White Paper

A Tour Beyond BIOS UEFI Variable Extension for Confidentiality in the EFI Developer Kit II Annex

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

November, 2016

Executive Summary

In [Variable Confidentiality], we introduced a way to add the UEFI Variable extension for the confidentiality support in a UEFI BIOS. We did not discuss the source of the key because we leave this as an implementation choice. In this paper, we provide additional information to discuss the key management to help people choose the appropriate solution to resolve the problem of key management in different scenarios.

Prerequisite

This paper assumes that audience has the basic EDKII/UEFI firmware development experience, and the basic knowledge of UEFI variable and cryptography.

Table of Contents

Overview

Introduction to the UEFI Variable Extension for Confidentiality

In [Variable Confidentiality], we introduced a way to add the UEFI Variable extension for the confidentiality support in a UEFI BIOS. The caller needs to input a key data as part of the payload to indicate that the caller has authority to read or write this variable.

```
typedef struct {
  UINT32                      KeyType;
  UINT32                      KeySize;
  //  union {
  //    UINT8                       RawData[KeySize];
  //    CHAR8                       AsciiData[KeySize];
  //    CHAR16                      UnicodeData[KeySize/2];
  //  } Data;
} EDKII_VARIABLE_KEY_DATA;
```

We added 2 more variable attributes as the EDKII extension to indicate the integrity of confidentiality of the variable.

- **`EDKII_VARIABLE_KEY_AUTHENTICATED`** is for the key-based integrity.
- **`EDKII_VARIABLE_KEY_ENCRYPTED`** is for the key-based confidentiality.

Introduction to the key selection

This solution does not handle the key storage or the source of key. The key can be retrieved from the user input, say as a password, directly. The key can be retrieved from other media, such as USB disk, TPM, or co-processor such as Intel Management Engine or a Baseboard Management Controller. The key can be derived from the user biometrics, such as fingerprint. Or the key content can be retrieved from a UEFI variable and decrypted by a root key as an implementation choice.

Summary

This section provided an overview of UEFI Variable Extension for Confidentiality.

Source of the Key – From a User

One source of key is from the platform user. The user needs to provide the identity information as a means to prove the resource access is authorized. We can have any of the below 3 ways for user authentication.

What the user knows

User password is the most used authentication factor.
Since most PC BIOS implementations have the console input and console output, it is not difficult to support the password based user authentication.

The limitation of the password based authentication is below:
1) When does the BIOS pop up a UI to let user input the password?

The PI specification defines **END_OF_DXE_EVENT**. "From SEC through the signaling of this event, all of the components should be under the authority of the platform manufacturer and not have to worry about interaction or corruption by 3rd party extensible modules." [UEFI PI Specification]

On one hand, we hope to do user authentication before the **END_OF_DXE_EVENT** because it is in the trusted state. We define the "trusted console" concept in EDKII so that the platform BIOS only connects a "trusted console" before **END_OF_DXE_EVENT.** The trusted console means the console device is integrated in the platform so that the driver of console is on the flash part instead of the option ROM.
On the other hand, if a platform has an external 3rd party video card, the video card option ROM can only be dispatched after **END_OF_DXE_EVENT**. The user is not able to input a password in this case.

2) It might break fast boot.
Some UEFI firmwares support "fast boot". For example, the console is only connected when it is nessarry. The console might not be avaiable in some special boot path, such as warm reset, or S4 resume.

If the BIOS requires the user to input the password in every boot, it might impact the user experience.

What the user has

Another problem of password is: the size and pattern. If the size is too short or the pattern is too simple, then the password is weak and easily broken. If the size if too long or the pattern is too complex, the password is easily forgetten.

In order to mitigate this, we can use one unique thing the user has as the authentication factor.

The typical example is a special USE KEY, USB token, SmartCard, NFC card, or a phone, etc.

During boot, the platform BIOS may connect these devices, and get the user key from it. Or the platform BIOS may initiate a challendge/response sequence to authentication the user.

The limitation of the solution is:
1) User must remember to carry this authentication device during boot. If a user forgets to carry this device, then he cannot boot the system.
2) User must take care of the device. If the device is stolen, then the identity is stolen.

In the real world, this solution needs to work with other authentication factors based upon a policy AND or a policy OR. For example, if a user has the Card/Token, he can use it directly. Or the user has to input the password.

What the user is

In order to resolve the inconvinence of device based authentication, we may use what a user has when he is born.

Since every person is unique, we can use the biometrics as the authenitcation factor, such as fingerprint, iris, face recoginition, voice recognition, etc.

Biometrics are a powerful way to authenitcate a user without bringing a burden to the end user. The limitation of this solution is: the platform must carry the biometics reader, such as a fingerprint reader. It increases hardware cost and may not be avaible in every platform.

[NOTE] Besides BIOS variable, the key derived from the user authentication may be used for other purposes, such as ATA HDD password feature defined by T13.org [HDD Password] or TCG OPAL password feature defined by TCG [OPAL].

Summary
This section discussed the source of key from an end user.

Source of The Key – From a Platform

Sometimes, in pre-boot phase, we need check the integrity of the platform. If and only if the platform is in a trusted state, the platform BIOS may continue to boot. The platform authentication is typically supported by a trusted platform module (TPM), which is defined by trusted computing group (TCG).

Basic TPM Concept

Platform Configuration Registers (PCR)

PCR is one of the most important features in TPM. It provides the record on the system boot flow (the code) and the system configuration (the data). The PCR is used to present the system state to a later party. The PCR consumer can check if the current boot differs from the previous boot. A typical TPM module has 24 PCR registers. The typical usage is below:

PCR	Typical Measured Object
0	Platform Firmware Code, such as PEI FV, DXE FV
1	Platform Firmware Data, such as SMBIOS table
2	3rd part OPTION ROM (UEFI driver)
3	3rd part OPTION ROM configuration
4	OS loader (UEFI Application)
5	GPT Partition, UEFI Boot Variable.
6	N/A
7	UEFI Secure Boot Configuration (PK, KEK, db, dbx)
8~23	Not used in BIOS

TPM Non-volatile (NV) Storage

TPM module includes Non-Volatile storage. This NV region can be used to save TPM internal state, or to save a user define data.

The TPM NV variable should be defined before it is used. A TPM NV variable must have an Index, and Index-specific authorization policies, which can be used to authorize reading and writing the TPM NV. The policy includes: a password/key, PCR value, locality value, read/write lock, etc.

Enhanced Authorization (EA) Commands

In previous section, we mentioned that TPM NV can combine with one or more authorization policies. These policies can be defined by the TPM2 Enhanced Authorization (EA) Commands in a policy session. For a policy session, some commands require checking something at execution time. For example, **TPM2_PolicyCounterTimer** checks the TPMS_TIME_INFO

structure, **TPM2_PolicyLocality** checks the locality information, **TPM2_PolicyPCR** checks the PCR value, and **TPM2_PolicyPassword** checks the password value.

Policy Choice

The PCR value can be used as an authentication factor for a platform. However, the PCR is updated when the BIOS update happens. In TPM1.2, the data should be unseal and seal again. TPM2.0 resolved the problem by using "*Non-Brittle PCRs*", so that "you can seal things to a PCR value approved by a particular signer instead of to a particular PCR value (although this can still be done if you wish). That is, you can have the TPM release a secret only if PCRs are in a state approved (via a digital signature) by a particular authority. … This is done via the **TPM2_PolicyAuthorize** command" [TPM2 Guide].

Besides the PCRs, TPM2 provides a more flexible policy control via EA, which can be used to create the policy AND or policy OR. For example, one object may be accessed when the PCR value matches and the Locality value matches. Another object may be accessed when the PCR value matches or the password value matches.

User may set a different policy combination based upon the usage in an UEFI BIOS.

Storage Choice

There are lots of ways to bind the TPM to a key. In UEFI firmware, we can take advantage of the options below:

1) TPM Sealing + UEFI NV

The full encrypted variable data is in UEFI NV. The Key is sealed to be a TPM object by **TPM2_Create** command with AuthPolicyHash. The output is the public area (*outPublic*), and the encrypted sensitive area (*outPrivate*). The **PCR** value may be used as AuthPolicy. The *outPublic* and *outPrivate* are saved in UEFI NV. The *outPublic* and *outPrivate* can be used to unseal to the Key by **TPM2_Load** and **TPM2_Unseal** command.

2) TPM NV with EA

The full encrypted variable data is in UEFI NV. The Key is stored in TPM NV. We need use EA command *policySession* for TPM NV access, such as **TPM2_PolicyPCR**.

NOTE: No matter if we use TPM Sealing or TPM NV with EA, we can only handle the KEY. We do not recommend to save the variable data into TPM NV, which might cause TPM NV out of resource condition because the TPM NV region is limited. We also do not recommend sealing the variable data, which might cause TPM command error because the TPM command size is limited.

If a key is bound to the platform, it means below things:

1) It is independent of a user. Everyone can read the key, as long as he owns the machine. If the machine is stolen, the system can boot and the variable content can still be read.

2) If the platform is updated, the platform binding information (such as PCR) is updated. In order to recovery the key, we might need another way to back up the key for platform recovery.

   [NOTE] In some cases, a user may save a set of common keys in TPM NV with a password as authorization. In such case, it is a user authentication action. And this action binds to a specific platform. The TPM device works as another storage device beyond the main flash part on the mother board.

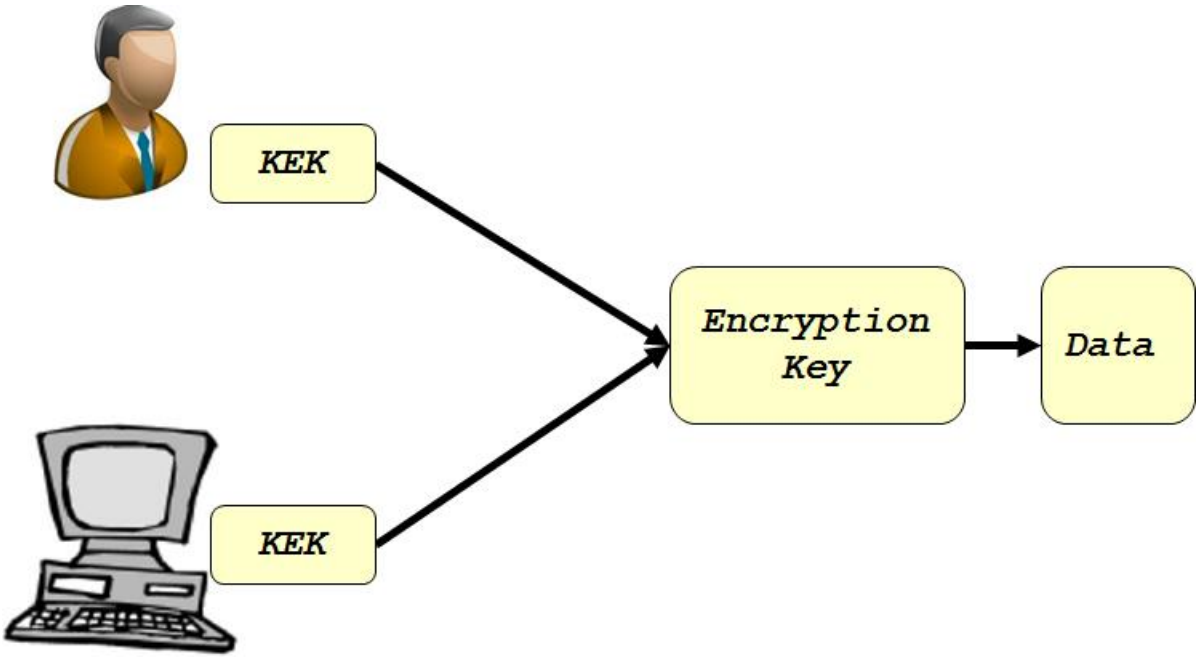
Summary

This section discussed the source of key from a platform.

Key Management

Key Encryption Key (KEK)

It is possible that we can use a user password or a platform PCR to seal the data encryption key directly. However, a better way is to use a key encryption key (KEK) to protect the encryption key and only bind KEK to the user or the platform. The data encryption key might be backed up to another source.



[TPM2 Guide] introduced 2 real examples **Microsoft BitLocker** and **IBM File and Folder Encryption** on how the TPM2 sealing is used for a KEK. More detailed information can be found at [Bitlocker] and [eCryptfs].

UEFI Key Management Service (KMS)

The UEFI specification also defines a KMS protocol to generate, store, retrieve, and manage cryptographic keys. The implementing could be an external key server over the network, or to a Hardware Security Module (HSM) attached to the system it runs on such as TPM.

If the use case just required a simple key to maintain the data confidentiality, UEFI KMS is not needed.

If the use case requires multiple keys and a rich key management feature, such as multiple users with multiple keys, then the UEFI KMS protocol is a good way to maintain these data objects.

The UEFI variable extension for confidentiality does not require UEFI KMS. It just checks the key data in the payload. It does not care if the key data is from a simple source (user password, PCR value), or a complex key management driver.

Summary
This section discussed the key management.

Summary

Integrity Consideration

UEFI Authentication Variable

UEFI authentication variable provide integrity support of a variable. The caller must provide the variable signature as payload in order to update the variable.

UEFI authentication variable is used for UEFI secure boot.
However, the solution is not a good one in pre-boot phase because it requires caller to sign the variable data. It means the caller needs to access the private key, which typically does not exist.

Variable Lock

In order to mitigate the limitation of UEFI Authenticated Variable, EDKII provides Variable Lock concept.

The platform may call EDKII_VARIABLE_LOCK_PROTOCOL.RequestToLock to lock the variable after the platform exits the platform manufacture authentication state (**END_OF_DXE_EVENT**).

Variable Lock and Unlock

In some cases, there is a need to lock a variable after **END_OF_DXE_EVENT** and update the variable with the evidence of the authorization.

Once example is [Secure MOR]. The caller may provide a key when it wants to lock the variable and provide the same key when it wants to unlock the variable.

However this is not a standard usage.

Key Based Authentication

A more generic key based authentication solution is discussed in [Variable Confidentiality].

For variable integrity, we should consider:

1) Do we just want to lock the variable to make it read-only (Such as Variable Lock)?
   a) If the variable is read-only, when it should happen? At the **END_OF_DXE_EVENT**, or after the **END_OF_DXE_EVENT**?
   b) After the variable is locked, is it OK the variable can still be updated by some special ways, such as in SMM?
2) Do we want to update the variable with authenticated information?
   a) If the asymmetric encryption algorithm is used (such as UEFI Authenticated Variable), when do we do the variable signing and where do we store the private key? When do we provision the public key? How do we revoke the public key?

b) If the symmetric encryption algorithm is used (such as Key Based Authentication), how do we handle the data encryption key? (See next section for more detailed questions)

Confidentiality Consideration

Key Based Encryption

We discussed a generic key based encryption solution in [Variable Confidentiality].

The most difficult part is the key. We should clearly define:

1) Which entity we want to trust: a user or a platform? Both or either?
2) When and where do we provision the key (register a user or a platform)? In BIOS or in OS?
3) How do we revoke the key (unregister a user or a platform)?
4) Where do we store the key? The UEFI NV, or TPM NV?
5) What is the authorization policy? If it is PCR, which PCR index we want to bind?
6) How do we update the key? Examples can include a user initiated password update, or platform BIOS upgrade by the user.
7) What is the recovery plan for the key lost? Examples can include a lost user password, or platform BIOS being damaged.
8) What is the solution for platform being stolen? Do we expect the platform boot? Is it OK that the variable data can be read? Is that OK that the variable data can be written?

Summary

This section is the summary of how we choose solution in different scenario.

Conclusion

This annex provides additional information on the key selection for UEFI variable extension for confidentiality.

Glossary

KEK – Key Encryption Key

MMIO – Memory Mapped I/O.

PCR – Platform Configuration Registers. See [TPM]

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

TCG – Trusted Computing Group.

TPM – Trusted Platform Module.

SMM – System Management mode.

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

[Bitlocker] BitLocker Drive Encryption Overview, https://technet.microsoft.com/en-us/library/cc732774(v=ws.11).aspx

[eCryptfs] Setting up eCryptfs with a TPM-managed key, https://www.ibm.com/support/knowledgecenter/linuxonibm/liaai.ecrypts/liaaiecryptfs.htm

[EDK2] UEFI Developer Kit www.tianocore.org

[HDD Password] Information Technology - AT Attachment 8 - ATA/ATAPI Command Set (ATA8-ACS). www.t13.org

[OPAL] TCG Storage Security Subsystem Class: Opal http://www.trustedcomputinggroup.org/

[Secure MOR] Microsoft Secure MOR implementation. https://msdn.microsoft.com/en-us/library/windows/hardware/mt270973(v=vs.85).aspx

[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

[STRIDE] The STRIDE Threat Model, https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx

[STRIDE2] Threat Modeling 2006 MSFT Press http://www.amazon.com/Threat-Modeling-Microsoft-Professional-Swiderski/dp/0735619913/ref=pd_bbs_sr_1?ie=UTF8&s=books&qid=1210363230&sr=8-1

[TPM2] Trusted Platform Module Library, http://www.trustedcomputinggroup.org/

[TPM2 Guide] Will Arthur, David Challener, A Practical Guide to TPM 2.0, 2015. ISBN-13: 978-1430265832

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.5 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

[Variable] Jiewen Yao, Vincent Zimmer, Star Zeng, A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDKII, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Implementing_UEFI_Authenticated_Variables_in_SMM_with_EDKII_V2.pdf

[Variable Confidentiality] Jiewen Yao, Vincent Zimmer, A Tour Beyond BIOS UEFI BIOS extension for confidentiality, https://github.com/jyao1/VariableEx/blob/master/doc/A_Tour_Beyond_BIOS_UEFI_Variable_Extension_For_Confidentiality.pdf

Authors

Jiewen Yao (jiewen.yao@intel.com) is an EDKII BIOS architect, EDKII TPM2 module maintainer, ACPI/S3 module maintainer, and FSP package owner with the Software and Services Group at Intel Corporation. Jiewen is member of UEFI Security Sub-team and PI Security Sub-team in the UEFI Forum.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer and chairs the UEFI networking and security sub-team with the Software and Services Group at Intel Corporation.