view of security touch-points and highlighted specific areas of concern and the parties who might have a specific interest in those areas. The reader should take away a strong notion that platform security is not adequately applied in just a single phase of the platform's life-cycle, rather, done right, platform security requires attention from concept to EOL.

In the next chapter, we will dive more deeply into the secure platform Production life-cycle phase and explore the application of specific security initiatives.

Application of Security Initiatives

In this chapter, we will explore the Production phase of the secure platform's life-cycle with emphasis on specific initiatives that can play a profound role in platform security. Having highlighted the entire secure platform life-cycle above, here we focus on the most active life-cycle phase.

There are currently several security initiatives being developed by various bodies around the PC industry. These range from Trusted Computing Group initiatives centered around the concepts of Trusted Measurement and Trusted Store, to methods of securing the loading of enabling firmware in a secure chain manner, to silicon-based techniques to create pure, unadulterated memory execution regions into which one can launch an Operating System or Virtual Machine Monitor with certainty that no unknown code resides in that space.

Many of these technologies are applicable to the boot process of a deployed platform in its normal production life-cycle phase. An example of how these various technologies and initiatives align during a PC boot cycle might look similar to the following:

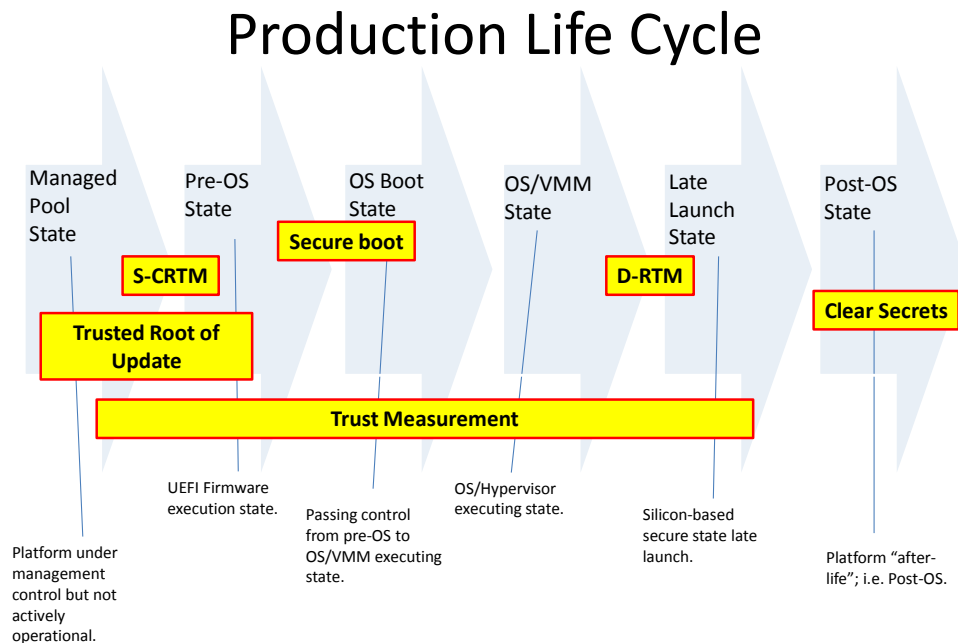


Figure 2 Production Life-Cycle

This diagram shows a detailed view of a portion of the secure platform's life-cycle focused around its deployed and managed Production state. This may apply to a managed

server or client platform, which are quite common use models. At the left of the diagram the secure platform resides in S5 state, powered off, but with a management entity, either service processor ([BMC](#)) or management engine awake and attached to a management infrastructure. We will call this the Managed Pool State. In this state the management infrastructure may be capable of doing many things through its out of band interface. The management infrastructure may be able to push firmware updates, power on the platform, change the platform's configuration via firmware settings, etc.

Managed Pool State

In this Managed Pool State, the platform's security must be protected from rogue firmware updates that might affect a root kit attack possibly infecting the most intimate regions of the platform (e.g. [SMM](#)). Techniques centered around a comprehensive Root of Trust for Update (RTU) such as described in NIST SP800-147A/B, when properly implemented, can prevent rogue firmware from being updated into a platform. The principles of SP 800-147 include locking the NVRAM device into which the boot firmware is to be loaded and only unlocking when in a controlled update process. It also requires that only a single pathway to the NVRAM device is available for updates and that there is no bypass scheme to get around this single pathway. Further, and perhaps most important is the requirement to update only signed images that authenticate by the RTU. The principles of secure update apply not only to the boot loader firmware of the platform (BIOS or UEFI firmware), but also to the service processor which might be in the update chain for the above firmware. Even more importantly, in some cases the service processor may be trusted to update the pre-OS firmware. So the Root of Trust for Update must be extended or shifted to the entity actually doing the update.

In this Managed Pool state, the platform must be protected from secure data manipulation or loss, from prohibited configuration settings which might cause failure to boot (effecting Denial of Service). In some cases BIOS Setup may contain settings which conflict with each other or the platform's capabilities or that may be configured in such a way as to cause a failure to boot. These must be searched for and removed before a platform is deployed.

Pre-OS State

Upon application of power to the managed platform's host processor complex, boot firmware starts to execute on the host processor in what we will call the Pre-OS State. On a PC platform this is the state classically represented as BIOS POST, or more currently as UEFI firmware execution. In this state, the processor, chipset, memory and platform peripherals, including add-in cards, are being initialized and configured to support the launch of the Operating System. The code in this state is typically run in a single threaded mode and with absolute highest privilege level.

On an x86 platform this early initialization code also is responsible for establishing System Management Mode (SMM) and populating SMM specific memory with handlers for various system maintenance support routines including server Reliability,

Availability, and Serviceability (RAS) and a very wide range of runtime support services which operate “behind” the Operating System itself.

The Pre-OS State configuration process may include execution of code introduced onto the platform via add-in PCIe card-based peripherals. Such “alien” code, regardless of its provenance, is presented to the host processor to be run at the same privilege level as the rest of the boot firmware and therefore presents an obvious avenue to bring in rogue code. And of course, the origins of the base Pre-OS firmware itself must also be questioned.

It is at this point in the Production life-cycle of the platform, namely at reset, where we can apply one of the most potentially significant security techniques; establishment of a Core Root of Trust. It is here that we have the distinct ability, with the right hardware and software, to establish a foundation from which we can not only securely go forward to loading the OS but also securely capture detailed data from which we can later assure acceptable secure state was achieved. The concept of a Core Root of Trust is based on the establishment of the minimal Trusted Compute Base opportunity presented at host processor reset.

The idea of a Trust Base is that someplace along the line one must decide to trust something about the secure platform. This implies that the platform owner knows enough about some part of the platform to be able to trust at least that much of it. As with a trusted colleague, for example, one knows enough about that person to be able to reasonably say what that person might do in a given predictable situation. Just as with a trustable person, one can know that a secure platform is trustable in its Managed Pool state because it is locked in their data center, for example. And they may know that their data center ensures that the secure platform was not physically accessed by untrusted persons. Therefore, that physical platform is trustable at least at the point of the beginning of execution of add-on enabling code such as the UEFI firmware. One can be reasonably sure that if the hardware itself was not accessed, then there is little chance indeed that the logic baked into that secure platform will do anything other than go to the reset vector address, fetch code, and begin executing that code according to the hardware’s design. This point of trust is the launching point of Core Roots of Trust for authenticating the balance of the code yet to come at run time and for launching a scheme known as code measuring. This entails the establishment of a Core Root of Trust for Measurement.

The first instance of a Core Root of Trust for Measurement (CRTM) is often known as the Static CRTM, or S-CRTM. The term Static denotes the fact that the CRTM is anchored in one place, namely at processor reset. If the code making up the S-CRTM is adequately trustable itself, it can be employed to authenticate the rest of the UEFI firmware before allowing it to execute. The technique used to ensure that the S-CRTM code is trustable is essentially to make that code immutable. That means that that code came from one place – the OEM in most cases – and there is no currently known way to alter that code after the platform leaves the OEM’s hands, short of physical possession and alteration of the platform. This, of course, means that the immutable portion of the code is locked or burned into a memory device and the host processor’s reset vector

points into that device. There is no field update of this immutable code, and there is tamper prevention on its storage device or said non-volatile memory is hidden away inside some other larger logic device where it cannot be altered without altering a major component of the platform. Of course it has to be noted that such measures do not come without a cost and potential risk. The work required to develop the immutable code and validate it to a point of such high reliability is costly indeed. For example, if the immutable code is found to be buggy or somehow compromised once deployed to Production, the secure platform must be returned to the OEM to be replaced.

The other S-CRTM usage is one of establishing a starting place from which to measure the code that will be executed on the platform. The idea is again that we have a trustable place to start doing some process that will span code from potentially many sources, most outside of the OEM or platform owner's control. For example, after the UEFI firmware core executes, it starts loading third-party drivers, applications, add-in card drivers, OS-loaders, Operating Systems, etc. The concept of "measuring" here means that the platform literally takes a cryptographic hash of the next piece of code it is scheduled to launch. Then subsequent code is also hashed, and so on and so forth. This presents a set of fingerprints of the code, or even of platform configuration data, that was run on that platform. These measurements must also be kept someplace safe or else they are themselves subject to "evidence tampering". The safe place is another TCG construct known as the Trusted Platform Module (TPM) and in TCG such usage is known as Trusted Storage. (See TCG specs tbd for details of Trusted Measure, Trusted Store.)

With the measurement data captured in the TPM, one can perform either a "real-time" state based check, or one can retroactively analyze the measurement data to determine if a certain state was indeed met. For example, taking the latter case, an agent running on the secure platform can at some point in the Production boot cycle halt the platform, grab the set of measurements from the TPM device on the secure platform, and utilizing a trusted service, typically found on a trusted third party controlled database, determine if the firmware executed on that secure platform matched the hashes of known good firmware.

This implies of course that the trusted third party doing the checking has access to a set of Golden Measures that match accepted secure platform measures. The process of checking actual measures against golden measure is one of attestation. There may be more than one set of golden measures. This is because there are likely more than one set of acceptable configurations – given optional BIOS settings, optional boot device selection, and so on possible for a given platform. With the determination of a match between present and Golden measures, or not, the secure platform agent may be used to inform secure platform policy as to whether and how secure platform may continue to boot and into what operating domain it may be permitted. So, for example, if our measured platform is found to have been configured to employ a USB boot device, policy may dictate that the secure platform may not join the corporate network.

Beyond the S-CRTM which reads on firmware construction for purposes of an S-RTM implementation, there are other concerns in platform construction. As an example, we

will discuss a BIOS based upon the Unified Extensible Firmware Interface (UEFI) Platform Initialization (PI) specification sets. The UEFI specifications are purposely silent on construction intent and policy. Instead, these are pure interface specifications that admit to conformance testing of the API's. As such, some of the intent on usage of these standards include which portions are platform supplier (PS) extensible and which are platform owner (PO) extensible.

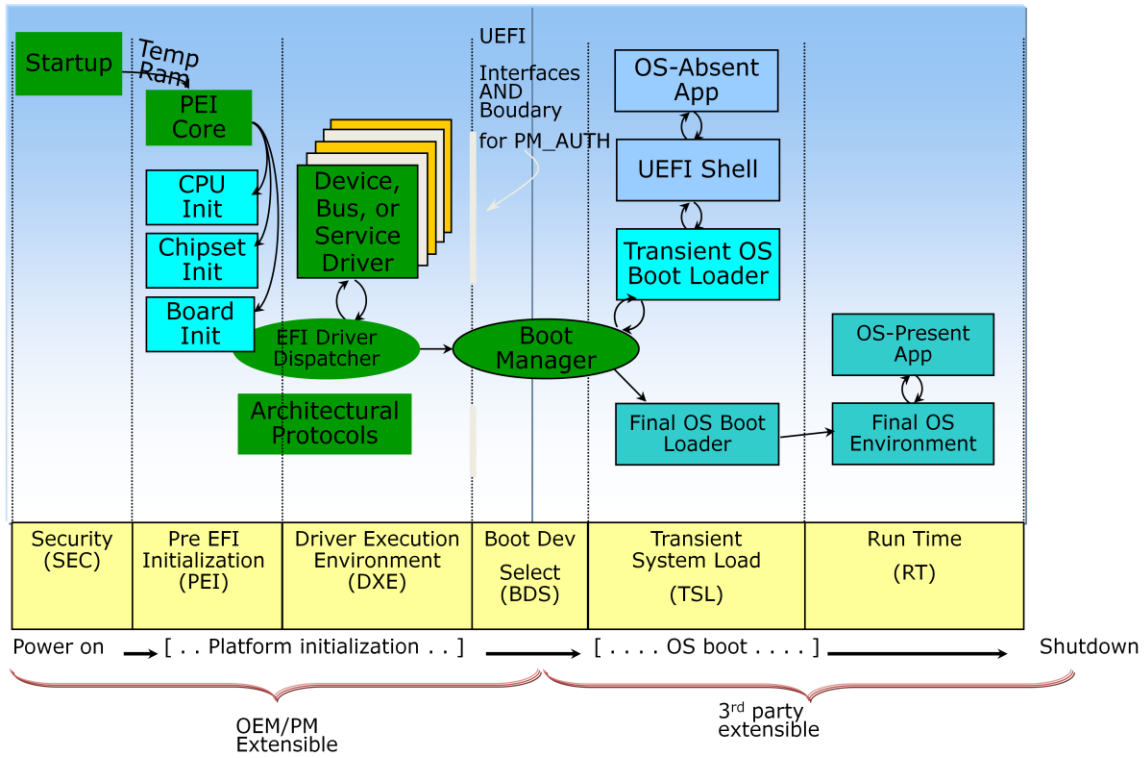


Figure 3 UEFI PI Boot Flow

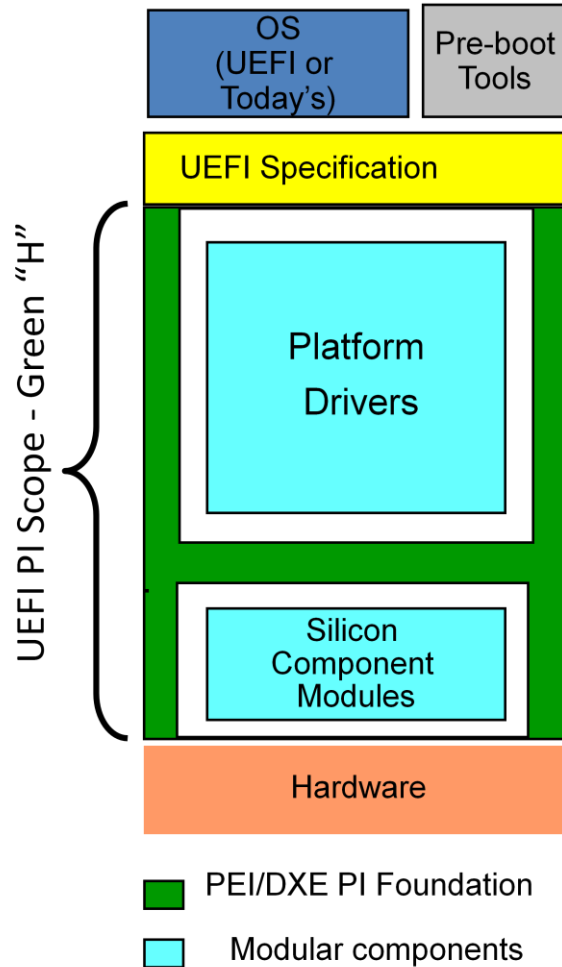


Figure 4 UEFI PI Software Layering Diagram

As you can see from the boot flow above, the SEC, PEI, and DXE all run prior to having the UEFI services available. The UEFI specification describes a set of interfaces to the platform, and the UEFI PI DXE phase acts as the UEFI core. In fact, the DXE core is the preferred embodiment of the UEFI interfaces.

What is important to take away from the boot flow is that the left hand side of the flow are platform supplier extensible. The UEFI PI components form the root of trust for updates (RTU), the root of trust for measurement (RTM), and the root of trust for verification (RTV). The RTU can be implemented via a DXE capsule update driver, the RTM via a DXE TPM driver that publishes the `EFI_TCG_PROTOCOL`, and finally, a set of DXE modules that implement the RTV by means of implementing the authenticated variables and UEFI 2.3.1c Secure Boot processing. The codes which implement the RTU, RTV, and the RTM must be from the platform supplier (PS) in order to have assurance regarding their behavior. The PS is sometimes known as the Platform Manufacturer (PM) and the codes which are from the platform manufacturer are known

as running under PM Authority, or PM_AUTH.

As distinct from PM_AUTH, the elements on the right hand side of the boot flow are typically under control of the platform owner (PO). These PO extensible elements include UEFI drivers, UEFI applications, and the operating system kernel.

Secure Boot

Regrettably, “Secure Boot” is a highly overloaded term in the industry. In the context of the taxonomy in this paper, we introduce “Secure Boot” as a more generic capability as distinct from measured boot. In measured boot, a record of the code execution is provided into the PCR’s but no policy decisions are enacted. Secure Boot, on the other hand, enforces policy prior to the execution of content. The code that implements the Secure Boot logic is known alternatively as a Root of Trust for Verification (RTV) or Root of Trust for Enforcement (RTE). The RTV can be a hardware embodiment that vets the provenance of the UEFI PI firmware prior to passing control or it can be implemented in UEFI for purposes of providing PO control of UEFI 3rd party executables. The latter usage is described in more detail below.

Spectrum of the verifiers in the Pre-OS

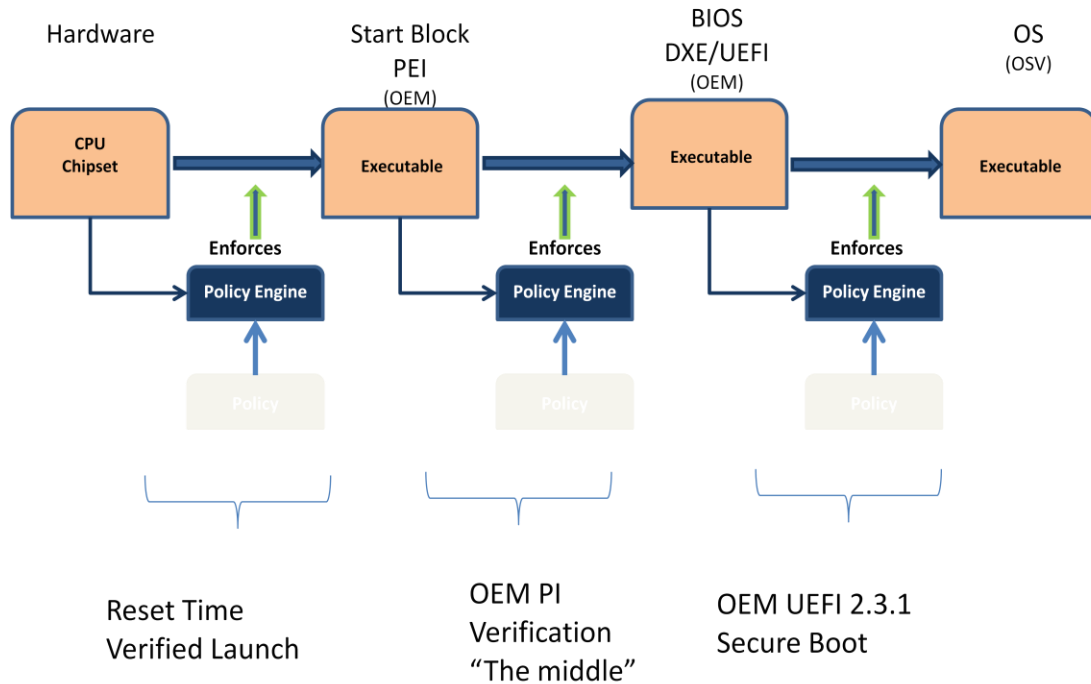


Figure 5 Verifiers in Pre-OS Space

In the figure above, the platform hardware complex will provide logic to verify that the ensuing BIOS storage container meets the policy of the platform supplier. Once this restart time verification occurs, the PS PI code, such as the UEFI PEI phase, is responsible for ensuring that the rest of the PS code meets policy. Today, the hardware action is PS and/or hardware-supplier specific, and the ‘middle’ verification is PS specification. Neither of these actions admits to industry standardization today.

Another very current security technique involves the authentication of the very images that the UEFI firmware itself will be asked to execute. This capability was introduced in UEFI Spec r2.3.1 and requires not only a compliant UEFI firmware but also requires drivers signed in a compliant manner by their producer or a trusted third-party (CA). Note that this technique does not apply to legacy BIOS because there is no standards-based way to orchestrate the compatible signing of loadable BIOS objects. For UEFI firmware, perhaps the most notable application of this technique, known as Secure Boot, is the signing and authentication of the OS Loader itself. This is but one of many possible applications of this technique, but is probably the most obvious because it involves the signing and authentication of the one UEFI driver responsible for loading the Operating System and because of its inherent security is now Required to load some operating systems.

The idea behind this application of “secure boot” is that if the UEFI driver for loading the OS, which comes from the OS vendor, is not found to be on the platform’s allowed list (or conversely is found on the disallowed list) the assumption is that some party has attempted to inject a rogue OS loader driver into the boot path and it is assumed, therefore, that they intend to launch an OS unknown or unintended by the secure platform owner. In other words, there is a strong chance that in such a case someone is trying to do something untoward. It should also be noted that this technique of signing OS loaders applies only to UEFI Native OS loading and does not apply to loading of legacy operating systems. That is because legacy operating system loaders, while possibly running “in” UEFI firmware itself, as is the case with a CSM-enabled platform, do not employ a UEFI driver, and therefore there is no way to ensure conformance with a set signing and authentication scheme.

Beyond the higher profile OS Loader signing, the UEFI image signing scheme can also be employed to sign UEFI shell applications, or UEFI drivers of any type including third-party drivers introduced via add-in cards. This is important because a secure platform vendor or owner can now require that add-in card vendors, or even vendors of on-board peripheral devices, provide signed UEFI drivers. This step helps ensure that the driver introduced onto the secure platform can be authenticated against a known good list, or rejected based on a known-bad list. Again, this capability does not apply to legacy BIOS “Option ROM drivers” because there is no set standard for signing and authentication in legacy BIOS. And again, this capability requires that the producer of the UEFI driver which enables the add-in card complies with UEFI specification mandating signing types and so on.

Features like UEFI Secure Boot on a PC without a lower-level hardware root of trust to cryptographically bind the NVRAM (flash) part to the motherboard should be resistant to the hack attack, but not a shack attack (a shack attack involves physically placing illicit components on a board, typically using off the shelf components e.g. Radio Shack). A likely shack attack could be a ROM swap where someone either de-solders the system board ROM or uses a Dediprog[®] to reprogram the ROM. The XBOX1 was susceptible to this class of attack, for example.

In general it is unlikely that Commercial off the Shelf (COTS) technology, as described in this paper, can be resistant to an adversary with the resources to launch a lab-class attack.

Late Launch State

Moving forward in boot time, we encounter other opportunities to apply platform security technologies at the firmware layer. Late Launch is a term that refers to the capability of placing a running platform, which has passed the Pre-OS and possibly the OS launch state, into a new, secure machine state. In TCG terms, this capability is referred to as D-RTM.

D-RTM

A less frequented Root of Trust for Measurement, perhaps, the D-RTM is a highly silicon centric means of establishing a secure launching place for an OS or VMM. This technique is enabled in vendor specific ways by at least a couple major platform silicon providers as of this writing, varies in its implementation, but always requires specific silicon to enable. If said silicon is available on our secure platform, we can affect a D-RTM at any point in time and repeatedly enter and exit the D-RTM state without a platform reset. The most obvious application of this capability is to establish a clean slate, if you will, from which to launch a Virtual Machine Manager.

The D-RTM requires that the BIOS be compliant in that it carries certain enabling code modules, originating from the silicon vendor and added to the OEM's BIOS, which is typically authenticated on the platform before being executed. This code is generally architecture-intimate, similar to micro-code, and is part of the total scheme to create the clean slate platform state. Of course, to make this work the platform must also somehow be configured to deal properly with DMA, and with IO accesses which could dirty the otherwise clean slate. To that end, there will often be associated requirements for other virtualization capabilities on your platform, and these must be enabled as directed.

The D-RTM is in effect less about measurement than about creating a pristine operational state on the fly, but measurement of the components of the D-RTM scheme and measurements of the providers of said components are captured and potentially used to unseal platform state information or to provide attestation opportunities as with the C-RTM described above and by the TCG.

Persistent Memory Resident Platform Information Spanning Boot

As part of informing the OS of platform specific configuration, requirements and specific methods to handle certain aspects of the platform, such as power management, the UEFI firmware builds tables in system memory and passes a pointer to same to the OS after UEFI firmware exits. Since these tables exist in memory passed from UEFI firmware to Operating System, they should be examined in a secure platform context as well.

ACPI

As part of managing platform power, ACPI information is passed from the platform UEFI firmware to the OS via persistent memory structure. Because this memory structure persists across the handoff from UEFI firmware to OS, and is unprotected, it must be examined as a potential attack surface. One could potentially alter the data in this table and conceivably cause abhorrent behavior which could overheat a platform or conversely severely limit its processing speed, for example. It is conceivable to augment this structure with a hash value or even to place this structure into protected memory. However, neither the ACPI specification nor the UEFI specification details requirements or means to protect this memory structure.

SMBIOS

The SMBIOS table consists of both static and dynamic data which is created by the UEFI firmware during execution of power on based in part on dynamic platform configuration. This data is also passed to the OS through a memory mapped structure. SMBIOS data is not generally considered critical operational data, and is rather used to inform platform policy and control. This data is most often consumed by platform management. Subversion of this data may cause misbehavior or worse, mis-configuration or mis-application of platform management policies. These can, in theory, lead to a DoS situation.

Post OS State

One typically assumes that after the OS shuts down the platform is quiesced, all data is gone, the slate is cleaned and we are ready to start all over again with application of power and a fresh boot cycle. Not necessarily true. Secrets may persist in memory through a reboot cycle, after OS exits, either intentionally or not.

One intentional use of persistent memory across a reset is to pass configuration or update information “back” to the UEFI firmware. There may also be other areas of memory that contain secrets that could unintentionally persist across a reset cycle. For these reasons, care must be taken to ensure that secrets in memory are either fully flushed, sequestered into protected memory regions, or are protected by encryption,

Summary

This chapter used a platform life-cycle view to point out specific security concerns and recommend application of various technologies and practices to address them. Many of the applied technologies are open standards based, which makes application and

interaction viable for many secure platform developers. Some of the technologies discussed require specific hardware devices, but these are considered important potential additions to a secure platform because of their relationship to a secure Root of Trust based in hardware, and therefore they are inherently more resistant to attack via introduced malware.

Conclusion

This paper has attempted to take a holistic view of platform security technologies and initiatives presently available to most x86 platform developers, whether client or server, and discuss their relevance based on a secure platform life-cycle. While this list is not exhaustive and there are certainly proprietary methods to achieve facets of platform security in practice, we hope to have given a grander view of the security challenges and the application of potential solutions in an easy to follow and digest narrative.

It should also be very clear from this paper that the present state of platform security is far from perfected and is best thought of as a palette of solutions which must be understood and properly applied; and even so, today's palette leaves opportunities for clever attack and exploitation.

Glossary

Glossary Item	Definition
ACPI	Advanced Configuration and Power Interface specification.
Attestation	Attestation refers to a process of confirming a platform state or measure.
BDS	A UEFI firmware phase where in boot devices are selected.
BIOS	The term for x86 boot firmware that predates UEFI compliant firmware.
BMC	Baseboard Management Controller; a ‘service processor’ as defined by the IPMI specification. Usually found on a managed platform such as a server.
CA	Certificate Authority;
CRTM	Core Root of Trust for Measurement; a platform operational state which serves as a trusted place to perform security operations like measurement.
CSM	Compatibility Source Module; a UEFI adjunct that enables legacy BIOS-like capabilities.
DoS	Denial of Service; a specific type of platform attack meant to make the platform in some way unable to perform its intended tasks.
DRTM	Dynamic Root of Trust for Measurement; a TCG term for a RTM that can be launched after a CRTM; is highly platform silicon centric.
DXE	A UEFI firmware execution phase that is by design open and enables add-on drivers and devices in pre-OS space.
EOL	End of Life; the last life-cycle phase for a platform.
Golden Measure	A Golden Measure is a measurement that is made of a known platform configuration and provided as input to an attestation scheme, for example.
Hack Attack	A platform attack that involves primarily software techniques and typically does not include physical access to the platform.
Hypervisor	See VMM
IPMI	Intelligent Platform Management Interface; an open standard defining a server management infrastructure and interfaces.
IT	Information Technology
IaaS	Infrastructure as a Service; managed platform provider.
ME	Management Engine; an embedded management service processor typical of Intel x86 platforms.
Measurement	This term refers to the act of capturing a uniquely identifying piece of information about a software or firmware component; in practice the component is usually hashed and that hash is stored securely for later analysis or decision making,
NIST SP800-147A/B	Special Publication detailing a means of performing secure BIOS

	update on client and server platforms.
NVRAM	Non-Volatile Random Access Memory
ODM	Original Device Manufacturer; often refers to a contract manufacturer of hardware platforms.
OEM	Original Equipment Manufacturer; the entity who produces the hardware platform.
OSV	Operating System Vendor
PCIe	PCI Express; used here largely to define add-in IO devices
PCR	Platform Configuration Register; a TCG term defining registers within a TPM used to capture platform measures.
PEI	Early phase of UEFI firmware execution that is closed by design and functions to enable critical onboard and firmware resources.
RTE	Root of Trust of Enforcement;
RTM	Root of Trust for Measurement; a TCG term that defines where secure “measurement” starts.
RTR	Root of Trust for Reporting; a TCG term that indicates a secure device that is used to convey security information such as measurements.
RTS	Root of Trust for Storage; a TCG term that indicates a secure device that can retain data such as measurements.
RTU	Root of Trust for Update; a term that defines a hardware rooted update scheme, for example SMI based firmware update.
RTV	Root of Trust for Verification; a term that defines a rooted verification entity.
SaaS	Software as a Service; a platform provided to host user applications typically in a Cloud.
S-CRTM	Static CRTM; a CRTM that is tied to a immutable hardware event such as reset.
SEC	A UEFI firmware phase that precedes PEI and is the first UEFI defined firmware phase. This firmware phase is typically quite brief and contains OEM specific code and not extensible at runtime.
Shack Attack	A platform attack that utilizes physical access and relatively low-cost methods based on tools and techniques readily available to the hobbyist or enthusiast.
SMBIOS	A standard defining a data structure and interface which details many aspects and capabilities of a platform. This table is typically created by UEFI firmware and consumed by OS.
SMM	System Management Mode; a processor functional mode that executes out of discrete memory segment and is invisible to the hosted OS or VMM. SMM is programmed into the platform by the UEFI firmware and operates during platform runtime/Production.
TCG	Trusted Computing Group; an open consortium of hardware, silicon, software and firmware vendors which seeks to define

	secure platform initiatives.
UEFI	Unified Extensible Firmware Interface; the open standard specification that dictates the interfaces to and services of a UEFI firmware implementation.
VM	Virtual Machine; a virtualized operating environment created by a VMM.
VMM	Virtual Machine Monitor; a software system that runs either on or in place of an OS and creates VMs. See also hypervisor.
VMMV	VMM Vendor

References

[Authenticode] Windows Authenticode Portable Executable Support
http://www.microsoft.com/whdc/winlogo/drvsign/Authenticode_PE.msp

[EDK2] EFI Developer Kit www.tianocore.org

[FRAMEWORK] Intel Framework Specifications www.intel.com/technology/framework

[HIPAA] Health Insurance Portability and Accountability Act, <http://www.hipaa.org/>

[IBM TCG] Challener, et al, *A practical guide to trusted computing*, IBM Press, 2008

[PCI] Payment Card Industry, <https://www.pcisecuritystandards.org/>

[RSA] PKCS#1: RSA Cryptography Standard, Version 2.1
<http://www.rsa.com/rsalabs/node.asp?id=2125>

[Secure Computing Book] David Grawrock, *Dynamics of a Trusted Platform*, 2nd edition, Intel Press, 2009

[SHA] Secure Hash Algorithm, FIPS 180-2 <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>

[TCG Overview] Trusted Computing Group Benefits,
http://www.trustedcomputinggroup.org/trusted_computing/benefits

[TCG Main] TCG TPM Main specification, version 1.2 www.trustedcomputinggroup.org

[TCG PC] TCG PC Client Specification, version 1.2 www.trustedcomputinggroup.org

[TCG EFI Platform] EFI TCG Platform Specification, version 1.2
www.trustedcomputinggroup.org

[TCG EFI Protocol] EFI TCG Protocol Specification, version 1.2
www.trustedcomputinggroup.org

[TCG TIS] TCG TPM Interface Specification www.trustedcomputinggroup.org

[THREAT] [Frank Swiderski](#), [Window Snyder](#) "Threat Modeling," Microsoft

[TRUSTZONE] http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[UEFI Book] Zimmer, Rothman, Marisetty, *Beyond BIOS 2nd edition*, Intel Press, 2010,
http://www.intel.com/intelpress/sum_efi2.htm

[UEFI Main Specification] UEFI Specification, Version 2.3 www.uefi.org

[UEFI Overview] Mark Doran, Vincent Zimmer, Michael Rothman, "Beyond BIOS: Exploring the Many Dimensions of the Unified Extensible Firmware Interface," in *Intel Technology Journal - UEFI Today: Bootstrapping the Continuum*, Volume 15, Issue 1, pp. 8-21, October 2011, ISBN 978-1-934053-43-0, ISSN 1535-864X <http://www.intel.com/technology/itj/2011/v15i1/index.htm>
<http://www.intel.com/technology/itj/2011/v15i1/pdfs/Intel-Technology-Journal-Volume-15-Issue-1-2011.pdf>

[UEFI PI] Isaac Oram, Tim Lewis, Vincent Zimmer, "Silicon Enabling in a Modular Architecture," in *Intel Technology Journal - UEFI Today: Bootstrapping the Continuum*, Volume 15, Issue 1, pp. 22-39, October 2011, ISBN 978-1-934053-43-0, ISSN 1535-864X <http://www.intel.com/technology/itj/2011/v15i1/index.htm>
<http://www.intel.com/technology/itj/2011/v15i1/pdfs/Silicon-Enabling-in-a-Modular-Architecture.pdf>

[UEFI Security] Magnus Nystrom, Martin Nicholes, Vincent Zimmer, "UEFI Networking and Pre-OS Security," in *Intel Technology Journal - UEFI Today: Bootstrapping the Continuum*, Volume 15, Issue 1, pp. 80-101, October 2011, ISBN 978-1-934053-43-0, ISSN 1535-864X <http://www.intel.com/technology/itj/2011/v15i1/index.htm>
<http://www.intel.com/technology/itj/2011/v15i1/pdfs/UEFI-Networking-and-Pre-OS-Security.pdf>

[UEFI Secure Boot] Rosenbaum, Zimmer, "A Tour Beyond BIOS into UEFI Secure Boot," Intel Corporation, July 2012,
http://iweb.dl.sourceforge.net/project/edk2/General%20Documentation/A_Tour_Beyond_BIOS_into_UEFI_Secure_Boot_White_Paper.pdf

[UEFI Shell Book] Rothman, Lewis, Zimmer, Hale, *UEFI Shell*, Intel Press, 2009,
http://www.intel.com/intelpress/sum_eshl.htm

[UEFI Overview] Zimmer, Rothman, Hale, "UEFI: From Reset Vector to Operating System," Chapter 3 of *Hardware-Dependent Software*, Springer, February 2009

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5,
Version 1.2 www.uefi.org

Authors

Bill Jacobs (billjac@cisco.com) is a Technology Leader in Cisco's UCS server development organization.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Principal Engineer with the Software and Services Group at Intel Corporation.

This paper is for informational purposes only. THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel, the Intel logo, Intel. leap ahead. and Intel. Leap ahead. logo, and other Intel product name are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright 2012 by Intel and Cisco. All rights reserved

