UEFI NETWORKING AND PRE-OS SECURITY

Contributors

Magnus Nyström
Partner Architect at Microsoft Corporation

Martin Nicholes
Security Architect at Insyde Software

Vincent J. Zimmer
Principal Engineer at Intel Corporation

Readers will get an understanding for the scope and objective of the Unified Extensible Firmware Interface (UEFI) specification and the UEFI technology's role as a foundation for pre-OS security and networking. This includes the platform boot process from local and remote media, assets to be protected, threats against those assets, and the various technologies that allow for their protection. In addition to a review of these technologies, forward-looking capabilities and approaches related to UEFI are discussed.

UEFI Architecture

This section provides a basic overview of the UEFI firmware architecture. For a more detailed view of UEFI architecture, see the UEFI Web site [1] and *Beyond BIOS*[2]. UEFI provides a standard interface to shield the operating system from hardware changes. It has the ability to host chipset and peripheral boot drivers. It provides services both during the boot process and during runtime, available through architected tables.

Architecture Diagram

Figure 1 shows how an operating system loader relies upon the services provided by UEFI to launch the OS kernel. Other industry standard interfaces, such as ACPI, are available on a UEFI platform.

"UEFI provides an orderly method for loading drivers and handling interdependence between them."

UEFI provides an orderly method for loading drivers and handling interdependence between them. Once the necessary drivers are loaded, a series of preconfigured boot paths or boot devices are attempted by a boot manager or dispatcher. During a boot attempt, a file is loaded using UEFI boot-time services and is executed. This file load operation is a good point to secure, as will be discussed in the section on UEFI 2.3.1 Secure Boot.

Failures of these boot attempts are handled by returning to the UEFI boot manager. However, once a boot attempt executes an OS boot loader that exits the UEFI boot-time environment, the UEFI boot-time environment cannot be re-entered, as shown in Figure 2.

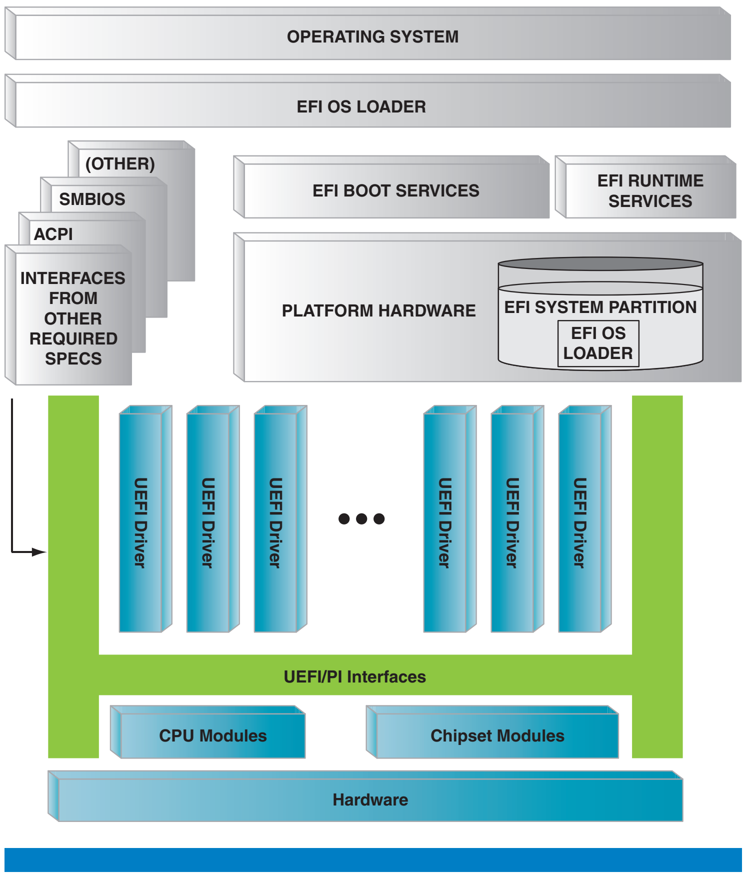

Figure 1: Firmware layering, including image loading
(Source: Intel Corporation, 2011)

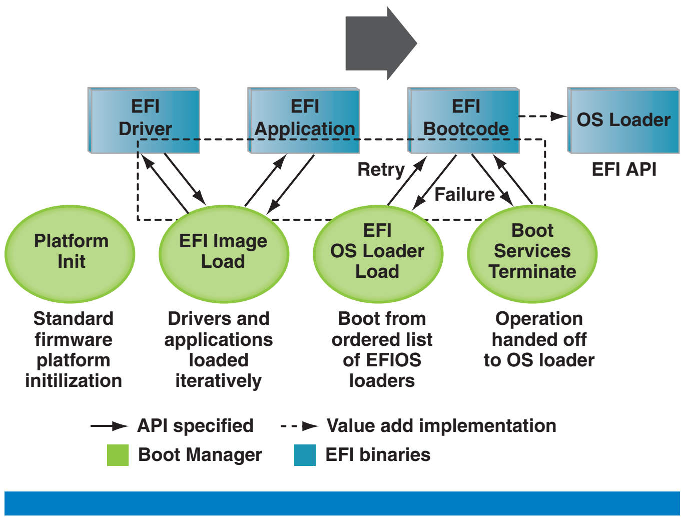

Figure 2: Firmware boot flow
(Source: Intel Corporation, 2011)

Services

Table 1 shows the boot-time services categories available to a UEFI application. The most critical services surround image handling, as this is way that new code is loaded and executed.

Name (Boot-time)	Description
Events	Create, Close, Signal, WaitFor, Check, SetTimer
Task Priority Level	Raise, Lower
Memory	Allocate, Free, GetMemoryMap
Protocol	Install, UnInstall, ReInstall, Notify, Locate, Open, Close, Information, Connect, Disconnect
Image	Load, Start, Unload, EntryPoint, Exit, ExitBootServices
Misc	SetWatchDogTimer, Stall, CopyMem, SetMem, GetNextMonotonicCount, InstallConfigurationTable, CalculateCrc32

Table 1: UEFI boot services
(Source: Intel Corporation, 2011)

Table 2 shows the runtime service categories available in the UEFI environment. These calls can change the state of the platform by modifying volatile data, nonvolatile data and real-time clock settings. Control of platform state is another critical area to review with respect to platform security.

"Control of platform state is another critical area to review with respect to platform security."

Name (runtime)	Description
Variables	Get, GetNext, Set, Query
Time	Get, Set, GetWakeup, SetWakeup
Virtual Memory	SetVirtualAddressMap, ConvertPointer
Misc	GetNextHighMonotonicCount, ResetSystem, UpdateCapsule, QueryCapsuleCapabilities

Table 2: UEFI runtime services
(Source: Intel Corporation, 2011)

Earlier versions of UEFI (UEFI 2.0) provided little in the way of security services. For example, there was no way to save platform data such that only the creator of the data could modify or delete it. In addition, earlier versions of UEFI did not provide basic authentication tools, such as hashing or decryption capabilities. This led to several development directions in an attempt to add security to the UEFI environment: one involved measuring and recording platform state (trusted computing), and the other moved to enhance UEFI security capabilities (secure boot). The next section starts with conceptual models of a computer system and leads into a detailed review of newer UEFI capabilities that enhance UEFI security.

Underlying UEFI Implementation

Although the prior discussions have been focused on aspects of UEFI, UEFI can be built upon another firmware standard called the Platform Initialization (PI) Standard [3]. The PI Standard provides a standard flow and architecture for early machine initialization. The security issues related to PI are discussed here, along with some related issues with respect to UEFI.

"The PI Standard provides a standard flow and architecture for early machine initialization."

UEFI by design allows files to be discovered and run from arbitrary locations in the system. Following is an overview of the power of this facility and touches on the security risks associated with loading files during the boot process.

Platform Initialization (PI) Standard

The PI specification allows a UEFI-based system to ride upon a standard firmware architecture. The boot process of such a system proceeds through a series of phases. Each phase has its own unique security advantages as well as its own risks.

SEC, PEI, and DXE

The security (SEC) phase of the PI platform boot must handle different types of platform reset events. SEC is also the root of trust for the system, providing a control point for further launch of firmware on the system. The main advantage of the SEC phase is to provide an anchor point from which to build an authenticated boot process. The SEC phase must find and transfer control to the Pre-EFI Initialization (PEI) phase of the boot process, once temporary memory is available. Of course, depending on location of the PEI code, as well as platform policy, the PEI code must be authenticated before execution.

The main purpose of the PEI phase is to provide an environment for PEI module execution. So early in the PEI phase, the PEI dispatcher is started. PEI modules typically perform low-level platform initialization of embedded devices and chipset, such as serial port initialization. Another requirement of the PEI phase is to discover platform information, create a database of this information in hand-off blocks (HOBs), and pass the database onto the DXE phase of platform boot. Again, PEI modules can be made subject to authentication before running the module. Typically PEI modules are part of the core firmware of the platform, and could be considered static and trusted for a particular platform model. However, there is no requirement in the PI specification regarding location of PEI modules, and so there could be platforms where some or all of the PEI modules would need to be authenticated to maintain platform integrity.

"The main purpose of the PEI phase is to provide an environment for PEI module execution."

DXE as Preferred UEFI Core Embodiment

The final stage of the PI specification starts when the DXE Initial Program Load (IPL) is located and executed by the PEI phase. The DXE stage consumes information about the platform through the HOBs and creates a more complete environment for drivers. The DXE dispatcher is responsible for finding and launching DXE drivers, and in this phase drivers may come from many sources, as will be described later.

"The DXE dispatcher is responsible for finding and launching DXE drivers."

Platform Manufacturer (PM) Authority

PI code and UEFI core only come from a system board manufacturer and are not arbitrarily extensible by third parties. A platform manufacturer is responsible for the delivery, authenticity, and protection of firmware that implements the PI and core UEFI services. In order to maintain platform integrity, this firmware must be changed under platform manufacturer control.

Achieving this control is platform-dependent, but true security methodology, including hashing and cryptographic authentication, is required. Some tools for firmware update have been provided in the UEFI specification and are touched upon later. Controlled firmware update is required for secure boot, as well as for TCG measured boot.

CRTM for TCG Boots

The Trusted Computing Group (TCG)[4] standard requires a Core Root of Trust for Measurement (CRTM). One method to implement a CRTM is to use a Static CRTM (S-CRTM), which is the core platform firmware provided in the flash part that comes with the system, as described earlier in this section. The S-CRTM is responsible for measuring any code that executes after the S-CRTM. Unlike secure boot, the measured boot only provides a record of all the firmware modules that have been run and does not provide any judgment about the integrity of the firmware modules.

Secure, Rollback-Protected Updates

In order to meet the need for a controlled method of updating platform firmware, several tools have been developed. UEFI 2.3 provides a protocol called Firmware Management Protocol. This protocol provides a standard way to control firmware on a platform. Any device in the system, from a PCIe card to the main system firmware, can expose this protocol so that a single tool can track and update firmware revisions. Although this protocol does not explicitly define firmware image authentication support, the support can be added into a system in various ways.

Another mechanism supporting secure firmware updates is the use of an EFI Update Capsule. If possible, firmware can perform the firmware update at runtime, or perhaps during the next reset of the platform. Again, security of a capsule update is the responsibility of the consumer of the capsule. Therefore, the structure of the capsule should contain authentication information necessary to maintain platform firmware integrity.

"the structure of the capsule should contain authentication information necessary to maintain platform firmware integrity."

Firmware update solutions based on either of these techniques must provide the ability to detect firmware rollback conditions. This is critical, due to the possibility of an attacker using an older firmware image with a known vulnerability. There may be situations where firmware must be rolled back due to some failure, but in those cases, platform operator intervention is required. This prevents an automated firmware rollback attack.

UEFI 2.3.1 Secure Boot

An introduction to UEFI's Secure Boot support is provided here as well as coverage of the various components associated with securely booting a platform.

Background

A core aspect of UEFI Secure Boot is its ability to leverage digital signatures to determine whether an EFI driver or application is trustworthy. After an initial brief introduction to the concept of digital signatures, this section describes how UEFI Secure Boot makes use of them—and other cryptographic constructs such as cryptographic hash functions—to establish a trustworthy system boot. We also discuss deployment aspects as well as how Secure Boot assists in the secure load of an operating system.

Cryptography Primer

Naturally, an article such as this one cannot provide anything but a high-level introduction to cryptography. The interested reader is referred to, for example, [5] for a more complete treatment of the topic.

Cryptographic Hash Functions

As is well known, a *hash function h* is a mathematical function from a large domain X of values into a smaller range Y. Normally, it is desirable that the projection of X into Y is such that the values of Y are evenly distributed; this reduces the risk of "hash collisions", that is, that two values x_1 and x_2 that belong to X will have the same image y: $h(x_1) = h(x_2)$.

A *cryptographic hash function f* is a hash function that meets some additional properties:

- It shall be computationally expensive to find x_1, x_2 such that $f(x_1) = f(x_2)$ (this property is often referred to as *collision resistance*)
- It shall be computationally expensive to find an x such that $f(x) = y$ for a given y (this property is often referred to as *pre-image resistance*)

In addition, cryptographic hash functions are usually designed to allow the quick computation of $y = f(x)$ given x. Some examples of well-known cryptographic hash functions are MD5 [6], SHA-1 [7], and the SHA-2 family [7].

"cryptographic hash functions are usually designed to allow the quick computation of y = f(x) given x."

Cryptographic hash functions are useful in applications that need to protect data integrity since their properties make it difficult for an attacker to modify a datum (such as the image of an executable) without detection of a party with access to the original hash value. Likewise, replacement of an image for another one will be infeasible if the party that checks the image has stored the original image's hash value (pre-image resistance).

Public-Key Cryptography and Digital Signatures

Public-key cryptography was discovered in the late 1970s by Whitfield Diffie, Ralph Merkle, and Martin Hellman [8]. The main notion until then was that in order to carry out confidential communication between two parties the two parties had to agree—out of band—on a shared secret. This secret was then used to encrypt the communication. Diffie, Merkle, and Hellman's discovery of a technique to establish shared secrets without the need for pre-agreements and without having to know in advance with whom to communicate securely was

"The public key can be presented to anyone whereas the private part of the key is known only to its owner."

revolutionary and was based on the concept of a new form of cryptographic keys consisting of a *public key* part and a *private key* part. The public key can be presented to anyone whereas the private part of the key is known only to its owner.

Ronald Rivest, Adi Shamir, and Leon Adelman built upon Hellman, Merkle, and Diffie's ideas when they invented the RSA cryptosystem a few years later. With RSA, an entity *A* holding a private key may not only use that key to decrypt data sent to him by someone knowing the public key of *A* but *A* may also use his private key to *digitally sign* some data, as shown in Figure 3. Anyone with knowledge of the public key of *A* and access to the signed data may then verify mathematically that the data could only have been signed with knowledge of *A*'s private key–that is, only *A* could have generated the message–and that it has not been modified.

This property is extremely useful and has numerous applications. One of them is of interest in the context of UEFI Secure Boot: an issuer of a firmware driver may *digitally sign* this driver to vouch for the driver's authenticity as well as provide assurance to a verifier that the driver has not been tampered with, as shown in Figure 4.

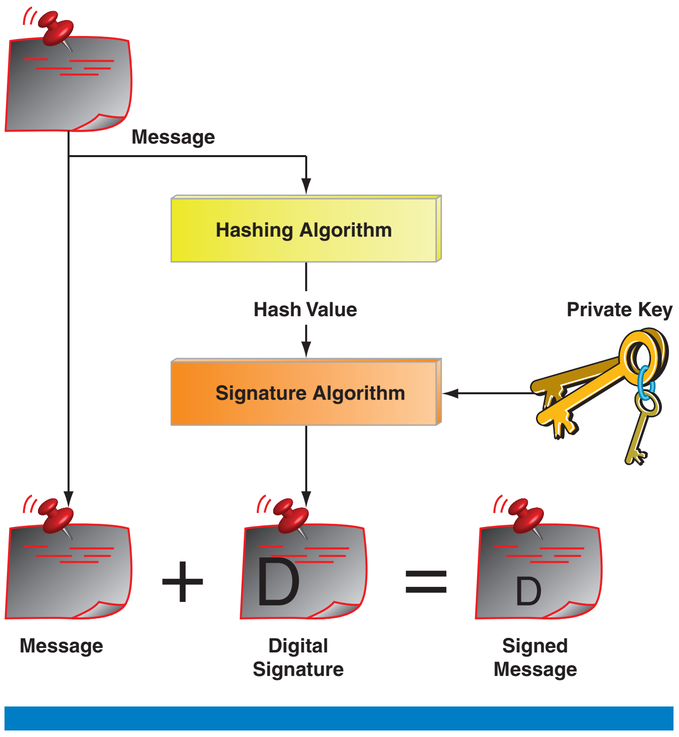

Figure 3: Digital signature creation process (Source: UEFI Specification)

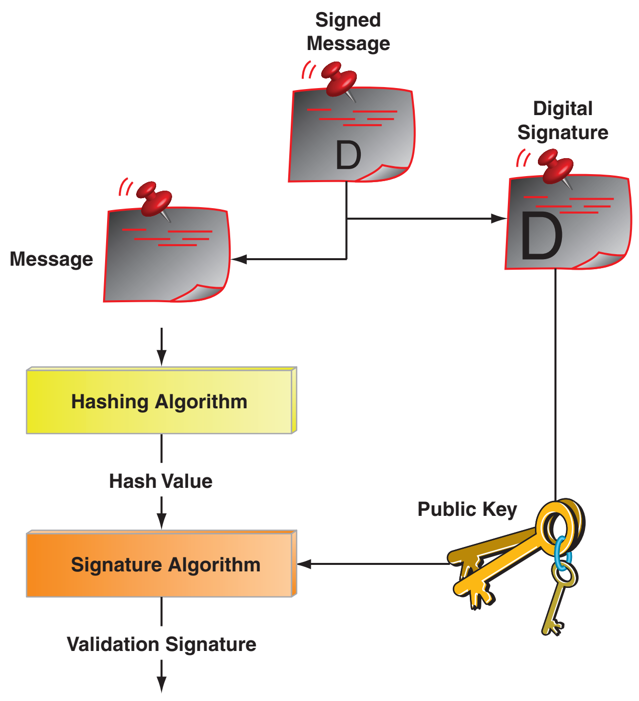

Figure 4: Digital signature verification process (Source: UEFI Specification)

Public-Key Infrastructures

One issue conveniently overlooked in the previous discussion is *how* a relying party (a party that needs to verify a digital signature) may determine that *A* actually is the holder of the public-private key pair that is claimed to be held by *A*. To achieve this, the concept of *Public-Key Infrastructures* (PKI) has been introduced. At a high level, a third party produces digital signatures binding a public key to the holder of the private key corresponding to the public key. Such signatures are commonly referred to as *digital certificates* and the issuer of the certificates are referred to as a *certificate authority* or CA. The problem is now transferred to one of distributing and trusting CAs. CAs may also be specialized; for example, a given CA may only issue certificates for individuals (stating that individual *Alice* holds public key *A*) whereas another CA only issues certificates for code-signing purposes (certificates stating that signer *S* can use his private key only to digitally sign computer code). Often these specialized CAs have certificates issued by a higher-layer CA. This is convenient as it reduces the distribution problem to the problem of distributing top-level CAs (commonly referred to as *trust anchors* or *root CAs*). The top-level CAs self-sign their certificates, meaning that they sign their own certificates. Technically this is not required; all that is required is for a relying party to know that a given root CA is in possession of a given public key, but for convenience's sake (consistent use of certificates) self-signed certificates are often used.

"At a high level, a third party produces digital signatures binding a public key to the holder of the private key corresponding to the public key."

PKIs with CAs also allow for *revocations*. A revocation is essentially a statement made by a CA that it is no longer the case that the unique binding between an entity *B* and the private key whose public key counterpart was present in *B*'s certificate holds. This could be due to *B* losing his private key or because it is suspected to have been compromised or some other reason. Once a certificate has been revoked, it can no longer be used to verify signatures made with the corresponding private key. For this reason, it is sometimes important to augment a digital signature with a *time stamp* allowing a relying party to determine if a signature was created before a given point in time (the time stamp itself is usually created by a third party called a timestamping authority, who also needs to be trusted by the relying party). In particular, it allows a relying party to determine if the signature was made before the certificate used to verify the signature was revoked.

"Once a certificate has been revoked, it can no longer be used to verify signatures made with the corresponding private key."

For a technical description of PKIs, see [9].

Functionality

With this introduction to the cryptographic building blocks that hash functions and digital signatures constitute, we are now ready to look at their application in UEFI Secure Boot.

Root of Trust

As mentioned, in order to rely on a digital signature, the relying party needs to know that the purported signer indeed was in possession of the private key that was used to create the digital signature. As CAs are commonly used as "trusted third parties" to convey this knowledge, the distribution of CAs (and in particular root CAs) is what enables a relying party to establish trust in a signature.

In UEFI Secure Boot, there is a particular key that constitutes the basis for the trust environment. This key is the Platform Key (PK) and the holder of this key is able to modify any of the other trust anchor lists that exist on a platform. Typically, the PK is held by a device manufacturer, but it may also be held by an enterprise that wishes to have full control of the UEFI Secure Boot environment of PCs in their organization.

"Typically, the PK is held by a device manufacturer, but it may also be held by an enterprise that wishes to have full control of the UEFI Secure Boot environment of PCs in their organization."

Trust Anchors

In addition to the PK, UEFI Secure Boot maintains two additional trust anchor databases:

- The Key Exchange Key (KEK) database
- The Allowed signature database

The former database contains those trust anchors that are allowed to modify the Allowed signature database. The Allowed signature database in turn contains those trust anchors that are used when verifying the signature on UEFI firmware images.

There is also a third database, the Forbidden database. This database identifies signers that have been revoked and can therefore no longer be trusted.

Signed Firmware

In UEFI Secure Boot, firmware that isn't explicitly *White-listed* (see below) must be digitally signed and time-stamped in such a way that the UEFI Boot Manager can verify the image's signature. UEFI uses the PE/COFF [10] format and the Microsoft Authenticode Specification [11] for signatures over PE/COFF images. The PE/COFF format contains a date/time field that may be used for timestamp purposes.

White-Listing and Black-Listing

UEFI Secure Boot extends beyond mere digital certificates for determining whether a signed UEFI firmware component is trustworthy or not, however. Besides digital certificates, UEFI Secure Boot also allows an authorized entity to identify a particular image hash as trustworthy (or not). If the hash of a given image is present in the Allowed database but not in the Forbidden database, then the fact that only an authorized entity (either the PK or a trust anchor in the KEK database) can modify the contents of the databases suffices to determine the validity of an image.

"UEFI Secure Boot extends beyond mere digital certificates for determining whether a signed UEFI firmware component is trustworthy or not."

This scheme carries an advantage over most existing PKIs in that individual images can be allowed (white-listed) or revoked (black-listed); there is no requirement to revoke signers—an operation with potentially much more far-reaching implications since *all* images signed by that signer would be inoperable on a Secure Boot system. The usefulness of this is apparent when one considers a scenario where a specific driver from a vendor has been found to contain a vulnerability. Clearly, the vendor need (and most often should) not be considered malicious because of this and hence the natural action is to just black-list the driver (by adding its hash to the Forbidden database). A useful application of white-listing is when a new driver is detected but isn't signed. In this situation, an authorized entity may explicitly add the hash of the driver image to the Allowed database, allowing future system boots with it.

Authenticated Variables

From the above logic regarding malicious code, it follows that a party that wishes to update any of the databases (KEK, Allowed, Forbidden), must be able to do so only with proper authorization. The authorization model chosen for UEFI Secure Boot follows a consistent design in that it leverages digital signatures itself and hence UEFI variables that requires authentication of the caller in order to be updated are referred to as *Authenticated Variables*. In short, in order to update an authenticated variable, the caller creates a signature over the new variable value. The trusted firmware then verifies this signature before updating the value of the variable.

"The authorization model chosen for UEFI Secure Boot follows a consistent design in that it leverages digital signatures."

What would happen if an early version of a Forbidden database update was captured by an attacker and later on replayed after the attacker's key had been revoked? In a worst-case scenario, the attacker would be able to continue to have his firmware booted because the platform's black-list would never include the attacker's certificate. To prevent this, any authenticated variable update includes a timestamp. Trusted UEFI firmware must ensure that the timestamp (which is part of the signature) is later in time than the timestamp currently associated with the authenticated variable before updating the variable's value. There is one exception to this, though: UEFI 2.3.1 introduces functionality that allows an entity to append a value to an existing authenticated variable such as the Forbidden database. In this case, the timestamp is unimportant as the update does not affect values already present in the database. However, even in this case the firmware must update the time associated with the variable if the timestamp is later than the current time associated with the variable, since it will serve as a "last known update" time and may be required if a later "write" (rather than "append") update request is made.

"Associated with the rollback prevention of the authenticated variables is the rollback prevention of drivers themselves."

Rollback Prevention

Associated with the rollback prevention of the authenticated variables is the rollback prevention of drivers themselves. Imagine a scenario where a driver is known to have a vulnerability and hence a new version is distributed. Unless the UEFI implementation has maintained some means to determine the version (in an abstract sense) of the driver, it would not be able to tell if the new driver version actually is an older version of the driver currently installed rather than a newer one. This situation would allow an attacker to redistribute the vulnerable version of the driver and potentially gain control over a large number of platforms.

To remediate this situation, compliant UEFI 2.3.1 platforms must implement rollback prevention of firmware components. They may do so by using the TimeDateStamp field from the PE/COFF image header of the firmware in question or by otherwise associating a version with the driver. Firmware updates are verified against the time (or version) of the currently used driver and only if the new firmware is found to be "later" than the currently used does the update occur. NIST has recently released [12], which describes further guidelines for the protection of firmware and firmware updates.

Integration

A UEFI Secure Boot–capable platform may consist of many firmware components, including:

- UEFI Boot Code
- UEFI Boot Manager
- UEFI Drivers
- UEFI Applications
- UEFI OS Loader

"When Secure Boot is enabled, the Platform Initialization module is expected to contain a public RSA key held in read-only memory."

When Secure Boot is enabled, the Platform Initialization module is expected to contain a public RSA key held in read-only memory. This key is used to verify the UEFI Boot Manager. Once the image has been verified, the UEFI

Boot Manager is launched and continues the platform boot by loading and initializing the EFI images as specified by boot order variables. Before the loading of each EFI image, the Boot Manager uses the Allowed and the Forbidden signature database to determine if the image should be allowed to be loaded. Only images whose signature matches an entry in the Allowed database (or whose signer is present in the Allowed database) and is not present in the Forbidden database (and whose signer is not present in the Forbidden database) are allowed to be loaded and initialized.

Information about images that are rejected due to failed signature verification or due to being associated with an entry in the Forbidden database will be stored in a UEFI table for later consumption (and possible remediation) by the platform operating system.

"Information about images that are rejected due to failed signature verification or due to being associated with an entry in the Forbidden database will be stored in a UEFI table."

Once the UEFI Boot Time and UEFI Runtime services are available, updates are possible to the KEK, Allowed, and Forbidden databases (the PK may also be modified, but this would be a rare operation). In order to protect against unauthorized updates to these databases (that normally are held in NVRAM) by a faulty EFI component, implementations should preferably only allow such updates to occur via a trusted platform component, thereby guaranteeing that the signatures on such updates are valid before committing to the updates.

Deployment

There are various phases of a platform lifetime, some of which are described in [22]. For UEFI Secure Boot, this provisioning and field maintenance entails additional considerations described below.

Manufacturing Time

During manufacturing of a platform that supports Secure Boot, it is expected that the platform vendor will provision the initial Secure Boot configuration. In a nutshell, this configuration will consist of:

- Creating the initial Allowed database
- Creating the initial Forbidden database
- Creating the initial KEK database
- Setting the Platform Key (PK).

Once the final step above has occurred, the platform will no longer be in Setup mode and any further changes to the databases will require the update to be properly authorized (digitally signed).

Assuming that the Allowed database contains signers for the OS loaders that later on will be booted on the platform, the initial Secure Boot configuration is now complete. If the default mode for the platform boot is Secure Boot, the platform manufacturer may now also set the Secure Boot variable to True; this provides information to the OS loader about the EFI platform's enforced security policy.

Field Management

Once a Secure Boot device has been deployed in the field, there may, as previously indicated, occasionally be a need for modifications to the Secure Boot-related databases. For example, a firmware image may have been found to be vulnerable to a security threat and hence should not be allowed to be in the boot path any longer. Likewise, the platform owner may want to add a new KEK signer to allow for more flexible updates to the Allowed or Forbidden database.

"any update to the Authenticated database variables needs to be signed."

Such management occurs through the EFI Runtime Services and in particular the Variable Services. As described, any update to the Authenticated database variables needs to be signed. The signature must be in the form of a PKCS #7 [20] signature and must contain a timestamp to allow the EFI Runtime Services to verify that the update is not a replay of a previous update.

Recovery Situations

There may be situations when a new driver has been found to be incompatible with the platform or contain a flaw that makes it unsuited for use on the device. Since Secure Boot does not allow for programmatic rollback of firmware (because this would open an attack vector in that earlier versions of the firmware with known vulnerabilities could be reintroduced), this could present an issue that in the worst case would render the device unusable.

Fortunately, however, platforms and platform administrators may avoid this situation in at least two ways:

- By requiring a physically present user to accept the "older" firmware component to be reintroduced.
- By re-signing the earlier firmware image, thereby creating a new image with a "fresher" time-stamp than the recently installed driver.

Such functionality provides robustness without compromising system security.

"Since the provenance of the OS and its loader is often different than that of the system board UEFI firmware, the secure boot of the loader is how a vendor's OS is cryptographically bound to a platform likely produced by another vendor."

OS Load Aspects

One usage of UEFI Secure Boot includes the invocation of the operating system loader. Recall that in UEFI, the loader is a UEFI executable with a subsystem type of boot service application. Since the provenance of the OS and its loader is often different than that of the system board UEFI firmware, the secure boot of the loader is how a vendor's OS is cryptographically bound to a platform likely produced by another vendor. The various modalities of this bootstrap are described below.

"The EFI Boot Manager concludes by verifying the signature of the OS Loader and then relinquishing control to the OS Loader."

Fixed Local Storage

In the most common case, the OS is present on some fixed local storage, such as a local hard disk. The EFI Boot Manager concludes by verifying the signature of the OS Loader and then relinquishing control to the OS Loader. Once in control, the OS Loader performs the OS load and, at some point in this process, calls the ExitBootServices() Boot Service function. After this call, only UEFI Runtime Services will be available.

Since EFI Variable Services are Runtime services, the OS may still perform updates to Secure Boot-related variables; however, the responsibility for verifying the validity of such updates still rests with the trusted firmware.

Removable Media
UEFI also allows OS load from removable media. The EFI Boot Manager reads the Boot order variable in order to determine whether to prioritize boot from removable media or from fixed local storage. Secure Boot still determines the validity of the OS Loader.

Network-based Access
A third alternative is to perform a network-based OS load, such as using the PXE [13] pre-boot environment protocol to download an OS loader.

The BIOS Integrity Services (BIS) protocol was deployed prior to the advent of the UEFI 2.3.1 Secure Boot. This was done because legacy BIOS loaders could not accommodate an embedded signature. As such, a detached signature and the Boot Object Authorization (BOA) were used for BIOS. During EFI1.02 definition, this usage was simply mapped to EFI. The BOA had limitations, like no chaining or multiple authorities, for either BIOS or EFI.

UEFI BIS also did not have a way to securely provision the BOA. Going forward, UEFI Secure Boot replaces BIS usage since Secure Boot mitigates these limitations via 1) certificate-based key storage with chaining, 2) authenticated variables to authorize the updates, and 3) a multiple-entry list of allowed signers.

In addition to authenticating the integrity of the code objects, the transport can either be integrity-protected and/or have confidentiality controls applied by means of the UEFI IPsec protocol [14].

Relationship with Measured Boot

There is often confusion surrounding the terms Secure Boot and Measured Boot. The previous section clearly described various components associated with Secure Boot, and this section describes Measured Boot and how the terms relate to one another.

Background
Measured Boot is the term used for a boot in which each component loaded during the boot process is measured into a trusted environment such as a TPM [4]. The measurements taken during the boot process may be provided to a third party for attestation purposes ("Authenticated Boot"), that is, a statement about the device's posture as indicated by the boot measurements. The combination of Secure Boot and Measured Boot is commonly referred to as *Trusted Boot* [15].

"Measured Boot is the term used for a boot in which each component loaded during the boot process is measured into a trusted environment such as a TPM."

For a more thorough treatment of Measured Boot, the reader is referred to [16].

Complementary Functionality
While Secure Boot provides *local verification* of boot integrity, Measured Boot provides the basis for attested statements about the platform's configuration. These statements may be provided to third parties for *remote* verification of the

"Measured Boot provides the basis for attested statements about the platform's configuration."

platforms configuration, such as, for example, in order to prove a platform's conformance to some defined security policy.

UEFI Secure Boot therefore works in tandem with, but independent of, Measured Boot. UEFI supports Measured Boot and the process for performing Measured Boot on a UEFI platform is described in [17] and [18]. [18] states that any UEFI variables that have an effect of platform configuration shall be included in the measurement process. Because of this, Measured Boot must include the Allowed, Forbidden, KEK, and PK variables (databases) in its measurements of a Secure Boot-configured platform.

Example Usage Scenarios

This section illustrates use of Secure Boot functionality and Signature Databases.

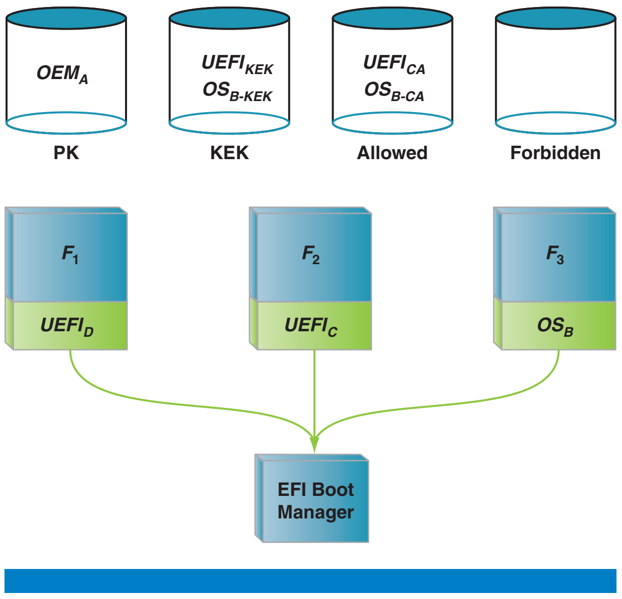

Figure 5: Boot succeeds
(Source: Microsoft Corporation, 2011)

"the boot succeeds because all of the image signatures are verified and no image (or image signer) is present in the Forbidden database."

In Figure 5, the boot succeeds because all of the image signatures are verified and no image (or image signer) is present in the Forbidden database. (In Figure 5, image F_1 is signed by a certificate issued by the UEFI CA $UEFI_{CA}$ identifying the vendor as D, image F_2 is signed by a certificate issued by the UEFI CA $UEFI_{CA}$ identifying the vendor as C, and image F_3 is signed by a certificate issued by the OS vendor B CA $OS_{B\text{-}CA}$ (identifying the vendor as B).

At this point, an update to the Forbidden database with the image hash of firmware image F_1 occurs through a UEFI *SetVariable*() call. The update is authenticated with a $UEFI_{KEK}$, and since the timestamp on the signature was fresher than the timestamp associated with the existing Forbidden database value (which was empty) the update succeeds. We now have the situation illustrated in Figure 6.

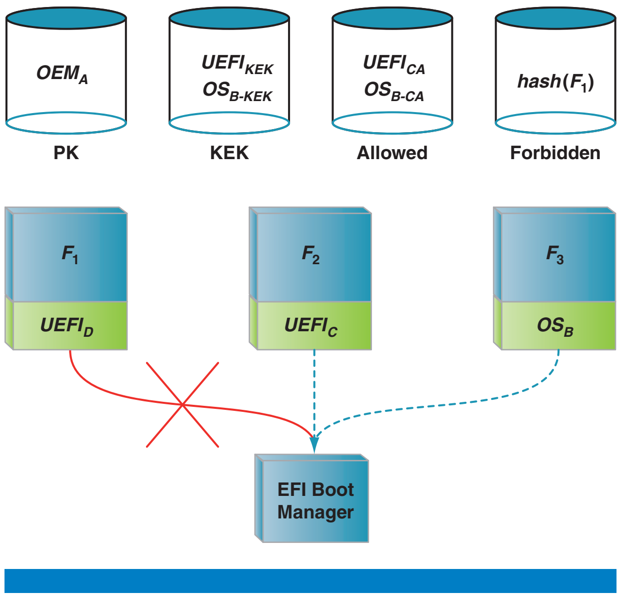

Figure 6: Failure due to presence of F_1 in the boot path
(Source: Microsoft Corporation, 2011)

A Secure Boot along this boot path will fail because the hash of image F_1 is now in the Forbidden database. The dashed line indicates that the UEFI Boot Manager never attempts to load F_2 or F_3.

Next, a firmware update occurs in which F_1 is replaced with F'_1. The update is signed with a key chaining back to $UEFI_{CA}$ and the timestamp in the signed image as well as the signature itself is verified before the update is committed. This gives the situation illustrated in Figure 7.

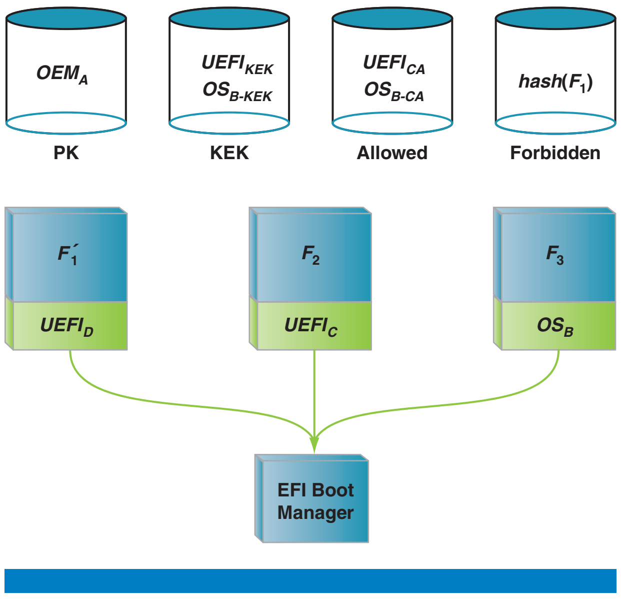

Figure 7: Boot succeeds thanks to new F'_1 image
(Source: Microsoft Corporation, 2011)

A Secure Boot now succeeds since F'_1 is signed by a signer chaining back to an entry in the Allowed database and none of the encountered images have a hash value equal to *hash*(F_1).

Finally, it is determined that no images issued by vendor *C* shall be allowed to be booted. While a rare event in practice, it is included here for illustrative purposes. In this situation, an update to the Forbidden database occurs through a UEFI *SetVariable* () call that *appends* the certificate of *C* to the Forbidden database. The *SetVariable* () call must naturally be signed by an entity whose certificate chains back to an entity in the KEK database, pass all authentication checks, and so on. The current platform state becomes that shown in Figure 8.

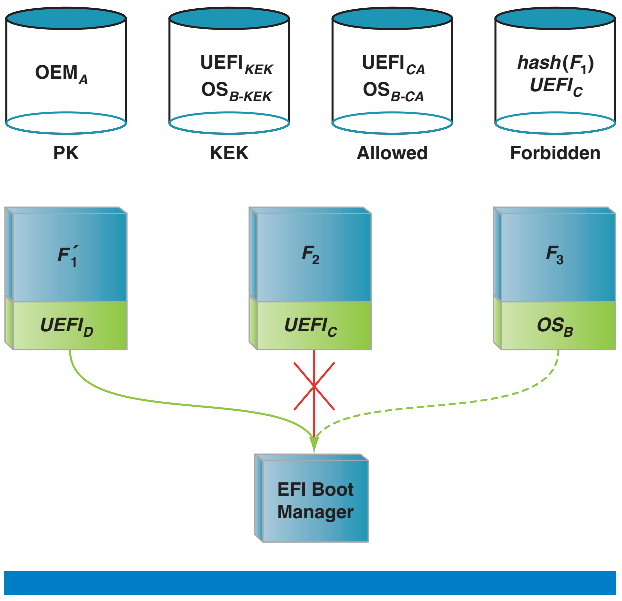

Figure 8: Boot fails due to image F_2 signature chaining to $UEFI_C$
(Source: Microsoft Corporation, 2011)

In this situation, Secure Boot in this boot path again fails because of the recognition by the UEFI Boot Manager that F_2 was signed by an entity now in the Forbidden database.

Networking

"The network should be considered a hostile environment because it is outside the perimeter of the platform protections."

One of the classic areas of concern with respect to security has been networking. The network should be considered a hostile environment because it is outside the perimeter of the platform protections. Some of the classic attacks include Man-in-the-Middle (MitM), wherein an agent can intercept traffic and corrupt traffic in flight. In addition, the end-point of the communication can impersonate the intended party; this is often referred to as the "*lying endpoint problem*" *[19]*.

For purposes of network boot, there are two perspectives. The first is identifying the machine to the network infrastructure. This is for corporate IT to protect its network from a possibly rogue machine. Technology to address this includes authentication protocols, such as 802.1X-controlled ports, and challenge-response capabilities in the Extensible Authentication Protocol (EAP) over LAN (EAPOL). EAP handlers can include EAP-CHAP, EAP-TLS, or EAP-KRB for Challenge-Handshake Application Protocol, Transport Layer Security or Kerberos, respectively. In this case, the UEFI machine needs to prove its identity or proof of some capability, such as a pre-shared key (PSK) for CHAP or Kerberos, or an asymmetric key pair for TLS, to the network infrastructure authentication server.

Technology such as hardware VLAN can be used to segregate this machine from the rest of the main network or quarantine the UEFI machine in case of an authentication failure/remediation.

The other perspective is for the client machine to protect itself from the network. This can include using the above-listed authenticated protocols in order to have the network prove it is authorized. A common use case here is ISCSI logon using CHAP, with the network operator setting the PSK in both the SAN target and the UEFI ISCSI initiator.

"A common use case here is ISCSI logon using CHAP, with the network operator setting the PSK in both the SAN target and the UEFI ISCSI initiator."

But the more common protection mechanism is the UEFI Secure boot. Just as a mutable EFI System Partition is an attack vector for UEFI images from host malware, the network opens up a larger class of malware. As such, the ability to verify the UEFI executable loaded across the network represents the last defense against network-delivered malware. This problem was appreciated in the late 1990s with the BIOS Integrity Services (BIS), which EFI 1.02 inherited in 1999. As mentioned earlier, the problem with BIS deployment is that it had a single certificate, the Boot Object Authentication (BOA). BIS was also a commercial failure in that it never specified a credential deployment strategy, such as that described for UEFI 2.3.1 Secure Boot. It was a single root, it didn't define any BOA protection mechanism/update, it didn't leverage industry-standard tool-flows, and it didn't cover bootstrap from boot paths other than the network.

An additional set of capabilities for authentication, confidentiality, and integrity includes UEFI IPSec support. This provides for secure Internet Protocol communication. This is protects any application traffic across an IP network and is mandatory for IPv6. Features include the Authenticated header (AH), the Encapsulating Security Payload (ESP), and the Internet Key Exchange (IKEv2). In addition, integrity mechanisms such as HMAC-SHA1, and encryption ciphers such as TripleDES-CBC and AES-CBC are incorporated. Finally, the stack can operate in both Transport or Tunnel modes for both IPv4 and IPv6 connections. And the authentication support includes pre-shared key and X.509 certificates.

"An additional set of capabilities for authentication, confidentiality, and integrity includes UEFI IPSec support."

"Beyond the IP6 and IPsec UEFI interfaces, the wire-protocol for network booting has commensurate evolution to the UEFI APIs."

Beyond the IPv6 and IPsec UEFI interfaces, the wire-protocol for network booting has commensurate evolution to the UEFI APIs. Specifically, in the

DHCPv6 extensions for IPv6 network booting, the boot file information is sent as a Uniform Resource Locator (URL); the network boot option details are described in both the UEFI 2.3.1 specification and in IETF RFC 5970. As such, the UEFI client machine and the boot server can negotiate various types of downloads, including TFTP, FTP, HTTP, NFS, or ISCSI. This allows the network capabilities to track the needs of the market and the machine's firmware capabilities. The default bootstrap on IPv6 is referred to as *netboot6*; this is a generalization of PXE 2.1 bootstrap on IPv4.

In order to exercise some of these capabilities, there are open source examples of these codes at [21].

Summary

This article has described how the various integrity preserving technologies in UEFI allow for the "extensibility" of UEFI to be a boon for companies, not a sharp edge that damages them. Technology that can be applied to that end includes support for measured boot and the trusted platform module and UEFI Secure Boot. UEFI was designed as a policy-driven boot loader environment, and it is with Secure Boot that a cryptographically strong set of controls are introduced, which ensures that the party with the appropriate administrative role, whether platform or OS, can manage this policy. And the policy-driven controls can be applied for boot scenarios that entail either locally-attached media or a network-attached server.

References

[1] Unified Extensible Firmware Interface (UEFI), at ***http://www.uefi.org/home***

[2] V. Zimmer et al. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*, Second edition. Intel Press. November, 2010

[3] The UEFI Platform Initialization (PI) Specification, Volumes 1–5, July 2010

[4] Trusted Computing Group (TCG), at ***http://www.trustedcomputinggroup.org/***

[5] A. Menezes et al. *Handbook of Applied Cryptography*, Fifth edition, CRC Press, August 2001

[6] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, April 1992

[7] "Secure Hash Standard", NIST, FIPS Publication 180–3, October 2008

[8] W. Diffie et al. "New Directions in Cryptography," IEEE Transactions on Information Theory, 22(6):644–654, November 1976

[9] R. Housley et al. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*, Wiley, March 2001

[10] "Microsoft PE and COFF Specification", Microsoft, Revision 8.2, at ***http://msdn.microsoft.com/en-us/windows/hardware/gg463119***

[11] "Windows Authenticode Portable Executable Signature Format", Microsoft, August 2008, at ***http://msdn.microsoft.com/en-us/windows/hardware/gg463180***

[12] D. Cooper, et al. "BIOS Protection Guidelines," NIST SP 800–147, April 2011

[13] Preboot Execution Environment (PXE) Specification, Version 2.1

[14] S. Kent et al. "Security Architecture for the Internet Protocol," IETF RFC 4301, December 2005

[15] W. Arbaugh et al. "A Secure and Reliable Bootstrap Architecture," in *Proceedings 1997 IEEE Symposium on Security and Privacy*, pp. 65–71, May 1997

[16] TCG Architecture Overview, Version 1.4, ***http://www.trustedcomputinggroup.org***, August 2007

[17] TCG EFI Protocol Specification, Version 1.20, Revision 1.0, ***http://www.trustedcomputinggroup.org***, June 2006

[18] TCG EFI Platform Specification, Version 1.20, Revision 1.0, ***http://www.trustedcomputinggroup.org***, June 2006

[19] R. Sahita et al. "Mitigating the lying-endpoint problem in virtualized network access frameworks", in *Proceedings of the Distributed Systems: operations and management 18th IFIP/IEEE international conference*, 2007

[20] B. Kaliski "PKCS#7: Cryptographic Message Syntax Version 1.5," IETF RFC 2315, March 1998

[21] UEFI Developer Kit, at ***http://www.tianocore.org***

[22] M. Rothman et al., *Harnessing the UEFI Shell: Moving the Platform Beyond DOS*, Intel Press, January 2010

Authors' Biographies

Magnus Nyström is a Partner SDE in the Windows Security team where he works on architectural matters related to the next Windows release. Most recently, he was a Distinguished Engineer at RSA, the Security Division of EMC, where he worked on various aspects of RSA's technical strategy. Magnus has extensive experience from international standardization work, both from intergovernmental standards bodies as well as industry consortiums. In 2005, he was selected to serve in the European Union's Network and Security Agency ENISA's permanent stakeholder's group together with 29 other experts from across the world. The tenure was 2.5 years and Magnus was reappointed for a second term in 2007. Magnus has about twenty patents or pending patents—all of them in the computer security space.

Martin O. Nicholes is a Firmware Architect at Insyde Software and has over 25 years of experience in software development, including firmware and operating system driver development. He worked 23 years for Hewlett-Packard in the areas of server firmware development, server firmware architecture, and server security. He holds 10 patents in the area of firmware architecture. He is CISSP certified. He received his B.S. in Engineering from San Jose State University and a M.S. and Ph.D. in Electrical and Computer Engineering from the University of California, Davis

Vincent J. Zimmer is a Principal Engineer in the Software and Services Group at Intel Corporation and has over 19 years experience in embedded software development and design, including BIOS, firmware, and RAID development. Vincent received an Intel Achievement Award and holds over 200 US patents. He has a Bachelor of Science in Electrical Engineering degree from Cornell University, Ithaca, New York, and a Master of Science in Computer Science degree from the University of Washington, Seattle. He can be contacted at http://www.twitter.com/VincentZimmer and vincent.zimmer@gmail.com