White Paper

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

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

June 1, 2016

Executive Summary

This paper introduces how we can add the UEFI Variable extension for the confidentiality support in a UEFI BIOS.

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 4

Introduction to the UEFI Authenticated Variable 4

Current limitation 4

Threat Model 5

Extension for UEFI Variable 7

Possible solutions 7

Key Encrypted Variable for Confidentiality Support 8

Key Authenticated Variable for Integrity Support 13

Assumption 14

Problem resolved 14

Problem not resolved 14

Potential Usage 16

Setup configuration management 16

Secret management 16

Conclusion 18

Glossary 19

References 20

Overview

Introduction to the UEFI 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 sets of documentation can be found on UEFI Secure boot and securing the platform [SECURE1][SECURE2][SECURE3].

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.

Anyone can read the UEFI authenticated variables, but only the holder of a private key can modify the variables. The UEFI authentication support library (https://github.com/tianocore/edk2/blob/master/SecurityPkg/Library/AuthVariableLib/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 at 7.2.1 of the UEFI Specification.

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.

Details on how EDKII implements the Authenticated Variables can be found at [AuthVariable]

Current limitation

Current UEFI defiend authentication variable solution is enough for the UEFI secure boot usage.

However, EDKII UEFI variable solution has some kown limitations on the below usages:

1) **Confidentiality**: There is no confidentiality support in current UEFI variable. We do some real usages which need the confidentiality support. For example, in order to support WIFI in the preboot phase, the BIOS may need save some passwords to the NV storage for the future connection. In order to support TLS/HTTPS or IPSec, the BIOS may need save the client side certificate with the private key to the NV storage.

2) **Integrity**: Current UEFI Authenticated Variable is a complicated solution for the integrity support. It must use the PKCS7-based signatures, including X509 certificates. A BIOS may just want a simpler solution. For example, the setup variable is protected by an administrator password, so that only an administrator is allowed to modify some setup variable fields.

Threat Model

A 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**. [STRIDE][STRIDE2]

EFI Variable Security Impact Examples

EFI Variable Name	Description / Data Type	What Can Happen to it	Security Impact Examples (from STRIDE set)
SetupVariable	Store System Configuration	At first boot, the firmware computes can the default configuration. A user may modify the system configuration in BIOS setup page based on the need.	**Tampering**: An attacker may update the configuration to meet his own need. **Denial-of-service**: An attacker may update the configuration to invalid option, or disable some features to prevent a system BIOS from booting.

EFI Variable Name	Description / Data Type	What Can Happen to it	Security Impact Examples (from STRIDE set)
WifiPassword	Store WIFI connection information	At first boot, a user may enroll the WIFI PSK(pre-shared key) in client mode or enroll the certificate in enterprise mode. In next boot, these configuration can be reused so that the user does not need enroll these info every time.	**Information Disclosure**: An attacker may read the WIFI configuration to know the password. **Denial-of-service**: An attacker may destroy the WIFI configuration prevent a system BIOS from using WIFI.

Summary

This section provided an overview of UEFI Authenticated Variable and its limitation.

Extension for UEFI Variable

Possible solutions

In order to support the confidentiality and simpler integrity usage, there might be some possible solutions:

1) Confidentiality Support:
   a) The variable consumer (variable service caller) may encrypt the variable content and save the cipher text as the variable data. Then the consumer need decrypt the variable content after it gets the variable data later. If we choose this solution, all variable consumer need aware this. It increases the burden on the secret data management.
   b) The variable producer (variable driver) may encrypt the variable content after the variable consumer calls SetVariable() service. Also, the variable producer decrypt the variable content after the variable consumer calls GetVariable() service. If we choose this solution, the data encryption and decryption is transparent to the variable consumer.
   c) The variable driver saves the content to a special variable storage. For example, TPM NV ram, or NV ram on the co-processor on the mother board. The implementation may choose a device specific way to encrypt the NV content read/write.

NOTE: In many current implementations, the variable content is saved to an SPI NOR flash region. This flash region can be read by MMIO access directly, so it is NOT acceptable to save the plain text variable data on an SPI NOR flash region directly.

2) Simple Integrity Support:
   a) The variable producer (variable driver) provides a hook for the platform defined variable checker. The checker function may do authentication check to decide if this SetVariable() can be processed or not. If we choose this solution, a platform may need be aware this and define its own authentication check.
   b) The variable producer (variable driver) may define a simpler key-based authentication besides UEFI defined count-based authentication or time-based authentication. If we choose this solution, the variable consumer just need a way to get the key and input it as a proof to modify the variable content. The integity check is integrated to the variable driver.
   c) The variable driver saves the content to a special variable storage. For example, TPM NV ram, or NV ram on the co-processor on the mother board. The implementation may choose a device specific way to authenticate the NV content update.

1.A) and 2.A) can be implemented with the current EDKII variable solution. For example, EDKII CryptoPkg provides a set of encryption/decryption API for 1.A). EDKII MdeModulePkg defines the VariableCheck protocol and library API for 2.A).

The other NV storage management in 1.C) and 2.C) is a special variable implementation. It might be platform and device specific. It is not a generic solution, so it is not discussed here.

In this whitepaper, we will introduce 1.B) and 2.B) as the generic variable extension. As an implementation choice, we use the key to provide both variable encryption and simple authentication.

Key Encrypted Variable for Confidentiality Support

UEFI specification does not define the special attribute for the key based authentication or encryption, so we added 2 more attributes as the EDKII extension:

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

In order to extend this capability without impact the UEFI defined API, we defined 2 new protocols and 1 new PPI for the attributes extension.

The DXE version protocol is at
https://github.com/jyao1/VariableEx/blob/master/VariableExPkg/Include/Protocol/VariableEx.h

The SMM version protocol is at
https://github.com/jyao1/VariableEx/blob/master/VariableExPkg/Include/Protocol/SmmVariableEx.h

The PEI version PPI is at
https://github.com/jyao1/VariableEx/blob/master/VariableExPkg/Include/Ppi/ReadOnlyVariable2Ex.h

Below is the protocol definition:

```
struct _EDKII_VARIABLE_EX_PROTOCOL {
  EDKII_GET_VARIABLE_EX            GetVariableEx;
  EDKII_GET_NEXT_VARIABLE_NAME_EX  GetNextVariableNameEx;
  EDKII_SET_VARIABLE_EX            SetVariableEx;
  EFI_QUERY_VARIABLE_INFO          QueryVariableInfo;
};

typedef
EFI_STATUS
(EFIAPI *EDKII_GET_VARIABLE_EX)(
  IN     CHAR16                     *VariableName,
  IN     EFI_GUID                   *VendorGuid,
  OUT    UINT32                     *Attributes,    OPTIONAL
  IN OUT UINT8                      *AttributesEx, // ← New attribute extension
  IN OUT UINTN                      *DataSize,
  OUT    VOID                       *Data           OPTIONAL
  );

typedef
EFI_STATUS
(EFIAPI *EDKII_GET_NEXT_VARIABLE_NAME_EX)(
  IN OUT UINTN                   *VariableNameSize,
  IN OUT CHAR16                  *VariableName,
  IN OUT EFI_GUID                *VendorGuid,
```

```
  IN OUT UINT32                   *Attributes,
  IN OUT UINT8                    *AttributesEx // ← New attribute extension
  );

typedef
EFI_STATUS
(EFIAPI *EDKII_SET_VARIABLE_EX)(
  IN  CHAR16                       *VariableName,
  IN  EFI_GUID                     *VendorGuid,
  IN  UINT32                       Attributes,
  IN  UINT8                        AttributesEx, // ← New attribute extension
  IN  UINTN                        DataSize,
  IN  VOID                         *Data
  );
```

The RED line is the additional parameters comparing with the ones in UEFI variable APIs.

All new APIs have AttributeEx parameter to carry the attributes extension: **EDKII_VARIABLE_KEY_AUTHENTICATED** or **EDKII_VARIABLE_KEY_ENCRYPTED.**

If a variable consumer wants to prevent a variable from being read, it may call SetVariableEx() API with attributes **EDKII_VARIABLE_KEY_ENCRYPTED.**

This attribute implies a new format of Data as the input parameter. See figure 1 – SetVariableEx() Data as the input parameter.

The variable Data includes an EDKII_VARIABLE_PASSWOR_DATA and variable user data. The EDKII_VARIABLE_KEY_DATA defines the key type and size, followed by a key. The EDKII_VARIABLE_KEY_DATA is not real user data, and it will not be returned by GetVariableEx() API. The variable user data is the real user data, and is returned by GetVariableEx() API.

This key encrypted variable design is similar as the current UEFI defined authenticated variable solution. The left 2 rectangles of the figure 1 show the input data format of the non-authenticated variable and UEFI defined authenticated variable. The right 2 rectangles of figure 1 show the input data format of the key authenticated variable and key encrypted with **EDKII_VARIABLE_KEY_AUTHENTICATED** or **EDKII_VARIABLE_KEY_ENCRYPTED.**

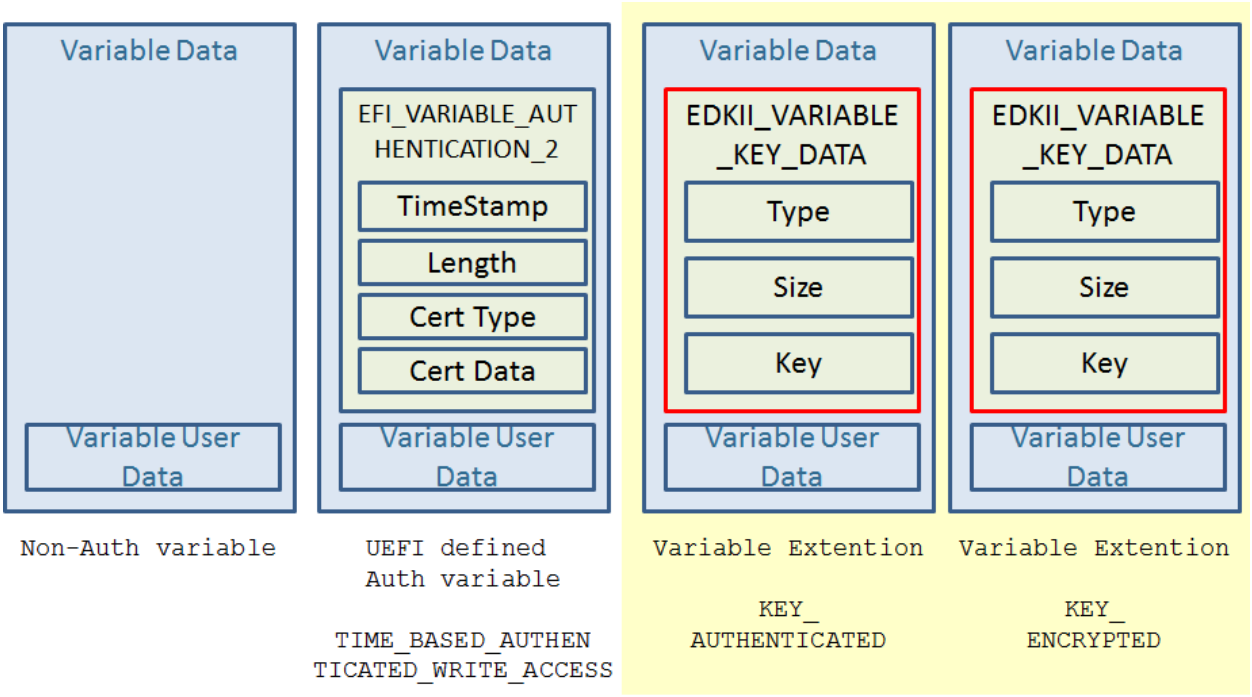

Figure 1 – SetVariableEx() Data as the input parameter

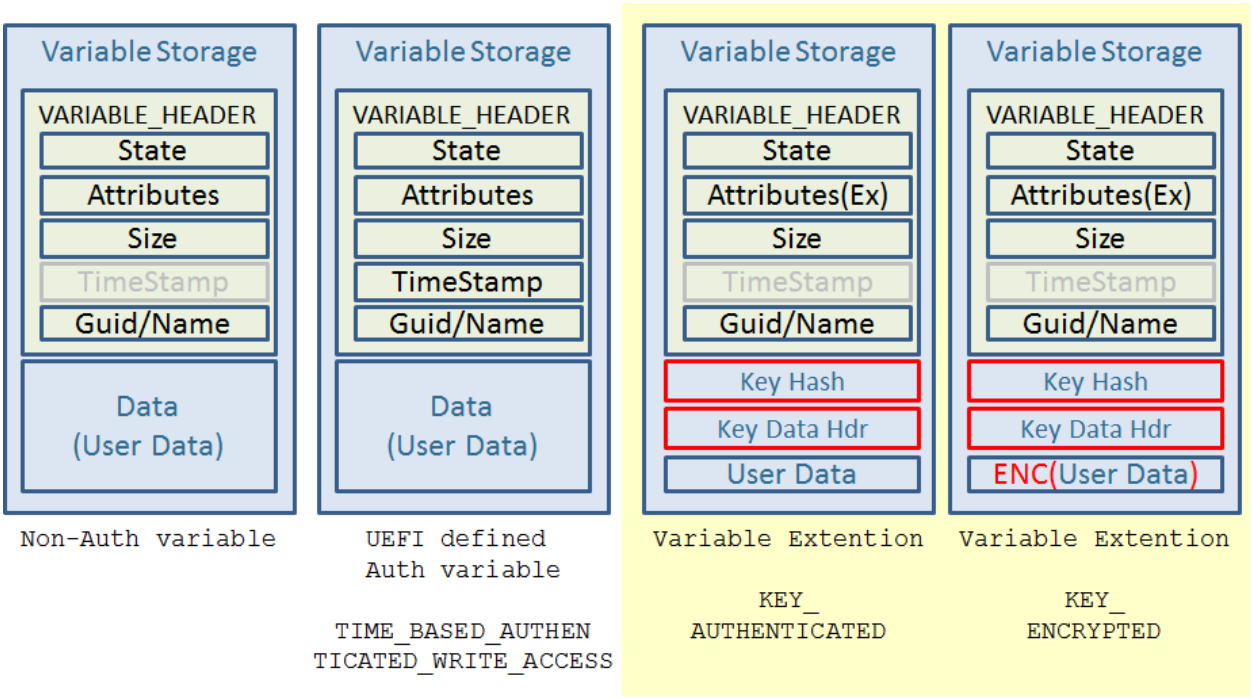

Figure 2 – Variable Storage

The variable service provider (variable driver
https://github.com/jyao1/VariableEx/blob/master/VariableExPkg/Universal/Variable/RuntimeDx

e/Variable.c) checks this AttributesEx and goes to a different path for the key encrypted variable. See figure 3 – SetVariableEx() flow for the key authenticated/encrypted variable.

1) If the key encrypted variable already exists, SetVariableEx() will get the original variable data storage. The new variable storage format is defined at https://github.com/jyao1/VariableEx/blob/master/VariableExPkg/Include/Guid/VariableFormatEx.h . See figure 2 – Variable Storage. The left 2 rectangles of the figure 2 show the variable storage of the non-authenticated variable and UEFI defined authenticated variable. The right 2 rectangles of figure 2 show the variable storage of the key authenticated and key encrypted variable.
2) SetVariableEx API generates a new hash value based upon the input key and the original salt value in the variable storage. Then it compares the new hash value with the original hash value in the variable storage. If the hash is matched, the authentication passes. Or the authentication fails and this API returns EFI_SECURITY_VIOLATION.
3) After the authentication, SetVariableEx API generates a new salt and a new hash value.
4) If the variable is an encrypted variable, SetVariableEx API will encrypt the data and save the salt, hash and encrypted data together into variable storage region.

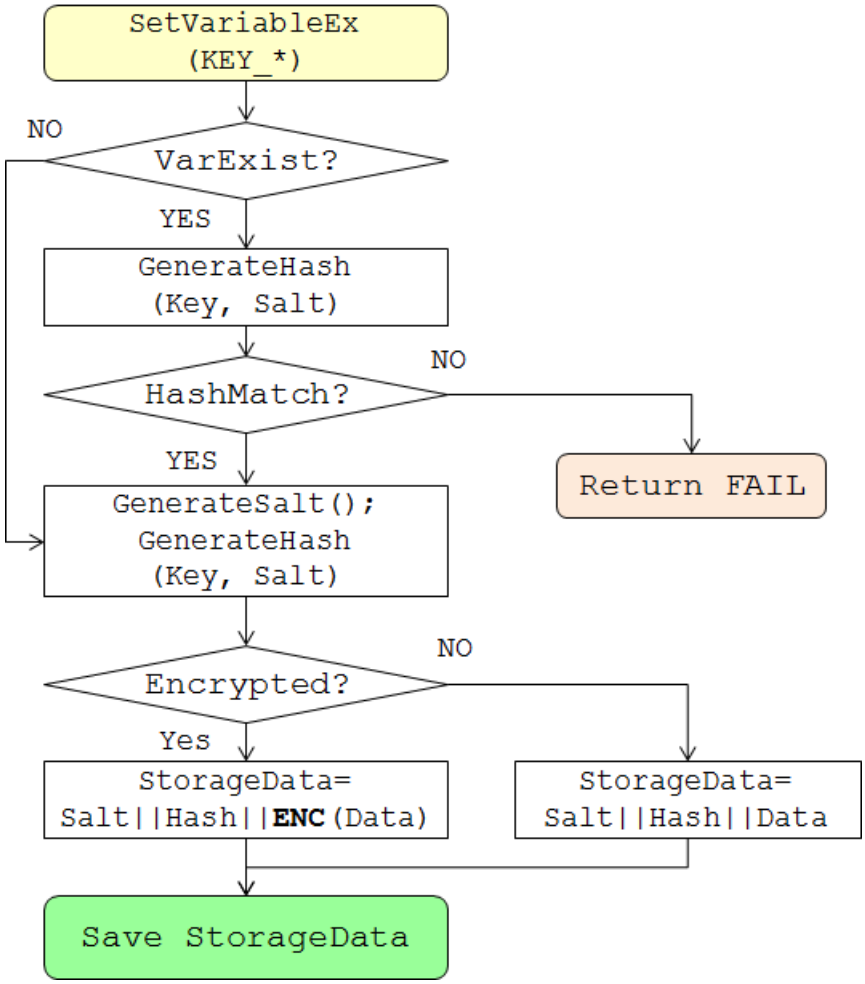

Figure 3 – SetVariableEx() flow for key authenticated/encrypted variable

After a variable consumer encrypts a variable, it may call GetVariableEx() API to get the data later. For the variable with attributes **EDKII_VARIABLE_KEYs_PROTECTED**, the caller must input the valid EDKII_VARIABLE_KEY_DATA. See below figure 4.
The reason is that the variable driver need the key to decrypt the variable data for the caller. On output, the variable driver just returns the plain text user variable data. See below figure 5.

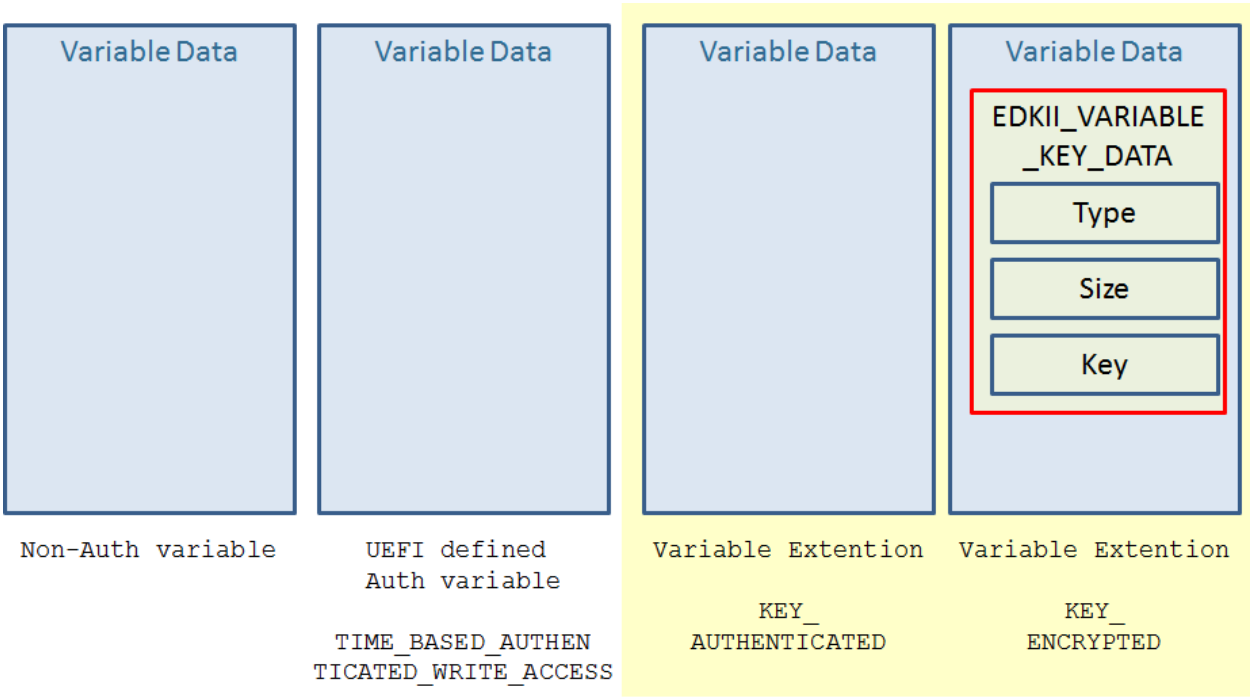

Figure 4 – GetVariableEx() Data as the input parameter

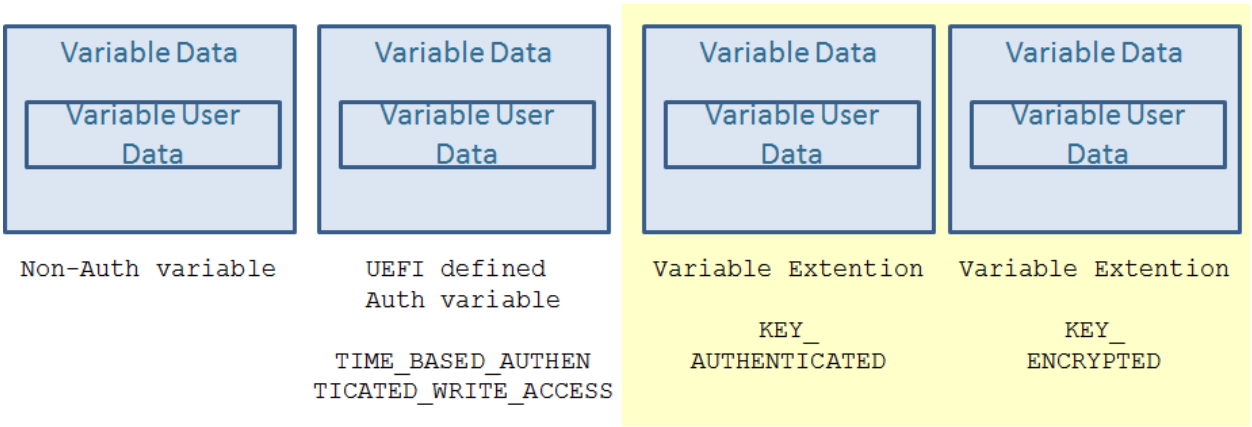

Figure 5 – GetVariableEx() Data as the output parameter

The flow of GetVariableEx() for key authenticated/encrypted variable is shown in figure 6.

1) If the key encrypted variable already exists, GetVariableEx() will get the original variable data storage.

2) GetVariableEx API generates a hash value based upon the input key and the original salt value in the variable storage. Then it compares this hash value with the original hash value in the variable storage. If the hash is matched, the authentication passes. Or the authentication fails and this API returns EFI_SECURITY_VIOLATION.
3) After the authentication, GetVariableEx API get the encrypted data and decrypt it with key. The plain text user data is returned.

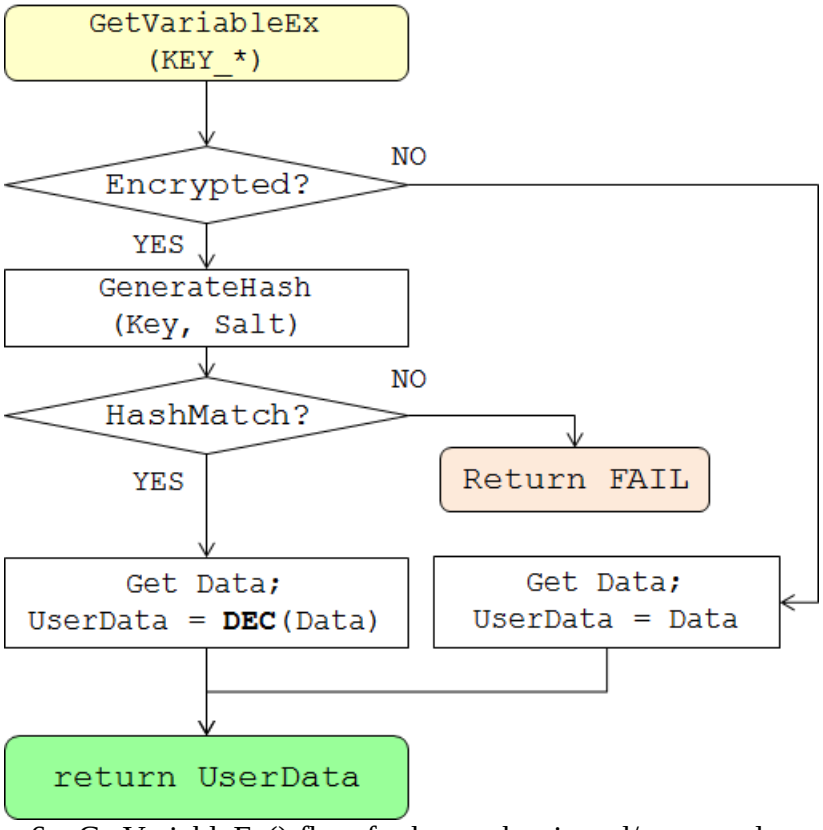

Figure 6 – GetVariableEx() flow for key authenticated/encrypted variable

Key Authenticated Variable for Integrity Support

The flow of the key authenticated variable is very similar as the flow of the key encrypted variable. A variable consumer may use `EDKII_VARIABLE_KEY_AUTHENTICATED` to indicate it is a key authenticated variable.

1) **SetVariableEx() API**: The data format is same - an EDKII_VARIABLE_KEY_DATA and variable user data. See figure 1 and 3.
2) **Variable storage**: A key authenticated variable saves plain text on the variable storage, while a key encrypted variable saves cypher text on the variable storage. See figure 2.
3) **GetVariableEx() API**: A key authenticated variable does not need the input data, while a key encrypted variable need EDKII_VARIABLE_KEY_DATA as the input data. The reason is that a key authenticated variable saves the plain text and returns the plain text, so there is

no need to provide a key to get the data. A key encrypted variable must have a key to decrypt the cypher text in order to return the plain text to caller. See figure 4, 5 and 6.

Assumption

- The key based encrypted variable and authentication variable solution assumes that there is a secure way to retrieve a key during the BIOS boot phase.

- If the caller of SetVariableEx/GetVariableEx can have a secure way to get key, `EDKII_VARIABLE_KEY_ENCRYPTED` can be used to maintain both confidentiality and integrity. For example, the WIFI password storage.

- If only the caller of SetVariableEx can have a secure way to get key, but the caller of GetVariableEx cannot have a secure way to get key, only `EDKII_VARIABLE_KEY_AUTHENTICATED` can be used to maintain the integrity. For example, the Setup Variable can only be updated if an admin password is provided. And this Setup Variable need to be read in the early PEI phase, when there is no UI to let user input password.

Problem resolved

- The main purpose of the key encrypted variable is to provide a generic, secure, non-volatile storage in a BIOS.

- The main purpose of the key authenticated variable is to provide a generic, simpler authenticated variable solution in a BIOS. It is not designed to replace the current EFI Authenticated variable solution. It just provides another simpler way to do that.

Problem not resolved

- This solution does not provide rich key management functions. The whole key management can be implemented as the other UEFI interfaces, such as EFI_KMS_PROTOCOL.

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

  NOTE: The source of key depends upon which entity we want trust. For example, if we want to trust the platform, the key can be sealed to TPM and unsealed when PCR match. But this does not protect the data when the platform is stolen. When a platform BIOS is updated, we need another way to get the key.
  If we want to trust the user, the key could be the user password (what a user knows). If the user feels too much burden to input a password in every boot, he or she may choose to save password to a USB key and let BIOS read the password from USB key directly (what a user

has). Or the user may just scan the fingerprint (what a user is) to derive the key, if the platform has fingerprint connected.

- This solution does not handle the data recovery or migration. For example, if the user loses the key, this variable will never be read or written. The variable service may clear all unknown variables when the variable region is out of resource. But there is no way to access the variable any more from the variable API without a valid key.

Summary

This section introduces the UEFI variable extension to support the key based encrypted variable and authenticated variable.

Potential Usage

Setup configuration management

A BIOS may have a setup variable to record the configuration of BIOS, such as an ATA device is in AHCI mode or RAID mode, FastBoot feature is enabled or disabled. This configuration might be important so that only an administrator is allowed to update them. A normal user is not allowed to update them.

Current UEFI authenticated variable solution requires an X509 certificate for a variable consumer. It might be too much burden if we want to create a certificate for each users. It is a nature way to create a password for each users.

When an administrator registers a password in setup UI, the BIOS setup driver may get setup variable, delete it and create a new setup variable with an **`EDKII_VARIABLE_KEY_AUTHENTICATED`** flag and the password as a key.

Then at next time, only an administrator can modify the setup configuration because only he or she knows the administrator password. The malicious person (eve) cannot modify the setup configuration because he or she does not know the password. See figure 7.

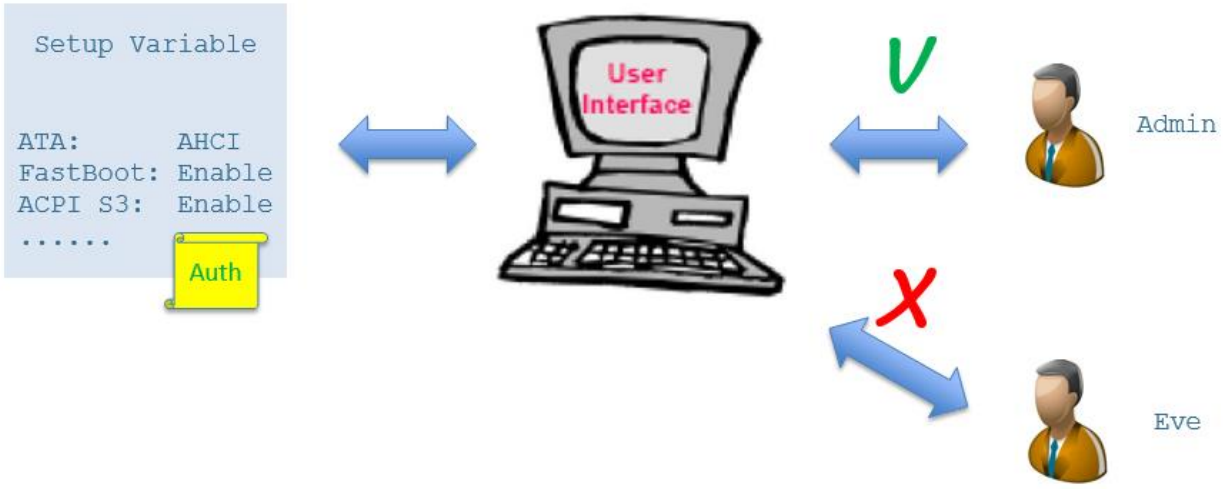

Figure 7 – Setup configuration management

Secret management

Assuming we have a root key, this root key can be used to save the other key or secret data to the BIOS NV storage region with an **`EDKII_VARIABLE_KEY_ENCRYPTED`** flag.

See below figure 8. The RootKey is used to encrypt the SubKey1 and the SubKey2. The SubKey1 is used to encrypt the SubKey3. Every SubKey 1, 2, 3 is used to encrypt the Secret 1, 2 and 3.

For example, the WIFI password or certificate can be treated as a secret. The WIFI driver may need encrypt them by using a WifiDriver key. The WIFI driver key can be encrypted by the RootKey.

This is treated as a key hierarchy in BIOS. The key encrypted the variable solution provides a secure storage for these keys and secrets.

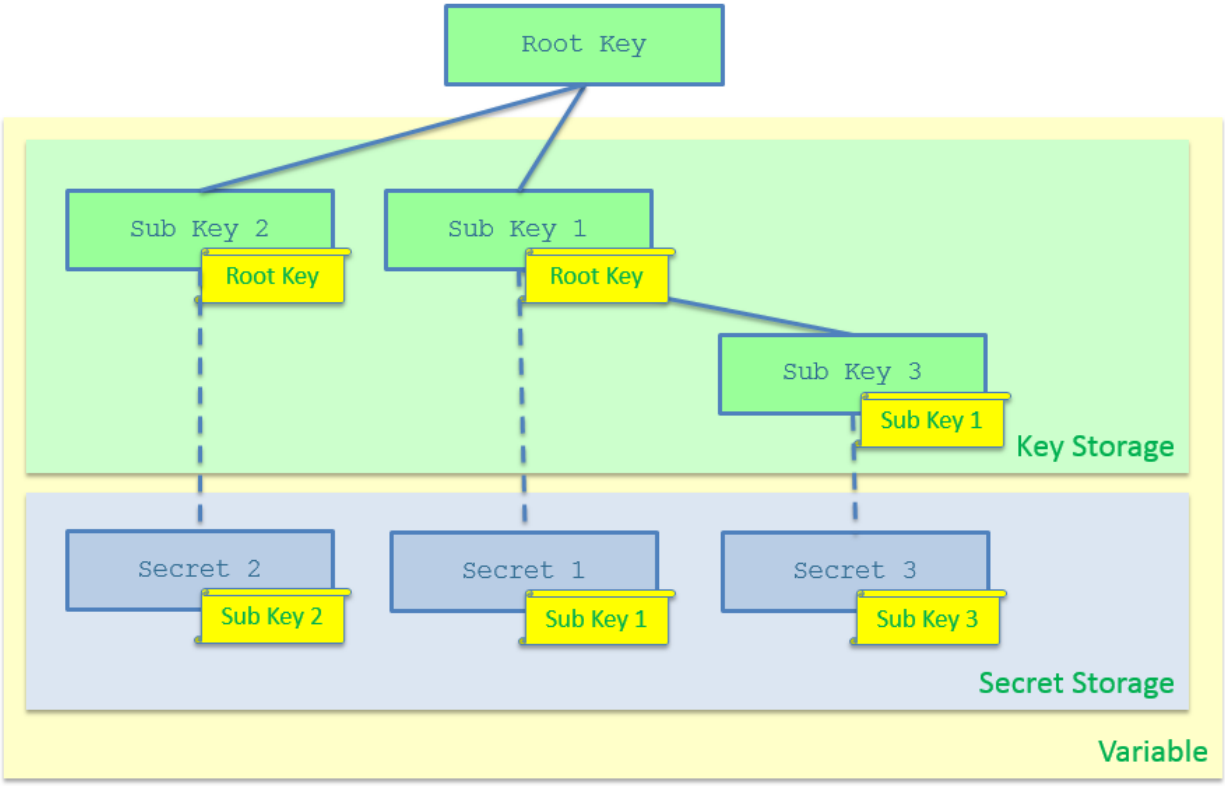

Figure 8 – Secret management

Summary

This section introduces the potential usage of the key based authentication and encryption.

Conclusion

Current UEFI specification does not define a standard way for the variable confidentiality, so that different OEMs may choose different solutions to achieve this. This paper discussed a generic and hardware-independent way to implement a key encrypted variable solution. This paper also discussed a key authenticated variable to provide a simpler authentication for UEFI variable.

Glossary

MMIO – Memory Mapped I/O.

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

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

[EDK2] UEFI Developer Kit www.tianocore.org

[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

[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

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.