Advanced x86:
BIOS and System Management Mode Internals
UEFI SecureBoot

Xeno Kovah && Corey Kallenberg

LegbaCore, LLC

All materials are licensed under a Creative Commons “Share Alike” license.

http://creativecommons.org/licenses/by-sa/3.0/

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

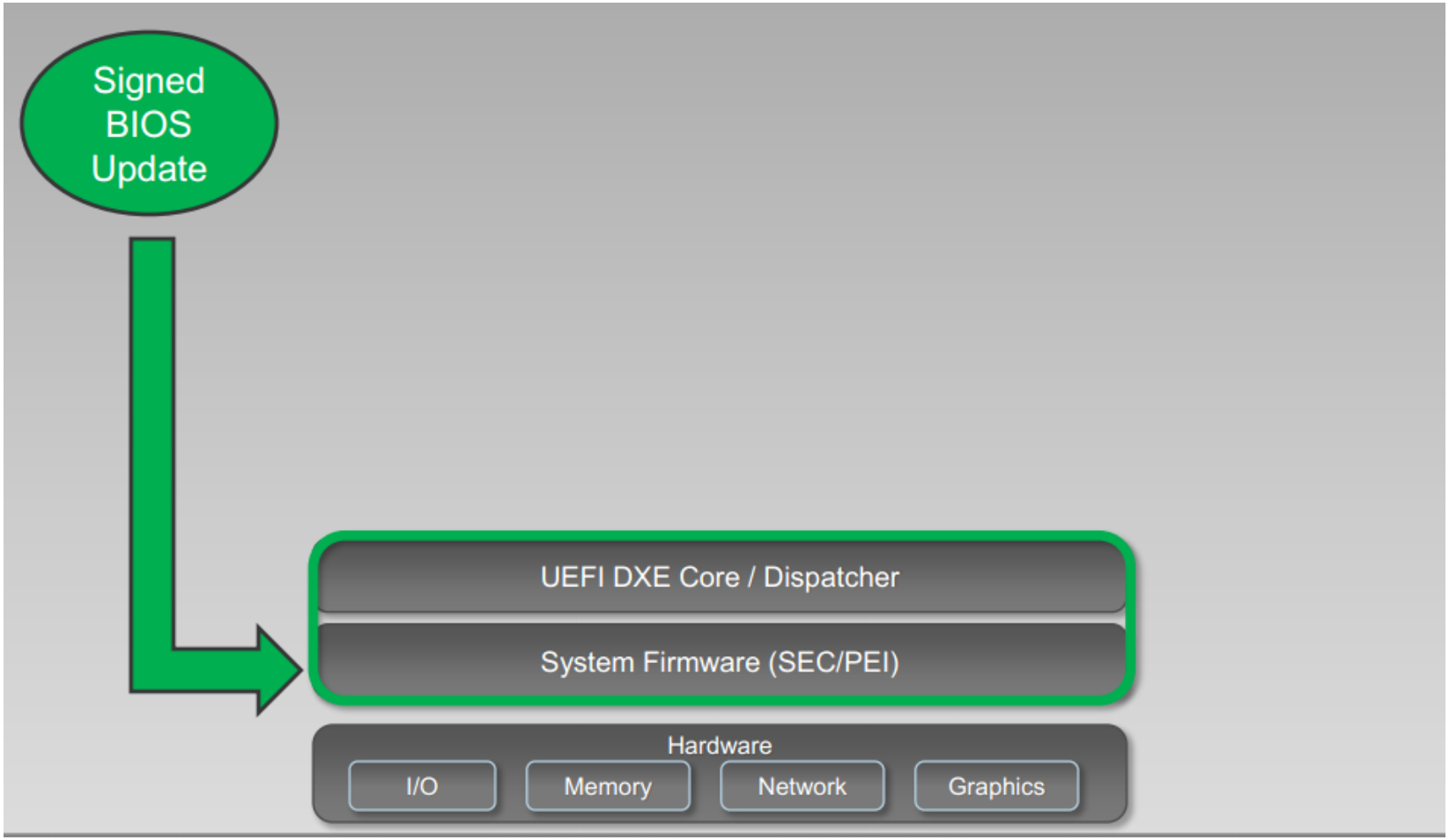
Attribution condition: You must indicate that derivative work
"Is derived from John Butterworth & Xeno Kovah’s ’Advanced Intel x86: BIOS and SMM’ class posted at http://opensecuritytraining.info/IntroBIOS.html”

Intro to UEFI Secure Boot

Intro to UEFI Secure Boot

- Verifies whether an executable is permitted to load and execute during the UEFI BIOS boot process
- When an executable like a boot loader or Option ROM is discovered, the UEFI checks if:
  - The executable is signed with an authorized key, or
  - The key, signature, or hash of the executable is stored in the authorized signature database
- UEFI components that are flash based (SEC, PEI, DXECore) are not verified for signature
  - The BIOS flash image has its signature checked during the update process (firmware signing)
- Yuriy Bulygin, Andrew Furtak, and Oleksandr Bazhaniuk have the best slides that describe the Secure Boot process
  - http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf (Black Hat USA 2013)

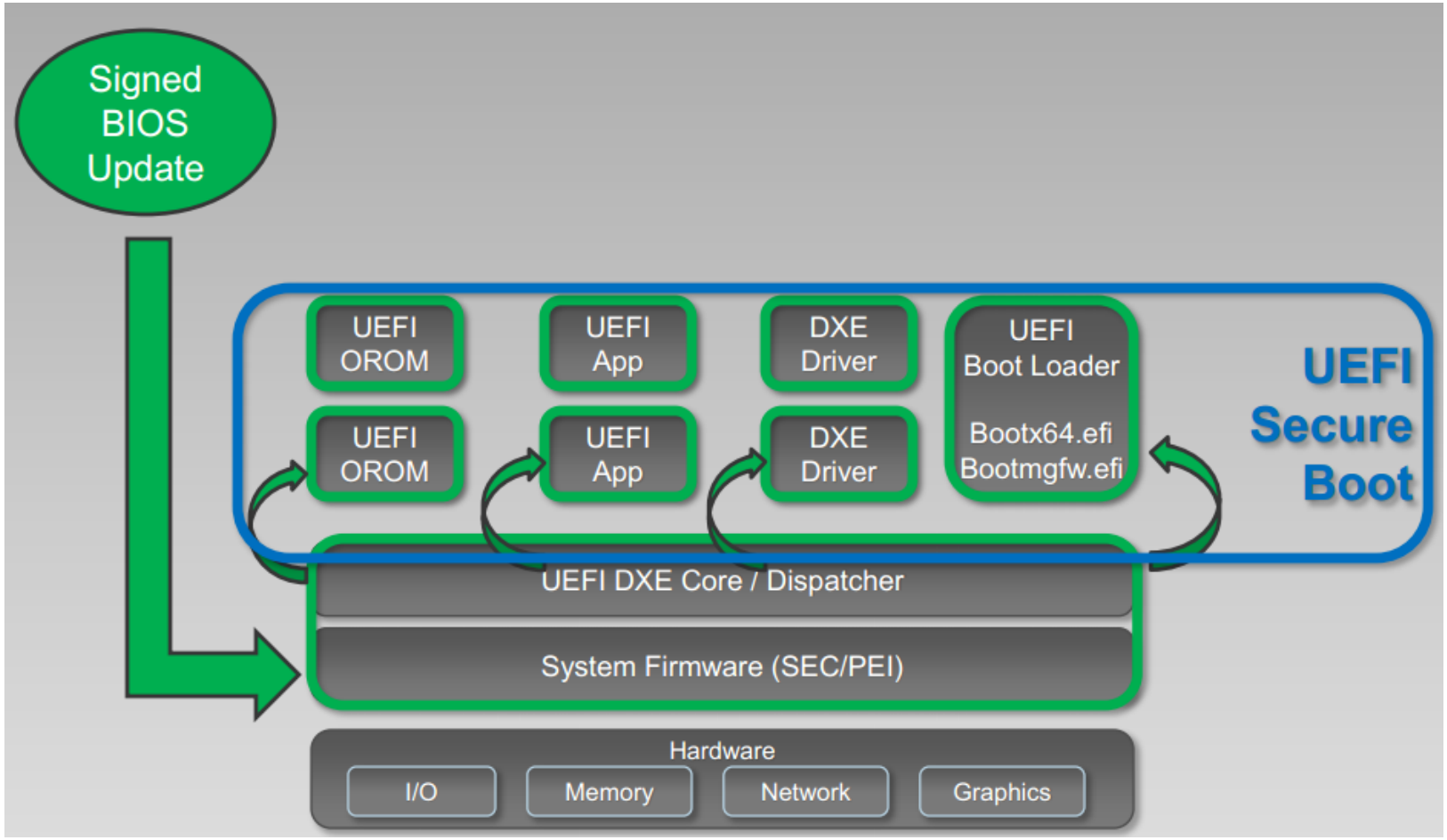
Firmware Signing



- Flash-based UEFI components are verified only during the update process when the whole BIOS image has its signature verified

http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

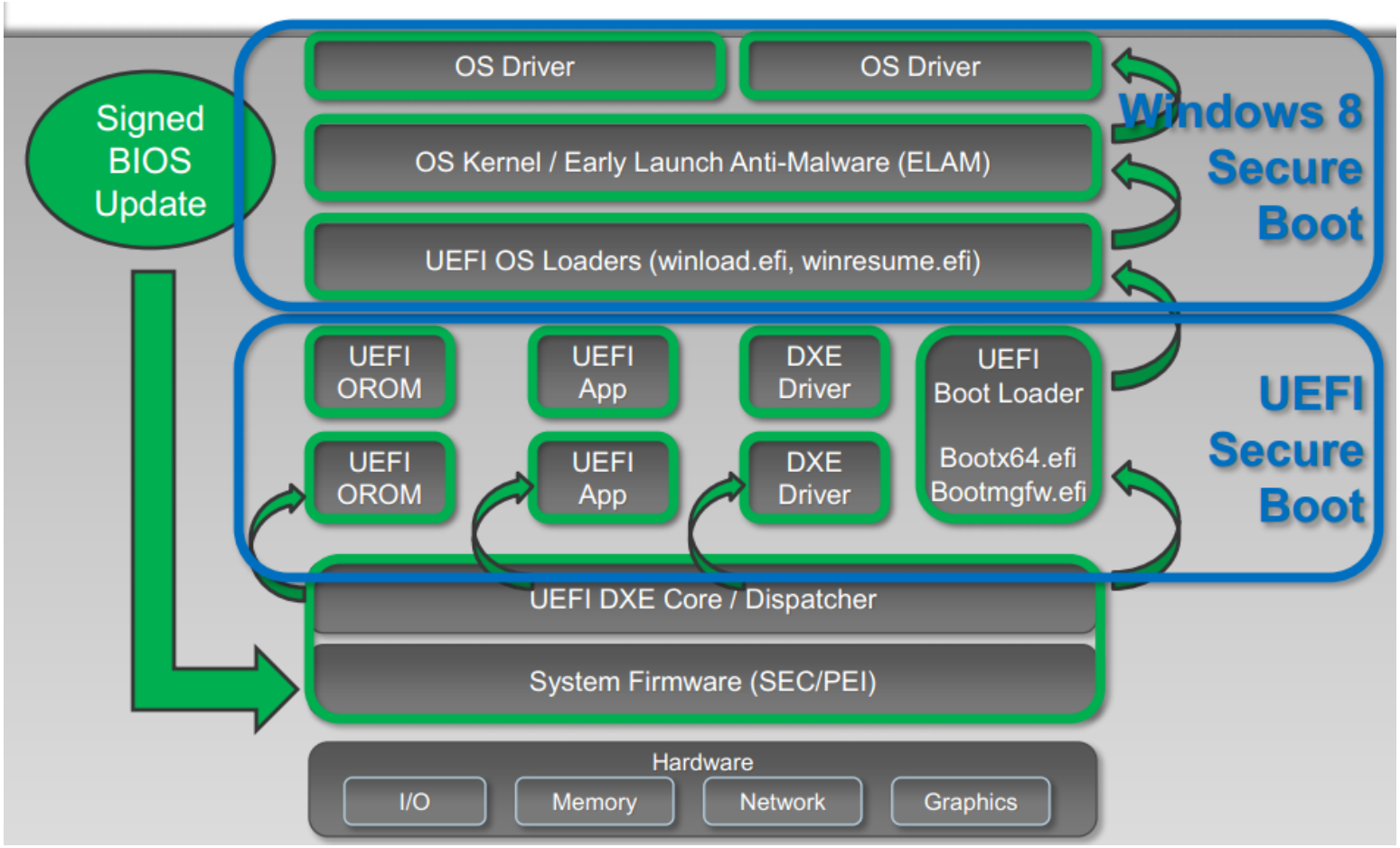
UEFI Secure Boot



- DXE verifies non-embedded XROMs, DXE drivers, UEFI applications and boot loader(s)
- This is the UEFI Secure Boot process

http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

Windows 8 Secure Boot



- Microsoft Windows 8 adds to the UEFI secure boot process
- Establishes a chain of verification
- UEFI Boot Loader -> OS Loader -> OS Kernel -> OS Drivers

http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

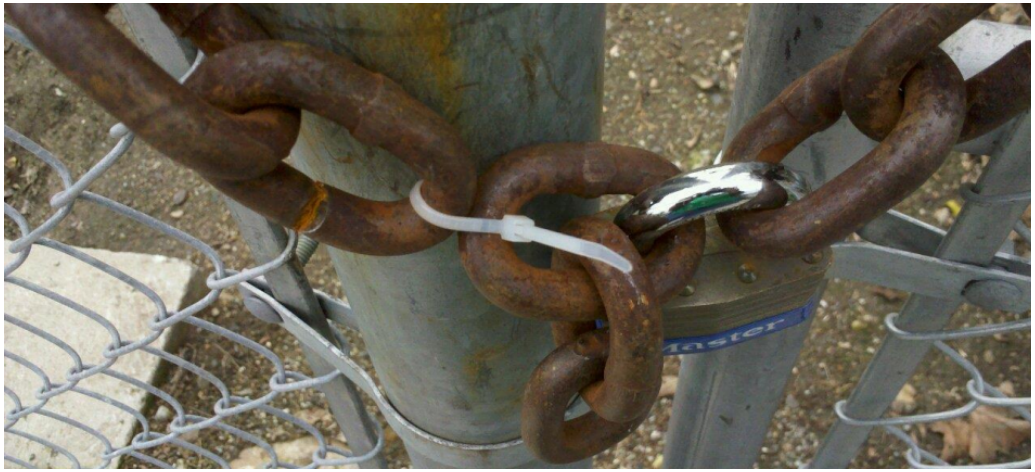
UEFI Variables (Keys and Key Stores)

- UEFI implements 4 “variables” which store keys, signatures, and/or hashes:
- Platform Key (PK)
  - Controls access to itself and the KEK variables
  - Only a physically present user or an application which has been signed with the PK is supposed to be able to modify this variable
  - Required to implement Secure Boot, otherwise the system is in Setup Mode where keys can be trivially modified by any application
- Key Exchange Key (KEK)
  - Used to update the signature database
  - Used to sign .efi binaries so they may execute
- Signature Database (DB)
  - A whitelist of keys, signatures and/or hashes of binaries
- Forbidden Database (DBX)
  - A blacklist of keys, signatures, and/or hashes of binaries

UEFI Variables (Keys and Key Stores)

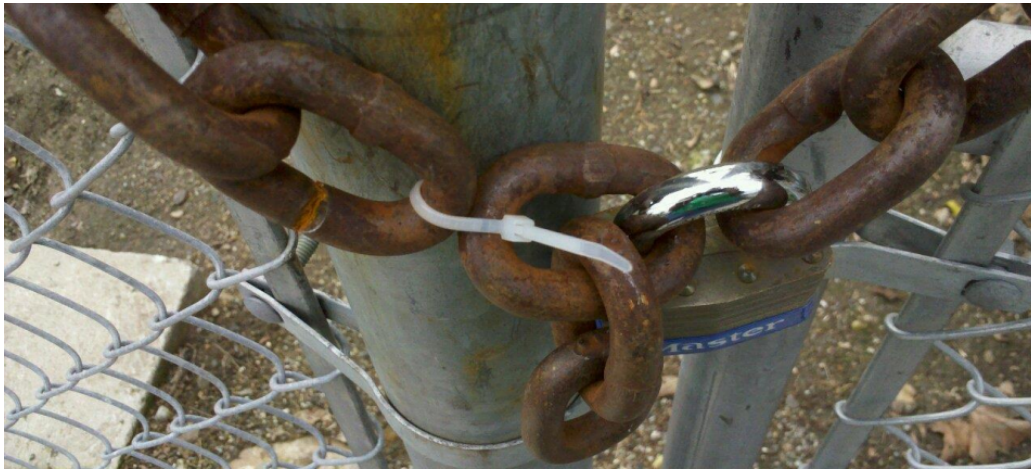
- As stated earlier, these variables are stored on the Flash file system
- Thus, if the SPI flash isn't locked down properly, these keys/ hashes can be overwritten by an attacker
- The problem is, the UEFI variables must rely solely on SMM to protect them!
- The secondary line of defense, the Protected Range registers cannot be used
- The UEFI variables must be kept writeable because at some point the system is going to need to write to them
- We saw this yesterday in the Charizard video where my colleague Sam suppressed SMI and wrote directly to the flash BIOS to add the hash of a malicious boot loader to the DB whitelist

(Easy) Secure Boot Bypass



- If signed firmware updates are not implemented properly, or if the SPI flash is not locked down properly, then Secure Boot can be trivially bypassed:
- http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf
- Some takeaways from this presentation:
  - Unprotected flash means UEFI variables can be overwritten
  - Add a hash to the DB for a malicious boot loader, then attack the boot loader to load a modified kernel
  - Secure Boot can be disabled by corrupting the PK
  - And more! Check it out.

Secure Boot Bypass



- From here on we'll assume that firmware signing has been enabled properly and the flash is locked down
- With that said, the firmware is still vulnerable
- Now we'll take a look at some vulnerabilities co-discovered by my colleague Corey Kallenberg and Yuriy Bulygin (Intel)
  - Presented first jointly with other Intel discoveries at CanSecWest 2013 as "All Your Boot are Belong to Us", and then later with the new material of Charizard at Syscan 2014 (and others) as "Setup for Failure: Defeating UEFI Secure Boot"

Secure Boot Signature Verification Policy

```
EFI_STATUS
EFIAPI
DxeImageVerificationHandler (
  IN  UINT32                           AuthenticationStatus,
  IN  CONST EFI_DEVICE_PATH_PROTOCOL   *File,
  IN  VOID                             *FileBuffer,
  IN  UINTN                            FileSize
  )
```

- Depending on the source location of the file, the signature check may be skipped
- When an image is discovered that needs to be authorized, the function 'DxeImageVerificationHandler' is called*
- Located in the file DxeImageVerificationLib.c

*Code from EDK2 open source reference implementation available at: https://svn.code.sf.net/p/edk2/code/trunk/edk2

Policy: ALWAYS_EXECUTE

```
//
// Check the image type and get policy setting.
//
switch (GetImageType (File)) {

case IMAGE_FROM_FV:
  Policy = ALWAYS_EXECUTE;
  break;

case IMAGE_FROM_OPTION_ROM:
  Policy = PcdGet32 (PcdOptionRomImageVerificationPolicy);
  break;

case IMAGE_FROM_REMOVABLE_MEDIA:
  Policy = PcdGet32 (PcdRemovableMediaImageVerificationPolicy);
  break;

case IMAGE_FROM_FIXED_MEDIA:
  Policy = PcdGet32 (PcdFixedMediaImageVerificationPolicy);
  break;

default:
  Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;
  break;
}
```

- If an executable is located on a Firmware Volume (SPI Flash) then it is always executed without authorization
- Makes sense assuming firmware signing is used and the BIOS flash was authorized prior to the update
- GetImageType gets its return value from DXE services that locate the source of the executable, not from a value stored in the executable

Code from EDK2 open source reference implementation available at: https://svn.code.sf.net/p/edk2/code/trunk/edk2

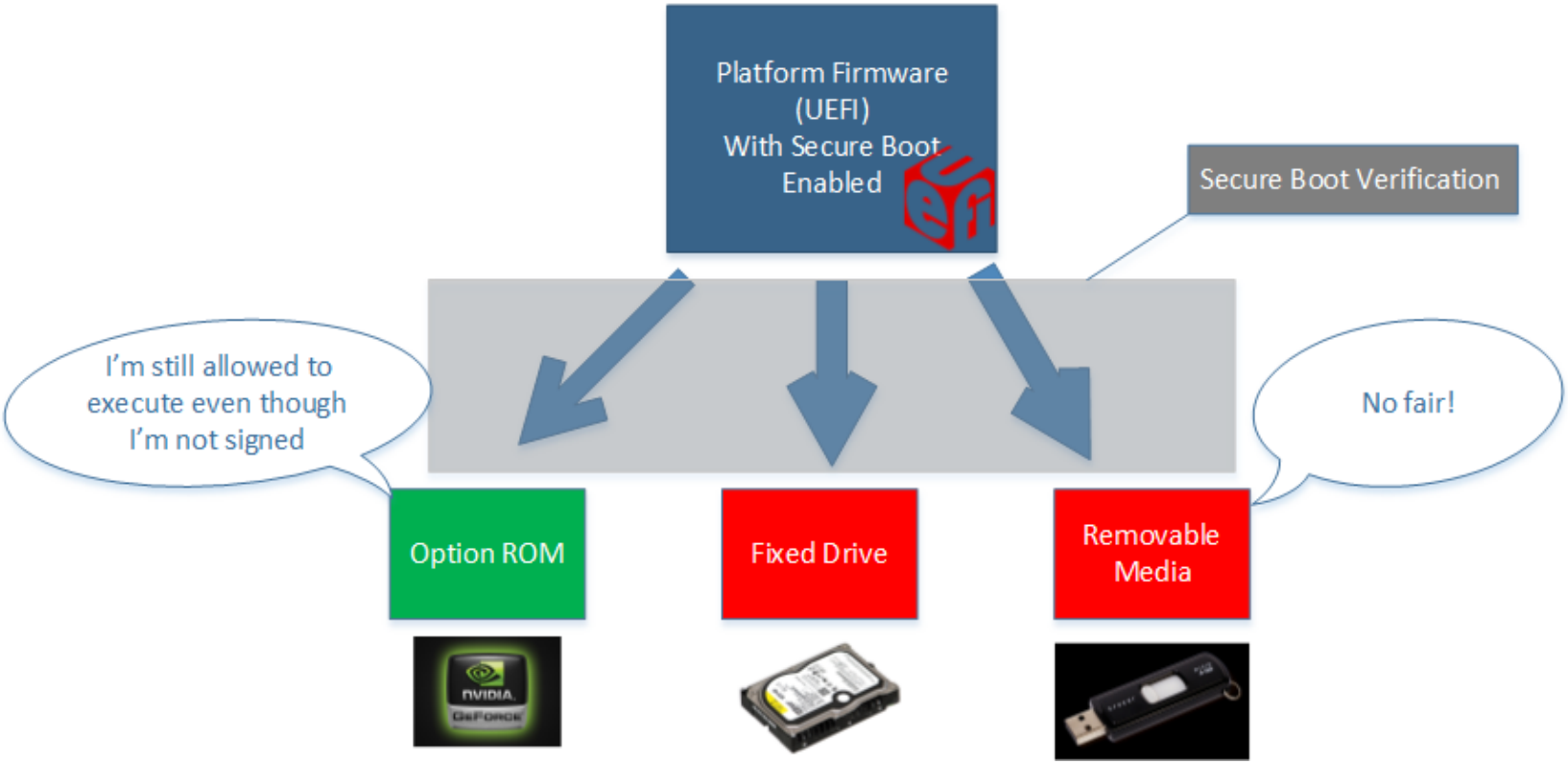
Flexible Signature Checking Policy

```
//
// Check the image type and get policy setting.
//
switch (GetImageType (File)) {

case IMAGE_FROM_FV:
  Policy = ALWAYS_EXECUTE;
  break;

case IMAGE_FROM_OPTION_ROM:
  Policy = PcdGet32 (PcdOptionRomImageVerificationPolicy);
  break;

case IMAGE_FROM_REMOVABLE_MEDIA:
  Policy = PcdGet32 (PcdRemovableMediaImageVerificationPolicy);
  break;

case IMAGE_FROM_FIXED_MEDIA:
  Policy = PcdGet32 (PcdFixedMediaImageVerificationPolicy);
  break;

default:
  Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;
  break;
}
```

- These policy values are hard-coded in the EDK2
  - OEMs can modify them as they see fit
- OEM's can specify custom policies, different from the reference specifications
- But they're likely not going to check everything from the FV at load time because that would be slow, and they have speed requirements they have to fulfill for their e.g. Windows 8 or Intel Ultrabook certifications

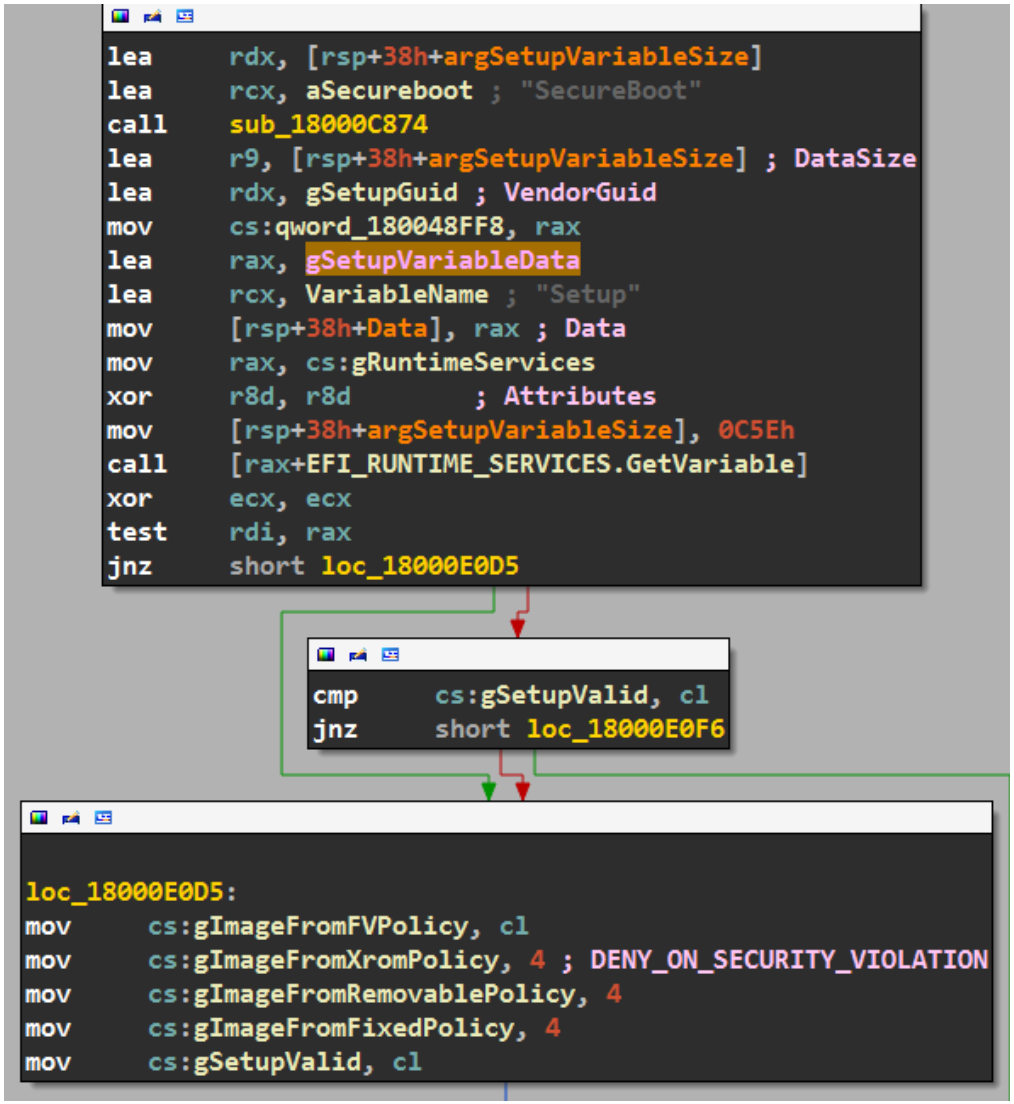
Code from EDK2 open source reference implementation available at: https://svn.code.sf.net/p/edk2/code/trunk/edk2

Flexible Signature Checking Policy



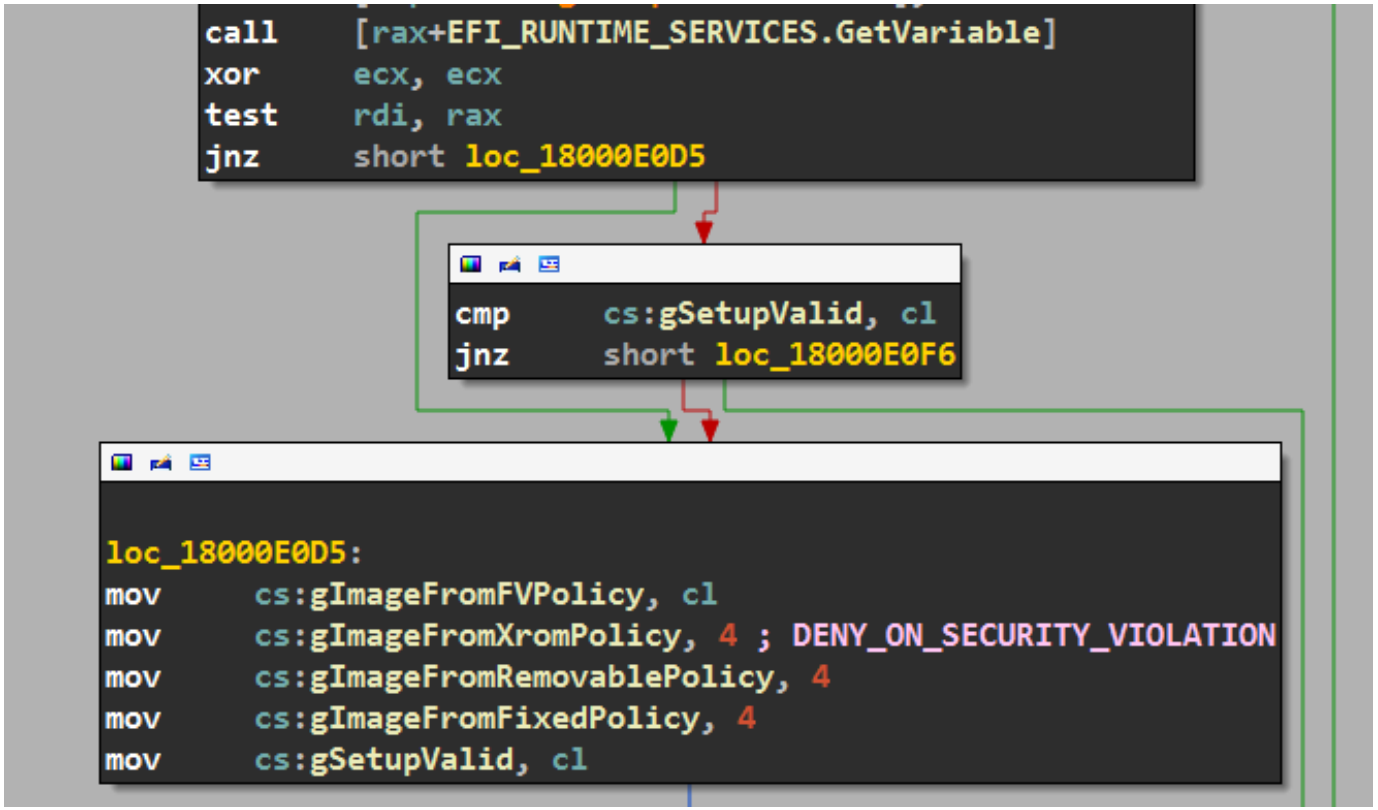
- Theoretical example: An OEM allows unsigned Option ROMs to run to allow aftermarket PCI cards, like graphics cards, to work seamlessly

Secure Boot Policy



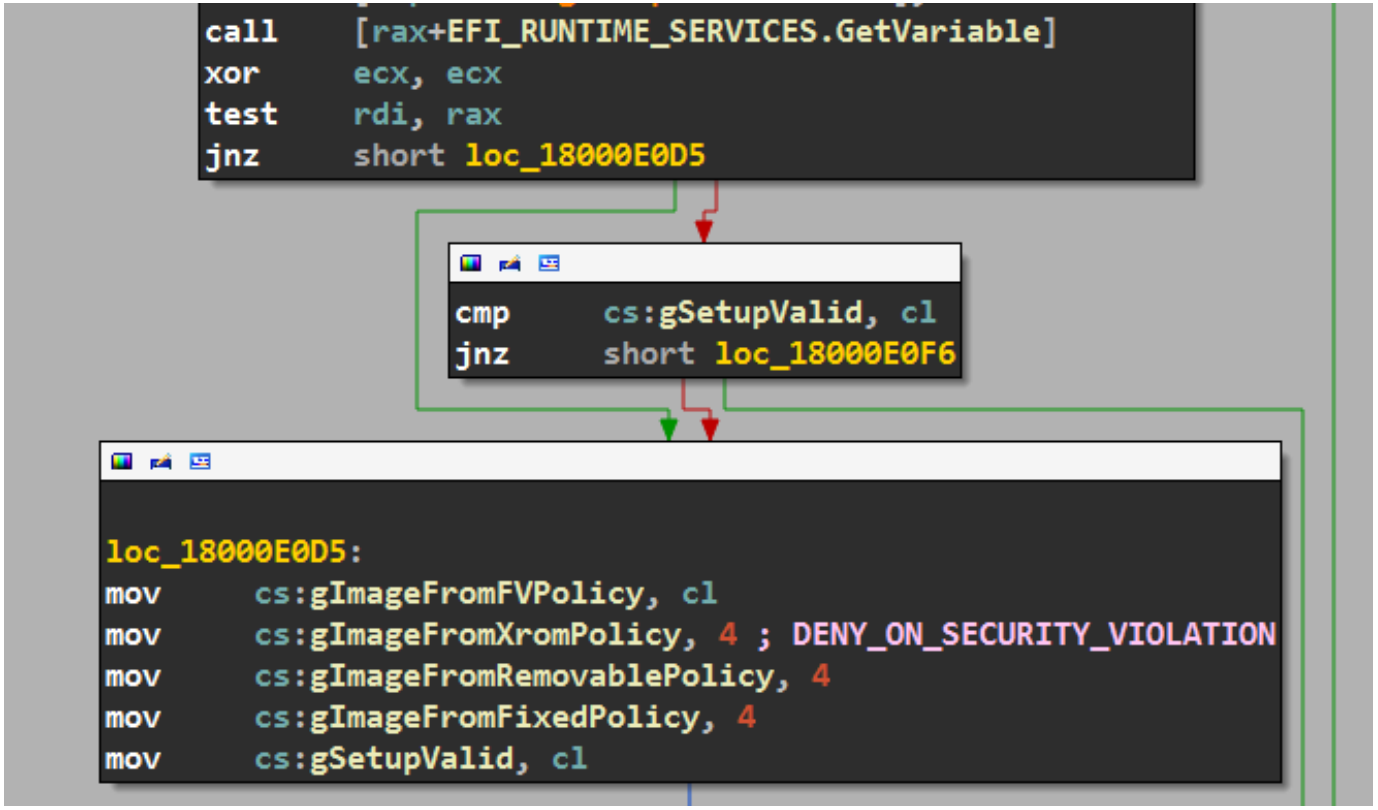
- Each OEM will have their own secure boot policy
- On the left is the disassembly of the secure boot policy initialization on a Dell Latitude E6430 BIOS revision A12
- You'll see that setup policy can come from either the flash NVRAM or be hardcoded in the BIOS
- Defined by the "Setup" variable

Secure Boot Policy



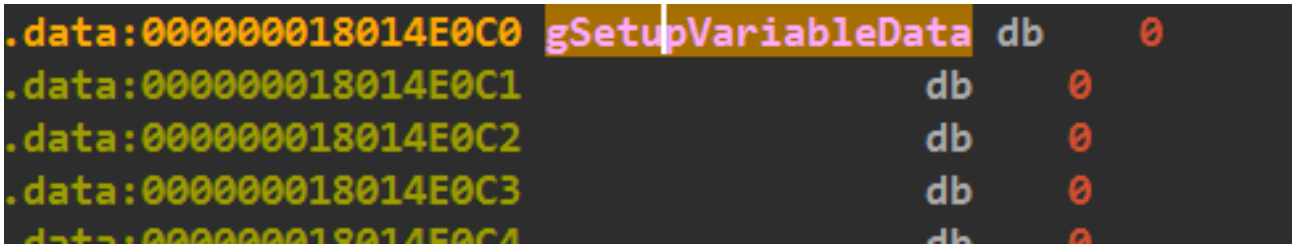
- gSetupValid determines whether to use the hardcoded secure boot policy, or if the policy embedded in the Setup variable should be used instead
- If it doesn't exist or it's invalid, then the hardcoded values will be used

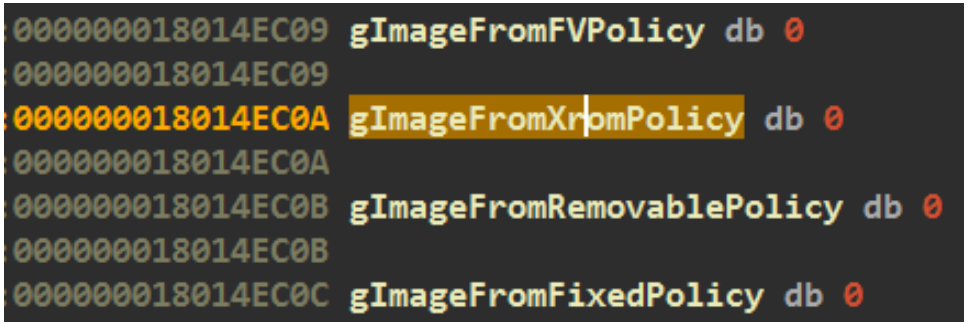
Default Hardcoded Policy

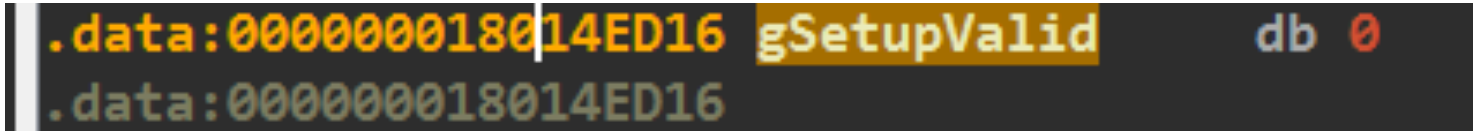


- Default hard-coded policy regarding unsigned executables originating from:
  - Option ROMs: Deny
  - Removable Drives: Deny
  - Hard Drives: Deny
  - Firmware Volume: Allow

Setup Variables Offsets

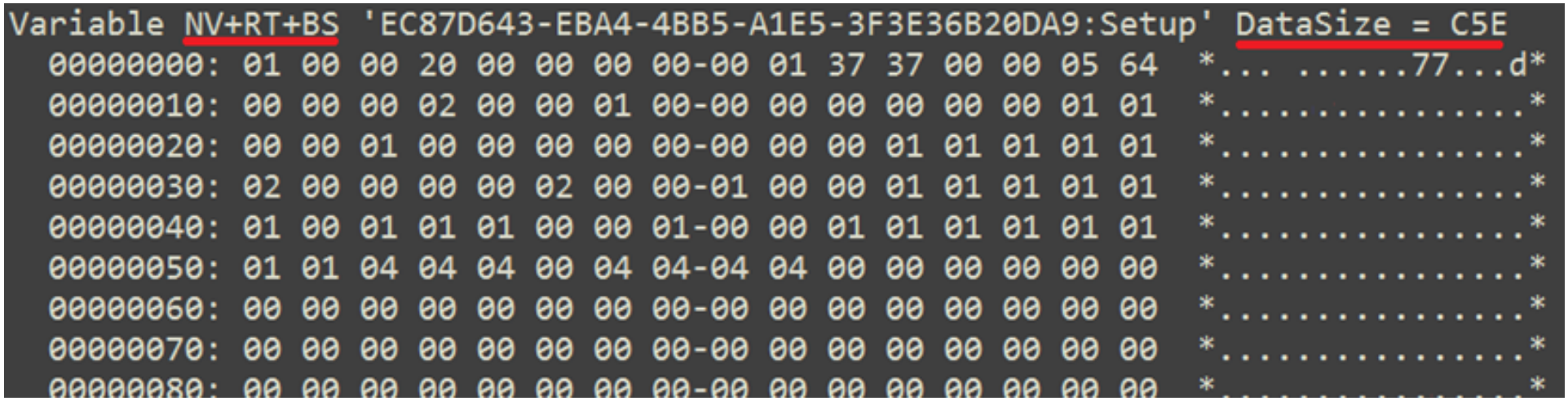






- The gSetupVariable data is loaded into memory at address 0x18014E0C0
- Secure Boot policy data starts at offset:
  - gImageFromFvPolicy – gSetupVariableData = 0xB49 (to 0xB4C)
- gSetupValid is at offset:
  - gSetupValid - gSetupVariableData = 0xC56

Setup Variable



- Setup variable is marked as:
  - NV: Non-Volatile (Stored on flash chip)
  - RT: accessible to Runtime Services
  - BS: accessible to Boot services
- Accessibility to Runtime Services means it should be modifiable from the operating system
- 0xC5E bytes long, chock full of stuff

EFI Variable Attributes

```
//*******************************************************
// Variable Attributes
//*******************************************************
#define EFI_VARIABLE_NON_VOLATILE                          0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS                    0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS                        0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD                 0x00000008
//This attribute is identified by the mnemonic 'HR' elsewhere in
this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \
0x00000020
#define EFI_VARIABLE_APPEND_WRITE                          0x00000040
```

- Each UEFI variable has attributes that determine how the firmware stores and maintains the data:
- ‘Non_Volatile’
  - The variable is stored on flash
- ‘Bootservice_Access’
  - Can be accessed/modified during boot. Must be set in order for Runtime_Access to also be set

* UEFI 2.3.1 Errata C Final

EFI Variable Attributes

```
//*******************************************************
// Variable Attributes
//*******************************************************
#define EFI_VARIABLE_NON_VOLATILE                          0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS                    0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS                        0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD                 0x00000008
//This attribute is identified by the mnemonic 'HR' elsewhere in
this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \
0x00000020
#define EFI_VARIABLE_APPEND_WRITE                          0x00000040
```

- ‘Runtime_Access’
  - The variable can be accessed/modified by the Operating System or an application
- ‘Hardware_Error_Record’
  - Variable is stored in a portion of NVRAM (flash) reserved for error records

* UEFI 2.3.1 Errata C Final

EFI Variable Attributes

```
//*******************************************************
// Variable Attributes
//*******************************************************
#define EFI_VARIABLE_NON_VOLATILE                          0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS                    0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS                        0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD                 0x00000008
//This attribute is identified by the mnemonic 'HR' elsewhere in
this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \
0x00000020
#define EFI_VARIABLE_APPEND_WRITE                          0x00000040
```

- 'Authenticated_Write_Access'
  - The variable can be modified only by an application that has been signed with an authorized private key (or by present user)
  - KEK and DB are examples of Authorized variables
- 'Time_Based_Authenticated_Write_Access'
  - Variable is signed with a time-stamp
- 'Append_Write'
  - Variable may be appended with data

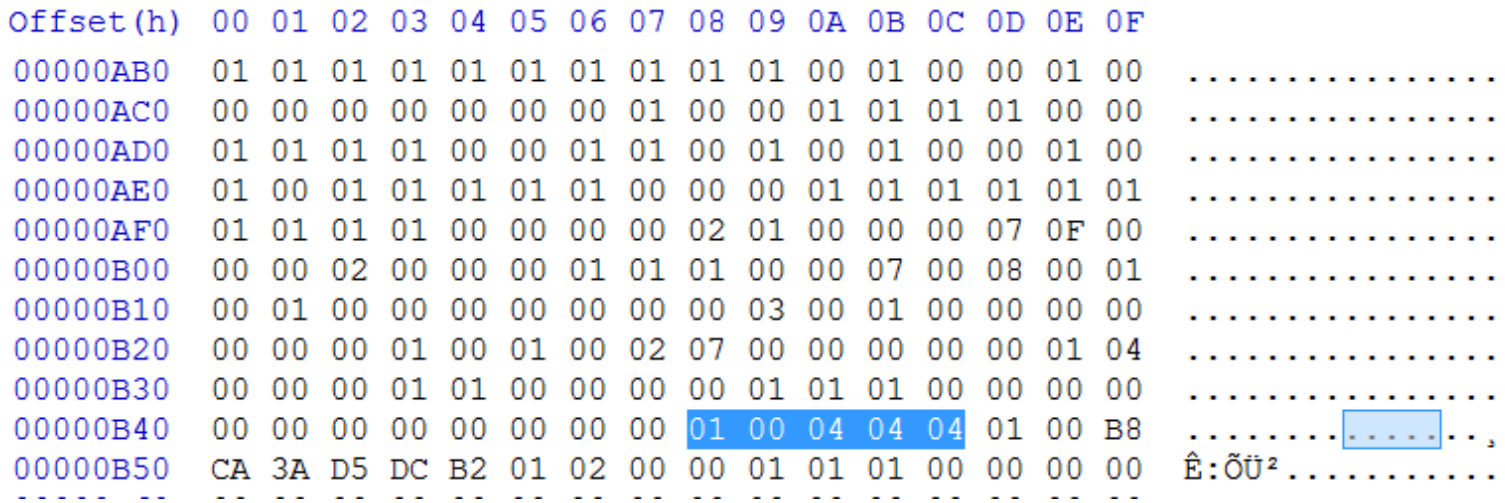
* UEFI 2.3.1 Errata C Final

EFI Variable Attributes Combinations

```
//*******************************************************
// Variable Attributes
//*******************************************************
#define EFI_VARIABLE_NON_VOLATILE                          0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS                    0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS                        0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD                 0x00000008
//This attribute is identified by the mnemonic 'HR' elsewhere in
this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \
0x00000020
#define EFI_VARIABLE_APPEND_WRITE                          0x00000040
```

- If a variable is marked as both Runtime and Authenticated, the variable can be modified only by an application that has been signed with an authorized key
- If a variable is marked as Runtime but not as Authenticated, the variable can be modified by any application
  - The Setup variable is marked like this

EFI Setup Variable Data



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000AB0	01	01	01	01	01	01	01	01	01	01	00	01	00	00	01	00	.....
00000AC0	00	00	00	00	00	00	00	01	00	00	01	01	01	01	00	00	.....
00000AD0	01	01	01	01	00	00	01	01	00	01	00	01	00	00	01	00	.....
00000AE0	01	00	01	01	01	01	01	00	00	00	01	01	01	01	01	01	.....
00000AF0	01	01	01	01	00	00	00	00	02	01	00	00	00	07	0F	00	.....
00000B00	00	00	02	00	00	00	01	01	01	00	00	07	00	08	00	01	.....
00000B10	00	01	00	00	00	00	00	00	00	03	00	01	00	00	00	00	.....
00000B20	00	00	00	01	00	01	00	02	07	00	00	00	00	00	01	04	.....
00000B30	00	00	00	01	01	00	00	00	00	01	01	01	00	00	00	00	.....
00000B40	00	00	00	00	00	00	00	00	01	00	04	04	04	01	00	B8	.....
00000B50	CA	3A	D5	DC	B2	01	02	00	00	01	01	01	00	00	00	00	Ê:ÕÜ².....

- Using the offsets we calculated earlier we can locate the secure boot policy settings in the Setup variable
- The Secure Boot policy settings started at offset 0xB49 from the start of the Setup variable data
- Byte B49 contains the “IMAGE_FROM_FV” policy and is set to ALWAYS_EXECUTE (0x00)
- Bytes B4A-B4C contain the policies pertaining to Option ROMs, Removable Storage, and Fixed Storage, respectively. All are set to “DENY_EXECUTE_ON_SECURITY_VIOLATION
  - We can change these to ALWAYS_EXECUTE (00)
- Byte B48 contains the Secure Boot on/off value (on)

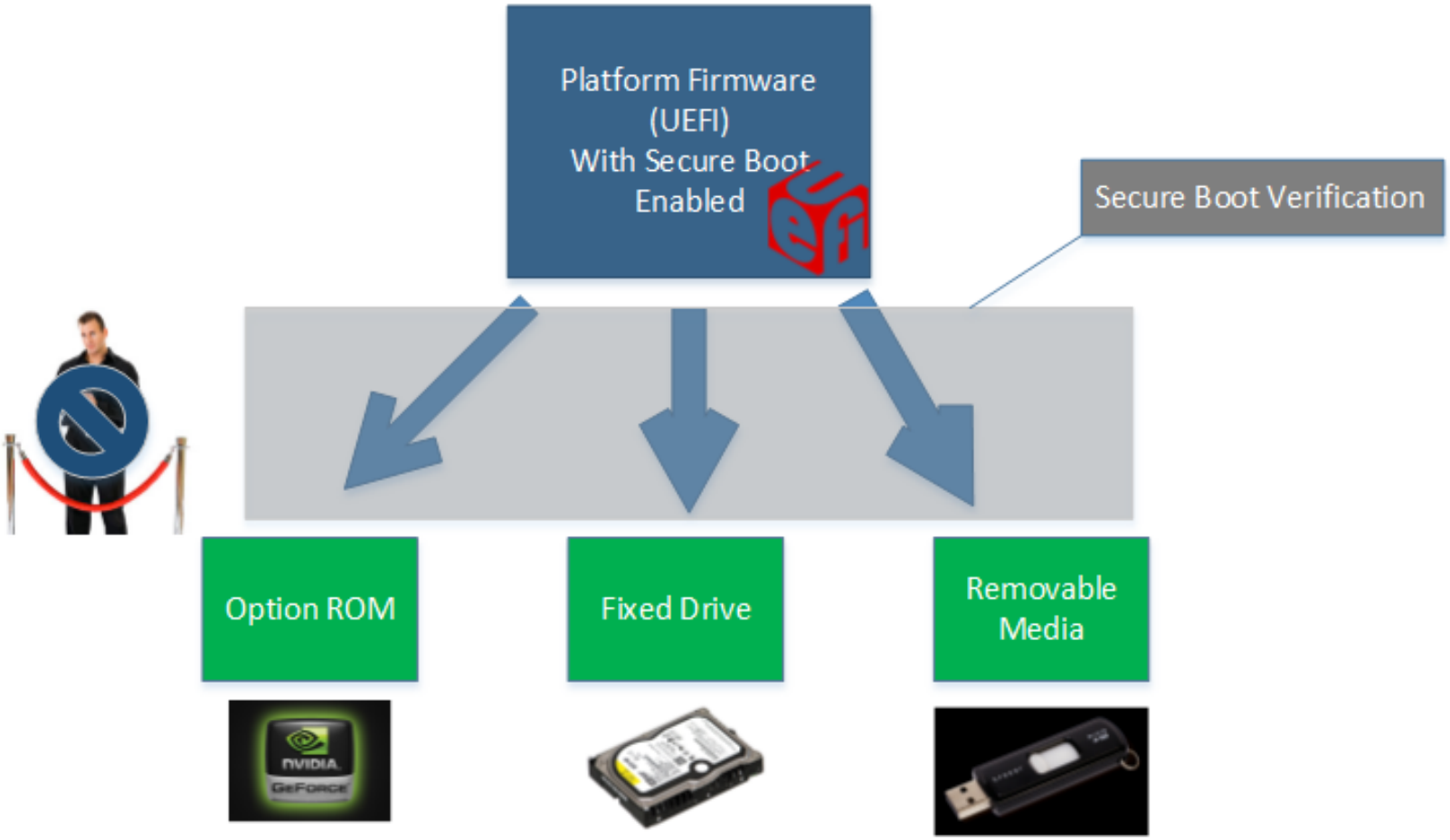
EFI Variable Access

```
DWORD WINAPI GetFirmwareEnvironmentVariable(
  _In_   LPCTSTR lpName,
  _In_   LPCTSTR lpGuid,
  _Out_  PVOID pBuffer,
  _In_   DWORD nSize
);
```

```
BOOL WINAPI SetFirmwareEnvironmentVariable(
  _In_  LPCTSTR lpName,
  _In_  LPCTSTR lpGuid,
  _In_  PVOID pBuffer,
  _In_  DWORD nSize
);
```

- Windows 8 provides an API to interact with EFI non-volatile variables
- http://msdn.microsoft.com/en-us/library/windows/desktop/ms724934(v=vs.85).aspx
- Because the Setup variable is marked as Runtime and not as Authenticated, we can modify it

Result: Modified Secure Boot Policy



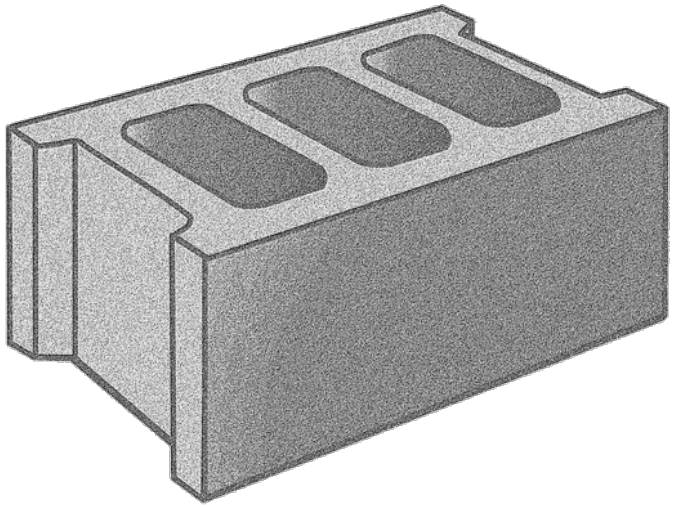
- An unsigned executable will always be executed regardless of whether it is signed or unsigned, based on the ALWAYS_EXECUTE policy associated with them now

Attack 1 Summary

- Malicious Windows 8 process can force unsigned executables to be allowed by Secure Boot
- Exploitable from privileged application in userland
- Bootkits will now function unimpeded
- Secure Boot will still report itself as enabled although it is no longer “functioning”
  - That secure boot ‘on’ value was not modified
- Co-discovered by Intel team

Attack 1 Addendum

- Malicious Windows 8 privileged process can force can "brick" your computer if it just writes Setup to all 0s
- Reinstalling the operating system won't fix this
- Intel didn't catch this and then we had to hold off on mentioning it until Hack in the Box AMS 2014



Attack 2: Delete Setup Variable

- Typically, setting a variables size to 0 will delete it
- Deleting the setup variable reverts the system to a legacy boot mode with secure boot disabled
- This is also effectively a secure boot bypass, as it will force the firmware to transfer control to an untrusted MBR upon next reboot

Attack 2 Summary

- Malicious Windows 8 process can disable Secure Boot by deleting “Setup” variable.
- Exploitable from userland
- Legacy MBR bootkits will now be executed by platform firmware
- Secure Boot will report itself as “disabled” in this case
  - More easily noticeable than the previous attack

Attack 3: Modify StdDefaults Variable

```
Dump Variable Stores
Variable NV+RT+BS '4599D26F-1A11-49B8-B91F-858745CFF824:StdDefaults' DataSize = D7F
  00000000: 4E 56 41 52 6F 0C FF FF-FF 83 00 53 65 74 75 70  *NVARo......Setup*
  00000010: 00 01 00 00 20 00 00 00-00 00 01 37 37 00 00 05  *.... ......77...*
  00000020: 64 00 00 00 03 00 00 01-00 00 01 01 02 01 00 01  *d...............*
  00000030: 01 00 00 01 00 00 00 00-00 00 00 00 01 01 00 01  *................*
  00000040: 01 02 01 00 00 00 02 00-00 01 00 00 01 01 01 01  *................*
  00000050: 01 01 00 01 01 01 00 00-01 00 00 01 01 01 01 01  *................*
```

- Actually, when the firmware detects the “Setup” variable has been deleted, it attempts to restore its contents from the “StdDefaults” variable
- This variable is also modifiable from the operating system, thanks to its non-authenticated and runtime attributes
- So we can corrupt this too to ensure that UEFI always restores our evil version

Attack 3: Summary

- Firmware would restore vulnerable Secure Boot policy whenever firmware configuration reverted to defaults
- This could make life very difficult

Summary

- CERT VU#758382
- Vulnerability allows bypass of secure boot on many systems.
- Co-reported by Intel and MITRE

- We first identified this vulnerability on a Dell Latitude E6430.
- Is this problem specific to the E6430?
- Is this problem specific to Dell?
- Is this vulnerability present in the UEFI reference implementation?

Summary

- CERT VU#758382
- Vulnerability allows bypass of secure boot on many systems.
- Co-reported by Intel and MITRE

- We first identified this vulnerability on a Dell Latitude E6430.
- Is this problem specific to the E6430? No.
- Is this problem specific to Dell? No.
- Is this vulnerability present in the UEFI reference implementation? No.