White Paper

A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDKII

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

September 2014

Executive Summary

This paper presents the internal structure and boot flow of the SMM-based UEFI Authenticated Variable driver in the Security Package of the EDKII.

Prerequisite
This paper assumes that audience has EDKII/UEFI firmware development experience. He or she should also be familiar with UEFI/PI firmware infrastructure, such as SEC, PEI, DXE, runtime phase.

Table of Contents

Overview

Introduction to the SMM Authenticated Variable

UEFI Authenticated Variable is designed to provision and maintain the UEFI secure boot status. The platform firmware keys, like 1) Platform Keys, 2) Key Exchange Keys, 3) Image Signature Database, are all stored in UEFI authenticated variable. Various documentation can be found on UEFI Secure boot and securing the platform [SECURE1][SECURE2][SECURE3].

EDKII is open source implementation for UEFI firmware. In SecurityPkg, VariableAuthenticated\RuntimeDxe is an implementation of authenticated variable services defined in UEFI specification. This document will introduce detail on how EDKII SMM Authenticated Variable driver works.

Today UEFI Authenticated variables are managed by the IA firmware.

UEFI Authenticated Variables are a type of UEFI variable that includes the "Authenticated Write" attribute **EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS.** When this bit is set, these variables need to be cryptographically verified for updates. The SetVariable() API is in the UEFI Specification, chapter 7.2 [UEFI]. This is an API exposed by the IA firmware.

Today to protect the UEFI authenticated variables, the IA firmware will generate a System Manage Interrupt and pass control to System Management Mode (SMM). In SMM, there is a UEFI Variable driver with sources at https://svn.code.sf.net/p/edk2/code/trunk/edk2/SecurityPkg/VariableAuthenticated/RuntimeDxe/ including files https://svn.code.sf.net/p/edk2/code/trunk/edk2/SecurityPkg/VariableAuthenticated/RuntimeDxe/VariableSmmRuntimeDxe.c and https://svn.code.sf.net/p/edk2/code/trunk/edk2/SecurityPkg/VariableAuthenticated/RuntimeDxe/VariableSmm.c. VariableSmmRuntimeDxe.c is UEFI Runtime phase driver. It passes the variable access request to VariableSmm.c by using Software System Management Interrupt (SW SMI). The variableSmm.c exists in system management ram (SMRAM). The UEFI PI specification defines a software model for running code in SMM in volume 4 of the UEFI Platform Initialization (PI) specification [UEFI PI Specification].

This driver runs in SMM and expects to have another SMM driver, such as the closed sourced SMM FVB (Firmware Volume Block) driver, in order to write to SPI NOR from SMRAM. The PCH or other IOH is programmed such that the write access to SPI NOR can only occur from SMM.

Anyone can read the UEFI authenticated variables, but only the holder of a private key can modify the variables. The above VariabledAuthenticed Driver authentication support component (https://svn.code.sf.net/p/edk2/code/trunk/edk2/SecurityPkg/VariableAuthenticated/RuntimeDxe

/AuthService.c) uses a public key in a stored authenticated variable to guarantee that the private key holder truly did the variable update.

Details on Authenticated Variables can be found in the UEFI spec at 7.2.1 of the UEFI Spec.

SO today's art includes SMM for isolated execution and volatile storage, with SMM-based access control for writes to the SPI NOR.

OS ring 0 calls UEFI runtime service set variable, and the UEFI runtime firmware implementation of set variable generates an SMI and control passes to SMM. The UEFI SMM variable driver controls SPI NOR flash to manage the authenticated variables

Most OS's use authenticated variables with *AuthInfo.CertType* set to **EFI_CERT_TYPE_PKCS7_GUID**.

As such, the code to support authenticated variables has to support PKCS7-based signatures, including X509 certificates, and cryptographic algorithms of SHA256 and RSA2048.

The EDK2 open source implementation exposes those primitives via the https://svn.code.sf.net/p/edk2/code/trunk/edk2/CryptoPkg/ The CryptoPkg has the subset of API's required by the SMM-based UEFI Authenticated Variable driver. Underneath this API we use OpenSSL-based code.

For size of the authenticated variable store, Windows8.1 recommends http://msdn.microsoft.com/en-us/library/windows/hardware/dn423132.aspx **Reserved Memory for Windows Secure Boot UEFI Variables**. A total of at least 64 KB of non-volatile NVRAM storage memory must be reserved for NV UEFI variables (authenticated and unauthenticated, BS and RT) used by UEFI Secure Boot and Windows, and the maximum supported variable size must be at least 32kB. There is no maximum NVRAM storage limit.

In the authenticated variable store we keep several special variables, namely the PK, KEK, db, and dbx (see 28.3 of the UEFI 2.4 spec).

Threat Model

The platform builder needs to avoid bypass of the logic that writes to the authenticated variable writes.

Physical attacks that replace SPI NOR or use Dediprog to modify the UEFI variables can roll-back or add errant entries. Similarly, software attacking where ring0 non-UEFI code can directly modify the flash region containing the authenticated variables poses another concern.

So the code that controls writes to the flash for the authenticated variables AND the code that verifies the signature in the AuthInfo of the authenticated variable MUST be protected at runtime from ring0 attacks (and provenance of this code must be guaranteed by some secure boot/update process).

From the above you can observe the need for an isolated execution (IsoW) to write to flash store of authenticated variables and verify (IsoV) signature of the authenticated variable when updating these variables.

Many platforms today use SMM and SPI NOR write trapping to have isolated, access controlled usage of this persistent store.

Summary
This section provided an overview of UEFI Authenticated Variable and EDKII. In next 3 sections, we will introduce 3 parts of SMM authenticated variable in EDKII. Part I focus on SMM, part II focus on authentication, part III focus on variable.

Part I - SMM

Why SMM?

An authenticated variable implementation requires an isolated execution environment to do the authentication and update flash. If there is no isolated environment, malicious code can bypass authentication check and update variable region directly.

System Management Mode (SMM) is an isolated environment defined in IA CPU architecture. SMM firmware infrastructure is defined in UEFI PI specification volume 4, and SMM core is implemented in EDKII.

SMM communication

SMM Authenticated Variable uses Smm Communication mechanism to pass parameter input by Variable service caller. During boot time, UEFI Variable driver will allocate a runtime communication buffer to use this buffer for parameter passing.

During runtime, when OS consumer calls variable services like GetVariable(), SetVariable(), the UEFI variable driver will copy data to communication buffer (copy the whole data, not the data pointer), then trigger Software SMI. In software SMI handler, SMM core infrastructure checks the “Header GUID” in SMM communication header, then dispatch to SMM Variable Handler. Then SMM Variable Handler checks the “Function ID” in Variable Communication Header, then dispatch to each variable services, like SmmGetVariable(), SmmSetVariable(). Then the SMM Variable services can get parameter, like GUID, Name, Data field, from VarirableAccess Communication Data.

Finally, SmmSetVariable() API can call the cryto services to do the authentication and update flash.

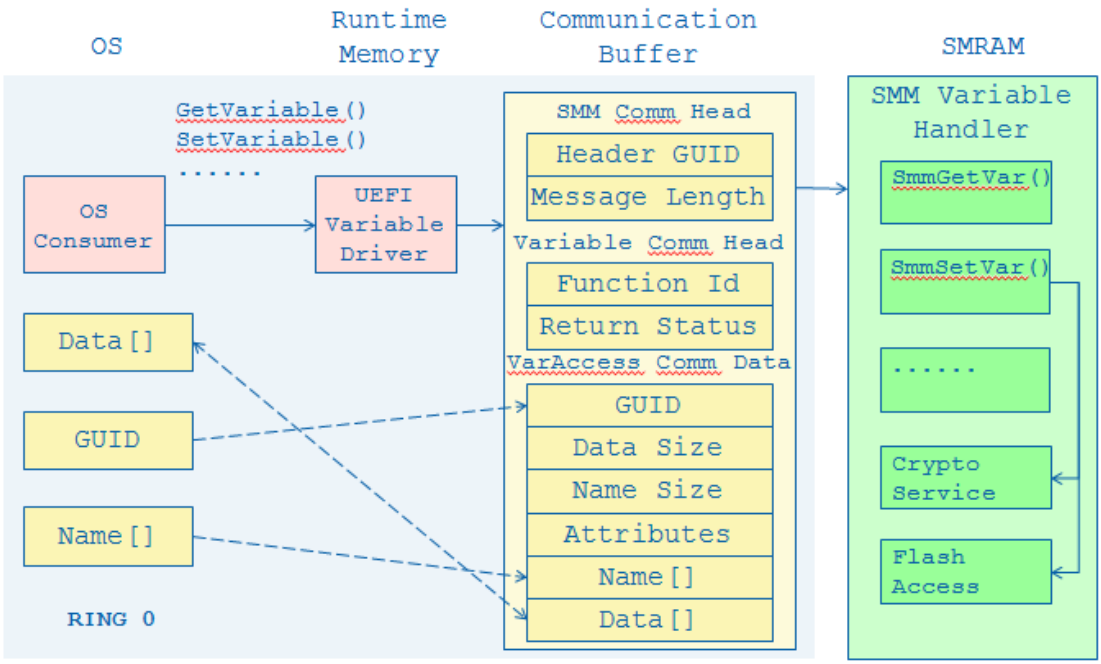

Figure 1 SMM Variable

Summary

This section describes the SMM part of SMM Authenticated Variable driver in EDKII, including why use SMM, and SMM communication.

Part II - Authentication

Why Authentication?

Authenticated Variable is designed to make sure the one who wants to update the variable is the one who has right to update the variable. If there is no authenticated, any malicious code can update a variable, which may break the system integrity and cause deny of service.

For a system with UEFI secure boot enabled, it need enroll 1) platform key, 2) key exchange keys, 3) image signature database. These keys are stored in variable, and the variable must be authenticated. Or the malicious code can break UEFI secure boot easily, by update the signature database.

Authenticated Variable Input Format

There are 2 kinds of authenticated variable.

1) Count based authenticated variable, if EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS attribute is set.
2) Time based authenticated variable, if EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS attribute is set. (Recommended)

When user calls SetVariable() API, the time based authenticated variable input data format is below.

There will be a time stamp associated with the authentication descriptor. Then the certificate type field, only PKCS7 is accepted for a time based authenticated variable. The certificate is DER-encoded PKCS #7 version 1.5 SignedData. Most important fields are 1) signer's DER-encoded X.509 certificate, 2) SHA256 hash of (VariableName, VariableGuid, Attributes, TimeStamp, New Variable Data). The Authenticated Variable driver will check the before update the variable content.

For count based authentication variable, the certificate type should be RSA_2048_SHA256. The certificate is in RSASSA_PKCS1-v1_5 format.

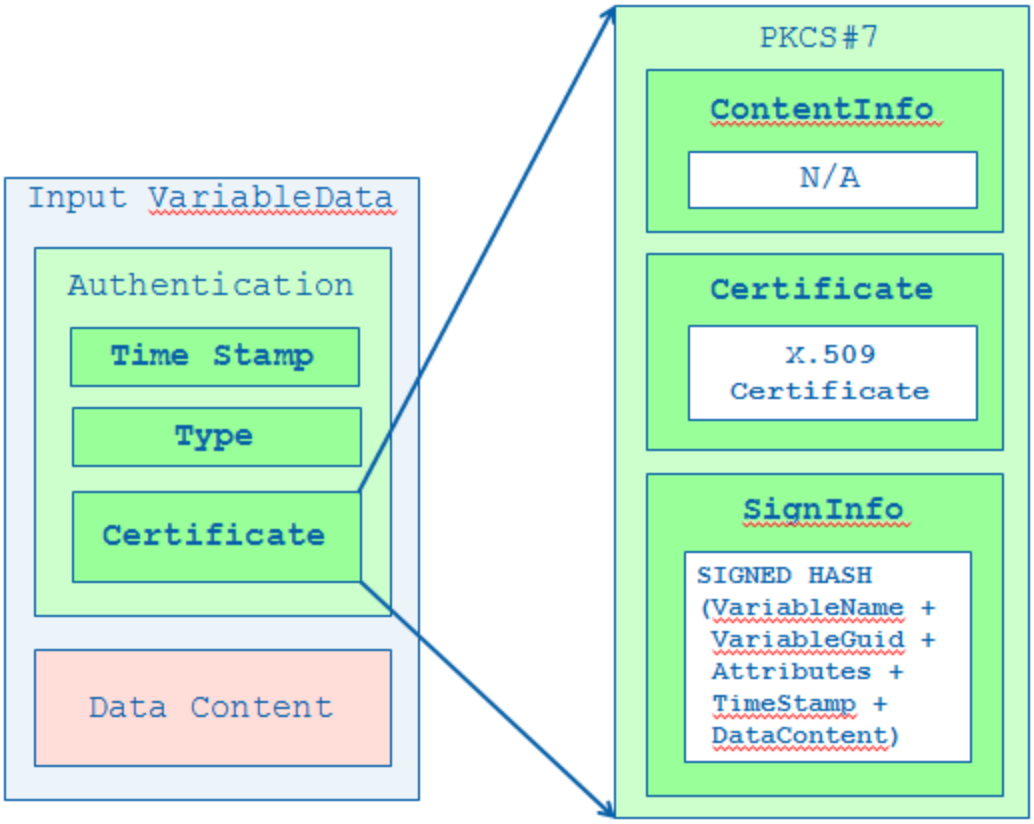

Figure 2: Authenticated Variable Input Format (Time based)

Authentication Flow

Authentication is need when consumer calls SetVariable() API. The SetVariable() will check if it is PK, or KEK, or signature database (db/dbx) to do different action.

Platform Key (PK) is the key to establish 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.

Key Exchange Key (KEK) is the key to establish a trust relationship between the operating system and the platform firmware. Each operating system (and potentially, each 3rd party application which need to communicate with platform firmware) enrolls a public key (KEKpub) into the platform firmware.

Image Signature Database (db/dbx) is the keys to maintain a list of authorized UEFI image (by db) and forbidden UEFI image (by dbx). The signature database is checked when UEFI boot manager is about to start a UEFI image. If UEFI image's signature is found in forbidden database, or not in authorized database, the UEFI image will be deferred and information placed in the Image Execution Information Table.

In general, below rules are applied during authenticated variable update:

1) PK variable update need be signed by old PK in USER MODE.
2) KEK variable update need be signed by PK in USER MODE.
3) Image signature database (db/dbx) update need be signed by PK or KEK in USER MODE.
4) Other authenticated variable update need be signed by creator of this authenticated variable.

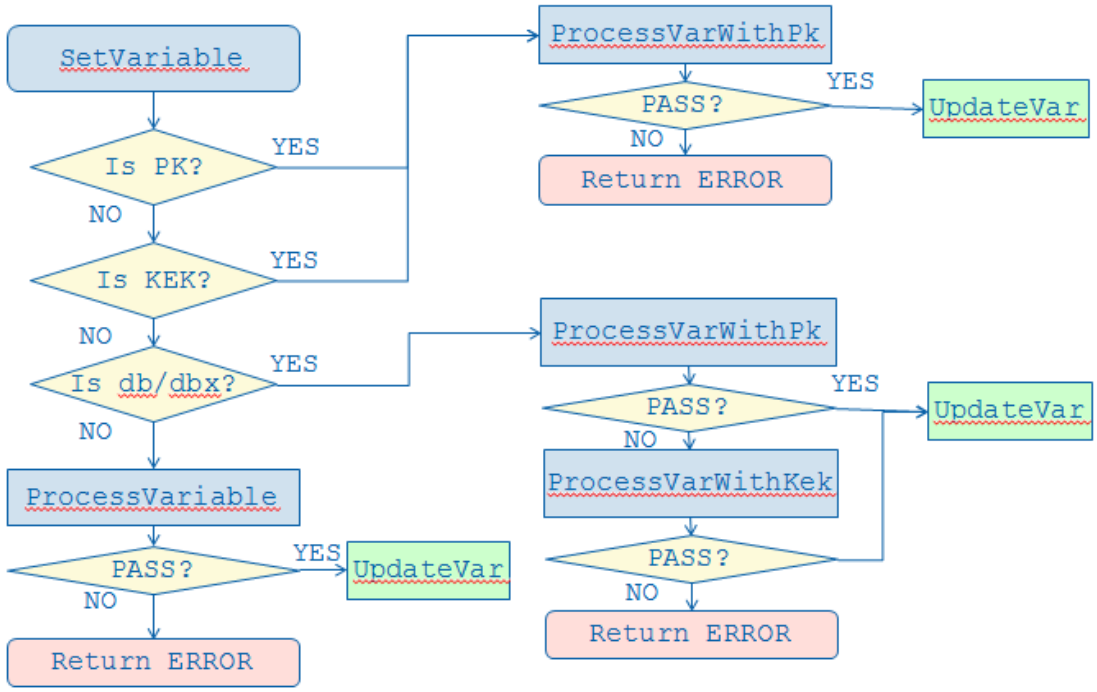

Figure 3: Variable Authentication Flow - Overview

Before we discuss detail authentication, let's introduce some important concept at first. They will be used later.

Concept: SETUP MODE V.S USER MODE

According to UEFI specification, the system is in SETUP MODE when PK is NOT enrolled, while the system is in USER MODE when PK is enrolled.
This platform mode is shown as UEFI specification defined L"SetupMode" variable.

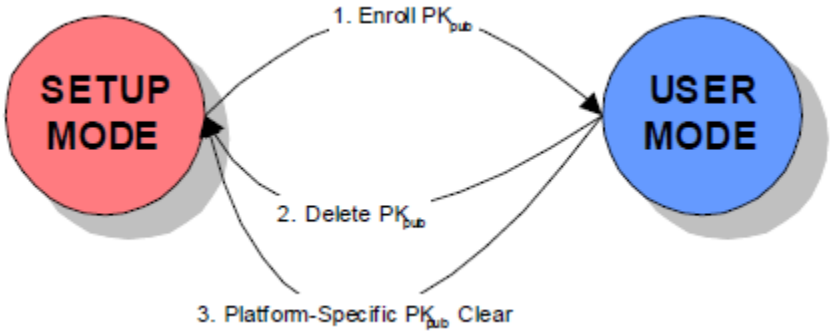

Figure 4: Setup Mode V.S User Mode

Concept: STANDARD MODE V.S. CUSTOM MODE

This is EDKII implementation specific mode for UEFI secure boot. Standard Secure Boot mode is the default mode as UEFI Spec's description. Custom Secure Boot mode allows for more flexibility as specified in the following:

1) PK variable update need NOT be signed by old PK.
2) KEK variable update need NOT be signed by PK.
3) Image signature database (db/dbx) update need NOT be signed by PK or KEK.

The switch between Standard Mode and Custom Mode need User Physical Present. It means a platform a way to detect a physical user present to do such action.
This mode is shown as EDKII implementation defined L"CustomMode" variable.

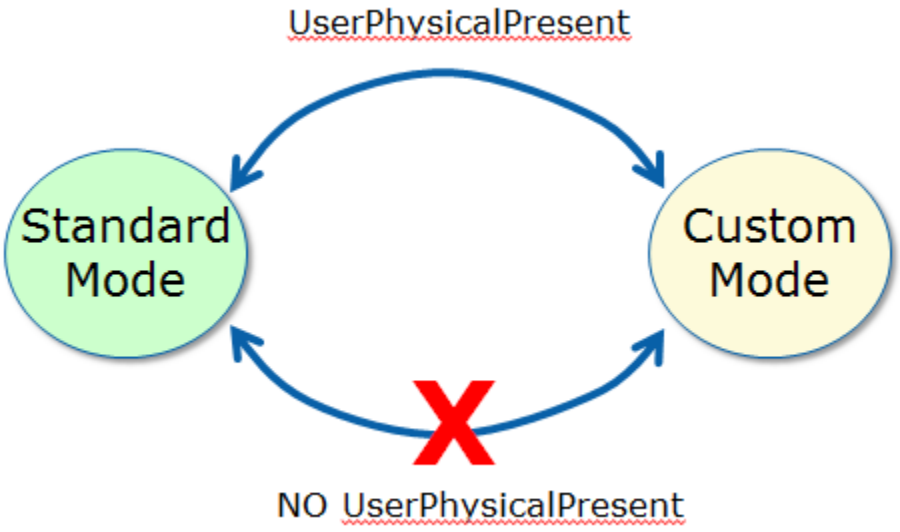

Figure 5: Standard Mode V.S Custom Mode

Concept: SecureBoot ENABLE V.S. DISABLE

In EDKII implementation, a platform can choose to enable or disable secure boot, if and only if user physical present. When SecureBoot mode is changed, it does not impact current boot. The new secure boot mode is adopted in next system boot.

This current secure boot state is shown as UEFI specification defined L"SetupBoot " variable. The new boot secure boot state is shown as EDKII implementation defined L"SecureBootEnable" variable.

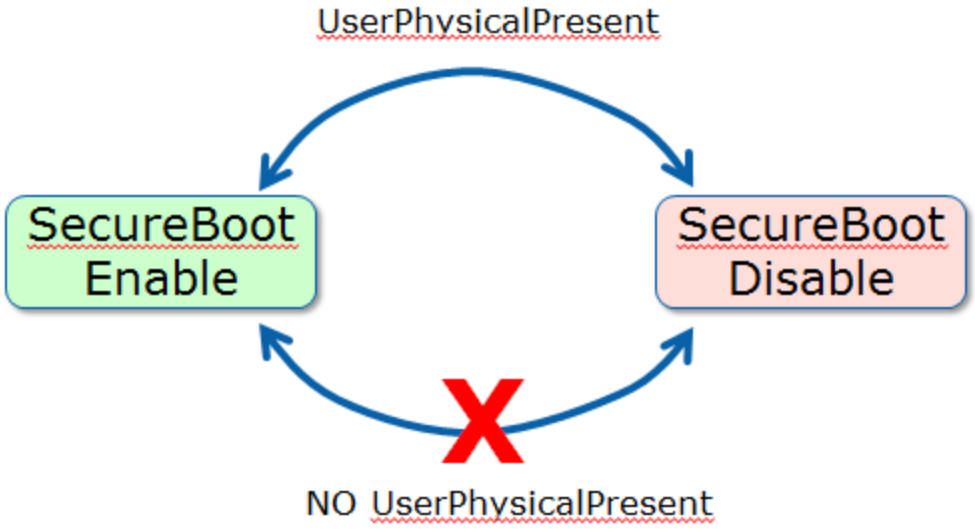

Figure 6: SecureBoot Enable V.S Disable

Concept: UserPhysicalPresent

In EDKII implementation, some secure boot variables change need user physical present to avoid remote attack. A platform need provide a library – PlatformSecureLib, which has UserPhysicalPresent() API. How to provide this API is platform implementation specific, and not addressed in this white paper

Now, we finish introduce the important concept, let's continue on detail of authentication flow.

Authentication Flow - ProcessVarWithPk

If in order to update PK or KEK, below authentication flow is used. First, if system is I CustomMode or UserPhysicalPresent, no authentication is needed. Second, if system is in SETUP mode not to enroll PK, no authentication is needed. Third, if system in USER mode, authentication with PK is performed. Last, if system in SETUP mode to enroll PK, authentication with this payload is performed.

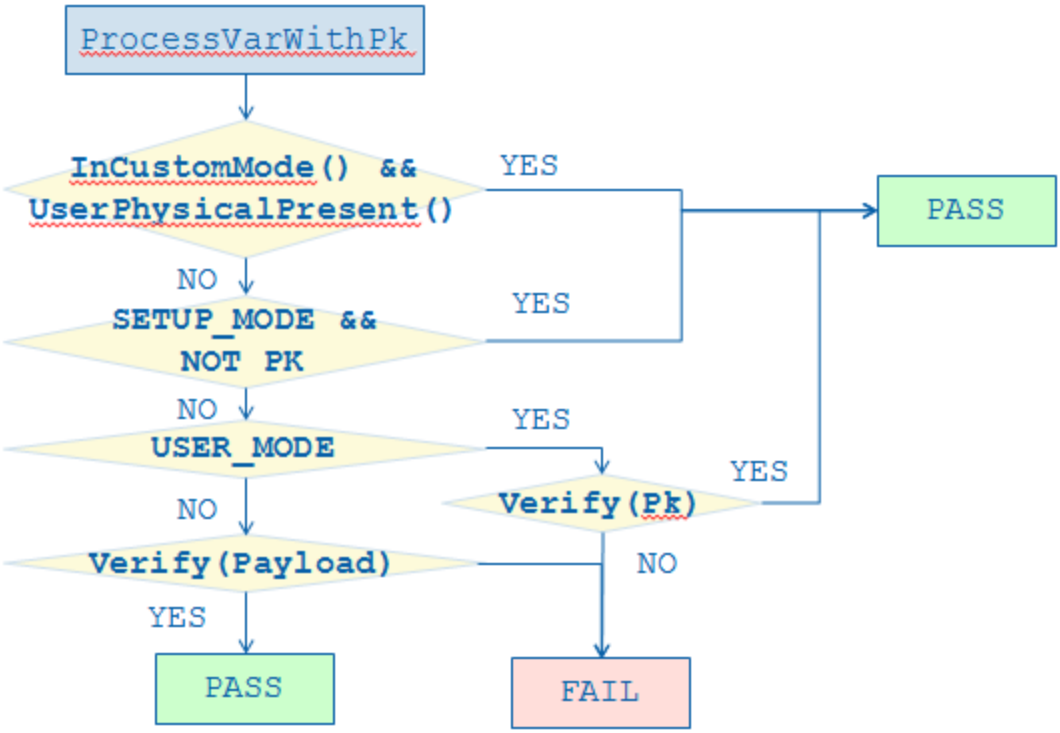

Figure 7: Variable Authentication Flow - ProcessVarWithPk

Authentication Flow - ProcessVarWithKek

If in order to update image signature database (db/dbx), system will invoke ProcessVarWithPk at firm. If fail, system will invoke ProcessVarWithKek. First, if system is CustomMode or UserPhysicalPresent, no authentication is needed. Second, if system is in SETUP mode, no authentication is needed. Last, authentication with KEK is performed.

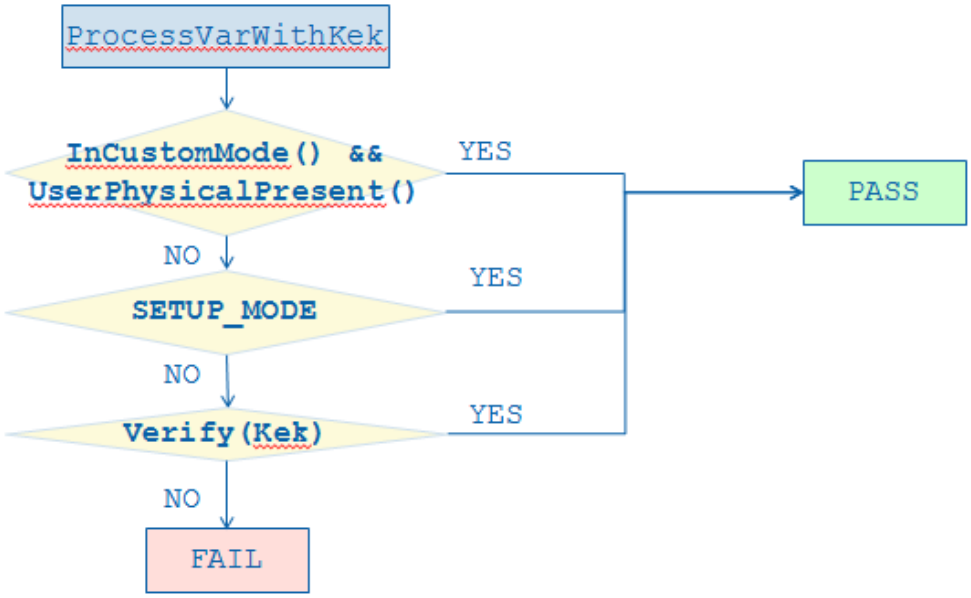

Figure 8: Variable Authentication Flow - ProcessVarWithKek

Authentication Flow - ProcessVariable

If system wants to update other authenticated variable, below flow is used.
First, if the variable requires PhysicalPresent, but system has no UserPhysicalPresent, update request is rejected. Second, if the variable is time based authenticated variable, authentication with time and creator's key is performed. Third, if the variable is count base authenticated variable, authentication with count and creator's key is performed. Last, if old variable has AUTH attribute, which does not patch current one, update request is rejected. Or it means no authentication is needed.

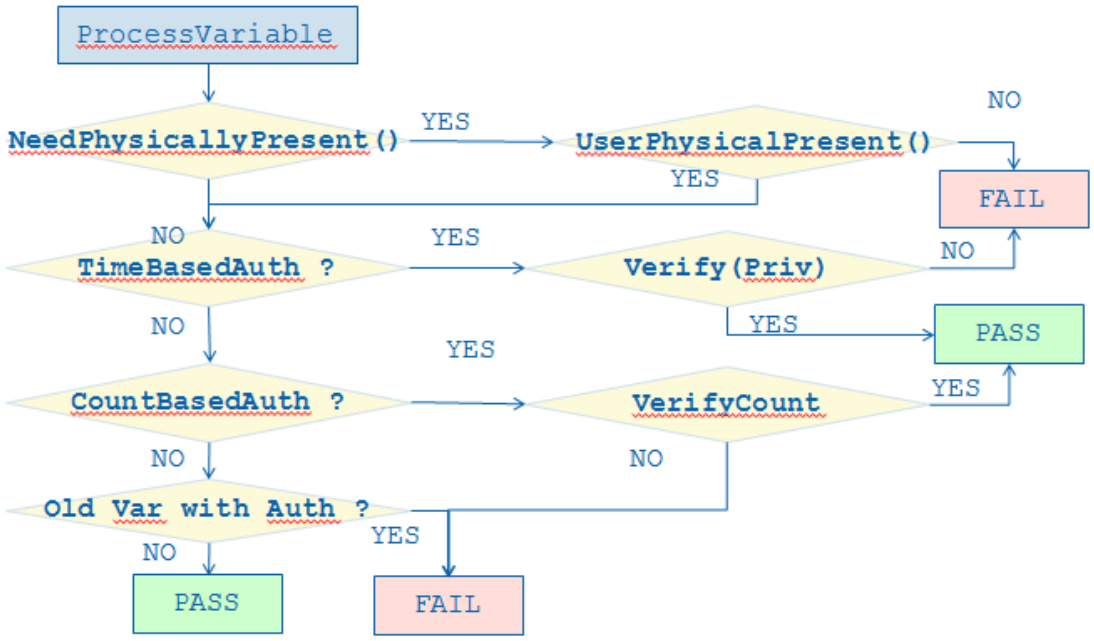

Figure 9: Variable Authentication Flow - ProcessVariable

Summary

This section describes the authentication part of SMM Authenticated Variable driver in EDKII, including why use authentication, authenticated variable input format, and authentication flow.

Part III - Variable

We have discussed SMM and authentication in previous chapters. Now let's focus on variable itself.

Variable Storage Format

First, how a variable is stored on a non-volatile storage, UEFI specification does not define the storage format. EDKII implementation chooses below format.

First variable exists in a special Variable Firmware Volume (FV). The variable FV can be identified by gEfiSystemNvDataFvGuid in File Syetem GUID field of FV header. The Authenticated Variable can be identified by gEfiAuthenticatedVariableGuid in GUID field of Variable Store Header. 0x5A in Format field of variable store header means this region is formatted. 0xFE in State field of variable store header means healthy.

Each individual variable is saved after variable storage header. 0x55AA in StartId of variable header means there is a new variable storage. 0x3F in State field means it is a valid variable added. 0x3D in state field means it is deleted. In some rare case, system reset during variable update, 0x7F in State field means only header is valid, no real variable data written. 0x3E in State field means this variable is delete transition.

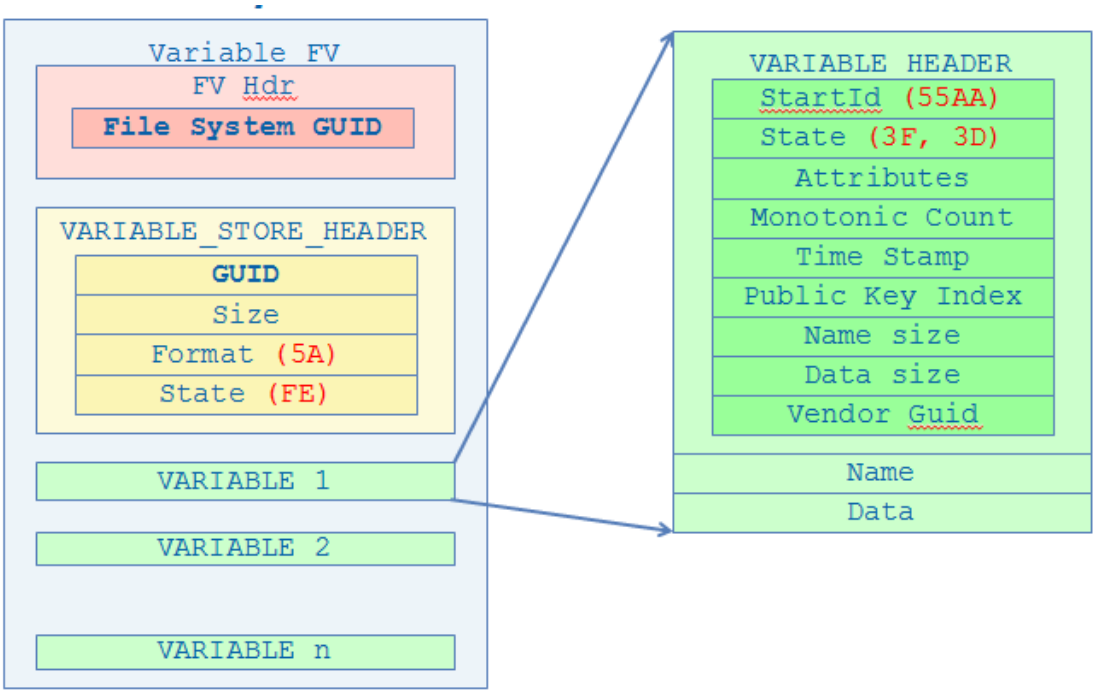

Figure 10: Variable Storage Format

Variable Update Flow

The State field is extremely useful on variable update. When system wants to update a variable, below flow is used.

1) The old variable state is marked as InDeleted.
2) New variable full header is added with state unchanged (0xFF).
3) New variable state is changed to Header Valid state.
4) New variable full data is added.
5) New variable state is changed to Added.
6) Old variable state is marked as deleted.

For a NOR flash, only one byte write from bit 1 to bit 0 can be atomic from hardware perspective. This State is designed to maintain atomicity of individual variable from software perspective. Each variable exists in ALL-or-NONE state. Partial variable is not allowed.

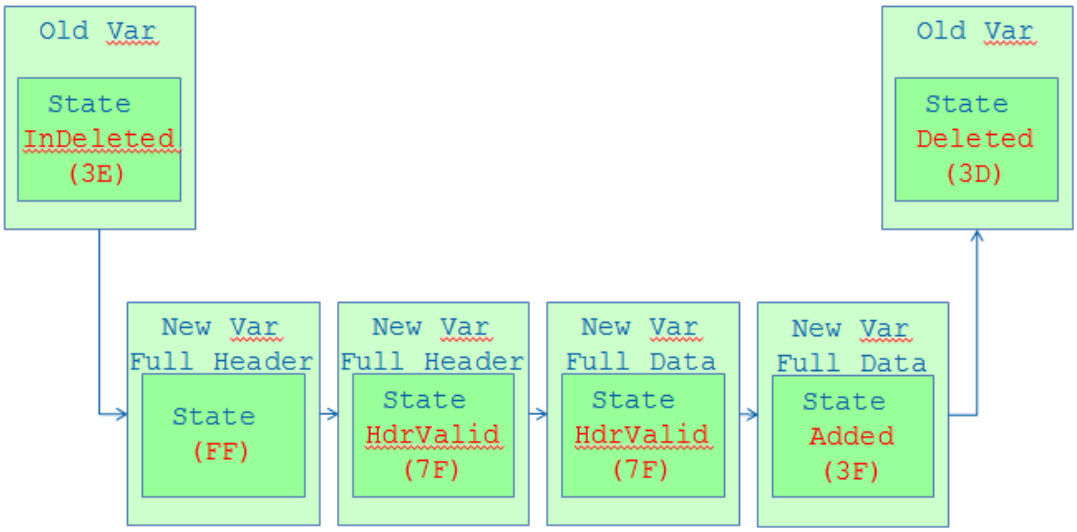

Figure 11: Variable Update flow

Variable Reclaim

From above variable update flow, when a variable is update. It is actually not UPDATED in original place, but marked as DELETED in original place, then ADDED in new place. If user updates variable several times, the non-volatile storage will have many DELETED variable, which is useless but occupies storage space.

In this case, there is a way called RECLAIM, to reorganize the variable region. It removes the DELETED variable to save space. Reclaim is triggered in below condition:

1) When update a variable or add a new variable, there is no enough free space.
2) On ready to boot event, there is no enough remaining free space.
3) On initialization, variable storage free space is not all 0xFF. (There must be something wrong.)

However, in EDKII reclaim is NOT supported during OS runtime, because it need erase entire flash block (NOR flash does not support write from bit 0 to bit 1), which is time consuming.

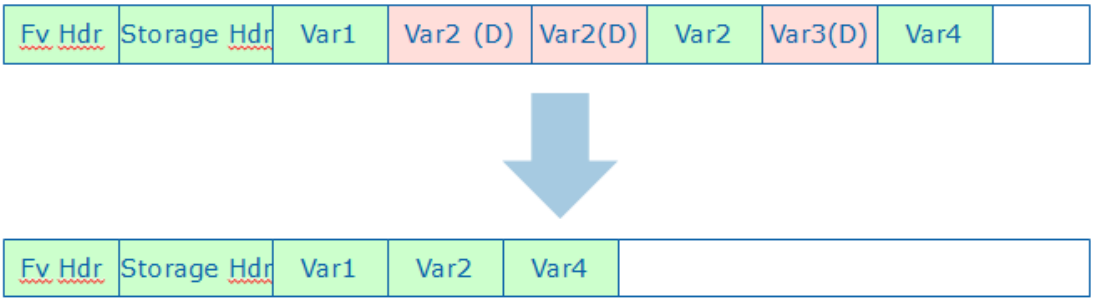

Figure 12: Variable Reclaim

Fault Tolerant Write

Individual variable atomicity is maintained by variable update flow. However, during variable reclaim, the flash block will be erased and be written again. In that period of time, if power lost or system shutdown due to user mistake, the variable firmware volume will be partially destroyed. That means if a variable crosses flash block, it might be partially correct. The atomicity is still broken. It is not acceptable.

Fault Tolerant Write (FTW) is designed to handle this situation. EDKII FTW driver is not included in variable driver. It is a standalone driver to provide capability of fault tolerant write. Every flash update driver can consume FTW protocol API to update flash part in a safety way.

Below is a high level picture of fault tolerant write flash layout. A FTW driver requires 2 flash parts:

1) FTW working block. This is a block to record write action. It is record block, not data block.
2) FTW spare block. This is a real data block to save the data. It must be bigger than the size of block required to update. In this case, it must be bigger than variable region.

In FTW working block, FTW driver put a data structure to record the write request and write status. See below picture for detail.

Signature field of WORKING_BLOCK_HEADER is used to identify it is FTW working block. After header, there will be multiply WriteQueueEntry. Each WriteQueueEntry has one WRITE_HEADER and 1 or more WRITE_RECORD. The number of WRITE_RECORD is NumberOfWrites field of WRITE_HEADER. The most important fields are Complete field of WRITE_HEADER, SpareComplete and DestinationComplet of WRITE_RECORD. Those fields record the status of write and guarantee the fault tolerant.

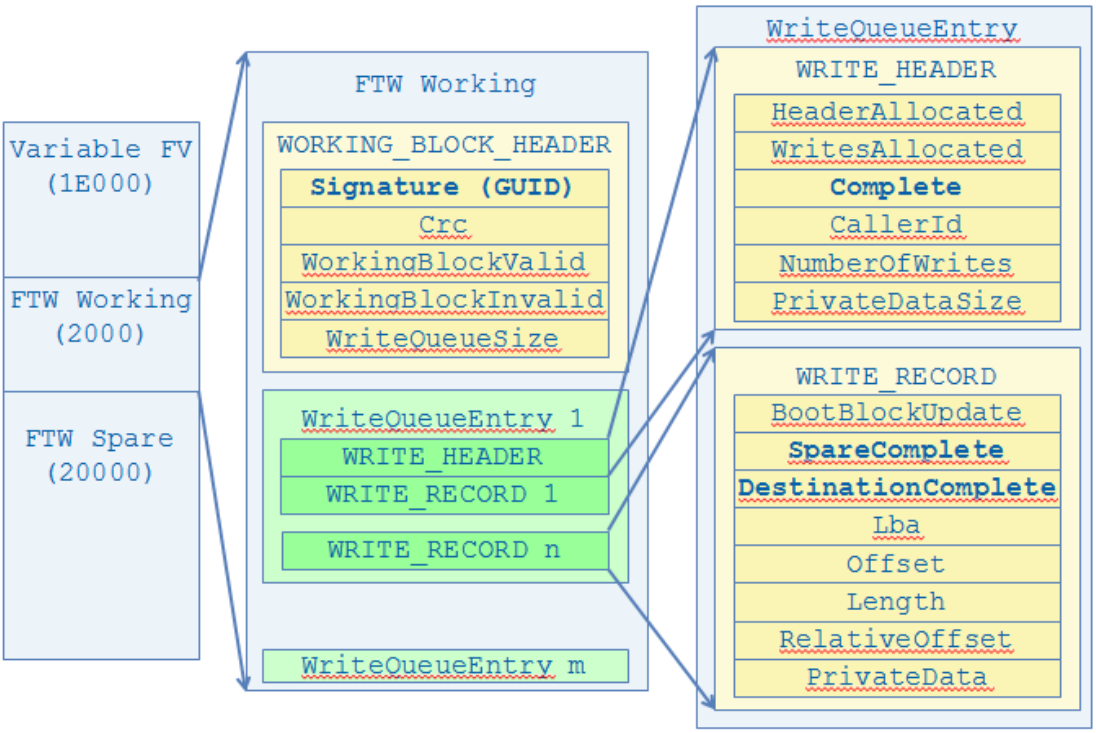

Figure 13: Fault Tolerant Write Flash Layout

Now, let's see how FTW->Write() work. Totally 8 steps are involved.

1) Step 1, when FTW->Write() is invoked, this API will record the request in FTW working block.
2) Step 2, this API finds SpareBuf on FTW spare flash area and back up to memory.

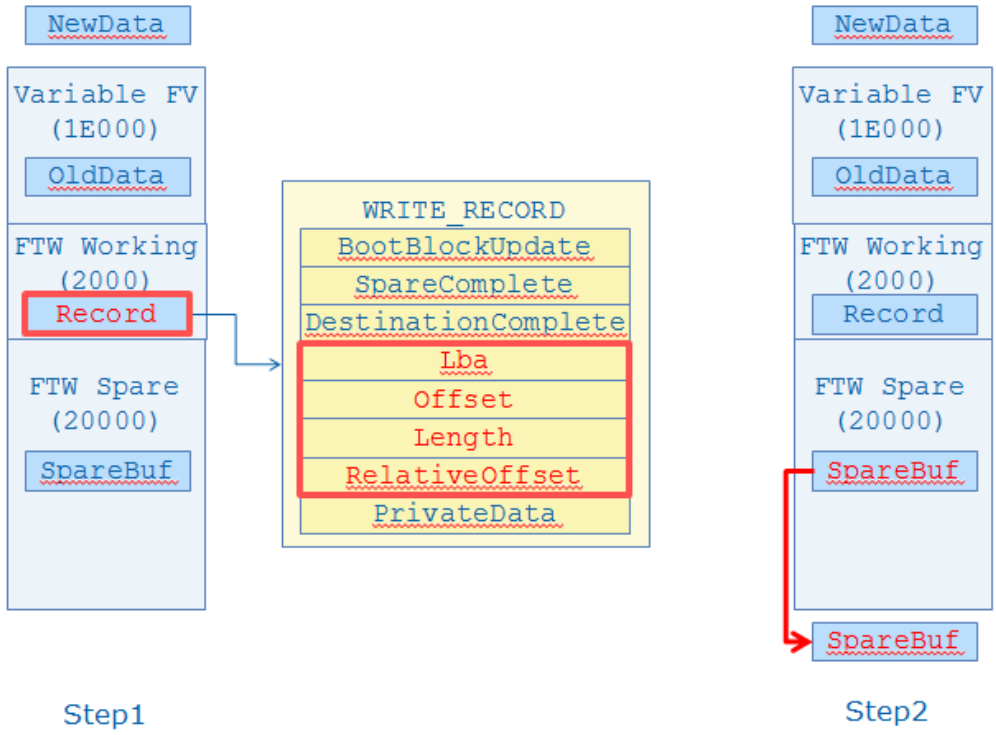

Figure 14: FTW write step 1 and 2

3) Step 3, this API writes NewData to FTW spare block, instead of Variable FV.
4) Step 4, after that, it sets SpareComplete flag in FTW working block.

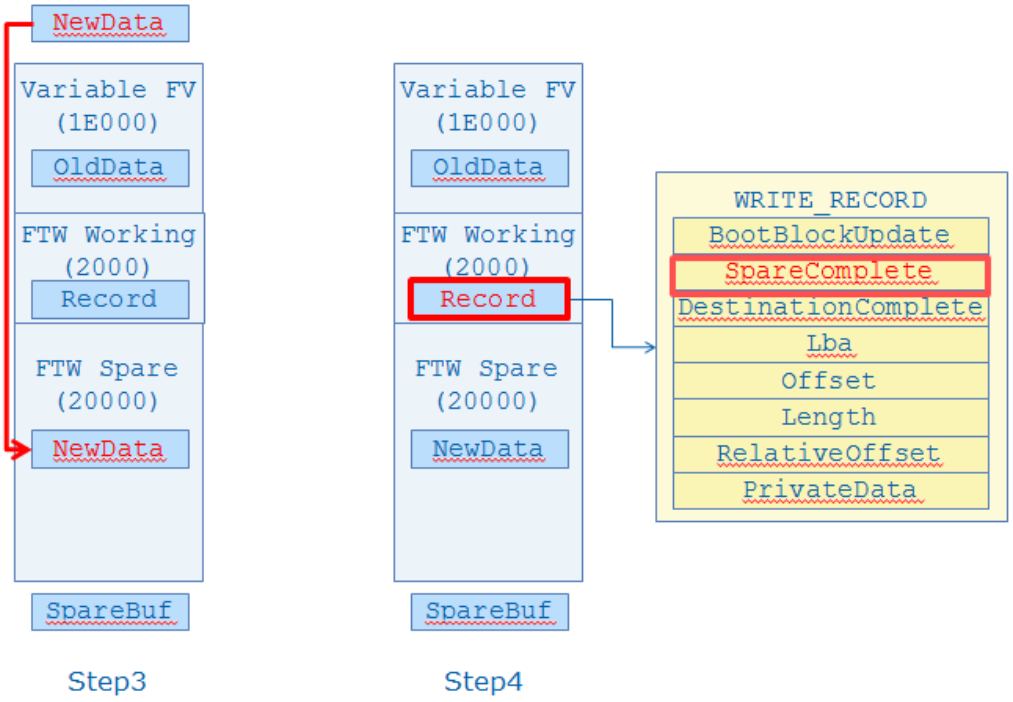

Figure 15: FTW write step 3 and 4

5) Step 5, this API writes NewData from FTW spare block to Variable FV.
6) Step 6, after that, it sets DestinationComplete flag in FTW working block.

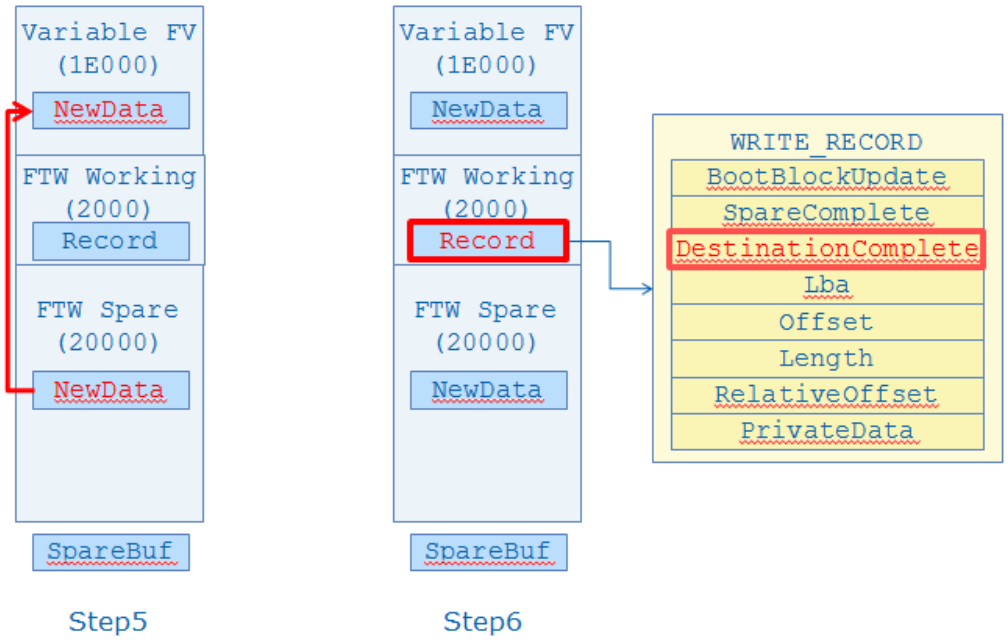

Figure 16: FTW write step 5 and 6

Step 5 is most important step, and we want to make sure it is fault tolerant. If system reset during step 5. Then SpareComplete flag is set, but DestinationComplete flag is not set. In next boot, the FTW driver will detect this situation and try to do recovery. The data is inside of FTW spare block, and all the LBA/size information is in FTW working block. So the corrupted variable region will be recovered in next boot.

7) Step 7, if it is last WRITE_RECORD associated with WRITE_HEADER, this API sets Complete flag in WRITE_HEADER.
8) Step 8, SpareBuf is restored in FTW space block. Then FTW->Write() finishes.

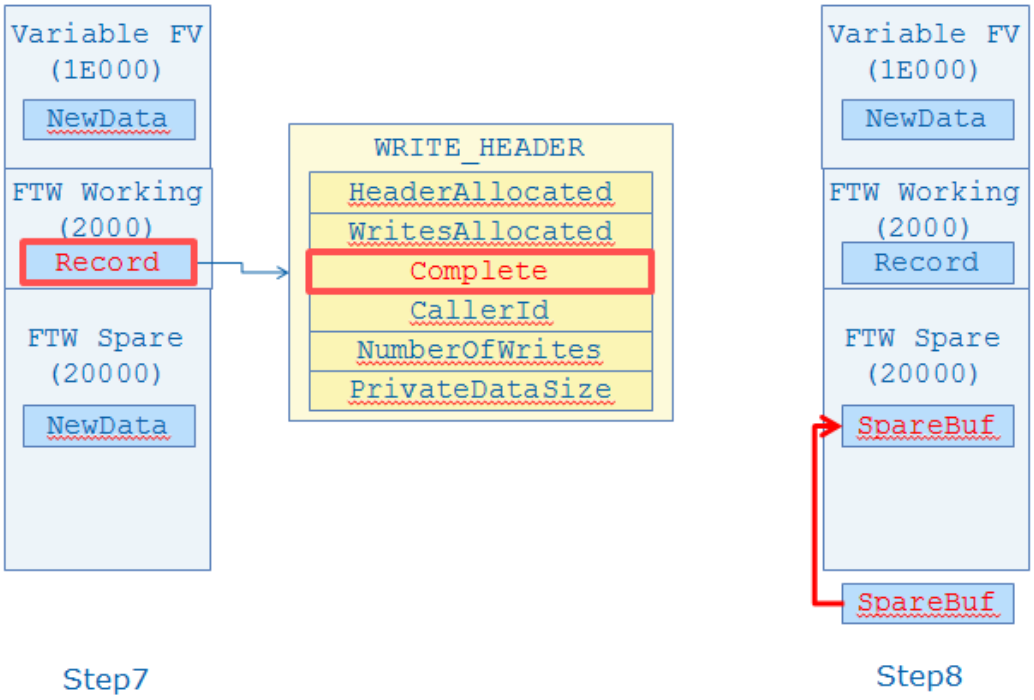

Figure 17: FTW write step 7 and 8

Variable Lock

UEFI specification defines variable attributes could be non-volatile (NV), boot services access (BS), runtime access (RT), hardware error record (HR), count based authenticated write access, time based authenticated write access (AT), and append write. However, there is a need to support read only (RO) variable for some special usage.

EDKII implements EDKII_VARIABLE_LOCK_PROTOCOL, to support mark some variable to be READ ONLY, by adopt below rule:

1) Before EndOfDxe event, EDKII_VARIABLE_LOCK_PROTOCOL.RequestToLock() API is used to submit Variable Lock request.
2) This Lock request itself is volatile. That means user need call RequestToLock() in each boot.
3) After EndOfDxe event, RequestToLock() API is closed.

Variable lock policy rule is below:

1) Before EndOfDxe event, Variable Lock does not take effect.
2) After EndOfDxe event, Variable Lock takes effect in DXE or OS runtime phase. A locked Variable cannot be deleted or updated. A locked variable cannot be created, if it does not exist before.
3) Variable Lock does not take effect in SMM phase. Inside SMM, the variable can still be updated.

A full summary below:
RO: means read only.

RW: means read write.
RWC: means read write, with reclaim feature.
RWL: means read write, with lock feature.
RWCL: means read, with reclaim and lock features.

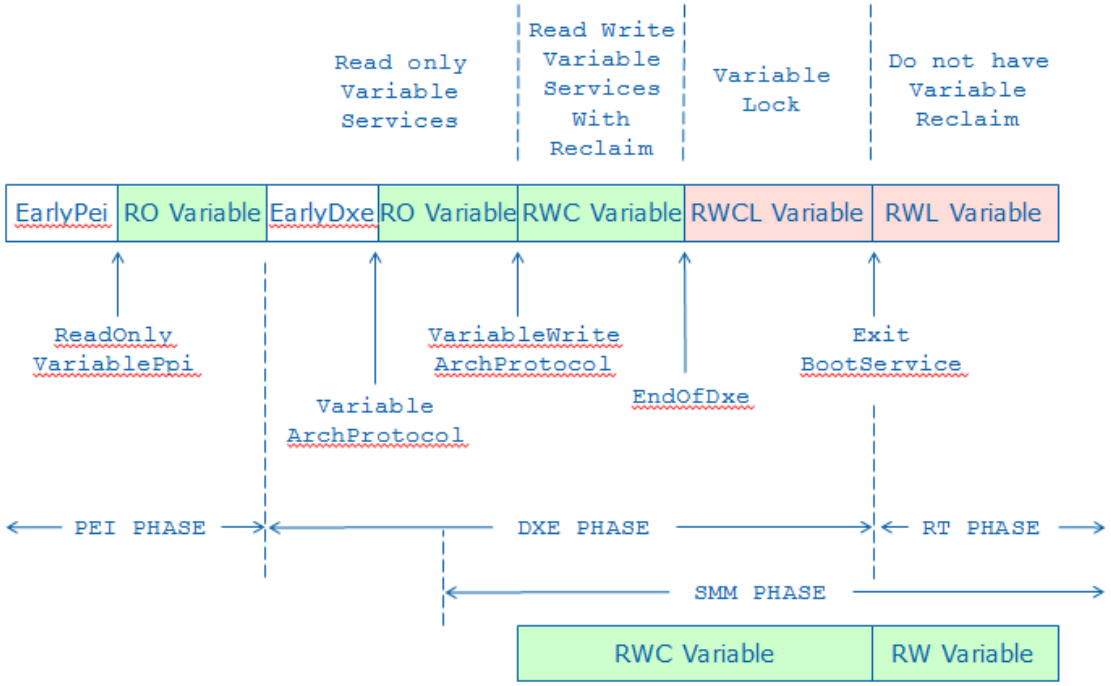

Figure 18: Variable Features in Each Phases

Summary

This section describes the variable part of SMM Authenticated Variable driver in EDKII, including variable storage format, variable update flow, variable reclaim, and variable lock.

Conclusion

SMM Authenticated Variable is an important component in UEFI secure boot. This paper describes detail work flow and data structure of SMM Authenticated Variable driver in the SecurityPkg.

Glossary

db/dbx – Image Signature Database, see UEFI specification, secure boot section.

KEK – Key Exchange Keys, see UEFI specification, secure boot section.

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

PK – Platform Key, see UEFI specification, secure boot section.

SMM – System Management Mode. x86 CPU operational mode that is isolated from and transparent to the operating system runtime

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system. Predominate interfaces are in the boot services (BS) or pre-OS. Few runtime (RT) services.

References

[SECURE1] Jacobs, Zimmer, "Open Platforms and the impacts of security technologies, initiatives, and deployment practices," Intel/Cisco whitepaper, December 2012, http://uefidk.intel.com/sites/default/files/resources/Platform_Security_Review_Intel_Cisco_White_Paper.pdf

[SECURE2] Magnus Nystrom, Martin Nicholes, Vincent Zimmer, "UEFI Networking and Pre-OS Security," in *Intel Technology Journal - UEFI Today: Boostrapping the Continuum*, Volume 15, Issue 1, pp. 80-101, October 2011, ISBN 978-1-934053-43-0, ISSN 1535-864X https://www.researchgate.net/publication/235258577_UEFI_Networking_and_Pre-OS_Security/file/9fcfd510b3ff7138f4.pdf

[SECURE3] Zimmer, Shiva Dasari (IBM), Sean Brogan (IBM), "Trusted Platforms: UEFI, PI, and TCG-based firmware," Intel/IBM whitepaper, September 2009, http://www.cs.berkeley.edu/~kubitron/courses/cs194-24-S14/hand-outs/SF09_EFIS001_UEFI_PI_TCG_White_Paper.pdf

[EDK2] UEFI Developer Kit www.tianocore.org

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.4.b www.uefi.org

[UEFI Book] Zimmer,, et al, "Beyond BIOS: Developing with the Unified Extensible Firmware Interface," 2nd edition, Intel Press, January 2011

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

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.3 www.uefi.org

Authors

Jiewen Yao (jiewen.yao@intel.com) is EDKII BIOS architect, EDKII FSP package maintainer with Software and Services Group at Intel Corporation.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with the Software and Services Group at Intel Corporation. Vincent chairs the UEFI Security and Network Sub-teams in the UEFI Forum.