UEFI Variable Policy

Specification proposal, version 1.0

Microsoft Core UEFI Team

January 28, 2020

# Summary

UEFI Variable Policy spec aims to describe the DXE protocol interface which allows enforcing certain rules on certain UEFI variables. The protocol allows communication with the Variable Policy Engine which performs the policy enforcement.

The Variable Policy is comprised of a set of policy entries which describe, per UEFI variable (identified by namespace GUID and variable name) the following rules:

* Required variable attributes
* Prohibited variable attributes
* Minimum variable size
* Maximum variable size
* Locking:
  + Locking “immediately”
  + Locking on creation
  + Locking based on a state of another variable

The spec assumes that the Variable Policy Engine runs in a trusted enclave, potentially off the main CPU that runs UEFI. For that reason, it is assumed that the Variable Policy Engine has no concept of UEFI events, and that the communication from the DXE driver to the trusted enclave is proprietary.

At power-on, the Variable Policy Engine is:

* Enabled – present policy entries are evaluated on variable access calls.
* Unlocked – new policy entries can be registered.

Policy is expected to be clear on power-on. Policy is volatile and not preserved across system reset.

# DXE Protocol functions

### DisableVariablePolicy

Function prototype:

EFI\_STATUS

DisableVariablePolicy(

VOID

)

DisableVariablePolicy call disables the Variable Policy Engine, so that the present policy entries are no longer taken into account on variable access calls. This call effectively turns off the variable policy verification for this boot. This also disables UEFI Authenticated Variable protections including Secure Boot. DisableVariablePolicy can only be called once during boot. If called more than once, it will return EFI\_ALREADY\_STARTED. Note, this process is irreversible until the next system reset – there is no “EnablePolicy” protocol function.

### IsVariablePolicyEnabled

Function prototype:

EFI\_STATUS

IsVariablePolicyEnabled(

OUT BOOLEAN \*State

)

IsVariablePolicyEnabled accepts a pointer to a Boolean in which it will store TRUE if Variable Policy Engine is enabled, or FALSE if Variable Policy Engine is disabled. The function returns EFI\_SUCCESS.

### RegisterVariablePolicy

Function prototype:

EFI\_STATUS

RegisterVariablePolicy(

IN CONST VARIABLE\_POLICY\_ENTRY \*PolicyEntry

)

RegisterVariablePolicy call accepts a pointer to a policy entry structure and returns the status of policy registration. If the Variable Policy Engine is not locked and the policy structures are valid, the function will return EFI\_SUCCESS. If the Variable Policy Engine is locked, RegisterVariablePolicy call will return EFI\_WRITE\_PROTECTED and will not register the policy entry. Bulk registration is not supported at this time due to the requirements around error handling on each policy registration.

Upon successful registration of a policy entry, Variable Policy Engine will then evaluate this entry on subsequent variable access calls (as long as Variable Policy Engine hasn’t been disabled).

### DumpVariablePolicy

Function prototype:

EFI\_STATUS

DumpVariablePolicy(

OUT UINT8 \*Policy,

IN OUT UINT32 \*Size

)

DumpVariablePolicy call accepts a pointer to a buffer and a pointer to the size of the buffer as parameters and returns the status of placing the policy into the buffer. On first call to DumpVariablePolicy one should pass NULL as the buffer and a pointer to 0 as the Size variable and DumpVariablePolicy will return EFI\_BUFFER\_TOO\_SMALL and will populate the Size parameter with the size of the needed buffer to store the policy. This way, the caller can allocate the buffer of correct size and call DumpVariablePolicy again. The function will populate the buffer with policy and return EFI\_SUCCESS.

### LockVariablePolicy

Function prototype:

EFI\_STATUS

LockVariablePolicy(

VOID

)

LockVariablePolicy locks the Variable Policy Engine, i.e. prevents any new policy entries from getting registered in this boot (RegisterVariablePolicy calls will fail with EFI\_WRITE\_PROTECTED status code returned).

# Policy structure

The structure below is meant for the DXE protocol calling interface, when communicating to the Variable Policy Engine, thus the pragma pack directive. How these policies are stored in memory is up to the implementation.

#pragma pack(1)

typedef struct {

UINT32 Version;

UINT16 Size;

UINT16 OffsetToName;

EFI\_GUID Namespace;

UINT32 MinSize;

UINT32 MaxSize;

UINT32 AttributesMustHave;

UINT32 AttributesCantHave;

UINT8 LockPolicyType;

UINT8 Padding[3];

// UINT8 LockPolicy[]; // Variable Length Field

// CHAR16 Name[] // Variable Length Field

} VARIABLE\_POLICY\_ENTRY;

The struct VARIABLE\_POLICY\_ENTRY above describes the layout for a policy entry. The first element, Size, is the size of the policy entry, then followed by OffsetToName – the number of bytes from the beginning of the struct to the name of the UEFI variable targeted by the policy entry. The name can contain wildcards to match more than one variable, more on this in the Wildcards section. The rest of the struct elements are self-explanatory.

#define VARIABLE\_POLICY\_TYPE\_NO\_LOCK 0

#define VARIABLE\_POLICY\_TYPE\_LOCK\_NOW 1

#define VARIABLE\_POLICY\_TYPE\_LOCK\_ON\_CREATE 2

#define VARIABLE\_POLICY\_TYPE\_LOCK\_ON\_VAR\_STATE 3

LockPolicyType can have the following values:

* 0 – NoLock – means that no variable locking is performed. However, the attribute and size constraints are still enforced. LockPolicy field is size 0.
* 1 – LockNow – means that the variable starts being locked immediately after policy entry registration. If the variable doesn’t exist at this point, being LockedNow means it cannot be created on this boot. LockPolicy field is size 0.
* 2 – LockOnCreate – means that the variable starts being locked after it is created. This allows for variable creation and protection after LockVariablePolicy() function has been called. The LockPolicy field is size 0.
* 3 – LockOnVarState – means that the Variable Policy Engine will examine the state/contents of another variable to determine if the variable referenced in the policy entry is locked.

typedef struct {

EFI\_GUID Namespace;

UINT8 Value;

UINT8 Padding;

// CHAR16 Name[]; // Variable Length Field

} VARIABLE\_LOCK\_ON\_VAR\_STATE\_POLICY;

If LockPolicyType is LockOnVarState, then the final element in the policy entry struct is of type VARIABLE\_LOCK\_ON\_VAR\_STATE\_POLICY, which lists the namespace GUID, name (no wildcards here), and value of the variable which state determines the locking of the variable referenced in the policy entry. The “locking” variable must be 1 byte in terms of payload size. If the Referenced variable contents match the Value of the VARIABLE\_LOCK\_ON\_VAR\_STATE\_POLICY structure, the lock will be considered active and the target variable will be locked. If the Reference variable does not exist (ie. returns EFI\_NOT\_FOUND), this policy will be considered inactive.

# Variable Name Wildcards

Two types of wildcards can be used in the UEFI variable name field in a policy entry:

1. If the Name is a zero-length array (easily checked by comparing fields “Size” and “OffsetToName” – if they’re the same, then the Name is zero-length), then all variables in the namespace specified by the provided GUID are targeted by the policy entry.
2. Character “#” in the Name corresponds to one numeric character (0-9, A-F, a-f). For example, string “Boot####” in the Name field of the policy entry will make it so that the policy entry will target variables named “Boot0001”, “Boot0002”, etc.

Given the above two types of wildcards, one variable can be targeted by more than one policy entry, thus there is a need to establish the precedence rule: a more specific match is applied. When a variable access operation is performed, Variable Policy Engine should first check the variable being accessed against the policy entries without wildcards, then with 1 wildcard, then with 2 wildcards, etc., followed in the end by policy entries that match the whole namespace. One can still imagine a situation where two policy entries with the same number of wildcards match the same variable – for example, policy entries with Names “Boot00##” and “Boot##01” will both match variable “Boot0001”. Such situation can (and should) be avoided by designing mutually exclusive Name strings with wildcards, however, if it occurs, then the policy entry that was registered first will be used. After the most specific match is selected, all other policies are ignored.

# Use cases

1. Variables containing values of the setup options exposed via UEFI menu (setup variables). These would be locked based on a state of another variable, “ReadyToBoot”, which would be set to 1 at the ReadyToBoot event. Thus, the policy for the setup variables would be of type LockOnVarState, with the “ReadyToBoot” listed as the name of the variable, appropriate GUID listed as the namespace, and 1 as value. Entry into the trusted UEFI menu app doesn’t signal ReadyToBoot, but booting to any device does, and the setup variables are write-protected. The “ReadyToBoot” variable would need to be locked-on-create. *(THIS IS ESSENTIALLY LOCK ON EVENT, BUT SINCE THE POLICY ENGINE IS NOT IN THE UEFI ENVIRONMENT VARIABLES ARE USED)*

For example, “AllowPXEBoot” variable locked by “ReadyToBoot” variable. The green-shaded part of the table describes the fields in the VARIABLE\_LOCK\_ON\_VAR\_STATE\_POLICY structure.

|  |  |
| --- | --- |
| Size | … |
| OffsetToName | … |
| NameSpace | … |
| MinSize | … |
| MaxSize | … |
| AttributesMustHave | … |
| AttributesCantHave | … |
| LockPolicyType | 0x3 |
| Namespace | … |
| Value | 1 |
| Name | “ReadyToBoot” |
| //Name | “AllowPXEBoot” |

1. Manufacturing Variable Provisioning Data (VPD) is stored in variables and is created while in Manufacturing (MFG) Mode. In MFG Mode Variable Policy Engine is disabled, thus these VPD variables can be created. These variables are locked with lock policy type LockNow, so that these variables can’t be tampered with in Customer Mode. To overwrite or clear VPD, the device would need to MFG mode, which is standard practice for refurbishing/remanufacturing scenarios.

Example: “DisplayPanelCalibration” variable

|  |  |
| --- | --- |
| Size | … |
| OffsetToName | … |
| NameSpace | … |
| MinSize | … |
| MaxSize | … |
| AttributesMustHave | … |
| AttributesCantHave | … |
| LockPolicyType | 0x1 |
| // Name | “DisplayPanelCalibration” |

1. Bluetooth pre-pairing variables are locked-on-create because these get created by an OS application when Variable Policy is in effect.

Example: “KeyboardBTPairing” variable

|  |  |
| --- | --- |
| Size | … |
| OffsetToName | … |
| NameSpace | … |
| MinSize | … |
| MaxSize | … |
| AttributesMustHave | … |
| AttributesCantHave | … |
| LockPolicyType | 0x2 |
| // Name | “KeyboardBTPairing” |

1. Software based variable policy.

Example: “Boot####” variables (a name string with wildcards that will match variables “Boot0000” to “BootFFFF”) locked by “LockBootOrder” variable.

|  |  |
| --- | --- |
| Size | … |
| OffsetToName | … |
| NameSpace | … |
| MinSize | … |
| MaxSize | … |
| AttributesMustHave | … |
| AttributesCantHave | … |
| LockPolicyType | 0x3 |
| Namespace | … |
| Value | 1 |
| Name | “LockBootOrder” |
| //Name | “Boot####” |