A Tour Beyond BIOS into UEFI Secure Boot

Lee Rosenbaum
Vincent Zimmer

Intel

July 2012

Executive Summary

This document provides an overview of the implementation and intent behind the UEFI Secure Boot [UEFI MAIN SPECIFICATION] capability. This includes implementation practices of UEFI PI-based firmware [UEFI BOOK][UEFI SHELL BOOK][UEFI OVERVIEW], such as those provided by source infrastructure in the UEFI Developer Kit (UDK2010) [TIANO CORE], for this feature.

The goal of the paper is to provide

- an understanding of the motivations behind this capability
- a walk-through of the implementation
- a future evolution.

This paper targets firmware, software, and BIOS engineers.

Contents

Figures

Tables

1 Integrity in an extensible pre-OS

This section describes the technical motivation for protections in an extensible pre-OS based upon UEFI [UEFI BOOK][UEFI SHELL BOOK][UEFI OVERVIEW].

The UEFI specification [UEFI MAIN SPECIFICATION] set includes the initial boot flows, as codified in the UEFI Platform Initialization (PI) specifications [UEFI PI SPECIFICATION], and the main UEFI specification. The former specification set describes how to build the system board and should be created and updated under the authority of the platform Manufacturer (PM). The PM may be referred to as a Multi-National Corporation (MNC) or an Original Equipment Manufacturer (OEM) in various contexts. PI is not intended to be third party extensible once the machine has shipped. The UEFI specification, on the other hand, describes the interoperability surface for third party independent hardware vendors (IHV's) that build drivers for boot devices, independent software vendors (ISVs) who build pre-OS applications, and operating system vendors (OSV's) who create operating system loaders. UEFI Secure Boot allows for having a policy-based invocation of these various UEFI executable images, using cryptographic signatures to identify the software publishers.

The relationship of the PI-based PM code and the ISV, IHV, and OSV interoperability will be treated in the context of a more formal integrity model. Then one aspect of the integrity model, namely the UEFI Secure Boot based launch of the third party UEFI executables, will be described in successive layers of detail. This detail will include but is not limited to a review of an open source verifier implementation. There are additional integrity preservation mechanisms, such as preservation and protection of the PI code via signed updates, which are mentioned in the integrity model but not treated in detail via an implementation.

1.1 UEFI Overview

"UEFI" is an industry group that is standardizing what were the EFI1.10 specification and the Framework PEI and DXE specifications. Within UEFI, the main UEFI specification (beginning with UEFI 2.0) is handled by the UEFI Specification Working Group (USWG) and the Platform Initialization (PI) Architecture PEI/DXE content are handled in the Platform Initialization Working Group (PIWG). The USWG is focused on the OS-to-platform interfaces, whereas the PIWG is focused upon platform initialization. The USWG parties of interest are the OS vendors, pre-OS application writers, and independent hardware vendors who write boot ROM's for block or output console devices. The PIWG parties of focus include the platform builders (MNC's/PMs), chipset vendors, and CPU vendors. The PIWG work can accommodate both UEFI operating systems and today's conventional/Int19h-based operating systems.

PIWG-based components provide one means of producing the UEFI interfaces. What is common across both UEFI and PI Architectures is the extensibility of code and

interfaces. For UEFI and PI DXE, the extensibility point is a loadable driver model and for PI PEI, extensibility is through firmware files with PEI Modules.

Figure 1 below shows how the PI elements are used for construction and can include a capability to product the interfaces required by the UEFI Specification.

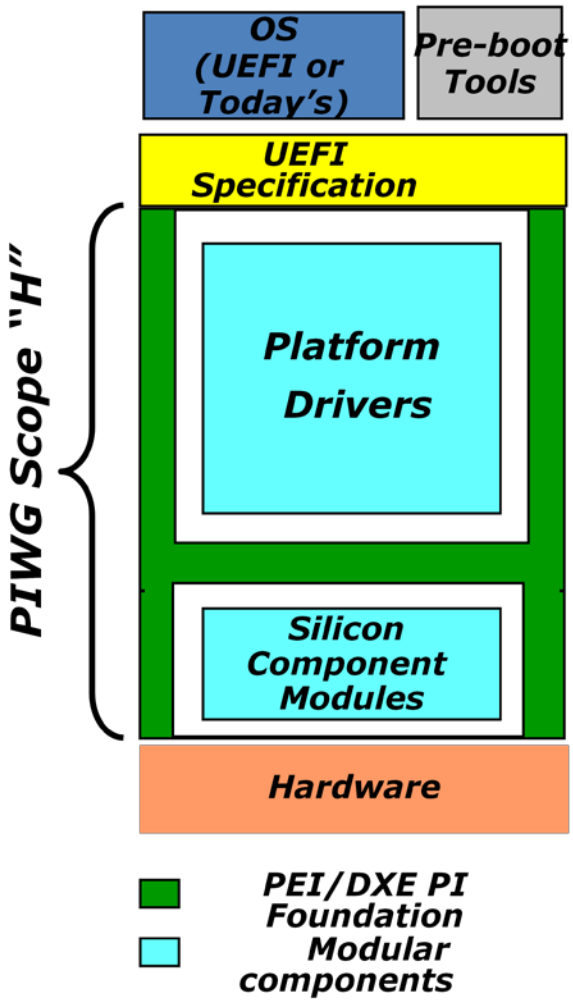


Figure 1 - PI versus UEFI

The UEFI main specification describes interfaces, whereas the UEFI Platform Initialization Working Group (PIWG) and the Platform Initialization (PI) components describe the building blocks that may produce a set of UEFI conformant interfaces.. Beyond UEFI interfaces, though, the PI components are primary responsible for initializing the systems and handing off control.

1.2 Integrity Model

To motivate the usage of UEFI secure boot, a quick review of the platform boot process is provided. The figure below describes the boot process. The important point to note is

that the initial code that runs after the restart or “Startup” event should be under manufacturer control, or the platform manufacturer (PM). This is distinct from later UEFI application and operating systems that can be under the platform owner (PO) control. This distinction between the PM and PO allows for describing the sphere of integrity control in terms of a more formal integrity model like Clark Wilson (CW) [CLARK].

The system diagram below indicates that the pre-OS can be divided into two parts, a PM controlled platform part for which extensibility is not necessarily a consideration (with the exception of upgrades and fixes) and a non-PM controlled extensible portion of the pre-OS boot where extensibility is inherent in the design via the loadable UEFI drivers and applications.

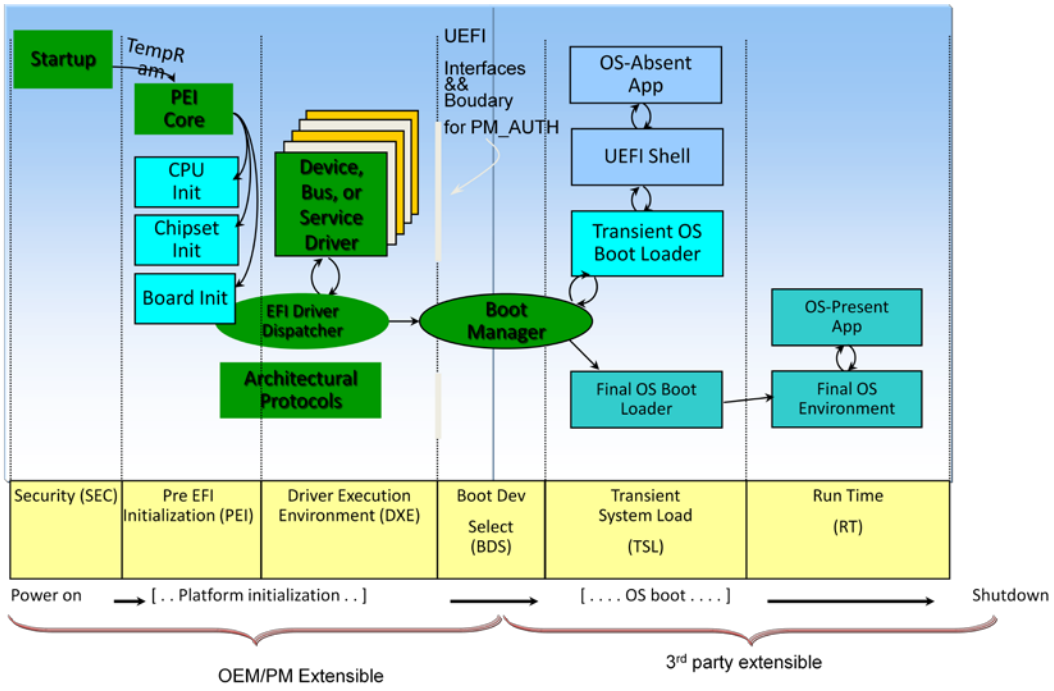


Figure 2 - Boot Flow

1.2.1 Compartments

In order to perform a more formal security analysis, we use the concept of an integrity compartment. Formally, a compartment is an integrity protected isolation boundary holding code and data that obeys the Clark-Wilson integrity rules. An informal definition of the Clark-Wilson rules for compartments follows below (1.2.1.1 through 1.2.1.4):

1.2.1.1 Guards:

1. The entry of all code and data into the compartment is **guarded.** This means that the installation of all code and data has to be verified by a guard (such as a code signing guard). This includes initial install and upgrade of code/data. Signed capsule updates provide such a mechanism and

UEFI Authenticated Variables to store signing keys used for integrity verification of code or data capsules being installed in a compartment.

The UEFI Secure Boot verifier is an instance of a code signing guard. This paper treats more detail on this capability.

2. Transitions of code/data into the compartment must be validated by a guard in the compartment. In general, this means things like bounds checking/type safety as well as code development practices like the SDL [SDL]

3. If the compartment has UI, then the UI transition into and out of the compartment must be guarded by a Trusted Path – that is the user (usually platform administrator) has to have assurance that he is using the UI for that compartment and not some other compartment. In the case of pre-OS, the PM compartment should not have any UI and platform administrative tasks must be absolutely minimized

The platform administrator, which may be the PO, is the party that can update the PK or KEK remotely, or the physically present local user manipulating the custom mode setup screens. Custom mode is distinct from Setup and User Mode, which are codified in the UEFI Specification. Details on Custom mode requirements can be found in [WIN LOGO].

1.2.1.2 Verifiable Integrity of Code and Data in the Compartment:

This means that the integrity of all code and data in the compartment must be verifiable at any time in the lifetime of the compartment. In general, this implies that the compartment needs memory (execution) isolation as well as storage isolated from other compartments. Having storage isolation implies that when a thread of execution within the compartment writes data, that the data is automatically protected by the compartment.

Authenticated variable drivers with isolated execution, such as System Management Mode (SMM), provide one means by which to verify the code integrity.

1.2.1.3 Admin:

The admin model for the compartment has to be fully defined in the sense that the tasks that the compartment admin can do have to be specified. Furthermore, the admin needs to be protected from other lower integrity compartments when he is performing admin actions. This means that the designer should absolutely minimize admin actions that can compromise the integrity of the compartment. For example, if the admin sets the security policy for a guard that authorizes code or data into a compartment insecurely, he should not expect that the system could be subsequently booted in a secure state even if the policy was reset to be secure. In the case of Authenticated Variables, the principal *platform admin* admin action was that of zeroing the platform key in order to put the system into *setup* mode – which should be

done by asserting remote or local physical presence. After that, the platform admin could reset an Authenticated Variable. Since remote reset of an Authenticated Variable does not need UI, remote platform admin operations on Authenticated Variables are more securable than local platform administrative actions.

1.2.1.4 Audit:

All integrity affecting operations including administrative operations in the compartment have to be audited. TPM Quote and Extend functions form the basis of audit in pre-OS [TRUSTED]. The availability of the audit log is also a critical issue and requires storage isolation. A TPM as a root-of-trust for reporting (RTR) is one embodiment.

UEFI has the deferred image execution table and other event recording mechanisms, too.

1.2.2 Security Requirements

4. PM-CPT-PROTECT:

   PM provided components of the platform must be a compartment separated from the non-PM provided pre-OS part of the platform. The PM compartment is administered by a PM administrator role and this role does not need interactive logon to the system.

   This requirement is typically met by signed BIOS update, including but not limited to what is requires by the “BIOS Protection Guidelines” in NIST 800-147 [NIST].

   Other defenses can include stack checking, address space randomization (ASLR), and a non-executable stack and data region [HARDENING].

   Figure 2 above describes the OEM or PM Compartment. This includes SEC, PEI and DXE. This compartment is not intended to be third party extensible.

5. NON-PM-PREOS-PROTECT:

   Non-PM provided (extensible) pre-OS part of the platform should be a compartment separate from the OS. This compartment is under the control of a platform admin role, which may need to be entered by an interactive logon. It is recommended that physical presence be at least one way of authenticating the platform administrator.

   What the above two compartments mean is that the SEC, PEI, and DXE elements of the platform and their storage in the firmware volume are separate from the third party extensible elements, such as UEFI drivers and applications. The **EFI_END_OF_DXE_EVENT_GUID** [UEFI PI SPECIFICATION] provides a line of demarcation between the PM-CPT and the NON-PM-PREOS.

   The Transient System Load (TSL) or the extensible UEFI pre-OS in Figure 2 describes this compartment. This compartment does not include the operating

system runtime. Each operating system (OS) or virtual machine manager may include its own integrity protection and admission scheme. Such OS protection art lies beyond the UEFI Secure Boot art. UEFI Secure Boot interacts with a successive OS integrity model by allowing for verification of the OS loader UEFI application.

6. NON-PM-PREOS-SECURE-POLICY:

   There exists at least one set of security policy for configuring the non-PM pre-OS compartment such that in such *secure configurations* the non-PM pre-OS compartment does not damage the integrity of an OS booting on the system.

   The UEFI Secure Boot **KEK**, **db**, and **dbx** provides for such policy objects.

   a. POLICY-MDL: The policy has minimal description length while capturing all *secure configurations* of the compartment. This is an administration requirement – for example, if the policy uses code or data hashes to specify secure configurations, a policy that uses digital signing has shorter description length and does not change as often as the former policy. This means that the platform admin does not need to administer the system often, reducing the TCO.

   b. POLICY-REVOKE: The policy must be able to revoke compartment configurations with security bugs and transition to a secure configuration based on platform administrator input. In practice, this means that if there was a code/data bug in the compartment, the policy should allow for patching the bug without intervention by the platform administrator to change the policy. For example, if patching policy was signature based then new patches could be installed without administrative intervention. On the other hand, if the policy was based on hashes, the patch could not be installed without a policy change – adding the new hash(es) to the list of allowed hashes.

   c. POLICY-AUDIT: The security policies (including policy for authenticating the admin) guarding the PM compartment as well as the non-PM pre-OS compartment must be auditable. Note that the succinctness requirement POLICY-MDL means that the audit log will change rarely and only in case of a change in the (succinct) policy represented by the POLICY-MDL requirement.

      d. TPM measurement of these variables is one means of audit.

1.3 Summary

This section provided details on a formal integrity model. This analytic framework helps to motivate the design intent behind UEFI Secure Boot. It also details those complementary technologies that need to be deployed along with UEFI Secure Boot in order to have a robust platform deployment experience.

2 History of UEFI Specifications and Implementations

This section describes the history of the UEFI specifications and the associated implementations. The important point to note is that the UEFI2.0 Specification was published in 2006, but it was largely based upon the EFI1.10 specification from 2001. The EFI specification, in turn, grew out of the work with Itanium in 1998. Therefore, the genesis of UEFI includes elements that are now fourteen years old. Figure 3 below describes the timeline of the UEFI specifications and implementations, respectively.

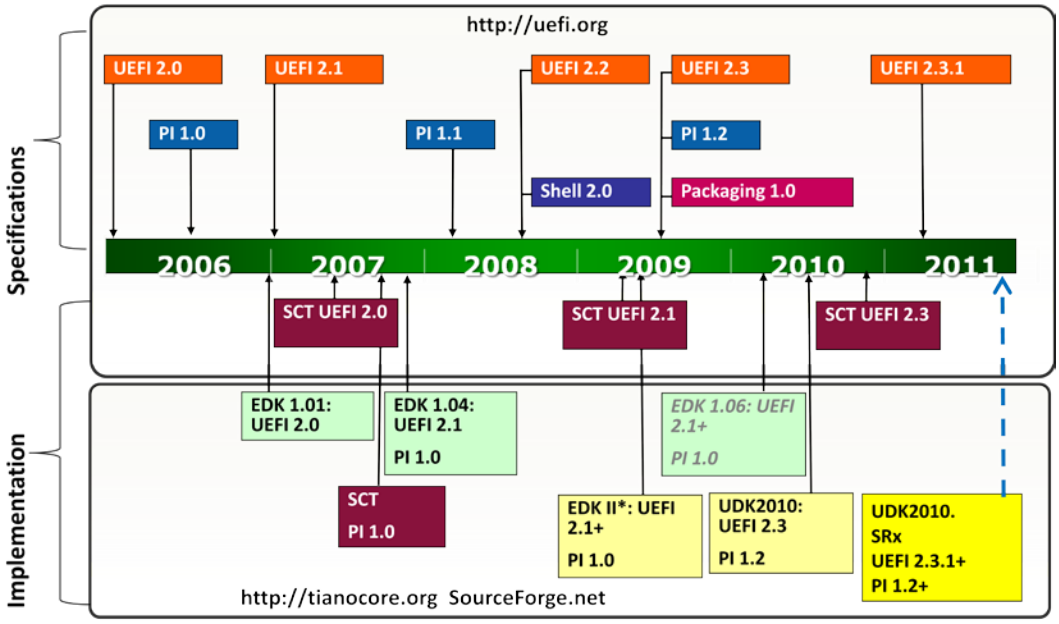


Figure 3 - UEFI Timeline

During that decade, the platform has evolved to include more CPU micro architectures, bus types, and networking technology. Along with the march of technology, the firmware implementations have evolved. The initial implementation of EFI was called the “EFI Sample” and was posted on Intel’s website. It layered on top of an Itanium SAL or a PC/AT BIOS. At the time, there was no definition of an underlying EFI-style initialization code.

Around 2001, Intel defined the Intel Platform Innovation Framework for the Extensible Firmware Interface, or “Framework” specifications. These 30 odd specifications were posted to the Intel website and described the platform boot flow, including SEC, PEI, DXE, BDS, CSM, etc. The Framework specification had an associated implementation, along w/ an implementation of EFI, under the guise of the project code name “Tiano.” Tiano, in turn, evolved through several iterations as part of the EFI Developer Kit (EDK). The EDK, or as it is now known, “EDKI”, evolved into various instances,

including EDK1.01 and EDK1.04. The EDK, like the EFI sample, was a monolithic code base.

With the advent of EDK2, several software engineering techniques were applied to the code base construction. These techniques include packages, which are ensembles or source and/or binaries. The EDK2 also includes more purpose-built libraries, as found in the Module Development Environment (MDE) package, along with industry standard .h files. In addition, EDK2 includes the MDE Module Package, which has implementations of the generic modules that map to the respective UEFI and PI industry standards. Other packages can be composed, with the Security Package being a notable addition. The Security Package contains the implementation of UEFI Secure Boot and works in concert with the Crypto Package, with the latter providing the security primitives. Many of the ensuing details in chapter 3 treat the Security Package contents.

The UEFI Development Kit (UDK2010) is a specific instance of the EDK2 open source project and other components that realizes the UEFI2.3.1C specification, including UEFI Secure Boot and other contemporary UEFI-aware operating system requirements.

2.1 Summary

This section has provided a brief history of the UEFI implementations and the specification development. The advent of EDK2 and the UDK2010 implementation are also introduced because of their role in UEFI Secure Boot enabling.

3 Secure Boot Code Implementation

This section describes the code flow as the UDK2010.SR1.UP1 implementation of UEFI Secure Boot [TIANO CORE] validates a UEFI image.

This supplements and extends the information in Signing UEFI Images [SIGNING], which focused on how to sign images that will be authorized for execution under UEFI Secure Boot by the code described below.

The section's primary audience is developers who wish to understand the UDK2010.SR1.UP1 implementation of UEFI Secure Boot. To that end, it lists the packages and modules involved and describe their functions that are executed as an image is authorized for execution.

The description begins when a UEFI image is invoked to be executed by the UEFI Boot Service LoadImage, for example when a UEFI application such as *HelloWorld.efi* is invoked using the UEFI Shell, and ends when functions within OpenSSL are invoked. OpenSSL [OPENSSL] internals are not discussed.

3.1 Background

UEFI Secure Boot defines how a platform's firmware can authenticate a digitally signed UEFI image, such as an operating system loader or a UEFI driver stored in an option ROM. This provides the capability to ensure that those UEFI images are only loaded in an owner-authorized fashion and provides a common means to ensure platforms security and integrity over systems running UEFI-based firmware.

At a high level, the platform manufacturer or the platform owner are the administrative role on the left of the figure below. They will enroll the policy objects, which include the n-tuple of keys {**PK**, **KEK**, **db**, **dbx**} as a step 1. During each successive boot, the UEFI Secure boot implementation will assess the policy in order to verify the signed images that are discovered in a host-bus adapter or on a disk. If the images pass policy, then they are invoked.

From this schematic overview in Figure 4, more details on the policy objects, the algorithm of the verifier, and the software architecture will be discussed.

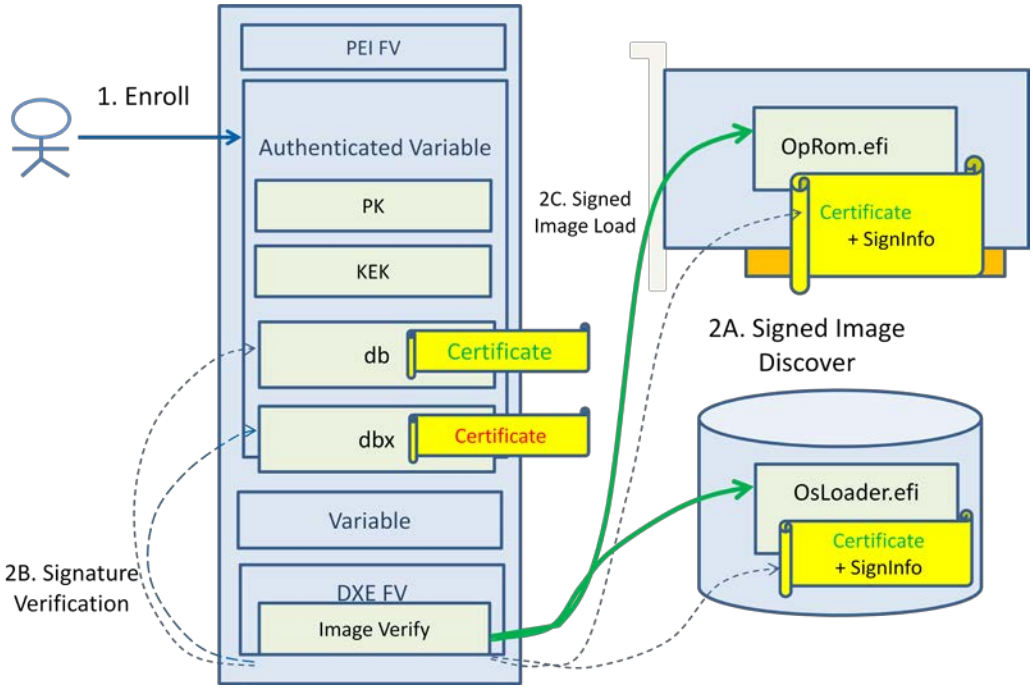


Figure 4 - Secure Boot Overview

The enrolled images can be signed with Authenticode [AUTHENTICODE] or simply have their hash registered.

3.1.1 Cryptography Primer

Cryptographic techniques are integral to the UEFI Secure boot processing. [UEFI-ITJ] p85 – 88 provides some background.

At a high level, there is the process of creating a digital signature, as shown below. This involves the processing of a message via a hashing algorithm. This flow is shown in the figure below.

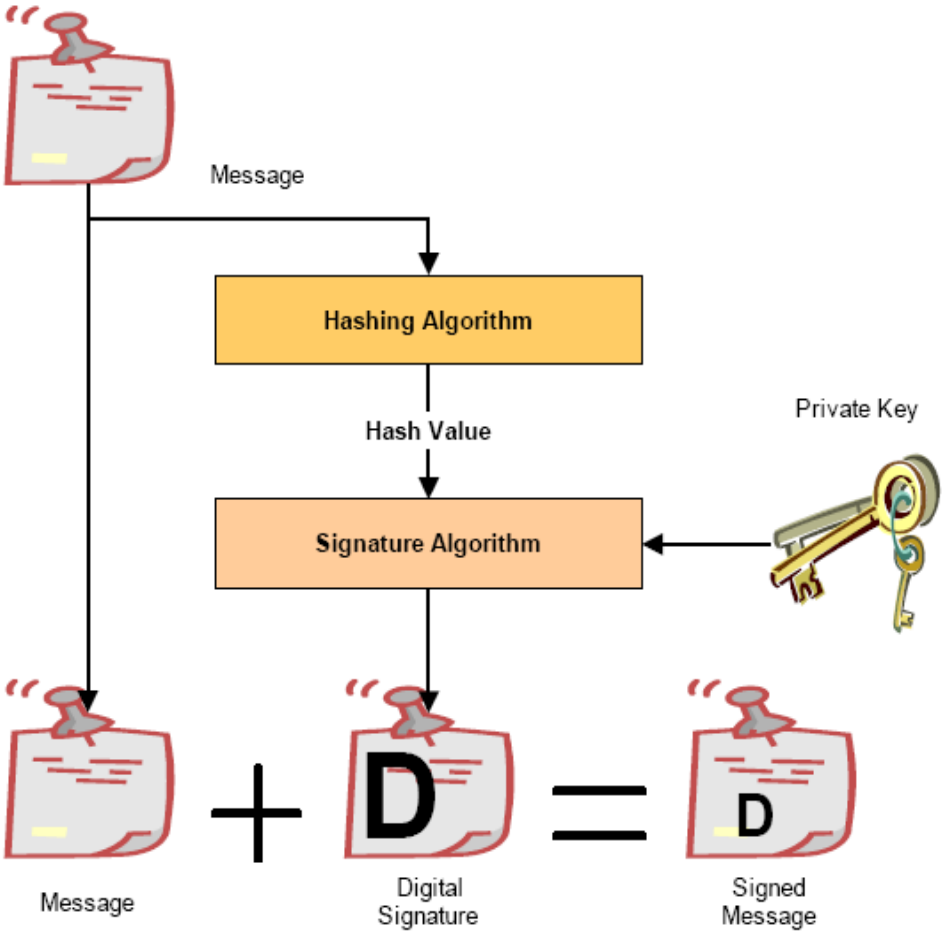


Figure 5 - Creating Digital Signatures

The verification process is the inverse of the signing process. The figure below describes this flow.

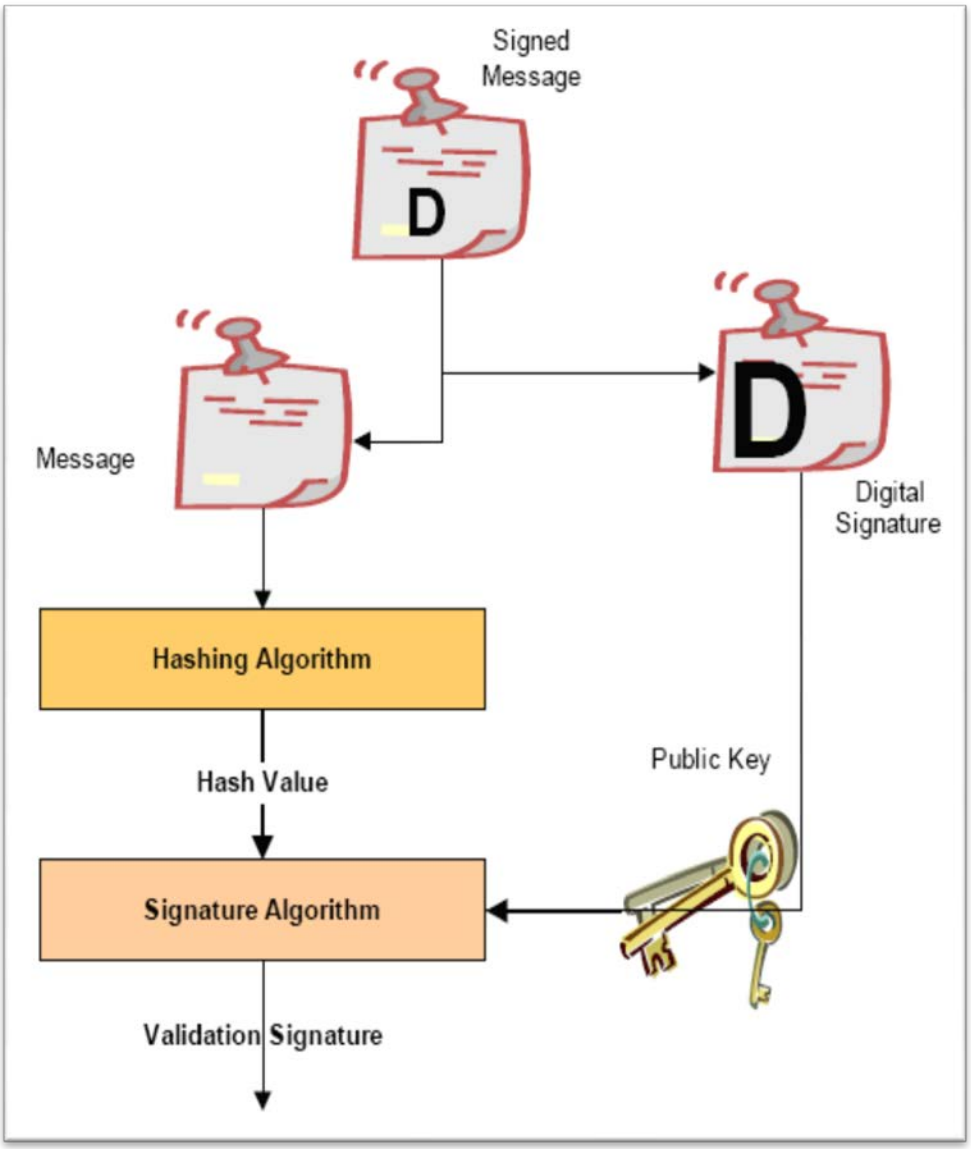


Figure 6 - Verifying Digital Signatures

Some hashing algorithms of importance include the Secure Hash Algorithm (SHA) family [SHA], such as SHA-2. Notable signature algorithms include RSA [RSA].

3.1.2 UEFI Secure Boot Policy

UEFI Secure Boot was designed with the following goals:

- Allow the platform owner to check the integrity and security of a given UEFI image ensuring that the image is only loaded in an approved manner.
- Allow the platform owner to manage the platform's security policy as defined by the UEFI Secure Boot authenticated variables described below.

UEFI Secure Boot is controlled by a set of UEFI Authenticated Variables, which specify the UEFI Secure Boot Policy. This policy includes which images and certificates are contained in the allowed list (a.k.a the authorized signature database) and the prohibited list (a.k.a the forbidden signature database). UEFI Authenticated Variables are defined in the UEFI 2.3.1C Specification and summarized in the figure below.

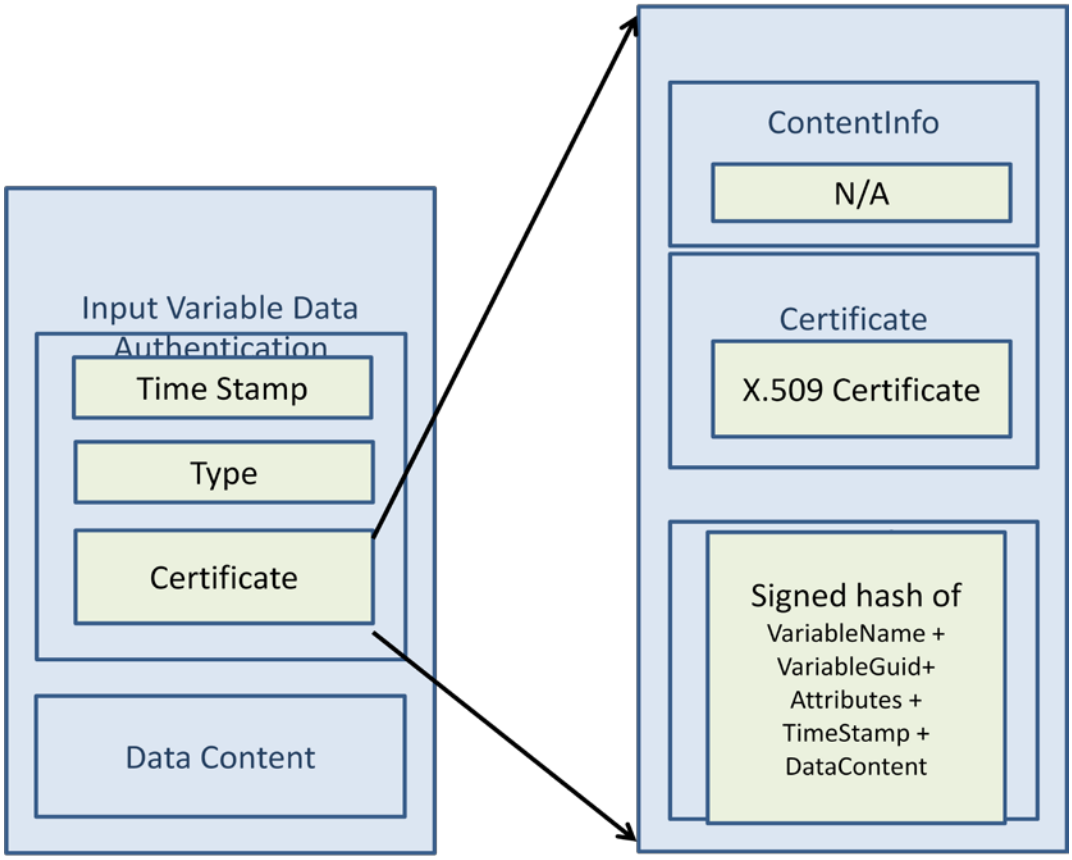


Figure 7 - Authenticated Variable Details

The authenticated variables use well-known X509 certificates to store the signature information, as shown in the figure below.

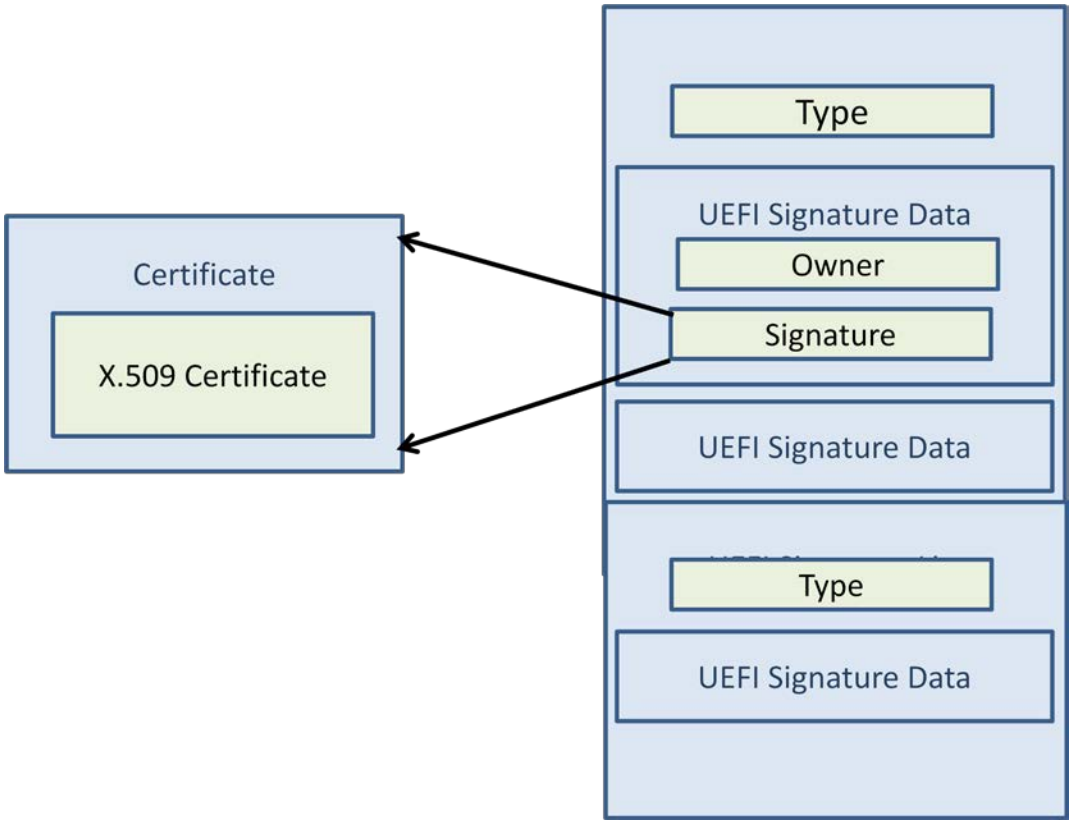


Figure 8 - X.509 Certificate Details

3.1.3 UEFI Image Formats

UEFI images use the PE/COFF format and are signed as defined in the Microsoft * Authenticode Specification [AUTHENTICODE]. The important detail is that the Authenticode signing mechanism is an embedded signature, versus a detached signature in other schemes like EFI BIS Protocol [BIS]. Details of an Authenticode signed image are shown below.

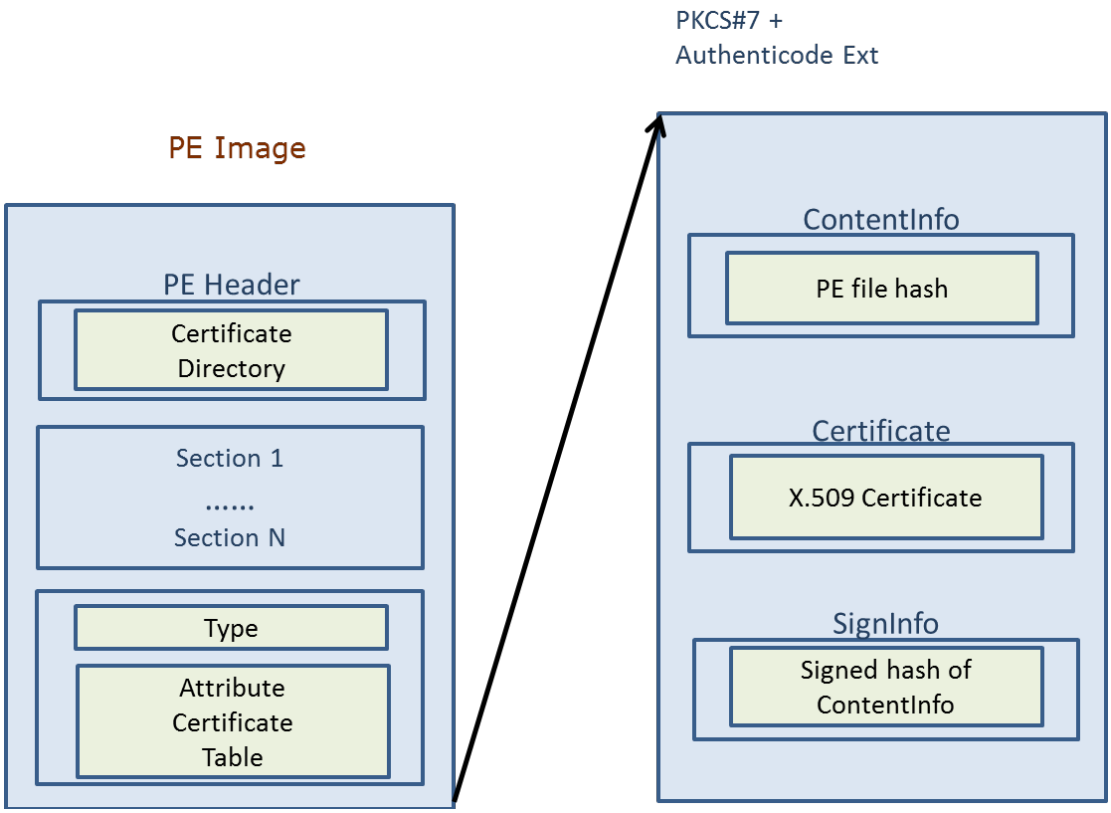


Figure 9 - ***PE/COFF and Authenticode Image Formats***

3.1.4 Overview of Secure Boot's Authenticated Variables

The UEFI Authenticated Variable Service is defined as an enhancement to the UEFI Variable Service. It provides a means to ensure the integrity of specified variables by prepending authentication data, shown as **EFI_VARIABLE_AUTHENTICATION** in the figure below. This authentication data is defined in chapter 3 of the UEFI 2.3.1C Specification.

If a variable's **EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS** bit is set when the SetVariable function is called, both the identity of the operator and integrity of data will be authenticated before the variable is written into storage.

Confidentiality was not a design goal, so there is no verification of data that is subsequently read using the **GetVariable** function. Therefore, anyone can read authenticated variables, but only parties possessing the correct key-pair can write or update authenticated variables.

The layout of the authenticated variable data is shown below:

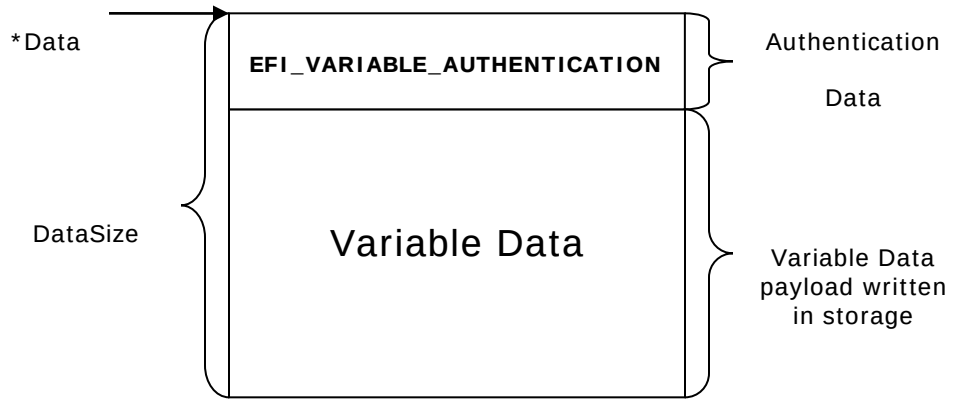


Figure 10 - Authenticated Variable Layout

3.1.5 Types of Authenticated Variables

Section 7.2 of the UEFI 2.3.1C Specification defines two types of Authenticated Variables:

- Counter Based
- Time Based.

In order to prevent replay attacks, Counter Based variables include a monotonic count within the **EFI_VARIABLE_AUTHENTICATION** section, while Time Based variables in **EFI_VARIABLE_AUTHENTICATION_2** include a timestamp for that purpose.

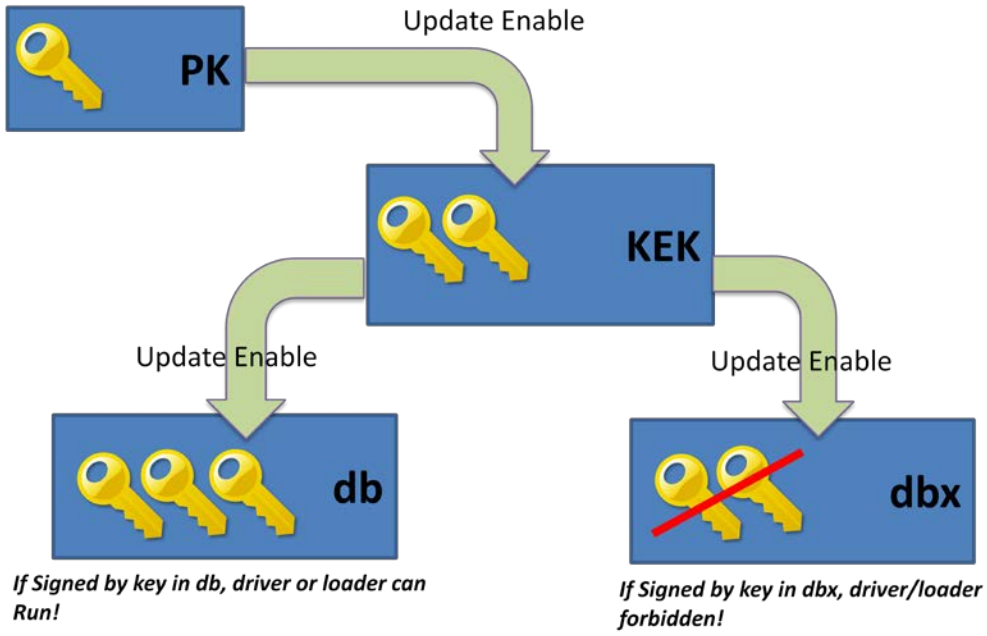
Each method requires different scenarios to deploy the corresponding secure boot policy. However, as the UEFI Specification states that the time-based method is preferred, counter based deployment scenarios are not described in this document.

3.1.6 Secure Boot's Authenticated Variable Descriptions

UEFI Secure Boot's image and certificate policies are controlled by the following UEFI authenticated variables, which are defined in section 3.2 of UEFI Specification 2.3.1C:

- **Platform Key (PK)** - The platform key establishes a trust relationship between the platform owner and the platform firmware. The platform owner enrolls the public half of the key (PKpub) into the platform firmware. The platform owner can later use the private half of the key (PKpriv) to change platform ownership or to enroll a Key Exchange Key. For UEFI 2.3.1C, the recommended Platform Key format is RSA-2048.
- **Key Exchange Key (KEK)** - The Key exchange keys establish a trust relationship between the operating system and the platform firmware. The public part of the key (KEKpub) is enrolled into the platform firmware. Each operating system (and potentially, each third party application which need to communicate with platform firmware) can later use the private half of the key (KEKpriv) to communication with firmware in trusted manner. For UEFI 2.3.1C, the recommended Key Exchange Key format is RSA-2048.
- **Authorized Signature Database (DB)** - This database contains authorized signing certificates and digital signatures. An image signed with a certificate enrolled in DB or whose digital signature is enrolled in DB is authorized to execute. DB is the allowed list
- Forbidden Signature Database (DBX) - This database contains forbidden certificates and digital signatures. An image signed with a certificate enrolled in **DBX** or whose digital signature is enrolled in **DBX**, is never allowed to run. **DBX** is the prohibited list.
- **Setup Mode** - When Setup Mode is null, no Platform Key is enrolled and the platform is said to be operating in setup mode. While in setup mode, the platform firmware does not authenticate images and secure boot policy can be configured by writing the PK, KEK, DB and DBX variables. When Setup Mode is not null, a Platform Key is enrolled and the platform is operating in User Mode. User Mode requires that all executables be authenticated before they are permitted to run.
- **SecureBoot** – When set (1), the platform is operating in secure boot mode and performs image verification based on the data stored in **KEK**, **DB** and **DBX**.

The authenticated variables are the policy for UEFI Secure boot. The keys have a specific hierarchy as shown below.



Attribution of [FACTORY]

Figure 11 - Secure Boot Authenticated Variables

3.2 Image Authorization High Level Flow

The process by which an un-trusted UEFI image might be authorized to run is described below. This process includes both signed and unsigned images. This section provides descriptions of the UDK2010.SR1.UP1 code involved in the verification of an untrusted UEFI image starting with a description of the high-level verification algorithm and then diving into the relevant packages, modules and functions. The descriptions are at a high level with the goal of helping to orient a new developer to UEFI image authorization in UDK2010.SR1.UP1. For complete details on the implementation, please the referenced code and for details on the requirements, please see chapter 27.7 of the UEFI 2.3.1C Specification.

During initialization of an UEFI executable based on the secure boot policy, the Boot Manager decides whether the UEFI executable should be initialized and run. Verification success means the executable passed authentication, which means that either its signing certificate or signature was found in the authorized signature database and was not found in the forbidden signature database. Otherwise, verification fails and the image is not executed, or “deny image.”

The verification steps for both unsigned and signed images are described below using the following definitions:

- **Signature** – is the SHA256 hash of the PE/COFF image.
- **Certificate** – is the image’s Authenticode certificate containing the public key that corresponds to the private key used to sign the image.

- **Certificate Authorized –** Authorizing a signing certificate allows any image containing that certificate to be run without the need to authorized individual signatures.

    - In UDK2010.SR1.UP1 and earlier implementations, an image's certificate is authorized *if it matches or chains to a* **KEK** *or* **DB** *entry*, (or both).

    - However, the UEFI 231C specification now states on page 227 that a certificate in **DB** can match or chain to certificates in **KEK**, or **PK**.

      Since certificates in **KEK** must chain to certificates in **PK**, this clarification implies that a certificate in DB is authorized if it chains to a certificate in PK.

      That is, during SetVariable, **KEK** is used to authorize **DB** (and **DBX**) variable entries and PK is used to authorize **KEK** entries. However, **KEK** is not used to authorize an image to be run during LoadImage.

      Implementations complying with this updated interpretation authorize an image's certificate *if it matches or chains to only a* **DB** *entry*.

- **Signature Authorized** –An image signature's is authorized by enrolling it in **DB**.

The image authorization process is summarized below and described in more detail in the text that follows the figure below.

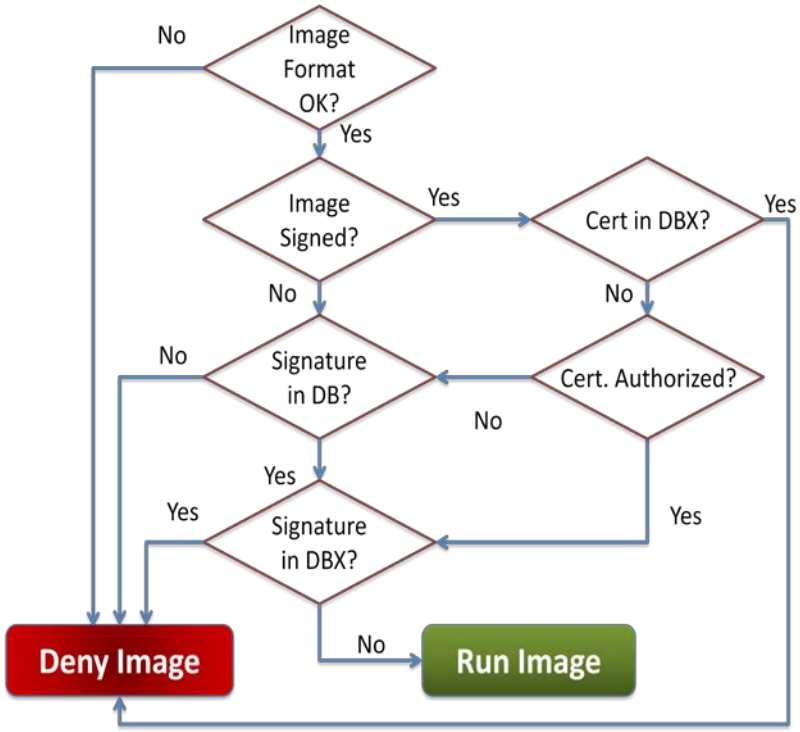


Figure 12 - Image Authorization Flow

1. First the image's format and structure is verified to ensure it is a valid PE/COFF image (if it is unsigned) or Authenticode image (if it is signed).

2. If the image is unsigned, then its signature must be found in the authorized database (**DB**) and not be found in the forbidden database (**DBX**). If so, run the image, otherwise the image is not executed.
3. If the image is signed, the process is more complex.
   a. First, check if its certificate has been authorized, and is not in the forbidden database (**DBX**).
   b. If the image's certificate is authorized, then unless the image's signature is in the forbidden database (**DBX**), run the image. Otherwise, the image is not executed.
   c. If the image's certificate has not been authorized, then it can still be run if its signature has been authorized. If its signature is in the authorized database (**DB**) and is not in the forbidden database (**DBX**), run the image, otherwise deny running the image.

3.2.1 Example Scenarios:

- If a signing certificate is enrolled in **DB**, then all images signed by that certificate are run, unless that certificate or individual signatures are found in **DBX**.
- If an image's certificate is not found in **DB** but its signature is found, then it is run, unless its certificate or signature is found in **DBX**.
- If an image's certificate or signature is found in **DBX**, then it is not run, even if they are in **DB**. This would occur when a certificate is revoked or an image is on the prohibited list.
- If an image is properly signed, but neither its certificate nor signature is found in **DB**, then it is not run.
- Of course, unsigned images do not contain a certificate so to be authorized; their signatures must be found in DB and are not found in **DBX**.

The following figures show screen shots from the UDK2010.SR1.UP1 release when the HelloWorld.efi application is authorized to run and not authorized to run.

Note: *The output seen on other implementations may vary.*

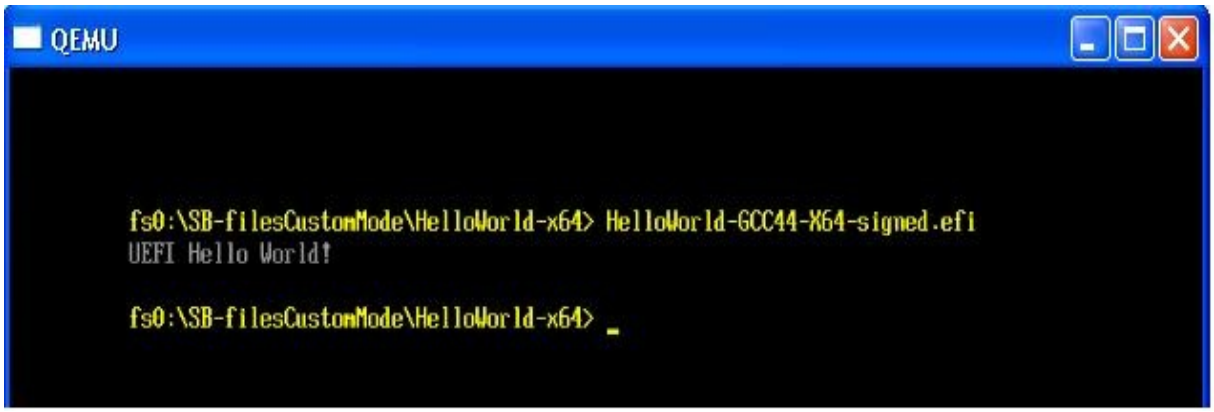


Figure 13 - Image Authorization Success – Run Image

Figure 14 - Image Authentication Failed – Deny Image

3.3 Packages and Modules Overview

This section lists the packages, modules and libraries that are involved in image verification. The diagram below shows the various modules.

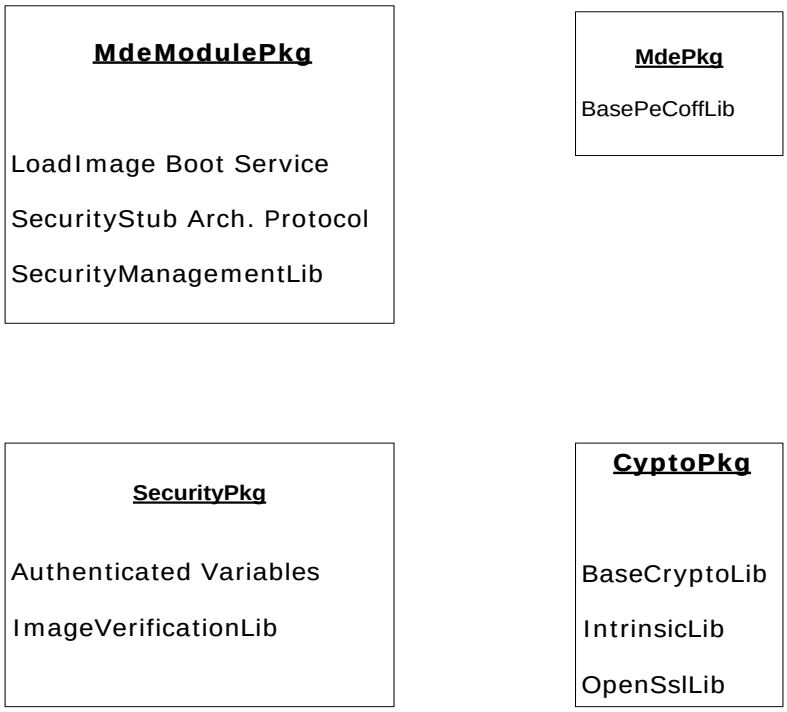


Figure 15 – Packages and Modules

3.3.1 MdeModulePkg

3.3.1.1 LoadImage UEFI Boot Service

The LoadImage UEFI Boot Service is defined in chapter 6 of the UEFI 2.3.1C Specification.

3.3.1.2 SecurityStub - SAP

The SecurityStub implements the Security Architectural Protocol defined in section 12.9.1 of the PI Specification revision 2.1.

Note that this revision added the definition of the definition of Security Architectural Protocol 2 (SAP2), and the UDK will soon be updated to utilize these additional capabilities. However, SAP2 is not discussed further in this document.

The SAP is an instance of an architectural protocols, or AP. The intent of the AP's is to decouple platform policy and implementation details from the PI DXE main executable. Since UEFI Secure Boot is an optional feature with respect to the UEFI Specification, per chapter 2.6, the present of the Secure Boot capability is abstracted from the DXE main via the SAP.

3.3.1.3 DxeSecurityManagementLib

DxeSecurityManagementLib provides routines to register and execute registered security handlers.

3.3.2 MdePkg

3.3.2.1 BasePeCoffLib

BasePeCoffLib provides common services for parsing Pe/COFF images.

3.3.3 SecurityPkg

3.3.3.1 DxeImageVerificationLib

DxeImageVerificationLib, which is implemented in SecurityPkg\Library\DxeImageVerificationLib\DxeImageVerificationLib.c, provides the routines used to verify a PE/COFF or Authenticode image prior to its execution. Authenticode images use SHA256 hashes and X.509 certificates.

3.3.3.2 Authenticated Variables

The Authenticated Variables modules provides the authenticated variable services defined in section 7.2 of the UEFI 2.3.1C Specification. This includes Time-based Authenticated Variables that provide a timestamp for roll back protection, support SHA256 hashes and signature algorithms using X.509 certificate chains,

3.3.4 CryptoPkg

3.3.4.1 BaseCryptoLib

BaseCryptoLib provides an interface between SecurityPkg and OpenSSL allowing SecurityPkg to be agnostic regarding low level security primitives such as ASN.1 decoding and PKCS#7 [PKCS7] parsing.

3.3.4.2 IntrinsicLib

IntrinsicLib provides an interface between OpenSSL and UDK2010 implementations for `memset()` and `memcpy()`.

3.3.4.3 OpensslLib

UDK2010.SR1.UP1 uses OpenSSL revision 0.9.8w.

3.4 Modules and Functions Overview

This diagram shows the modules and functions involved in image verification.

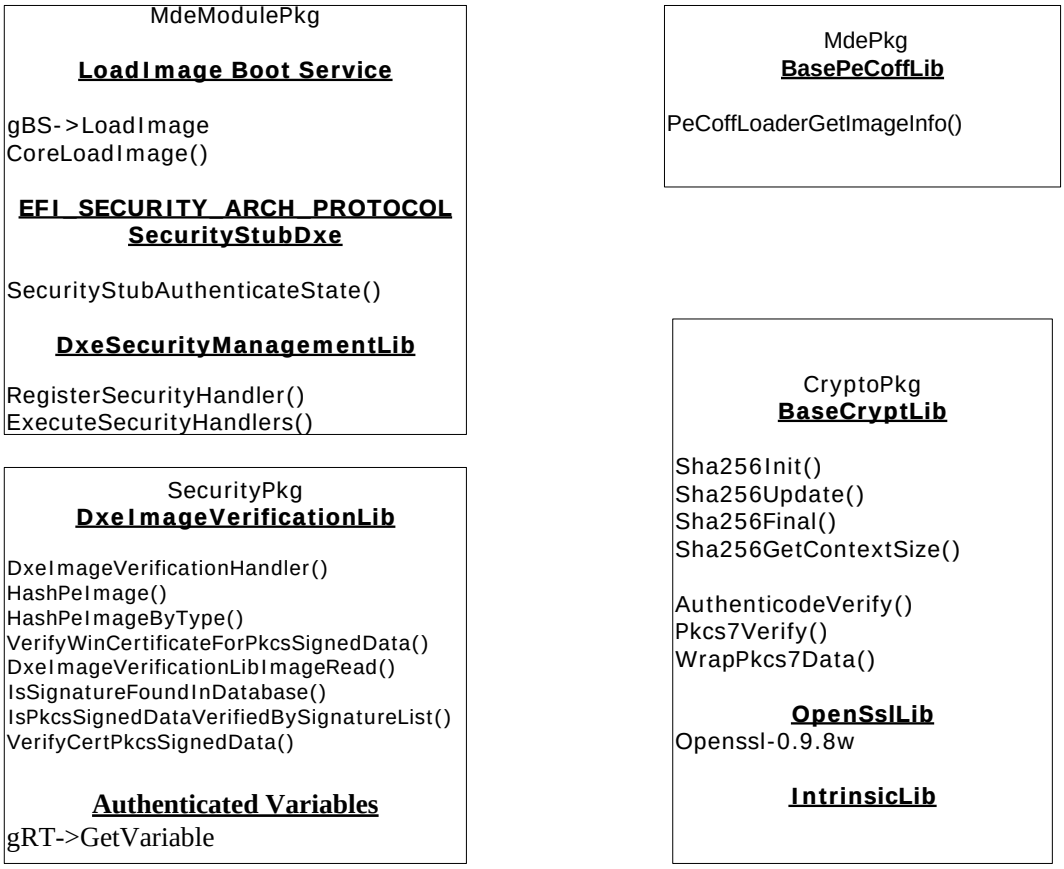


Figure 16 – Modules and Functions

3.5 Image Verification Flow Description

The image verification code flow is divided into three sections:

- Security Handler Initialization Flow
- LoadImage to Invocation of the Security Handler
- Security Handler Execution

3.5.1.1.1 Security Handler Initialization

The security handler commences execution in its initialization routine, as shown in the figure below.

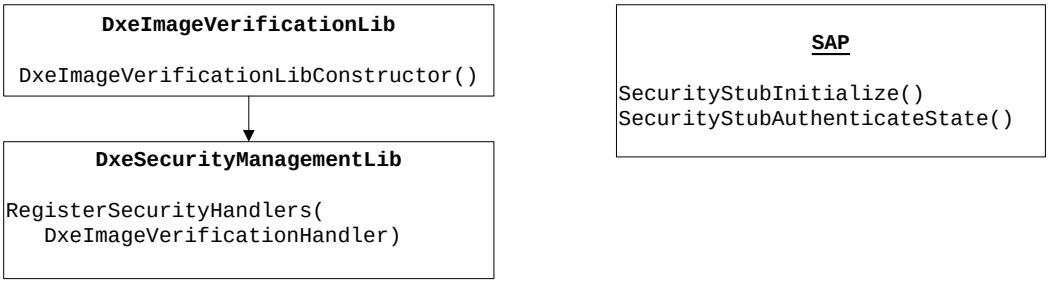


Figure 17 - Handler Initialization

Specifically, the SecurityStub is initialized by its SecurityStubInitialize routine which installs the Security Architectural Protocol (SAP), making **SecurityStubAuthenticateState()** available through SAP. SAP's **SecurityStubAuthenticateState()** is how registered security handlers will be invoked during image verification.

DxeImageVerificationLib is initialized by its constructor **DxeImageVerificationLibConstructor()** which calls DxeSeurityManagementLib's **RegisterSecurityHandler()** to register DxeImageVerificationLib's **DxeImageVerificationHandler()** as a SecurityHandler in the mSecurityTable[] array.

3.5.2 LoadImage To Image Verification Handler Flow

The actions of the verifier are invoked in response to a UEFI image load request, as shown in the diagram below.

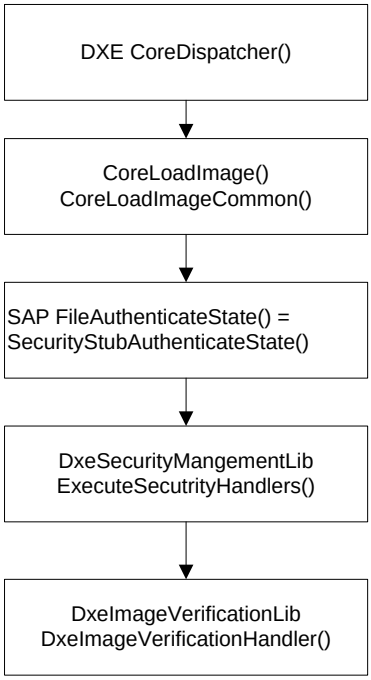


Figure 18 – LoadImage To Handler Invocation

Specifically, when an image is invoked for execution, which can occur when an image is entered through the UEFI shell or an option ROM is found during boot, the **LoadImage()** UEFI Boot Service is called to load the image into memory. These images can be unsigned PE/COFF images or signed images that comply with the Authenticode Specification. **LoadImage()** is implemented in **CoreLoadImage()**.

CoreLoadImage() calls **CoreLoadImageCommon()** which, among many other things, calls SAP->**FileAuthenticateSate()** to verify the image's authentication status. The image's authentication status is verified when **SecurityStubAuthenticateState()** executes the registered security handlers by calling DxeSecurityManagementLib's **ExecuteSecurityHandlers()** resulting in the image being passed to DxeImageVerificationLib's **DxeImageVerificationHandler()**.

3.5.3 Image Verification Handler Flow

The actual embodiment of the verification actions is detailed in the figure below.

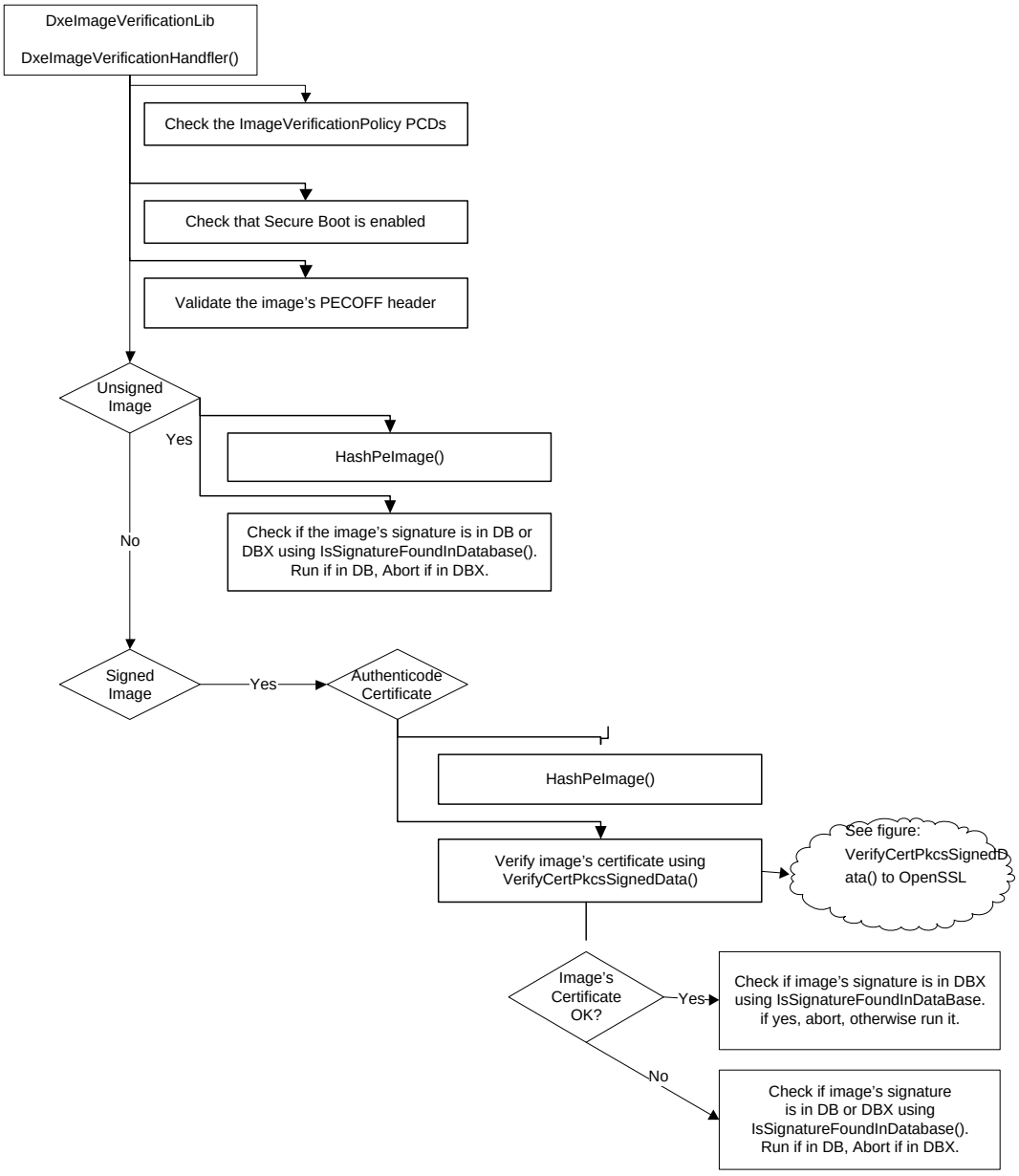


Figure 19 – Image Verification Handler

Specifically, the DxeImageVerificationLib's **`DxeImageVerificationHandler()`** performs the following steps to verify an image prior to its execution:

1. Check the policy settings in the **`ImageVerificationPolicy`** PCDs. These PCD's allow the platform to specify specific actions (including **`ALWAYS_EXECUTE`** or **`NEVER_EXECUTE`**) based on if the image was loaded from a fixed drive such as the EFI system partition, a removable drive, or an Option ROM.

Note: *A PCD or "platform configuration database" entry (PCD), when used as "fixed at build", are a stylized way to provide preprocessor directives. These are source artifacts outside of the UEFI specification.*

2. Call UefiLib's **GetEfiGlobalVariable2()** to check the "SecureBoot" variable. If it does not exist or is disabled, then exit since Secure Boot is disabled.
3. Validate the image's PeCoff hdr using PeCoffLib's **PeCoffLoaderGetImageInfo()** and other checks.
4. If the image is not signed,
   - Call **HashPeImage()** to calculate the image's hash as defined in the Authenticode Specification.
   - Call **IsSignatureFoundInDatabase()** to determine if the image's signature is in **DB** or **DBX**
     - Abort if the image's hash is in **DBX**.
     - Return **EFI_SUCCESS** if the image's hash is in **DB**.
5. The image is signed
6. Verify the image's certificate's size
   - Confirm that the image's certificate type is: **WIN_CERT_TYPE_PKCS_SIGNED_DATA**, signifying an Authenticode image.
     - Otherwise, abort.
   - Call **HashPeImageByType()**, which calls **HashPeImage()** to calculate the image's hash as defined in the Authenticode Specification.
   - Call **VerifyCertPkcsSignedData()** to verify the image's certificate. In UDK2010.SR1.UP1 and earlier implementations, an image's certificate is verified if it matches or chains to a **KEK** or **DB** entry, (or both).
     - If Image Certificate Verification passes:

Call **IsSignatureFoundInDatabase()** to check if the image's signature is in **DBX**,

If not, return **EFI_SUCCESS**

- If Image Certificate Verification fails:

Call **IsSignatureFoundInDatabase()** to determine if the image's signature is in **DB** or **DBX**

Abort if image's hash is in **DBX**

Return **EFI_SUCCESS** if image's hash is in **DB**

3.5.4 HashPeImageByType

DxeImageVerificationLib's **HashPeImageByType()** checks the hash algorithm encoded in the PKCS#7 data area of the image's Authenticode certificate. If a supported algorithm is found then **HashPeImage()** is called.

3.5.5 HashPeImage

DxeImageVerificationLib's **HashPeImage()** performs the steps defined in the PE/COFF and Authenticode Specifications to calculate the image's hash which excludes the Optional Header Checksum, Data Directory Certificate Table entry and the Attribute Certificate Table from the calculation since these fields change when the image is signed. The hash is calculated using BaseCryptLib's SHA256 hashing support.

3.5.6 IsSignatureFoundInDatabase

DxeImageVerificationLib's **IsSignatureFoundInDatabase()** checks if the specified signature is found in the specified variable (**DB** or **DBX**). The logic of this check is shown in the figure below.

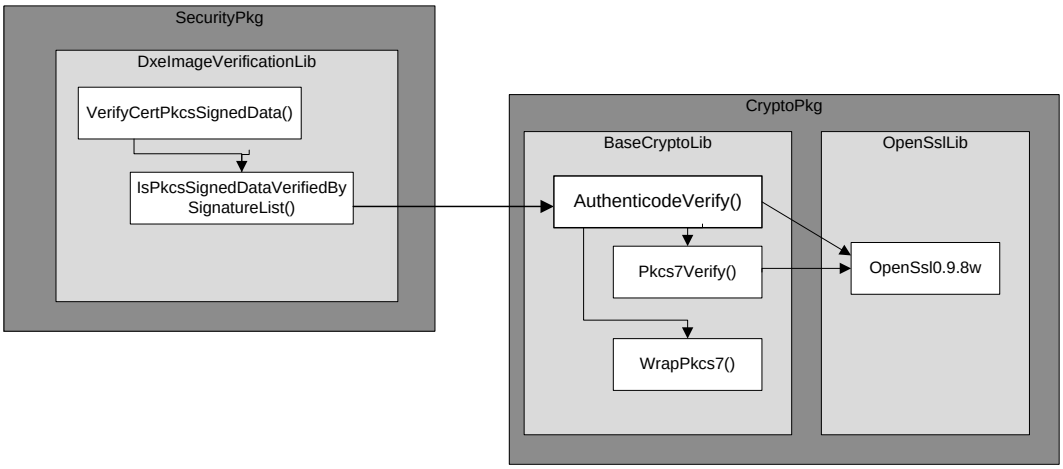


Figure 20 - VerifyCertPkcsSignedData to OpenSSL

3.5.7 VerifyCertPkcsSignData

DxeImageVerificationLib's **VerifyCertPkcsSignedData()** verifies if the image's certificate has been enrolled in **DBX**, **KEK** or **DB** using **IsPkcsSignedDataVerifiedBySignatureList()**. If found in **DBX**, abort. If found in **KEK** or **DB**, return **EFI_SUCCESS**. If not found in **KEK** or **DB**, abort.

3.5.8 IsPkcsSignedDataVerifiedBySignatureList

DxeImageVerificationLib's **IsPkcsSignedDataVerifiedBySignatureList()** iterates thru the specified variable's signature list checking it versus. the image's PkcsCertData by calling BaseCryptLib's **AuthenticodeVerify()**. In UDK2010.SR1.UP1 this check is hardcoded for a single certificate per image.

3.5.9 AuthenticodeVerify

BaseCryptLib's **AuthenticodeVerify()** verifies the PE/COFF image's Authenticode Signature as specified in the PE/COFF and Authenticode Specifications. This includes:

- Checking that the hash that was calculated for the image matches the hash retrieved from the image's Authenticode data using OpenSSL's PKCS7 parsing routines.
- Calling BaseCryptLib's **Pkcs7Verify()** to verify that the certificate in the image's Authenticode data matches the specified trusted certificate (i.e., the **KEK**, **DB** or **DBX** pointed to by the TrustedCert parameter.)

3.5.10 Pkcs7Verify

BaseCryptLib's **`Pkcs7Verify()`** verifies the image's PKCS#7 signed data as specified in the *PKCS #7 Cyptographic Message Syntax Standard.* The input signed data is wrapped in a PKCS#7 ContentInfo structure, which is verified by BaseCryptLib's **`WrapPkcs7Data()`**. OpenSslLib is used for parsing the ASN.1 DER encoded PKCS#7 data.

3.5.11 WrapPkcs7Data

BaseCryptLib's **`WrapPkcs7Data()`** checks the input PKCS#7 data is a wrapped in a ContentInfo structure. If not, a new structure is constructed to wrap the data. This is required before invoking OpenSslLib.

3.5.12 OpenSslLib entry

OpenSSL details are beyond the scope of this paper. Please see [OPENSSL].

3.5.13 More Information

Please see:

- MdeModulePkg\Core\Dxe\Dispatcher
- MdeModulePkg\Core\Dxe\Image
- MdeModulePkg\Universal\SecurityStubDxe
- MdeModulePkg \Library\DxeSecurityManagementLib
- SecurityPkg\Library\DxeImageVerificationLib
- CryptoPkg\Library\BaseCryptLib\Hash\CryptSha256.c
- CryptoPkg\Library\BaseCryptLib\Pk\CryptAuthenticode.c
- CryptoPkg\Library\BaseCryptLib\Pk\CryptPkcs7.c
- CryptoPkg\Library\OpenSslLib\openssl-0.9.8w

3.6 Summary

This section has provided an end-to-end description of the image verification process. This is the heart of UEFI Secure Boot.

4 Final thoughts

This portion of the document will delineate some of the items not covered in this document. Specifically, although the CW integrity model motivates the architectural choices made for inclusion of this feature, the correctness of the C code implementation of the capability is not treated. Such assurance and robustness claims are an important part of the software development processor [SDL] and should be pursued as the design is reduced to practice on a given system board.

Another challenge not treated here entails the business flow on how the PM pre-provisions certain keying material, including the initial PK, KEK, and db/dbx. This can include support for certain software publishers, such as a given set of operating system and device vendors. There is work afoot in the closed and open source operating system community to address these deployment issues.

Regarding deployments among many ecosystem participants, there has been a request to have multiple independent cryptographic signatures in a single Authenticode-signed PE/COFF image. The present signature is in the “Attribute Certificate Table” in a UEFI image. The current implementation of the UEFI Secure Boot verifier described in this document assumes that there is only one signature, but the PE/COFF specification [PECOFF] allows for a set of contiguous certificate entries (a list of certificates), and each certificate entry contains an independent Authenticode signature. The verification policy of the verifier is being updated to support this capability [MULTISIG]. This is a distinct capability from countersigning; Authenticode reads on having countersigning via PKCS#9, but the UEFI Secure Boot verifier does not have support.

Finally, this document omits implementation details on a significant aspect of the platform design, namely providing evidence and attestation. The UDK2010 has a TPM driver and integrated measured boot capability. These are complementary to the CW Code Signing Guard aspect of UEFI Secure Boot in that the measured boot provides the audit aspect of the security model. [UEFI-ITJ] also provides some detail on the “relationship to measured boot.”

Going forward, many aspects of the implementation will evolve. Some include architectural conformance, such as migrating the implementation to the PI1.2.1 security architectural protocol, different classes of platforms, optimizations for boot time and size. However, the contract between the various business parties, namely the PM, OSV, IHV, will be governed by the UEFI specification to ensure that the interoperability concerns are met.

4.1 Summary

This section has provided a review of the items treated in the paper, including a list of items not covered or in progress.

Appendix A References

[AUTHENTICODE] *Microsoft Portable Executable and Common Object File Format Specification,* http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx *Windows Authenticode Portable Executable Signature Format,* http://www.microsoft.com/whdc/winlogo/drvsign/Authenticode_PE.mspx

[BIBA] K. J. Biba, "Integrity Considerations For Secure Computer Systems," ESD-TR-76-372, NTIS#ADA039324, Electronic Systems Division, Air Force Systems Command, April 1977

[BIS] Boot Integrity Services http://www.intel.com/design/archives/wfm/downloads/bisspec.htm?wapkw=boot+integrity+services

[CLARK] D. Clark and D. Wilson, "A Comparison of Commercial and Military Security Policies," IEEE Symposium on Security and Privacy, 1987.

[EDK2] EFI Developer Kit www.tianocore.org

[HARDENING] Douglas MacIver, "Hardening the Attack Surface", UEFI Plugfest, February 2012, http://www.uefi.org/learning_center/UEFI_Plugfest_2012Q1_Microsoft_AttackSurface.pdf

[FACTORY] Kevin Davis, "Secure Boot Factory Tools," UEFI Plugfest, May 2012, http://www.uefi.org/learning_center/6_-_Insyde_Plugfest_May2012.pdf foil 5

[MULTISIG] Support for multiple signatures in a UEFI image https://sourceforge.net/apps/mediawiki/tianocore/index.php?title=SecurityPkg#Images_with_Multiple_Signatures

[NIST] Cooper, et al *BIOS Protection Guidelines*, NIST 800-147 http://csrc.nist.gov/publications/nistpubs/800-147/NIST-SP800-147-April2011.pdf

[OPENSSL] Network Security with OpenSSL, Viega, Messier and Chandra, O'Reilly 2002, ISBN: 978-0-596-00270-1

[PECOFF] PE/COFF Specification Version v8 http://msdn.microsoft.com/library/windows/hardware/gg463125

[PKCS7] PKCS#7: PKCS#7 Cyptographic Message Syntax Standard http://www.rsa.com/rsalabs/node.asp?id=2129

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

[SECURE BOOT] W. A. Arbaugh, D. J. Farber, and J. M. Smith, `
"A Secure and Reliable Bootstrap Architecture,'' in *Proceedings 1997 IEEE Symposium on Security and Privacy*, pp. 65-71, May 1997.

[SDL] Security Development Lifecycle
http://www.microsoft.com/security/sdl/default.aspx

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

[SIGNING] Signing UEFI Images
http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK_II_User_Documentation

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

[TIANO CORE] UDK2010 www.tianocore.org

[TRUSTED] Trusted Computing Group – TCG, www.trustedcomputinggroup.org

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

[UEFI-ITJ] UEFI Today: Bootstrapping the Continuum
http://www.intel.com/technology/itj/2011/v15i1/index.htm

[UEFI MAIN SPECIFICATION] UEFI Specification, Version 2.3.1C, June 2012
www.uefi.org

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

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

[UEFI PI SPECIFICATION] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.2.1, May 2012 www.uefi.org

[WIN LOGO] Microsoft Corp, *Windows 8 Hardware Certification Requirements: Windows 8 System Requirements*, July 2012, http://msdn.microsoft.com/en-us/library/windows/hardware/hh748188.aspx

Appendix B
Integrity Protection Models

This section introduces the Biba security model and the Clark-Wilson information flow rules that we'll be using in the rest of the paper. The intent is to educate readers who are not intimately familiar with these models.

B.1 Biba Model

The Biba model is built around the notion that there is a set of (ordered) integrity levels and sets up two rules:

- No-Write-Up – this means that lower integrity subjects (process/thread) cannot write over higher integrity objects to destroy their integrity.
- No-Read-Down – this means that high integrity subjects cannot read low integrity data. Cases where a high subject *executes-data* of a low integrity subject is also considered *read-down* for the purposes of this paper.

The problem with the Biba model is that the rules are very stringent. For example, reading data off the network for upgrade can lead to a violation of the rules. Instead of this, the Clark-Wilson rules relax Biba rules in a controlled manner – (CW-GUARD in Table 1) by saying that higher integrity subjects can read lower integrity data but must do so through the services of a guard – a data validation mechanism.

B.2 Clark-Wilson Information Flow Rules

In contrast to the Biba model, the Clark-Wilson information flow rules specify restrictions on information flows between pairs of integrity levels such as the PM compartment and the non-PM pre-OS compartment. This is not a limitation because information flows are transitive.

B.3 Definitions

1. **Constrained Data Items**: The Clark-Wilson information flow rules speak to information flow between a pair integrity levels, because one integrity level is of higher integrity than another; Clark-Wilson uses the term "Constrained Data Item" (CDI) to refer to objects or subjects (processes/threads) of the higher integrity level and "Unconstrained Data Item" (UDI) to refer to objects of the lower integrity level. Because there are multiple integrity levels the term "Constrained/Unconstrained" should be read relative to two different integrity levels. For example, data and code in the PM compartment is a CDI relative to the non-PM pre-OS compartment or the OS compartment.

2. **Integrity Validation Procedure** (IVP) or **Integrity Verification Guard**: An IVP/Guard checks to see if a CDI has the specified degree of integrity. Some examples of IVP are below.

a. **Code Integrity maintenance Guard:** An example IVP that checks the integrity of code is digital signing of the code. The set of valid root keys is policy data for the guard and this policy may be edited by the admin.

b. **Data Integrity maintenance Guard:** Data HMAC in transit, a guard that protects against direct modification of a physical resource for data – such as dropping direct writes to a volume with an active file system on it.

IVPs are guards for information flows between differing integrity levels. Guards allow for the controlled relaxation of the Biba No-Write-Up/No-Read-Down rules across integrity levels of the system. The main reason why guards are important is that there is no protection within an integrity level. The guard provides separation across integrity levels, so it is the guard that must be extensively validated in the analysis of the system. On the other hand the rest of the code running in the high level does not have to be validated, since the guard is the only way by which low information can enter a higher integrity level. Therefore, verification and penetration test should focus on the guard.

3. **Subjects**: Subjects are active entities with an associated security context like processes/threads/virtual machines etc. A user authenticates as a subject by providing a credential. The subject security context in Windows consists of a list of SIDs and Windows Privileges.

4. **Transformation Procedures** (TP): A Transformation Procedure is an *integrity preserving* mechanism by which a subject transforms data in the system (read/write/copy/delete/…..) which are sequences of atomic actions in the system. Transformation Procedures are used to implement IVP. Since a TP is integrity preserving, applying a TP to a CDI will preserve the integrity of the CDI.

B.3.1 *Clark-Wilson Information Flow Rules*

The table below details the CW information flow rules.

Table 1 CW Information Flow Rules

Rule	Rule Specification	What does it mean?
CW-VALIDATE-INTEGRITY INTEGRITY MUST BE VALIDATABLE	IVP's must be available on the system for validating the integrity of any CDI.	Any piece of code/data that claims to be at some integrity level must be verified to have it before being consumed at that level. For example, kernel mode code signing is an implementation of this rule for integrity validation of kernel mode code. There are two ways to ensure that a CDI (integrity protected item) remains integrity protected: Access Control – make sure that the item cannot be modified at all by items of lower integrity. Set an ACL on the item – for example, the data for the PM compartment can be stored in block-locked flash that is not writable by modules in the non-PM pre-OS. Integrity protect the item, perhaps by hashing or signing or sealing.

Rule	Rule Specification	What does it mean?
		Generally, (a) is preferable to (b).
CW-CHOKEPOINT MUST HAVE CHOKEPOINTS FOR INTEGRITY PROTECTED DATA	A CDI can be changed only by a TP.	The only way by which CDI can be modified maintains the integrity of the CDI since only TP are used to edit the CDI and TP maintain the integrity of CDI. If there are ways by which the CDI can be modified, without using a TP then we cannot assure the integrity protection of the high integrity object. For example, if one can install code into the PM compartment without going through the signed capsule update chokepoint then this rule is violated.
CW-ACCESS-CONTROL-INTEGRITY-DATA MUST RESTRICT WHO CAN EDIT INTEGRITY PROTECTED DATA	Subjects can only initiate certain TPs on certain CDI – there is a table (*s, t, d*) that specifies the set of TP, *t,* that a subject *s* can perform on a datum *d*	Recall that editing high-integrity data must be done via a TP due to CW-CHOKEPOINT. Since the TP may change the integrity of the system, such as changing the Platform Key, we must restrict the set of subjects (users) that can invoke the TP used to reset the Platform Key. For example, access to the TP used to reset the Platform Key should be gated on being in the *platform_admin* role.
CW-GUARD INTEGRITY UPGRADE DONE THROUGH GUARDS	Guards are TP that act on UDI and produce CDI as output. That is, low integrity data/code can be promoted to high integrity (after it is verified by a guard) using a special TP – the "un-tainting" guard.	Guards are used to validate UDI transitioning across a security boundary. For example when a CDI piece of code accepts data from an untrusted network (UDI), then the data transitions across the UDI->CDI boundary. This data needs to be validated and guards do this. Guards relax the no-read down rule of Biba integrity, so that read-downs are allowed via an integrity guard. The guard is an airlock (a one-way data flow) from a lower integrity level to a higher level. Guards can guard entry into the admin role as well. For example, if the capsule update mechanism is the only mechanism to upgrade the PM compartment, the signature checking guard prevents install of unsigned capsule updates. **It is very important that the guard must have integrity at least as high as the level that it is protecting. For example, the signing check should be done by code inside the Platform Supplier or PM compartment.**
CW-AUDIT MUST AUDIT INTEGRITY AFFECTING OPERATIONS	Each TP application must be audited – since TP can affect the integrity of code/data	An example of this being violated in the system today is modification of the audit log post-hoc.
CW-ADMIN-AUTH MUST AUTHENTICATE ADMINS	System must authenticate subjects attempting to perform a TP	By CW-ACCESS-CONTROL-INTEGRITY-DATA, only certain subjects like the *platform admin* can instantiate TP on protected objects. What this requirement says is that these subjects must be authenticated. We suggest authenticating the *platform admin* based on physical presence.

Rule	Rule Specification	What does it mean?
CW-SECADMIN MUST HAVE SECURITY ADMINS	Only special subjects (security administrators) must be able to change any authorization related lists such as the map (*s, t, d*). Generally this leads to the concept of a per-layer admin.	Since the administrator now executes code at every level, it is very hard to restrict information flow across levels because the administrator provides a natural conduit for such flows. Having the *platform admin (i.e., platform owner)* and *PM admin(i.e., platform manufacturer or platform supplier)* roles satisfies this requirement in our case.
CW-INTEGRITY-PRESERVED TRUSTED PROCEDURES PRESERVE INTEGRITY	Application of a TP to a CDI must preserve integrity of the CDI – the set of CDI are closed under TP actions	This means that a TP does not destroy the integrity of a CDI. How the TP preserves the integrity of a CDI depends, but if a procedure claims to be a TP, then it must be verifiable that the TP maintains the integrity of a CDI on which it acts.

Previously we stated that the C-W information flow rules were to be used to drive the Integrity Analysis Program.

B.3.2 *Integrity Protection Affecting Information Flows*

The C-W and Biba rules are rules that say what flows of information can result in integrity compromise. In the rest of this document we divide information flows into the following classes:

1. Write-like information flows: Write-like information flows are flows from a starting integrity type to an ending integrity type by which the starting integrity type can write chosen data to an ending integrity type. The C-W and Biba rules seek to ban write like information flows where the starting integrity type is of lower integrity than the ending integrity type.

2. Read-like information flows: Read-like information flows are flows from a starting integrity type to an ending integrity type by which the ending integrity type can read chosen data low integrity data from a starting integrity type. The C-W and Biba rules seek to ban un-guarded read like information flows where the starting integrity type is of lower integrity than the ending integrity type.

Our analysis tries to find such flows across the compartments defined in the security requirements.

Appendix C
STRIDE

Vulnerability in a software product can subject the computer on which it is running to various attacks. Attacks may be grouped in the following categories:

- **Spoofing**: An attacker pretends that he is someone else, perhaps in order to inflict some damage on the person or organization impersonated.
- **Tampering**: An attacker is able to modify data or program behavior.
- **Repudiation**: An attacker, who has previously taken some action, is able to deny that he took it.
- **Information Disclosure**: An attacker is able to obtain access to information that he is not allowed to have.
- **Denial of Service**: An attacker prevents the system attacked from providing services to its legitimate users. The victim may become bogged down in fake workload, or even shut down completely.
- **Elevation of Privilege**: An attacker, who has entered the system at a low privilege level (such as a user), acquires higher privileges (such as those of an administrator).

These six categories are encapsulated in the acronym **STRIDE.** [THREAT]

- A **vulnerability** is a defect in the design of a software product that makes it possible for an attacker to carry out an attack on a system.
- A **threat** is a possible way to attack a software product
- An **exploit** is a specific technique devised by an attacker to take advantage of a vulnerability. It is a realization of a threat.

Authors

Lee G. Rosenbaum (lee.g.rosenbaum@intel.com) is a UEFI Engineer with the Software and Services Group at Intel. Lee has over 35 years of experience in the computer industry. His system software development has ranged from printers to scale-up servers to his present role in UEFI security development.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Principal Engineer with the Software and Services Group at Intel Corporation. Vincent has over 20 years of experience in the computer industry. His system software development experience has ranged from embedded control for SCADA, hardware RAID, solid state disks, system management firmware, BIOS, and UEFI.

The authors also wish to acknowledge the input from the the Intel SSG team working on UEFI both in the US and PRC, along with participants of the UEFI Forum and contributors to the edk2 project and mailing list.