

# Setup For Failure: Defeating Secure Boot

Corey Kallenberg

@coreykal

Sam Cornwell

@ssc0rnwell

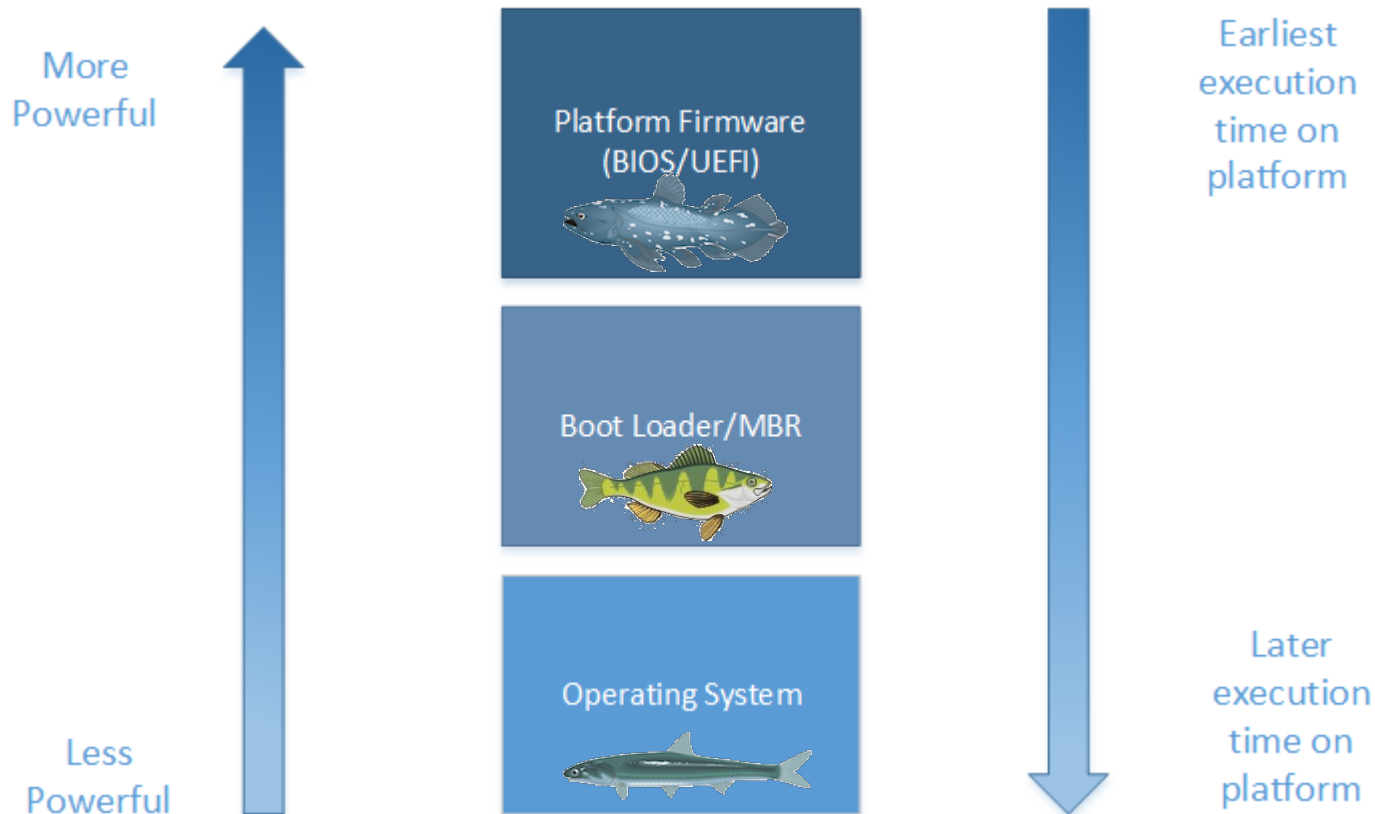
Xeno Kovah

@xenokovah

John Butterworth

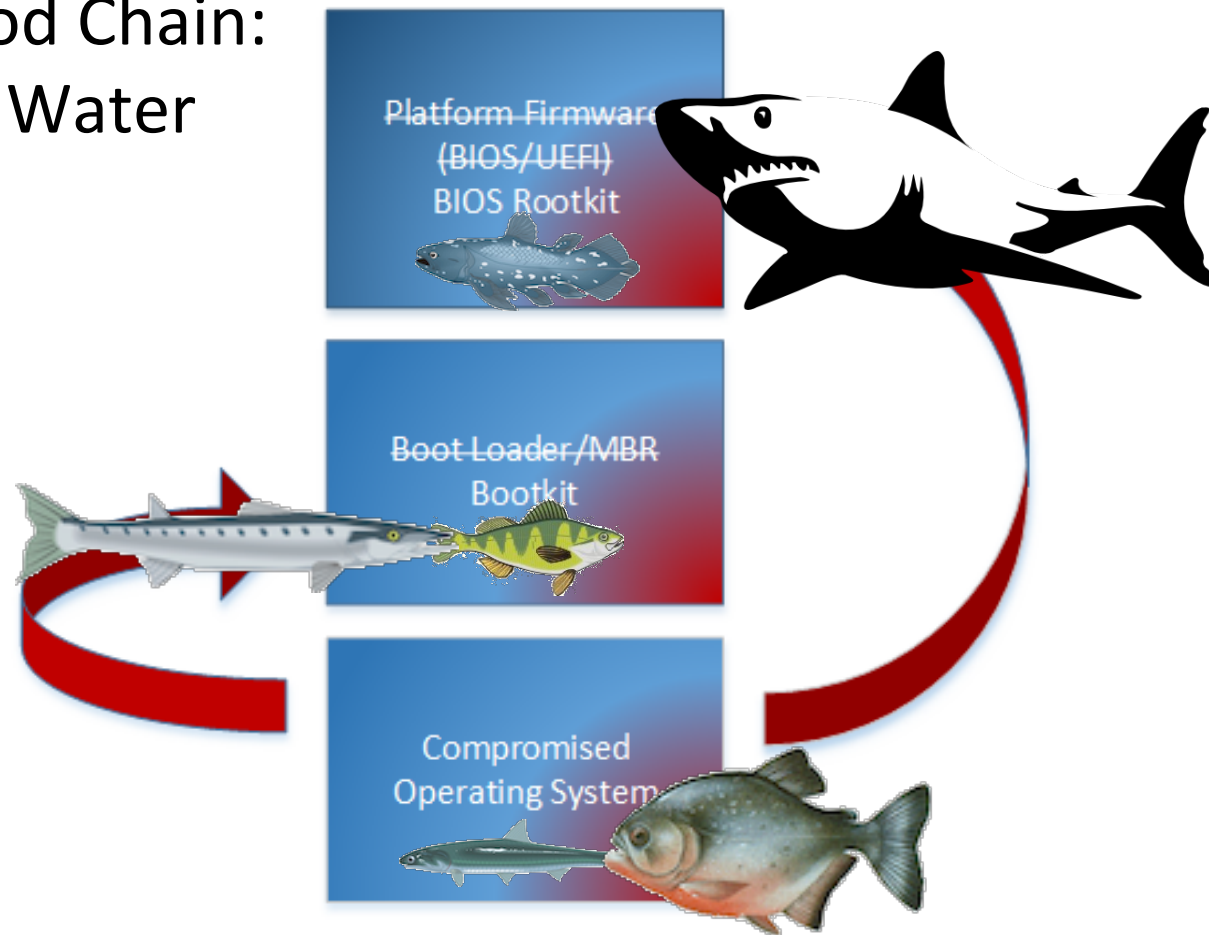
@jwbutterworth3

# The Malware Food Chain

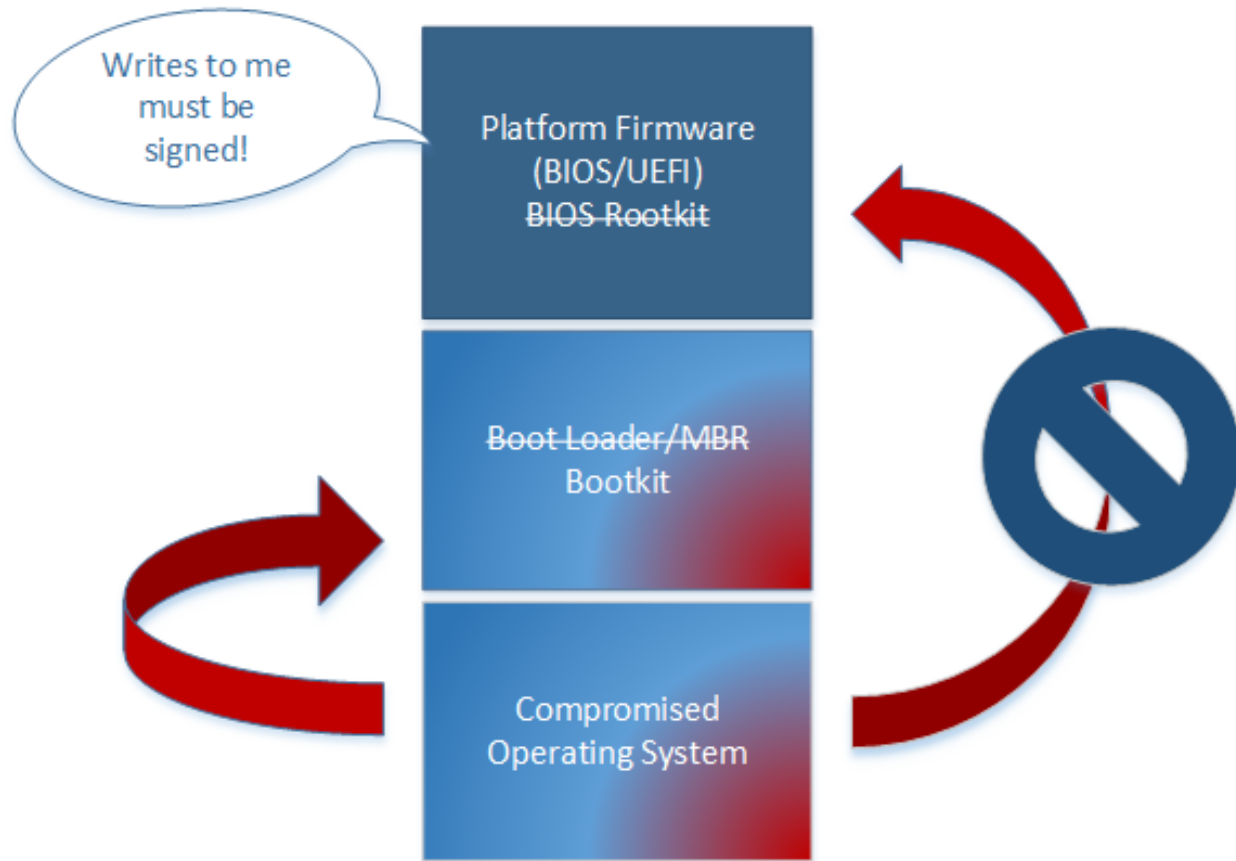


- Rootkits that execute earlier on the platform are in a position to compromise code that executes later on the platform, making earliest execution desirable.

# Malware Food Chain: Blood in the Water



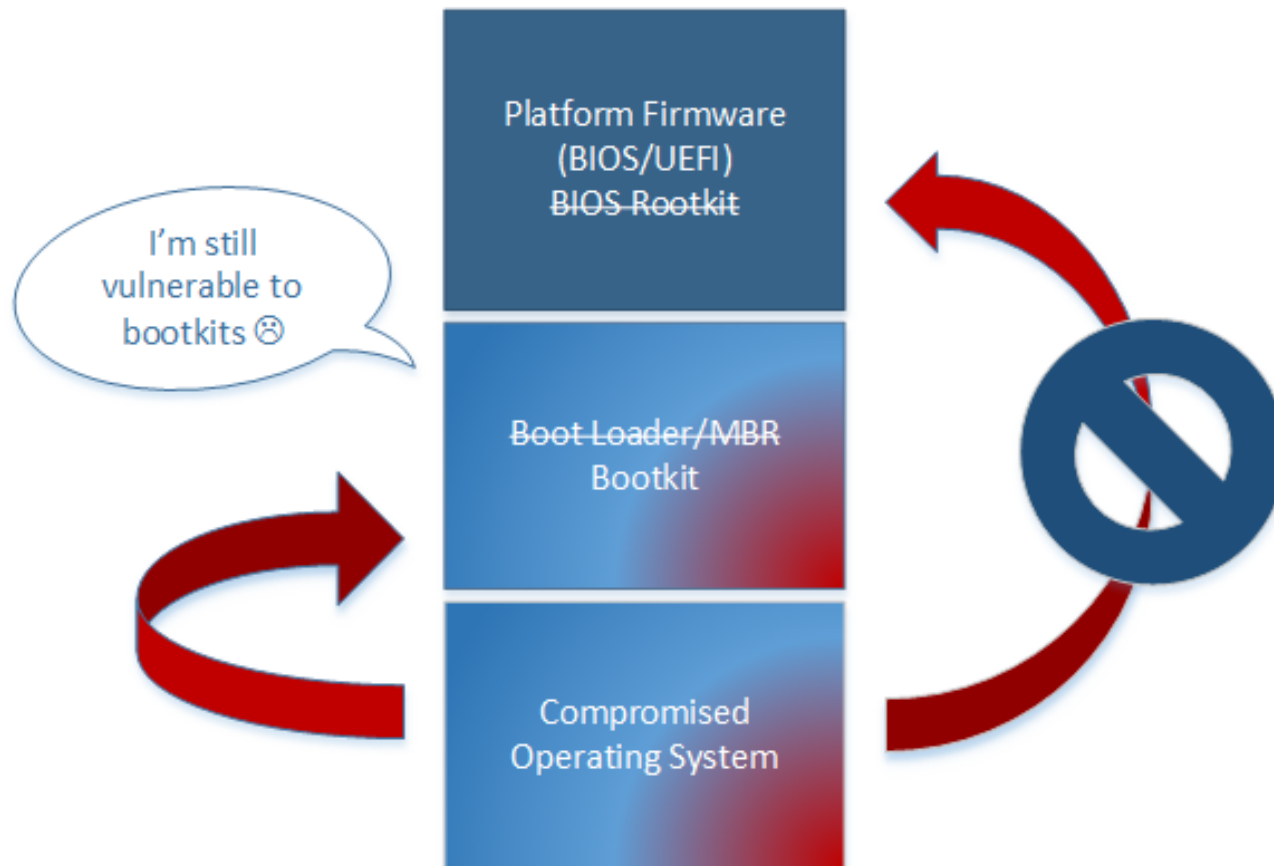
- It's advantageous for malware to claw its way up the food-chain and down towards hardware.
- Previously, malware running with sufficient privileges on the operating system could make malicious writes to both the Master Boot Record and the BIOS.



- Many modern platforms implement the requirement that updates to the firmware must be signed. This makes compromising the BIOS with a rootkit harder.

# More on Signed BIOS Requirement

- Signed BIOS recommendations have been around for a while now and preceded widespread adoption of UEFI.
- Not perfect, but significantly raises the barrier of entry into the platform firmware.
- “Attacking Intel BIOS” by Rafal Wojtczuk and Alexander Tereshkin.
- “Defeating Signed BIOS Enforcement” by Kallenberg, Butterworth, Kovah and Cornwell.

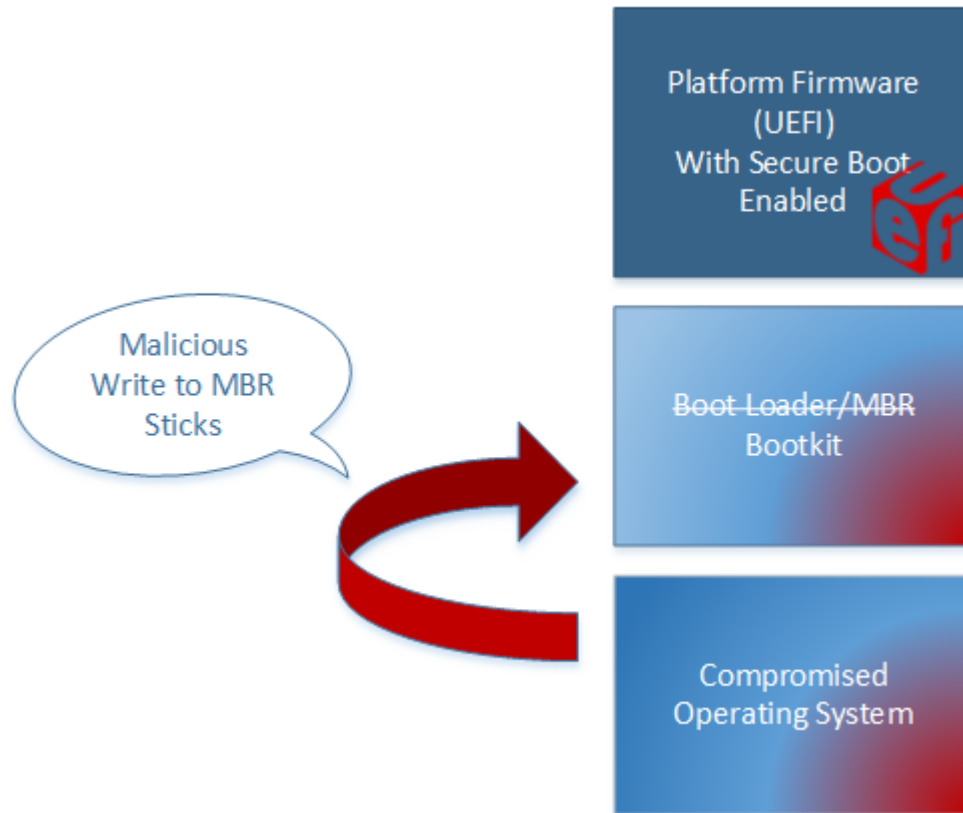


- Signed BIOS requirement did not address malicious boot loaders, leaving the door open for bootkits/evil maid attacks.

# Enter UEFI

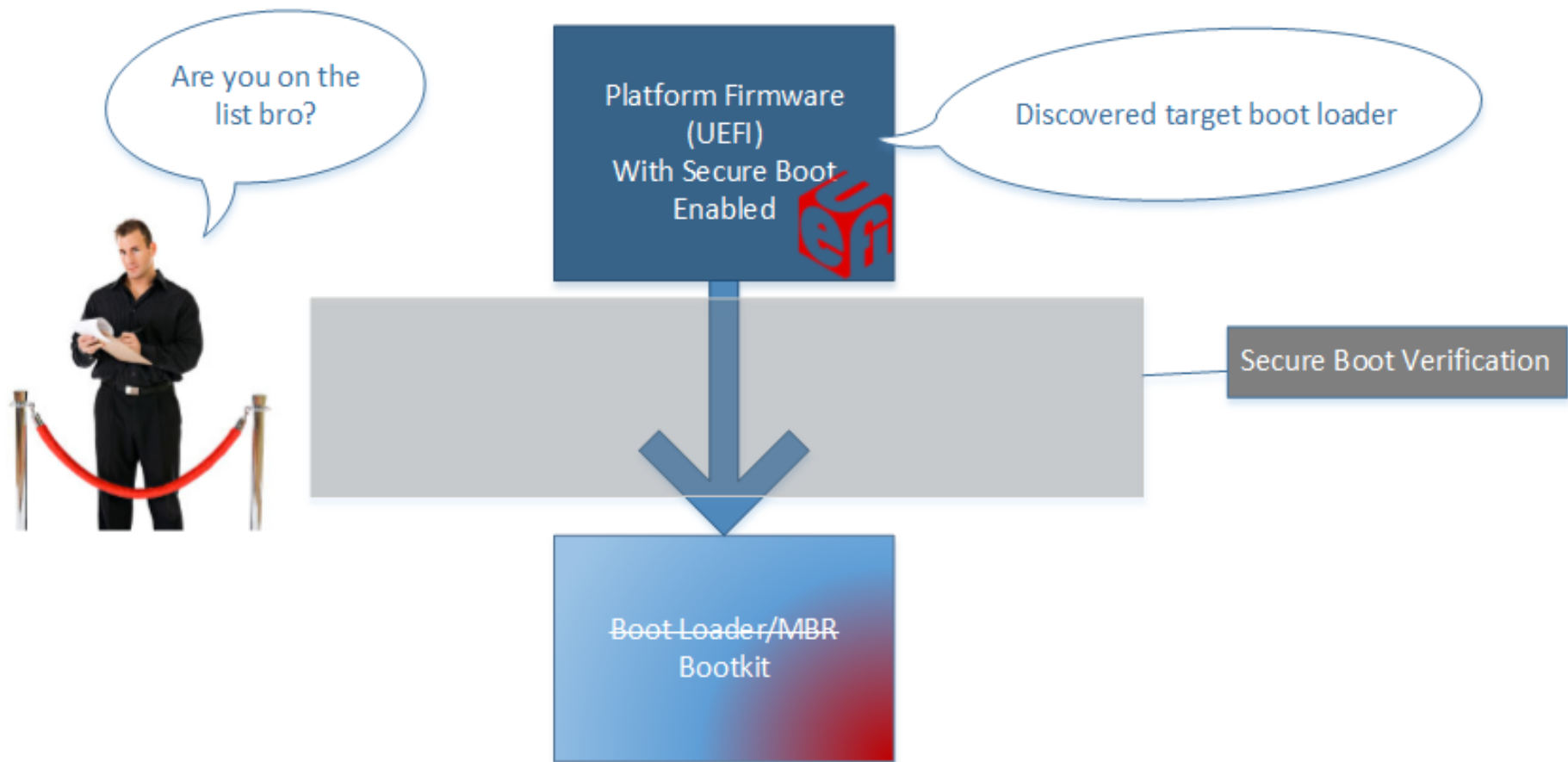


- UEFI has largely replaced conventional BIOS for PC platform firmware on new systems.
- UEFI 2.3.1 specifies a new security feature “Secure Boot” in order to address the bootkit vulnerability present on conventional BIOS systems.
- When enabled, Secure Boot validates the integrity of the operating system boot loader before transferring control to it.

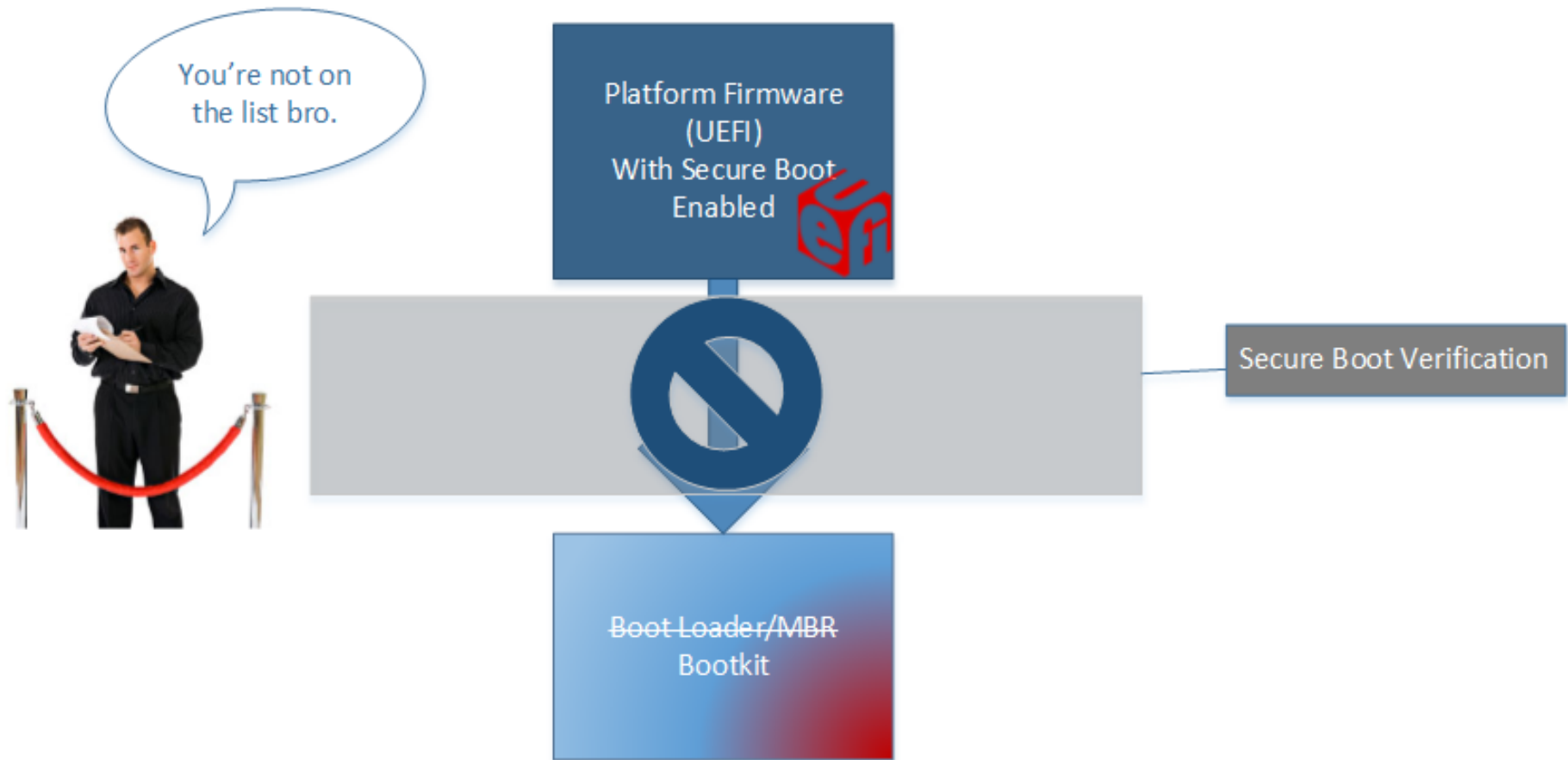


- Secure Boot does not prevent the initial malicious write to the boot loader (unlike signed bios enforcement)

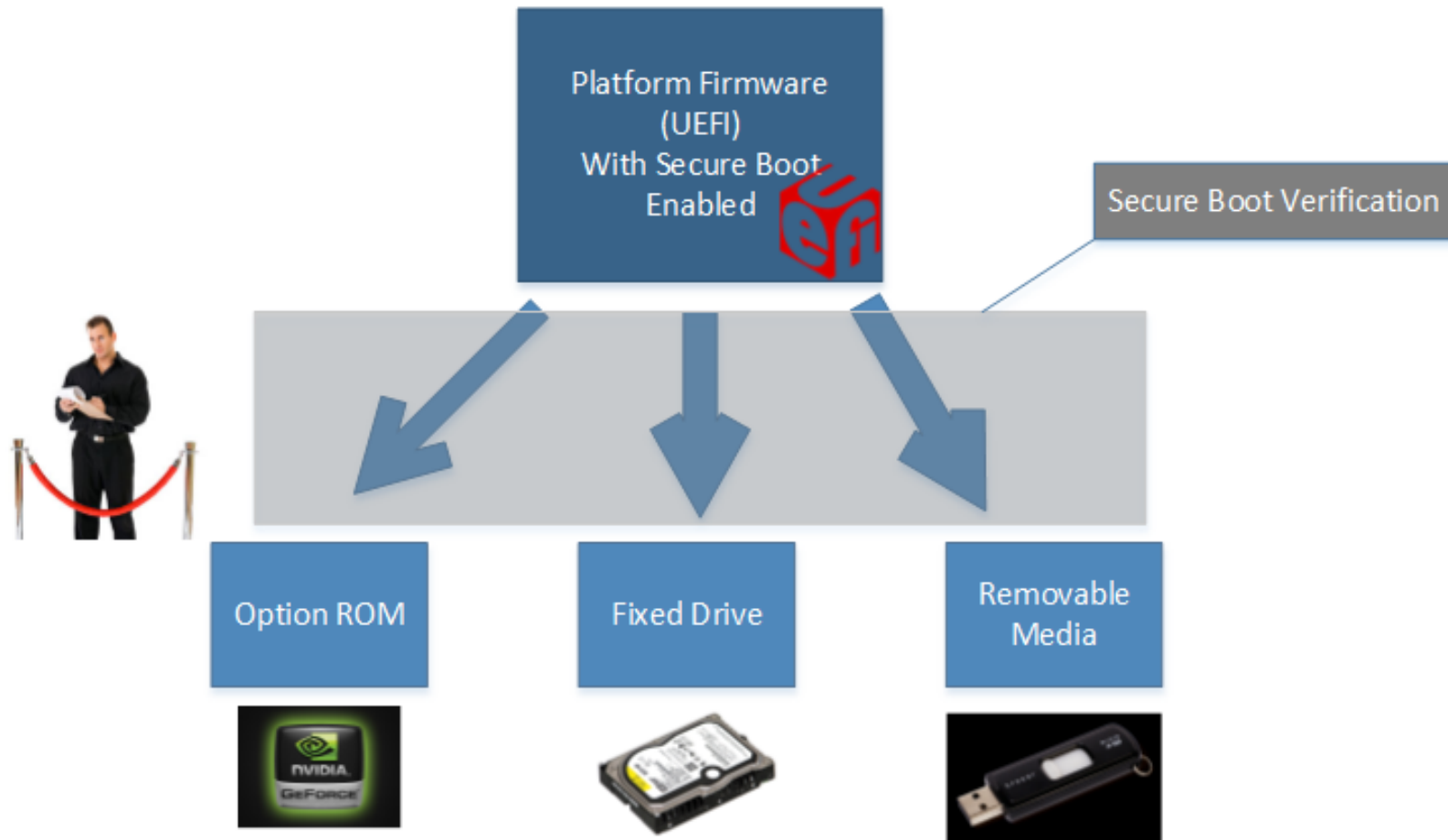




- Upon discovery of the overwritten (malicious) boot loader, the platform firmware will attempt to cryptographically verify the integrity of the target OS loader.



- Because the boot loader is not signed with a key embedded into the firmware, UEFI will refuse to boot the system.

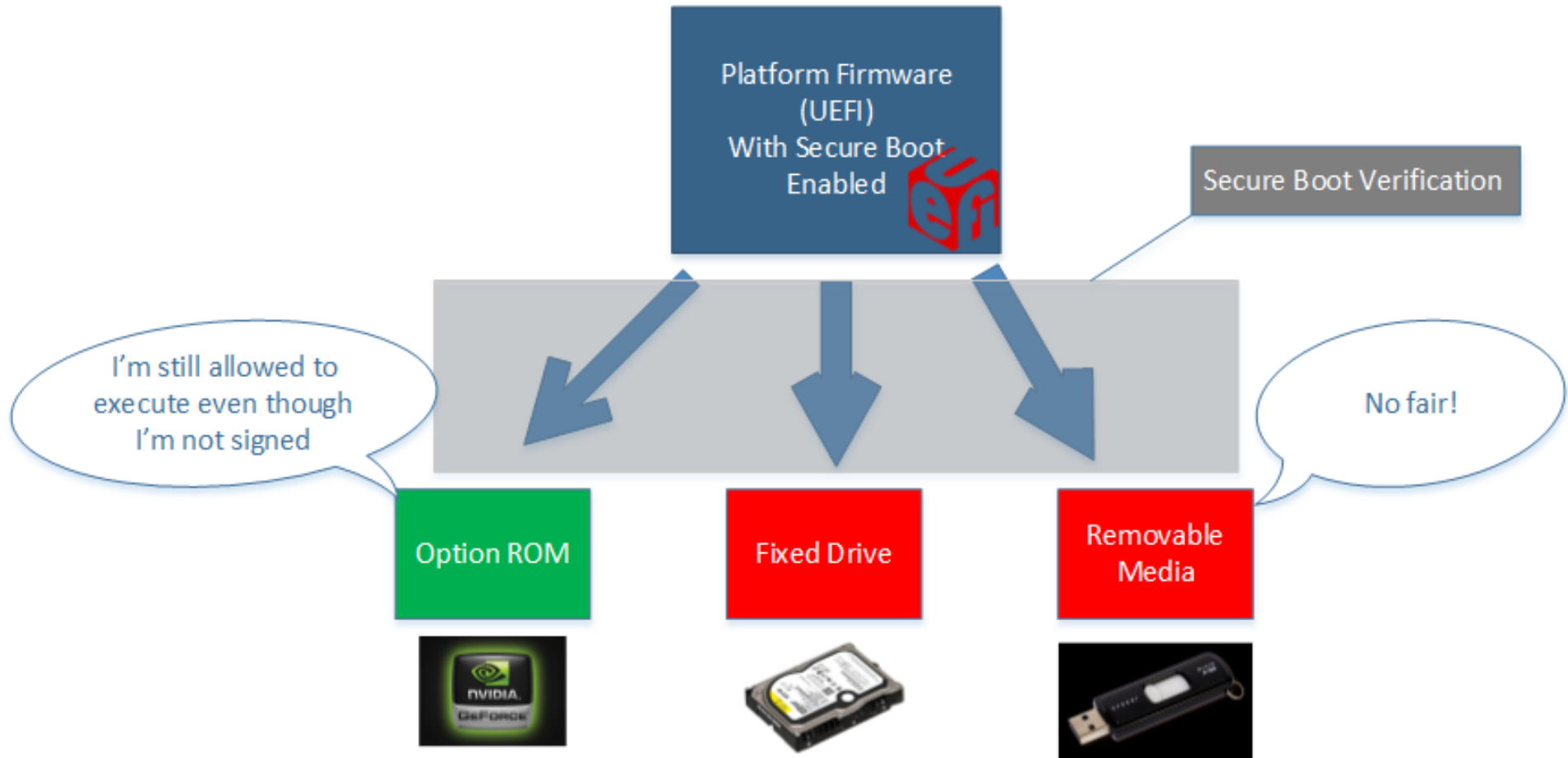


- Secure Boot is not limited to verifying only the boot loader.
- Secure Boot will attempt to verify any EFI executable that it attempts to transfer control to.
- Sort of..

```
//  
// 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;  
}
```

- The signature check on target EFI executables doesn't always occur.
- Depending on the origin of the target executable, the target may be allowed to execute automatically.
- In the EDK2, these policy values are hard coded.

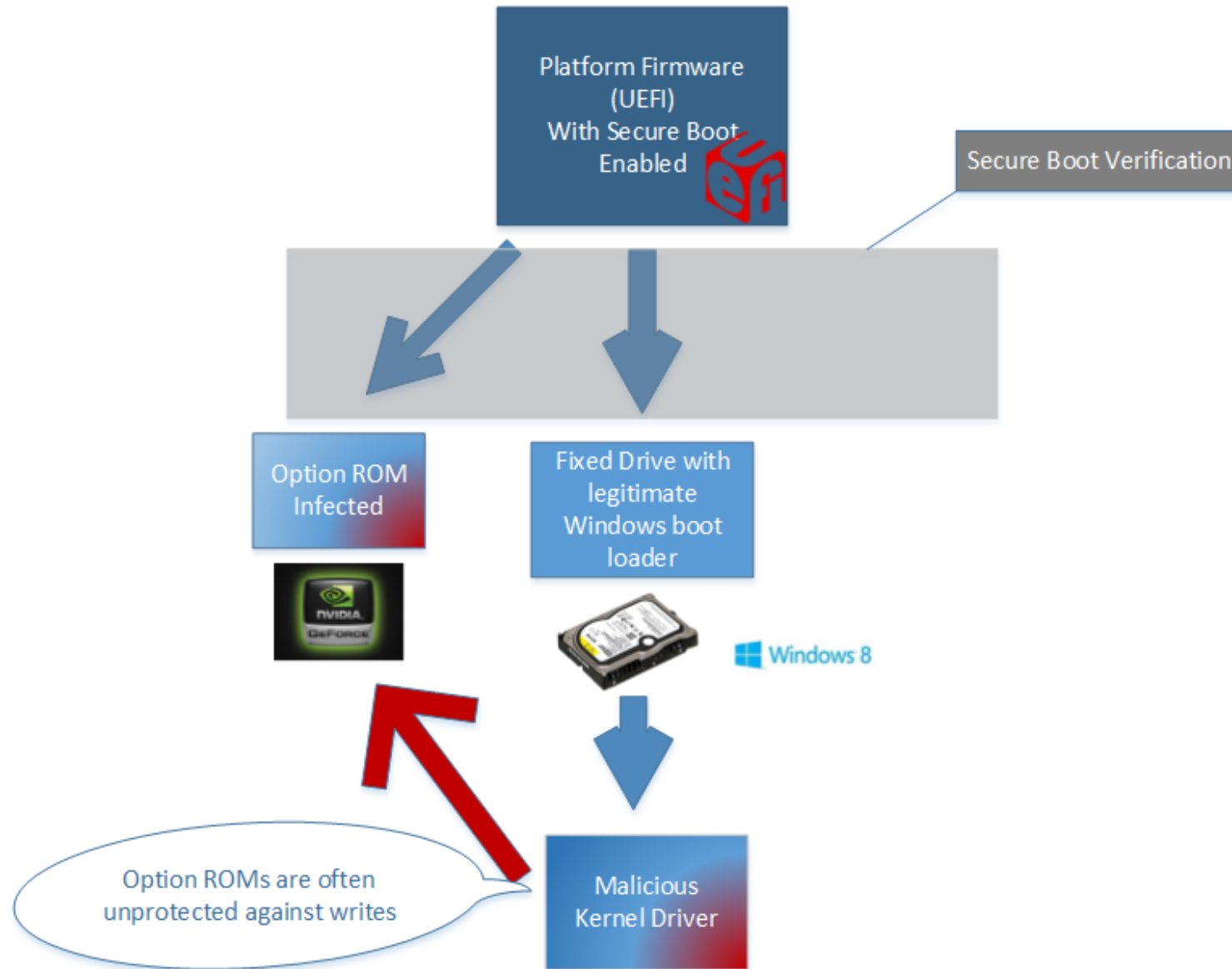
Code from EDK2 open source reference implementation available at:  
<https://svn.code.sf.net/p/edk2/code/trunk/edk2>



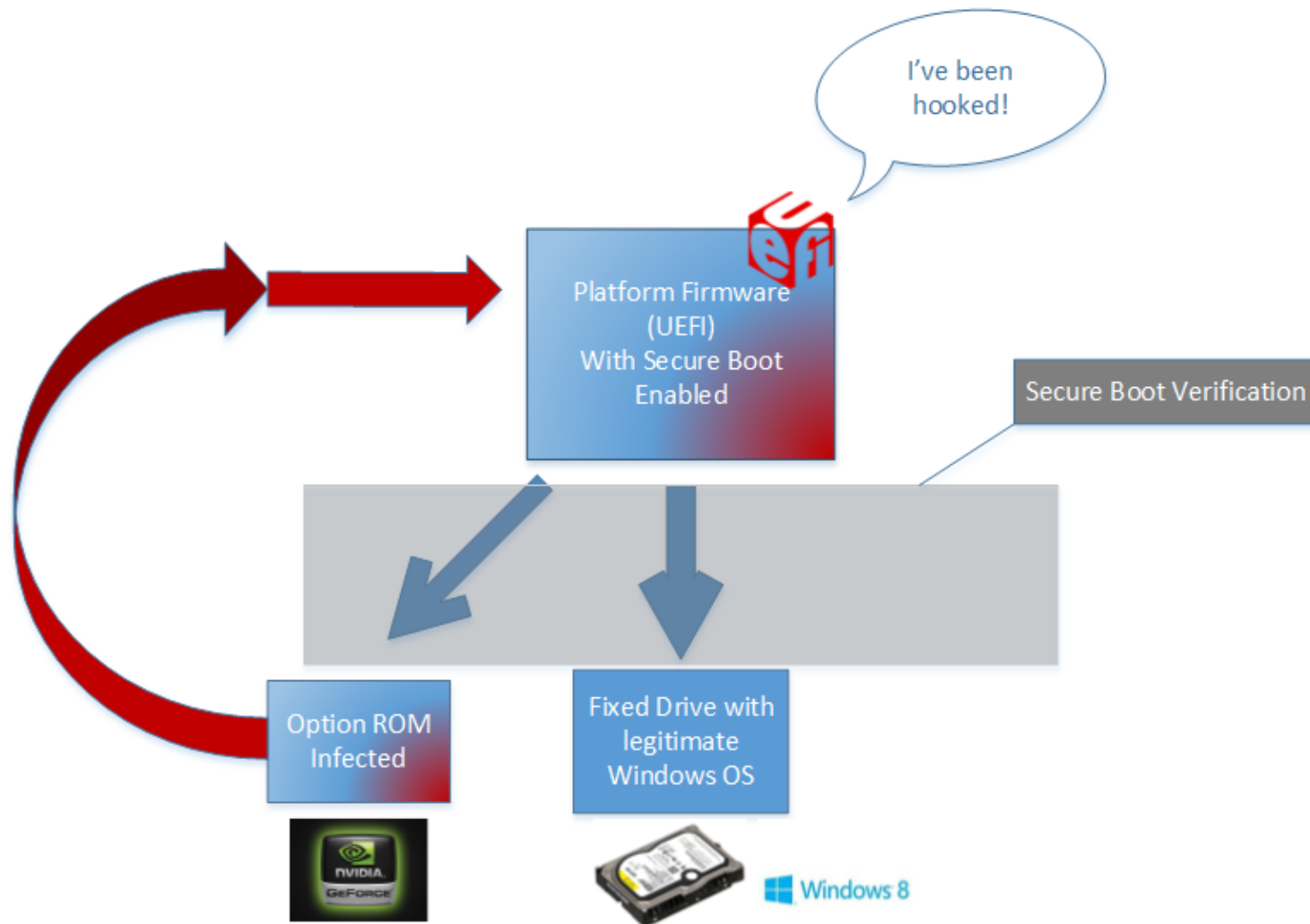
- For instance, an unsigned option ROM may be allowed to run if the OEM is concerned about breaking after market graphics cards that the user adds in later.

# Attack Proposal

- If a Secure Boot policy was configured to allow unsigned EFI executables to run on any mediums that an attacker may arbitrarily write to (boot loader, option rom, others...) then other legitimate EFI executables can be compromised later.



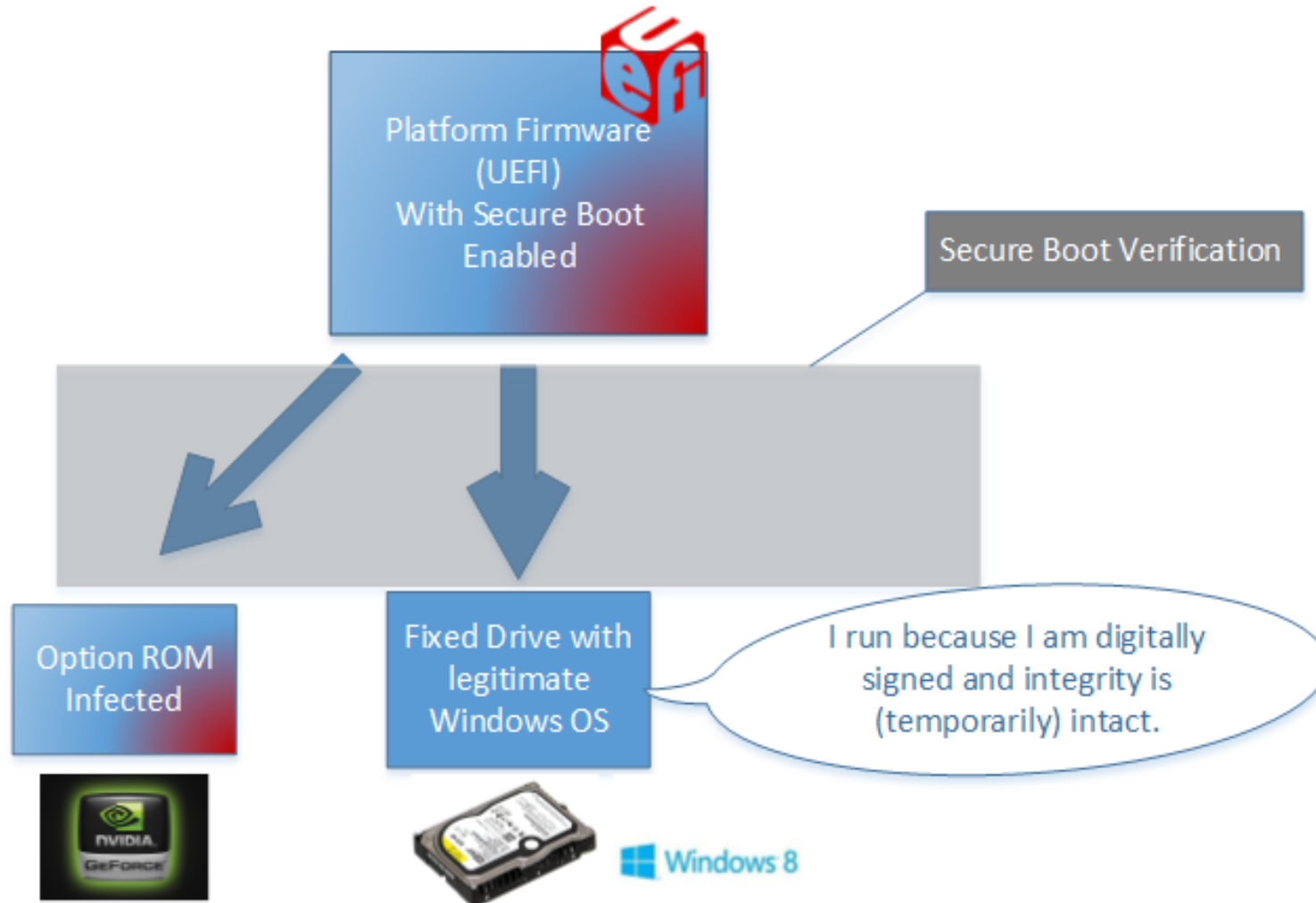
- The malicious option rom will run before the legitimate Windows boot loader.



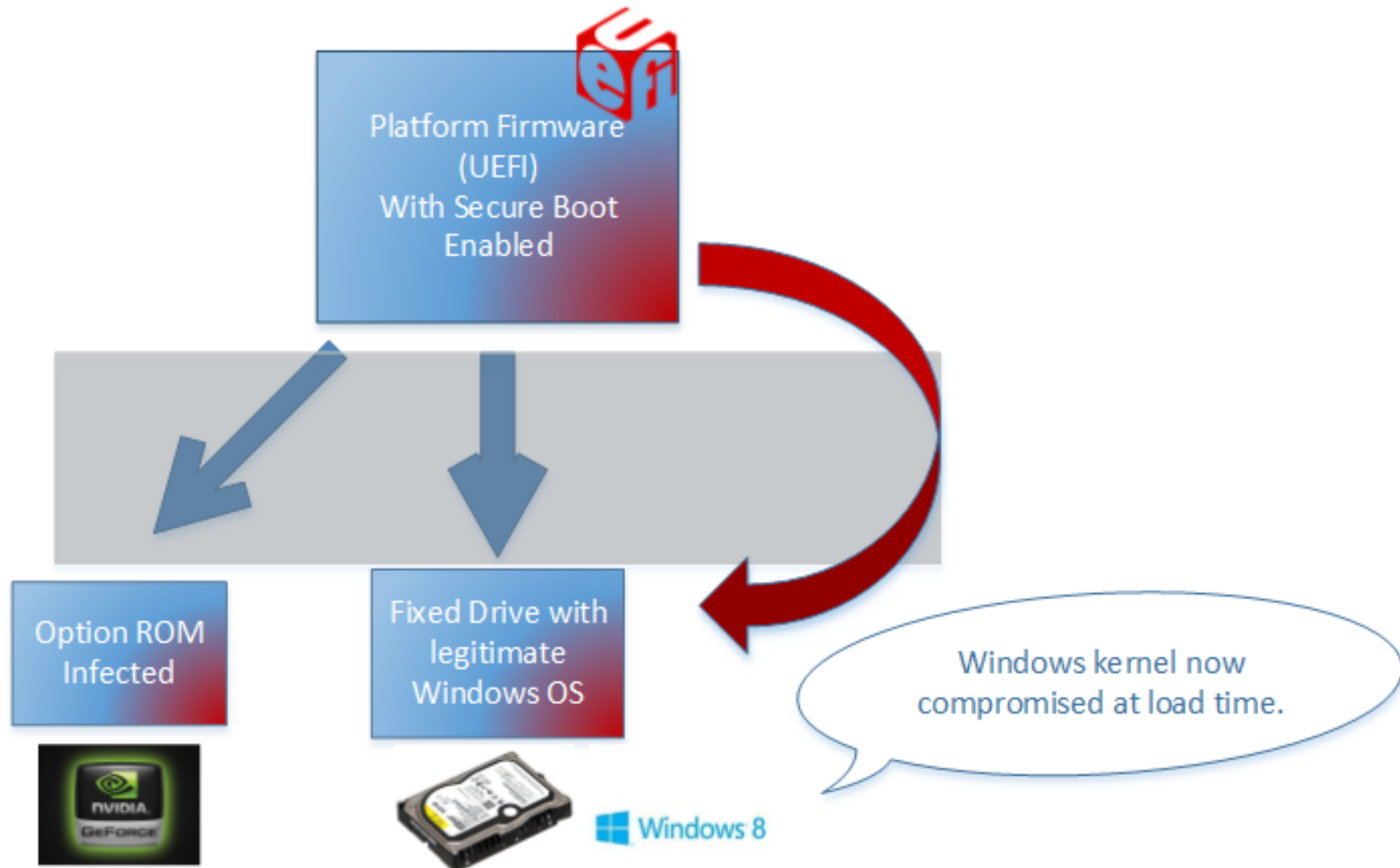
- Malicious option rom hooks some code that legitimate Windows boot loader will call later (think old school BIOS rootkit IVT hooking).

\* The actual flash chip contents aren't modified here, only in-memory copies of relevant firmware code/structures.

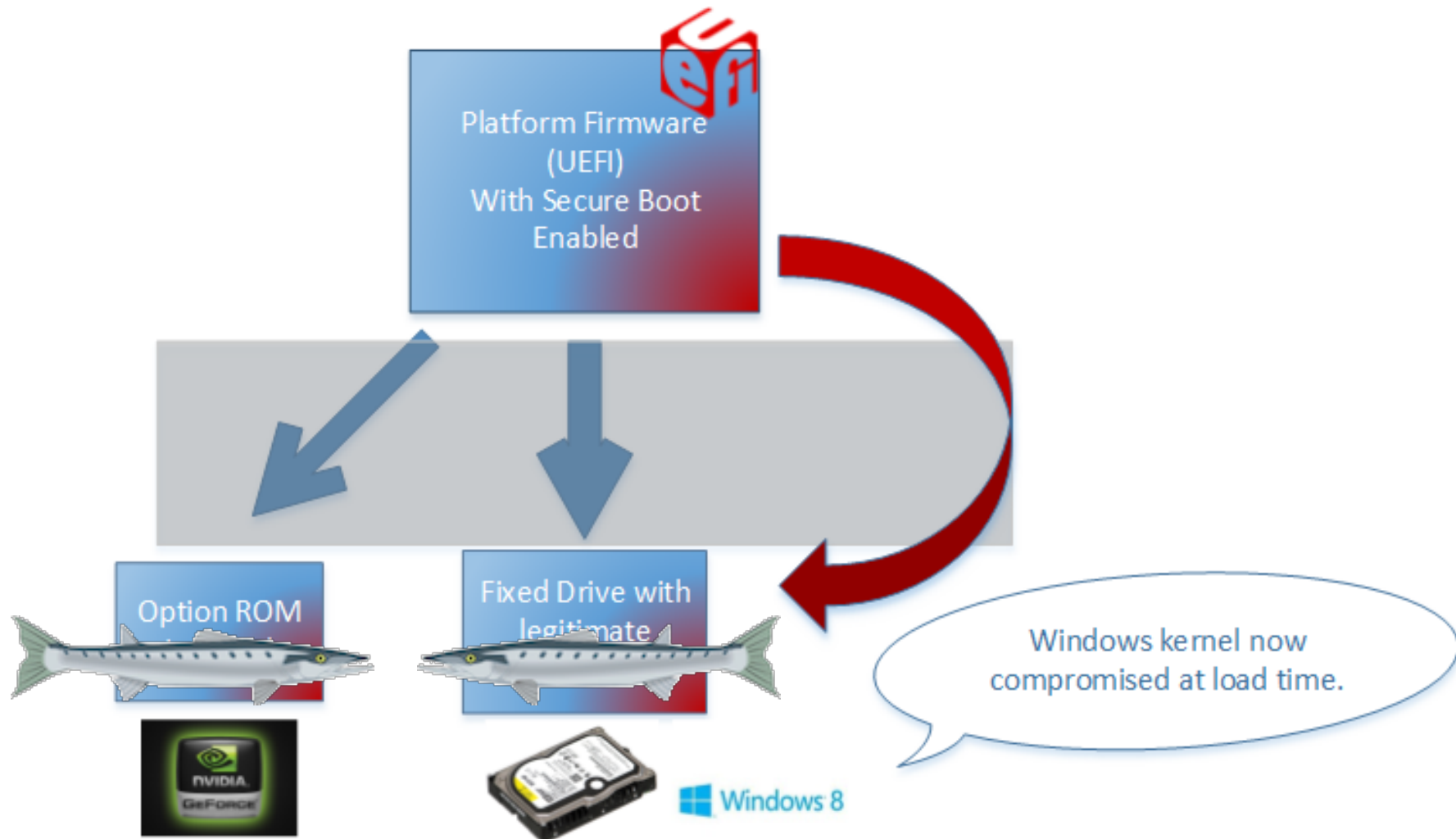




- Legitimate boot loader proceeds to execute as normal.



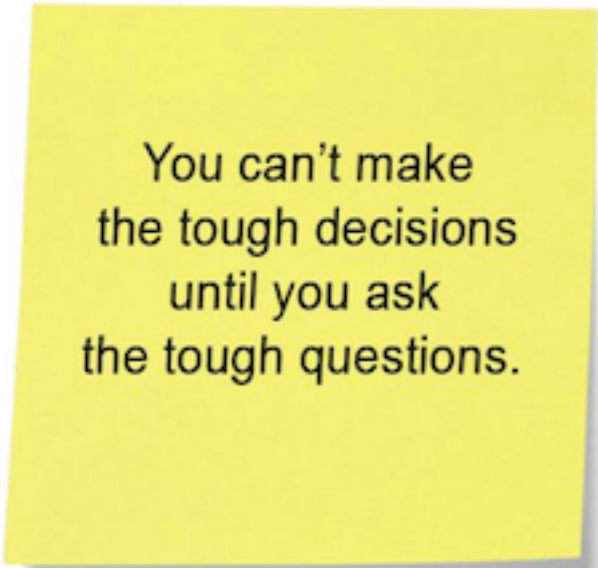
- Boot loader is compromised by BIOS code.
- Operating system is then later compromised.



- Malware has successfully evolved into a more dominant species on the malware food chain.

# Limitless possibilities

- This is just one example of how you could compromise the rest of a secure booted system if an attacker controlled executable was run due to a relaxed secure boot policy.
- There are many other ways an attacker could compromise the rest of the boot chain given this scenario.



You can't make  
the tough decisions  
until you ask  
the tough questions.

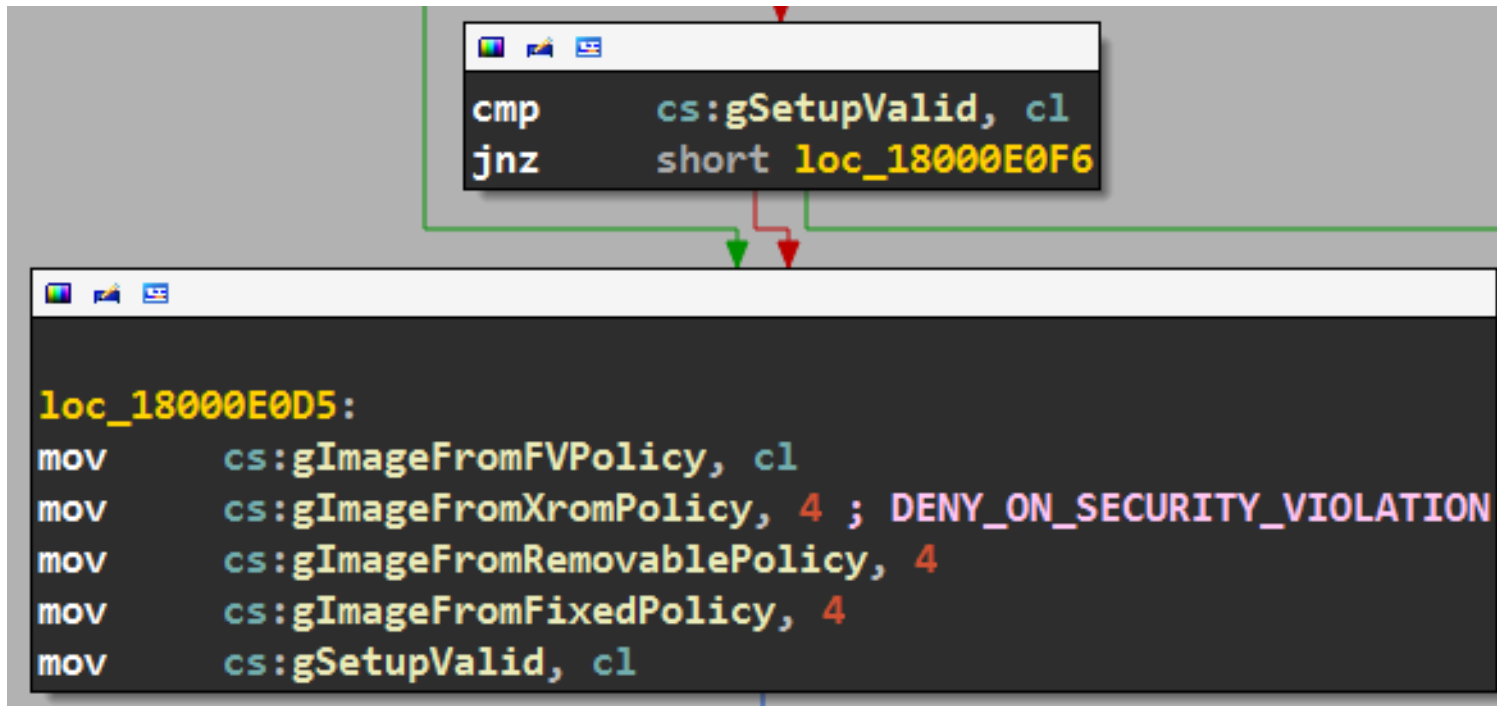
- What does the secure boot policy look like on real systems?
- How can you detect the secure boot policy of the system without manually testing?

```
lea    rdx, [rsp+38h+argSetupVariableSize]
lea    rcx, aSecureboot ; "SecureBoot"
call   sub_18000C874
lea    r9, [rsp+38h+argSetupVariableSize] ; DataSize
lea    rdx, gSetupGuid ; VendorGuid
mov    cs:qword_180048FF8, rax
lea    rax, gSetupVariableData
lea    rcx, VariableName ; "Setup"
mov    [rsp+38h+Data], rax ; Data
mov    rax, cs:gRuntimeServices
xor    r8d, r8d ; Attributes
mov    [rsp+38h+argSetupVariableSize], 0C5Eh
call   [rax+EFI_RUNTIME_SERVICES.GetVariable]
xor    ecx, ecx
test   rdi, rax
jnz    short loc_18000E0D5

cmp    cs:gSetupValid, c1
jnz    short loc_18000E0F6

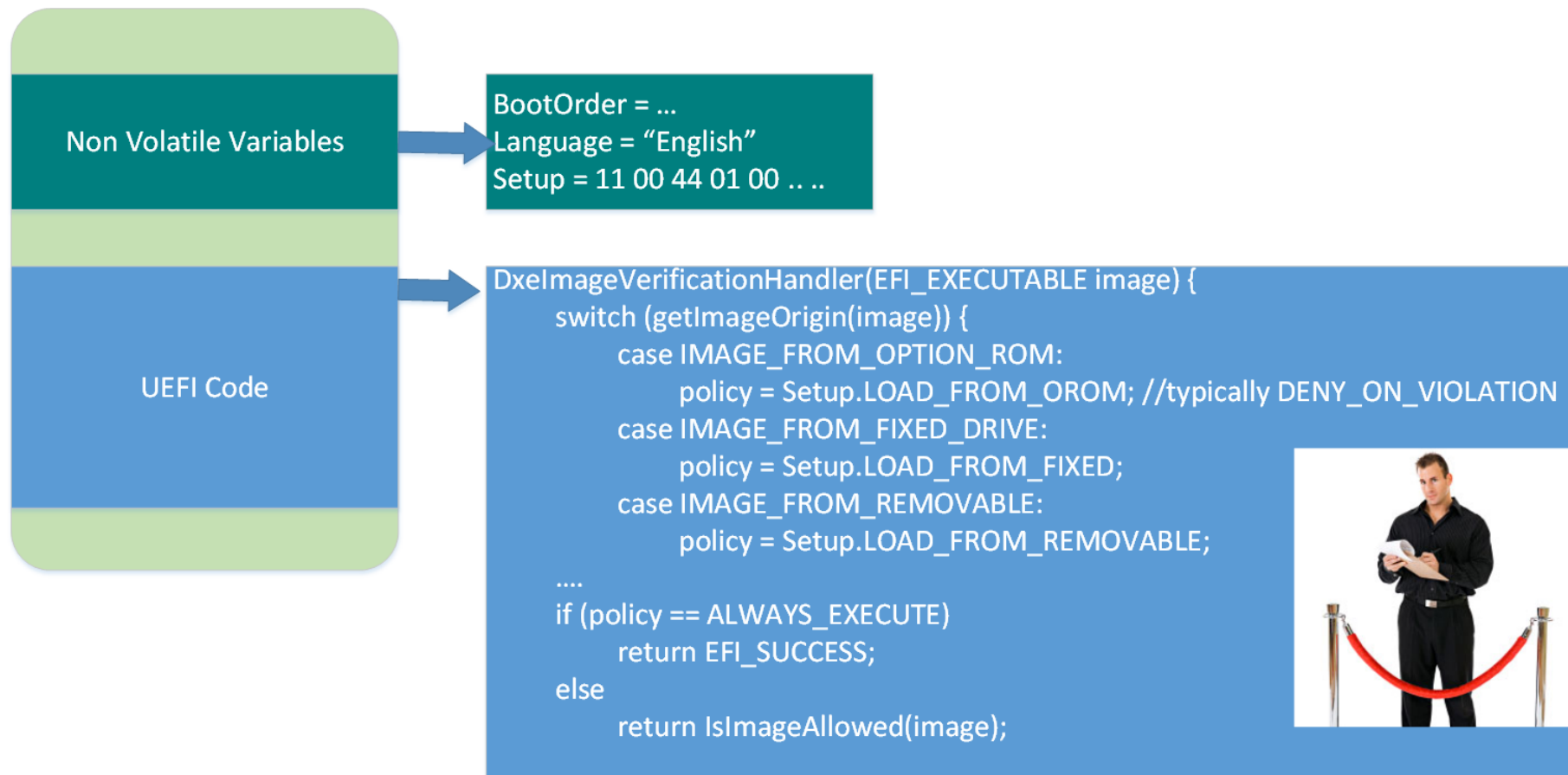
loc_18000E0D5:
mov    cs:gImageFromFVPolicy, c1
mov    cs:gImageFromXromPolicy, 4 ; DENY_ON_SECURITY_VIOLATION
mov    cs:gImageFromRemovablePolicy, 4
mov    cs:gImageFromFixedPolicy, 4
mov    cs:gSetupValid, c1
```

- The above shows disassembly of the secure boot policy initialization on Dell Latitude E6430 BIOS revision A12.



- The Secure Boot policy can be either hardcoded, or derived from the EFI “Setup” variable.
- It turns out the contents of the “Setup” variable makes this determination.

## SPI Flash



- The EFI variables are typically stored on the SPI Flash chip that also contains the platform firmware (UEFI code).



# Cross Roads

A painting of a man in a red cape and a dog on a dirt path. The man is shirtless, wearing a red cape and a hat, and is looking down at the dog. The dog is a brown and white long-haired breed, possibly a Weimaraner, and is looking up at the man. The background shows a dirt path leading through a field with a white picket fence and trees.

- The Dell's I looked at did not have relaxed option rom policies as I had previously hypothesized.
- The EFI "Setup" variable became my next target of attention.

```
Variable NV+RT+BS 'EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9:Setup' DataSize = C5E
00000000: 01 00 00 20 00 00 00 00-00 01 37 37 00 00 05 64 * .....77...d*
00000010: 00 00 00 02 00 00 01 00-00 00 00 00 00 00 01 01 * .....*
00000020: 00 00 01 00 00 00 00 00-00 00 00 01 01 01 01 01 * .....*
00000030: 02 00 00 00 00 02 00 00-01 00 00 01 01 01 01 01 * .....*
00000040: 01 00 01 01 01 00 00 01-00 00 01 01 01 01 01 01 * .....*
00000050: 01 01 04 04 04 00 04 04-04 04 00 00 00 00 00 00 * .....*
00000060: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 * .....*
00000070: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 * .....*
00000080: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 * .....*
```

- Setup variable is marked as Non-Volatile (Stored to flash chip), and as accessible to both Boot services and Runtime Services.
- We should be able to modify it from the operating system.
- It's also quite large... lots of stuff in here!

# Secret Protections?

- The Setup variable is of obvious importance to the security of the platform, which made me wonder if there was some secret protection that would prevent me from writing to it.
- As a first attempt to demonstrate I could write to the Setup variable from Windows, I tried to just zero out the variable. This turned out to be a very bad idea...

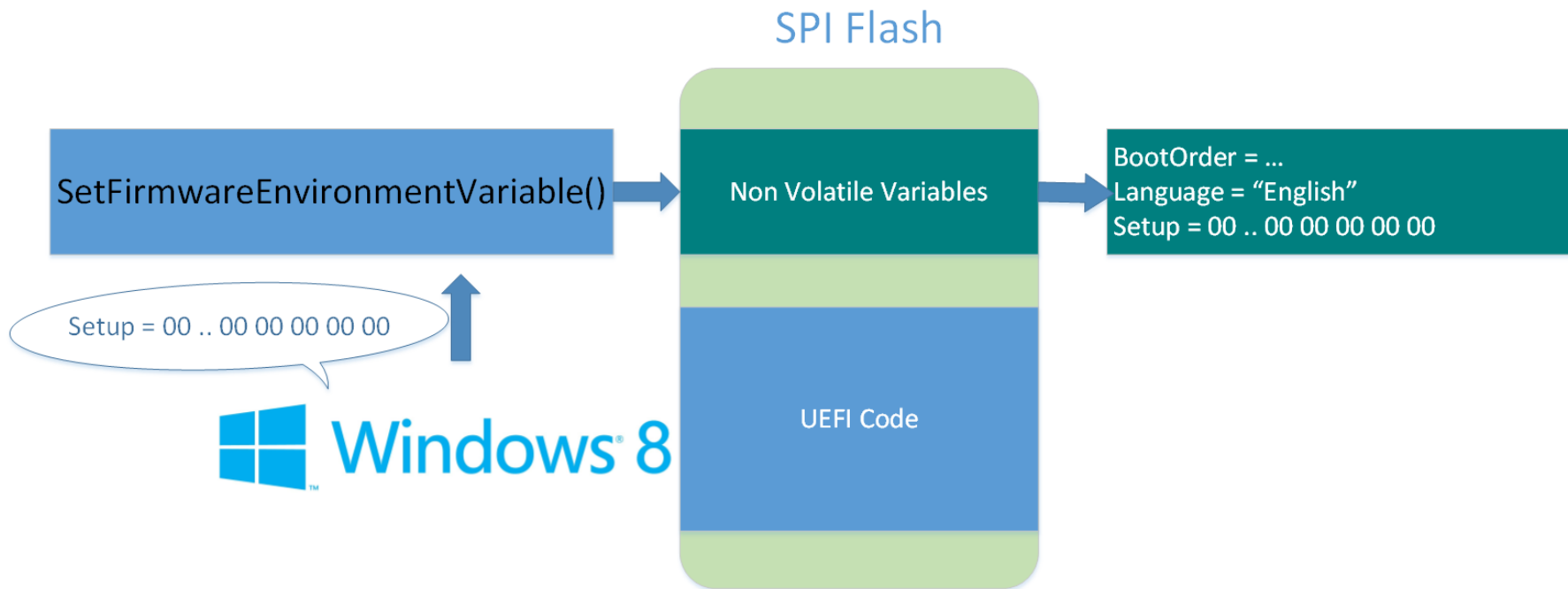
# SetFirmwareEnvironmentVariable function

Sets the value of the specified firmware environment variable.

## Syntax

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

- Starting in Windows 8, Microsoft provides an API for interacting with EFI non volatile variables.

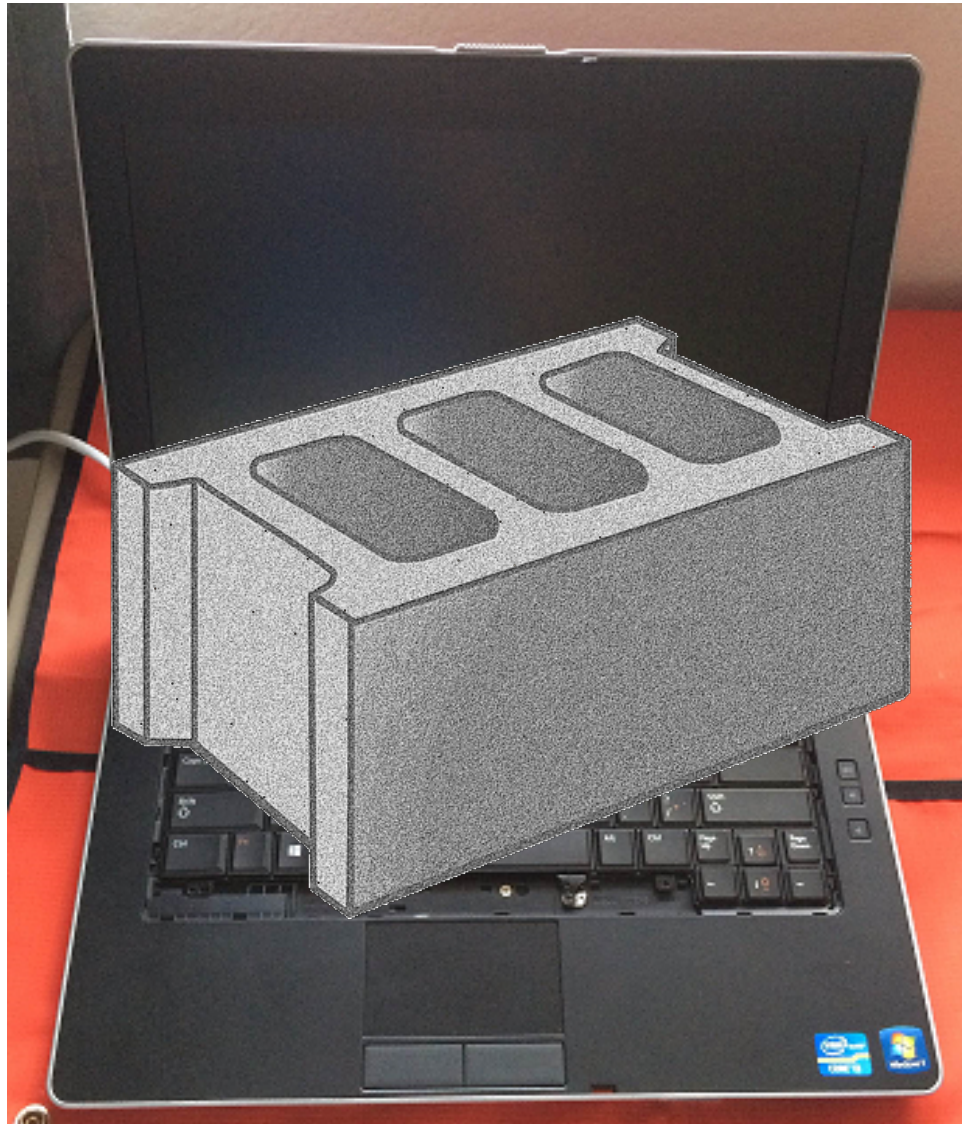


- Any guesses as to what happened?

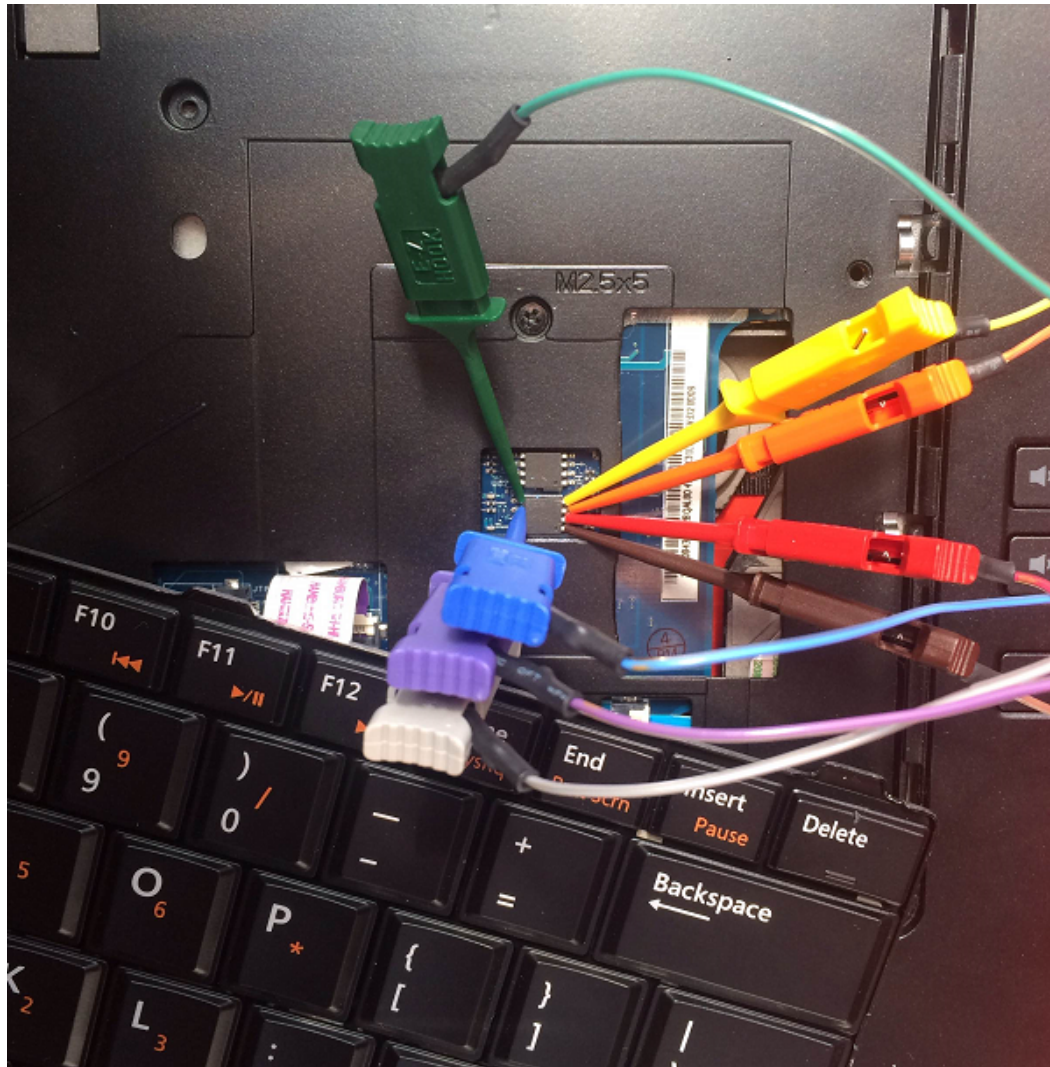




- Hint: you can tell I've already taken the laptop apart (this picture was taken post-surgical-recovery).



- Hint: you can tell I've already taken the laptop apart (this picture was taken post-surgical-recovery).

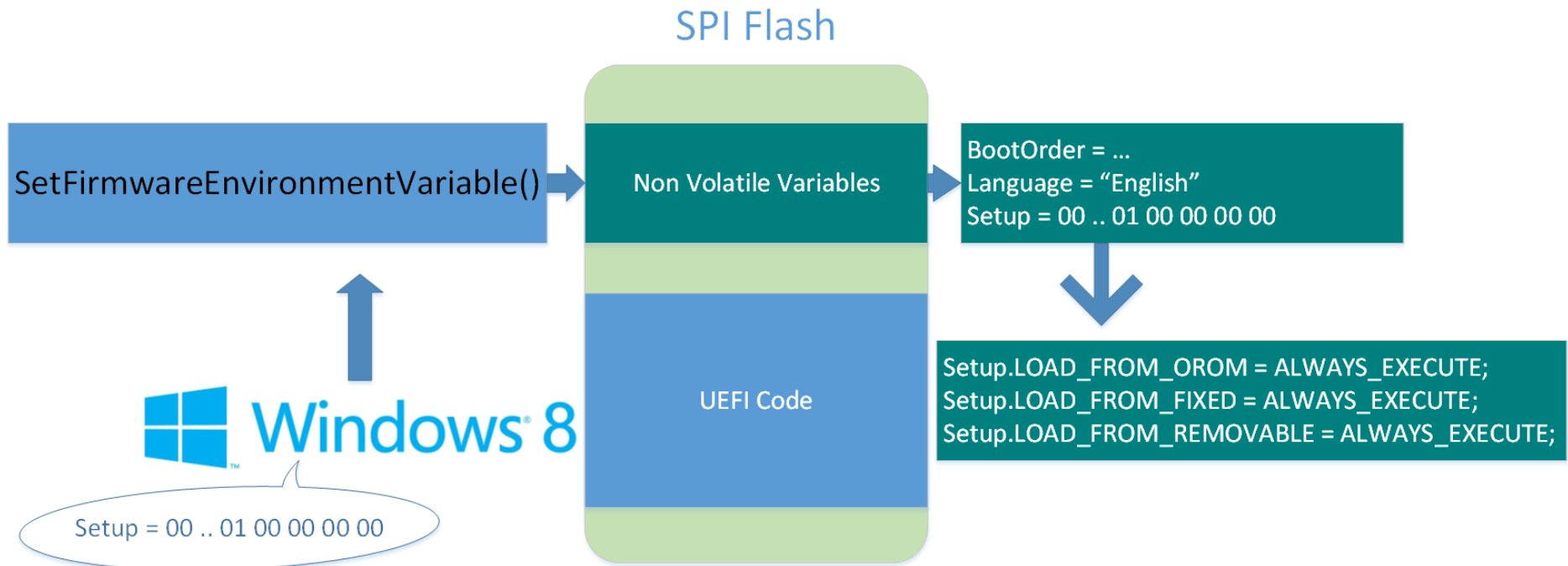


- Having to fix this would be very unpleasant for your organization.

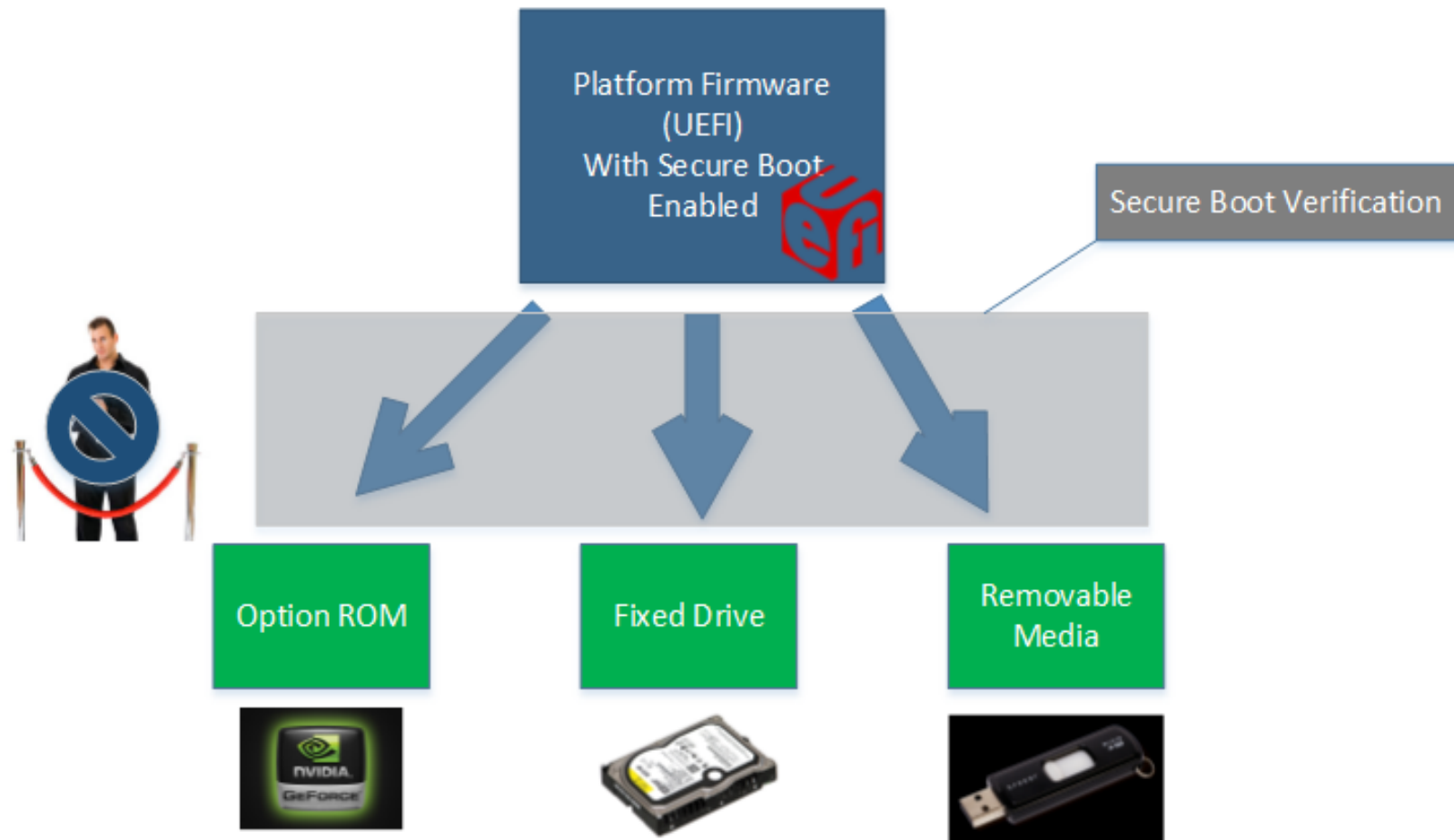


# Back on target

- Although devastating, bricking the firmware is “just” a denial of service attack.
- Let’s get back to trying to break secure boot.



- By twiddling the bits in the Setup variable, we can:
  - Force the firmware to use the Setup defined Secure Boot policy
  - Force the Secure Boot policy to "ALWAYS\_EXECUTE" everything, no matter if it is signed or not.



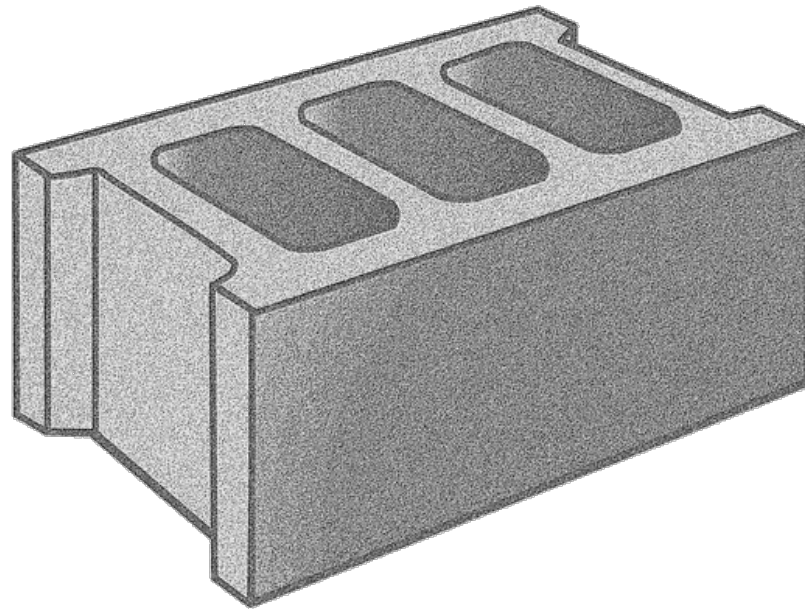
- All executables, no matter their origin or whether or not they are signed are now allowed to execute.
- Secure boot is still “enabled” though it is now effectively disabled.

# Attack 1 Summary

- Malicious Windows 8 privileged process can force unsigned executables to be allowed by Secure Boot
- Exploitable from userland
- Bootkits will now function unimpeded
- Secure Boot will still report itself as enabled although it is no longer “functioning”
- Co-discovered by Intel team

# Attack 1 Corollary

- Malicious Windows 8 privileged process can force can “brick” your computer
- Reinstalling the operating system won't fix this
- Exploitable from userland

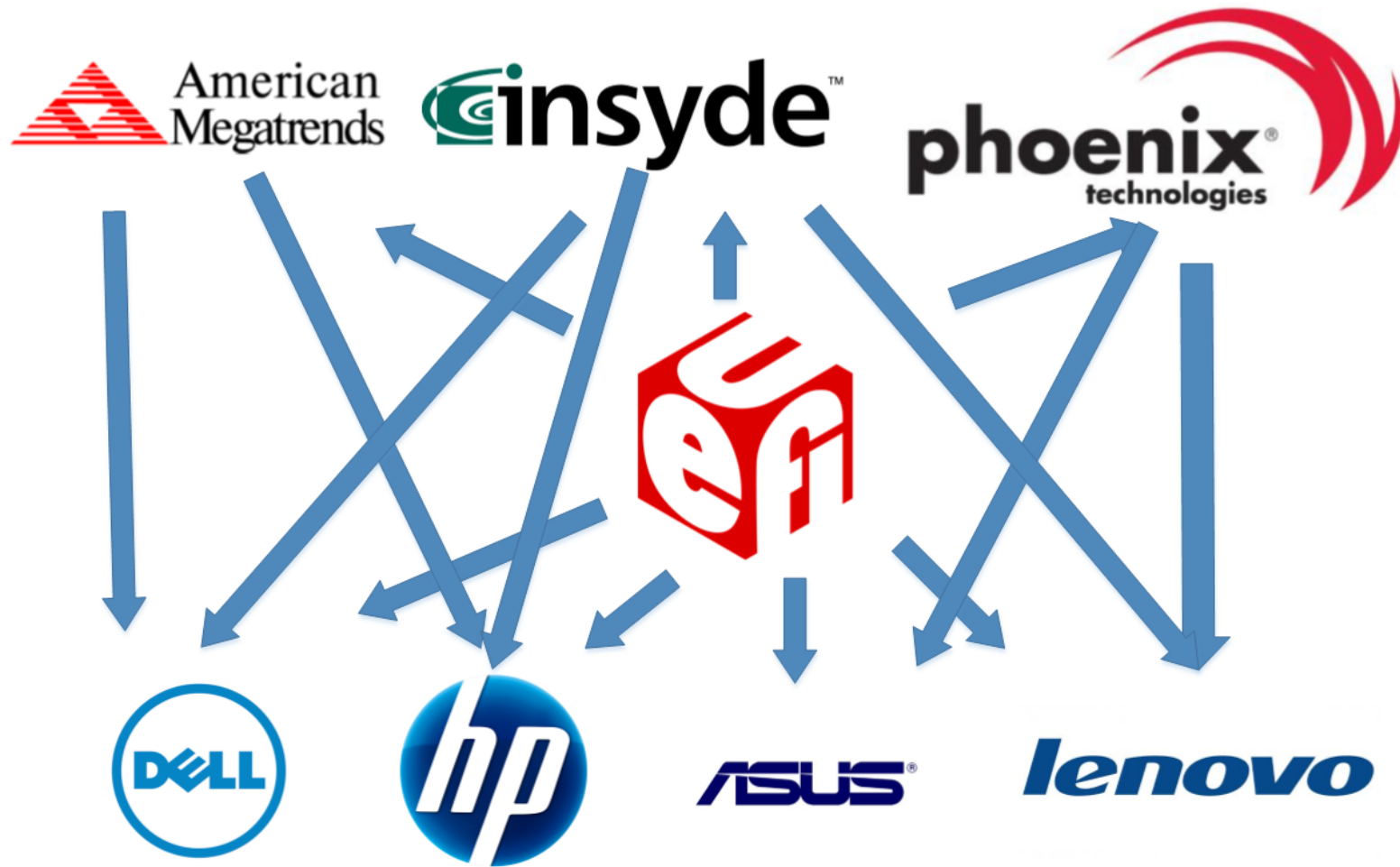


# Who is responsible?

- 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?

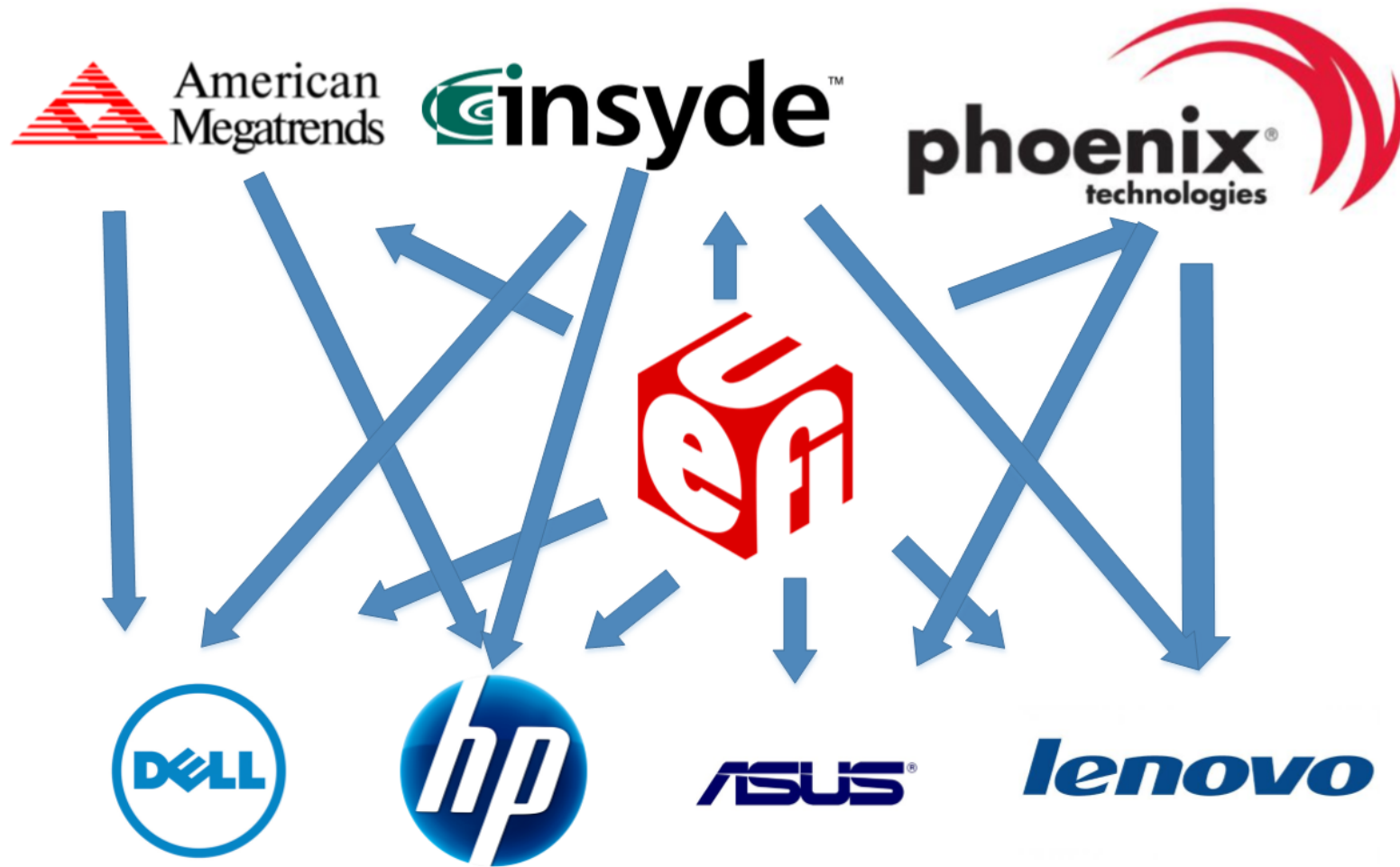
# Who is to responsible?

- 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.



- In theory:
  - UEFI specifies how platform firmware should be developed and provides a reference implementation.
  - Independent BIOS Vendors develop platform firmware based on UEFI specification and reference implementation.
  - OEMs get the firmware development frameworks from the IBVs, and customize it to their own needs.





- In practice:
  - OEMs will use different IBVs for different computer models. Firmware can vary dramatically between computers of the same OEM.
  - Sometimes OEMs won't use IBV code at all, and will instead choose to "roll their own."
  - IBVs may or may not actually use the UEFI reference implementation code.

# Who dunnit?

- Vulnerability does not appear in UEFI reference code.
- Vulnerability affects multiple OEMs, not just Dell.
- Conclusion: Vulnerability was introduced by one of the IBVs. Our analysis suggests that it was probably AMI.

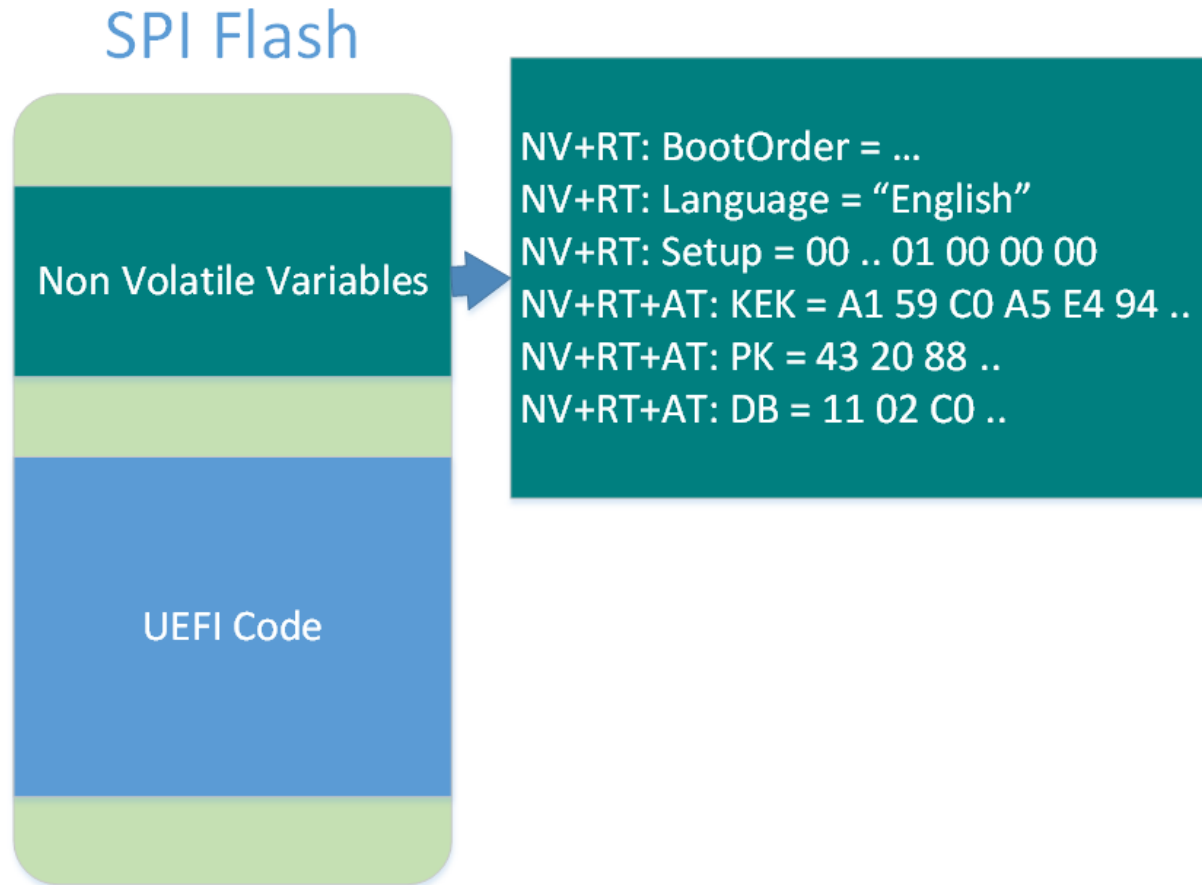


# Authenticated Variables

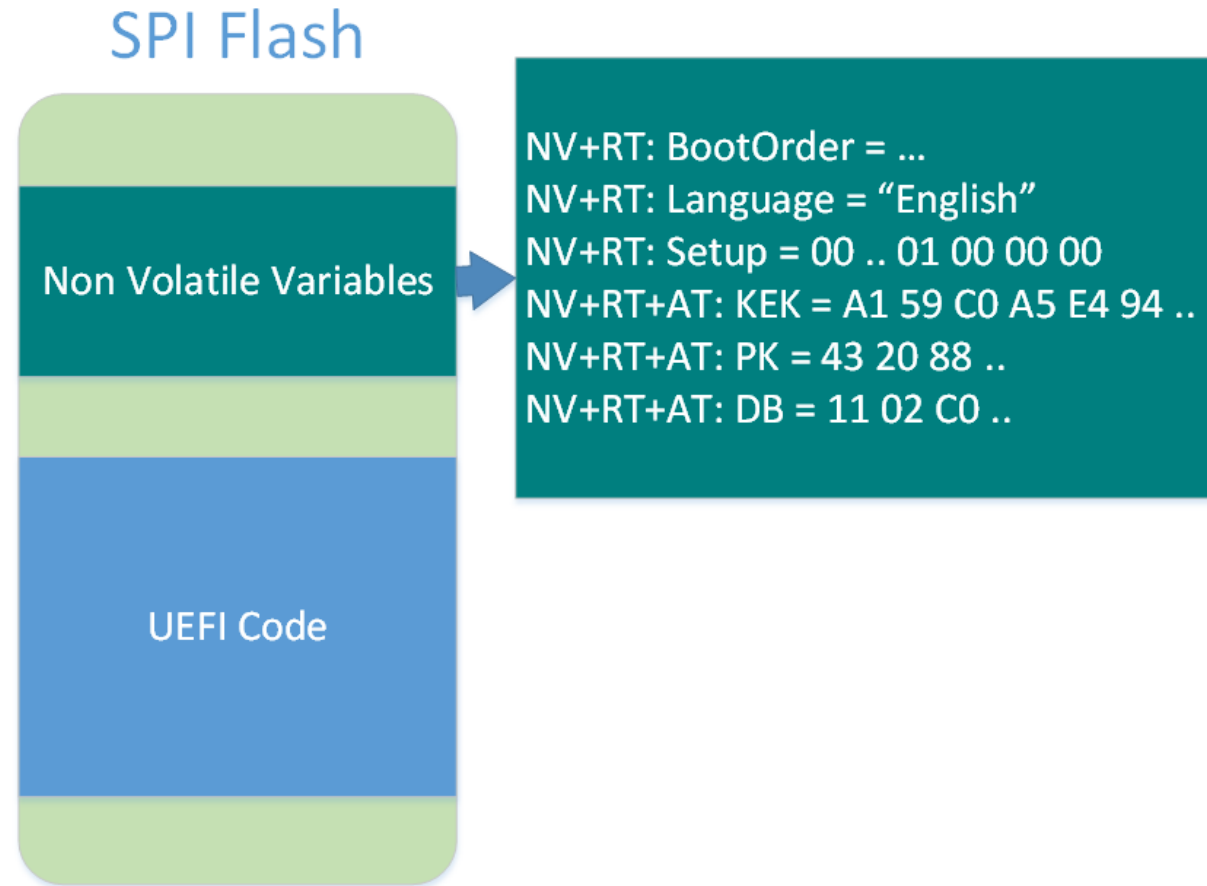
- We've seen how EFI variables that are writeable by the Operating System can potentially be abused.
- But not all EFI variables are arbitrarily writeable during the "runtime phase" of the system.
- Authenticated variables require knowledge of a private key in order to be modified.

# Authenticated Variables

- Authenticated variables store critical Secure Boot information such as:
  - The list of authorized keys by which an EFI executable can be signed in order to function with Secure Boot.
  - A list of “allowed hashes” for EFI executables.
  - A list of “denied hashes” for EFI executables.
  - Etc.
- Authenticated variables co-exist on the SPI flash chip with the platform firmware.



- SPI Flash is getting crowded, we have:
  - The UEFI code which should not be arbitrarily writeable at runtime.
  - Authenticated EFI Variables which are writable if knowledge of a private key is proven.
  - Non-Authenticated Runtime variables, which should be arbitrarily writeable by the operating system.



- How can OEMs implement the different writeability properties of these different components, which all exist on the same medium (SPI Flash)?

# Intel SPI Flash Protection Mechanisms

- Intel provides a number of protection mechanisms that can “lock down” the flash chip.
- It’s up to OEMs/IBVs to use these Intel provided mechanisms in a coherent way to implement things like:
  - UEFI variable protection
  - Signed firmware update requirement

# BIOS\_CNTL

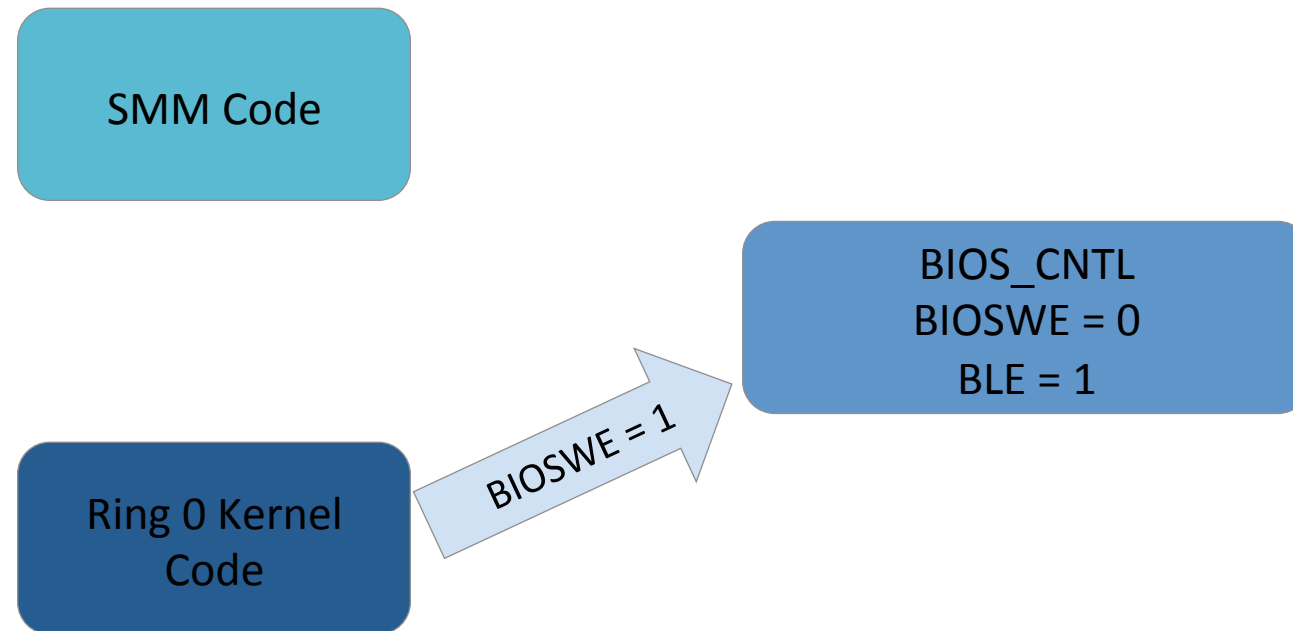
1	<b>BIOS Lock Enable (BLE) — R/WLO.</b> 0 = Setting the BIOSWE will not cause SMIs. 1 = Enables setting the BIOSWE bit to cause SMIs. Once set, this bit can only be cleared by a PLTRST#
0	<b>BIOS Write Enable (BIOSWE) — R/W.</b> 0 = Only read cycles result in Firmware Hub I/F cycles. 1 = Access to the BIOS space is enabled for both read and write cycles. When this bit is written from a 0 to a 1 and BIOS Lock Enable (BLE) is also set, an SMI# is generated. This ensures that only SMI code can update BIOS.

- The above bits are part of the BIOS\_CNTL register on the ICH.
- BIOS\_CNTL.BIOSWE bit enables write access to the flash chip.
- BIOS\_CNTL.BLE bit provides an opportunity for the OEM to implement an SMM routine to protect the BIOSWE bit.

from: <http://www.intel.com/content/www/us/en/chipsets/6-chipset-c200-chipset-datasheet.html>

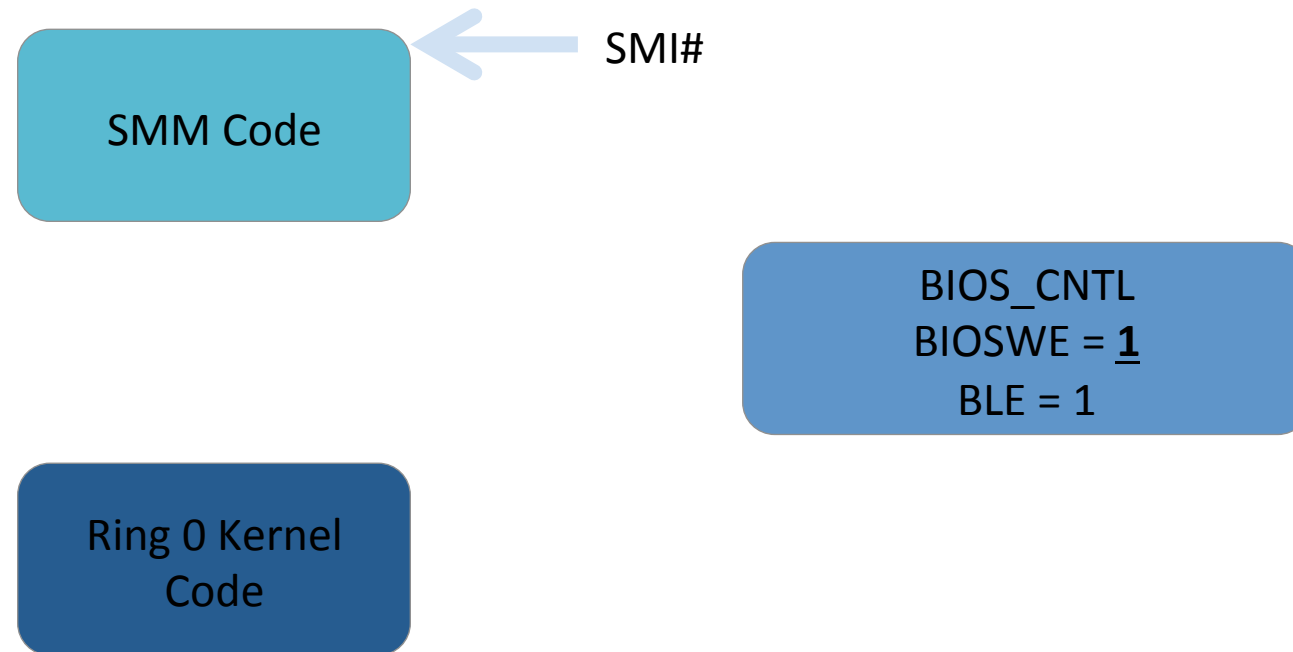


# SMM BIOSWE protection (1 of 2)



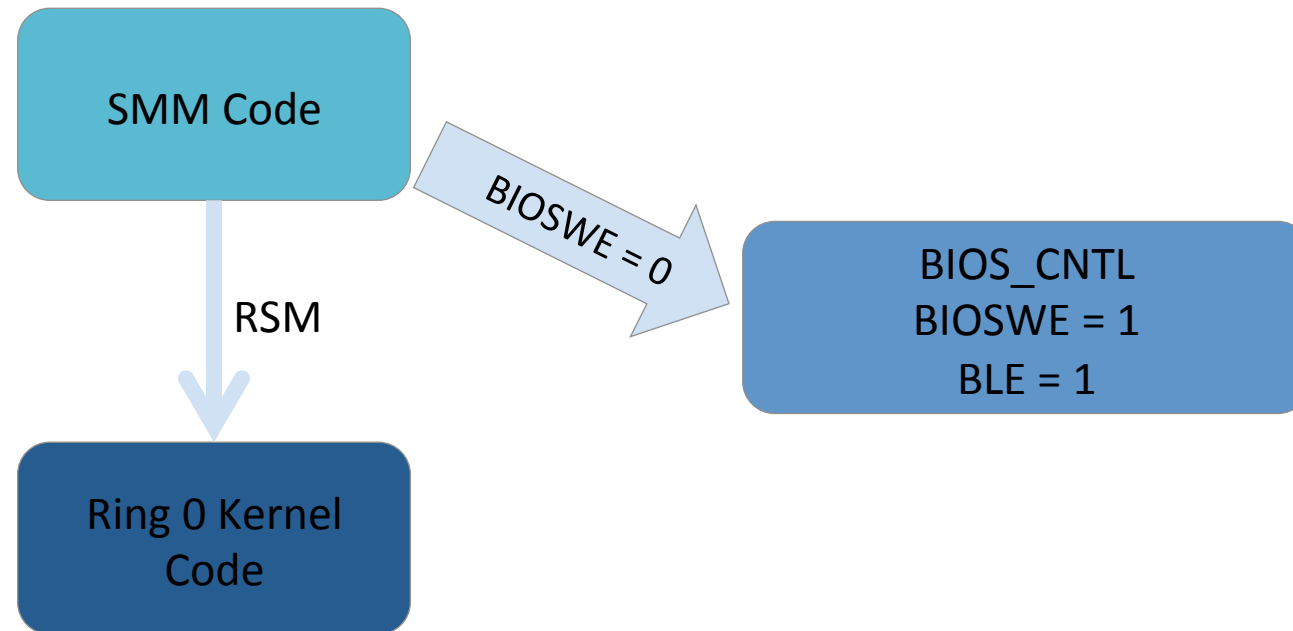
- Here the attacker tries to set the BIOS Write Enable bit to 1 to allow him to overwrite the BIOS chip.

# SMM BIOSWE protection (2 of 2)



- The write to the BIOSWE bit while BLE is 1 causes the CPU to generate a System Management Interrupt (SMI#).

# SMM BIOSWE protection (2 of 2)



- The SMM code immediately writes 0 back to the BIOSWE bit before resuming the kernel code

# SMM BIOSWE protection (2 of 2)

SMM Code

BIOS\_CNTL  
BIOSWE = 0  
BLE = 1

Ring 0 Kernel  
Code

- The end result is that BIOSWE is always 0 when non-SMM code is running.

# Protected Range SPI Flash Protections

## 21.1.13 PR0—Protected Range 0 Register (SPI Memory Mapped Configuration Registers)

Memory Address: SPIBAR + 74h      Attribute: R/W  
Default Value: 00000000h      Size: 32 bits

**Note:** This register can not be written when the FLOCKDN bit is set to 1.

Bit	Description
31	<b>Write Protection Enable</b> — R/W. When set, this bit indicates that the Base and Limit fields in this register are valid and that writes and erases directed to addresses between them (inclusive) must be blocked by hardware. The base and limit fields are ignored when this bit is cleared.
30:29	Reserved
28:16	<b>Protected Range Limit</b> — R/W. This field corresponds to FLA address bits 24:12 and specifies the upper limit of the protected range. Address bits 11:0 are assumed to be FFFh for the limit comparison. Any address greater than the value programmed in this field is unaffected by this protected range.
15	<b>Read Protection Enable</b> — R/W. When set, this bit indicates that the Base and Limit fields in this register are valid and that read directed to addresses between them (inclusive) must be blocked by hardware. The base and limit fields are ignored when this bit is cleared.
14:13	Reserved
12:0	<b>Protected Range Base</b> — R/W. This field corresponds to FLA address bits 24:12 and specifies the lower base of the protected range. Address bits 11:0 are assumed to be 000h for the base comparison. Any address less than the value programmed in this field is unaffected by this protected range.

- Protected Range registers can also provide write protection to the flash chip.

# HSFS.FLOCKDN

## HSFS—Hardware Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers)

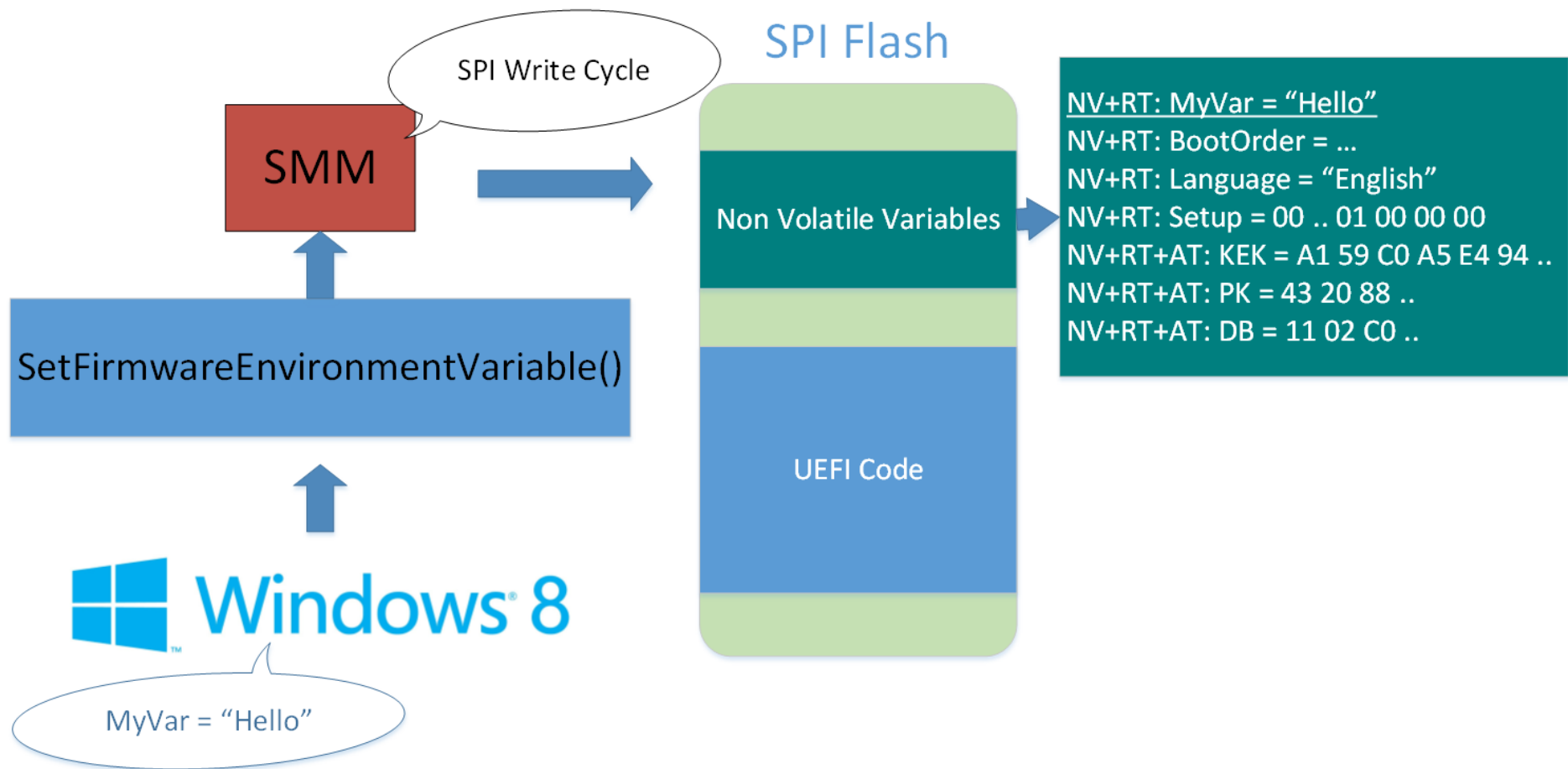
Memory Address: SPIBAR + 04h      Attribute: RO, R/WC, R/W  
Default Value: 0000h              Size: 16 bits

Bit	Description
15	<b>Flash Configuration Lock-Down (FLOCKDN)</b> — R/W/L. When set to 1, those Flash Program Registers that are locked down by this FLOCKDN bit cannot be written. Once set to 1, this bit can only be cleared by a hardware reset due to a global reset or host partition reset in an Intel® ME enabled system.

- HSFS.FLOCKDN bit prevents changes to the Protected Range registers once set.

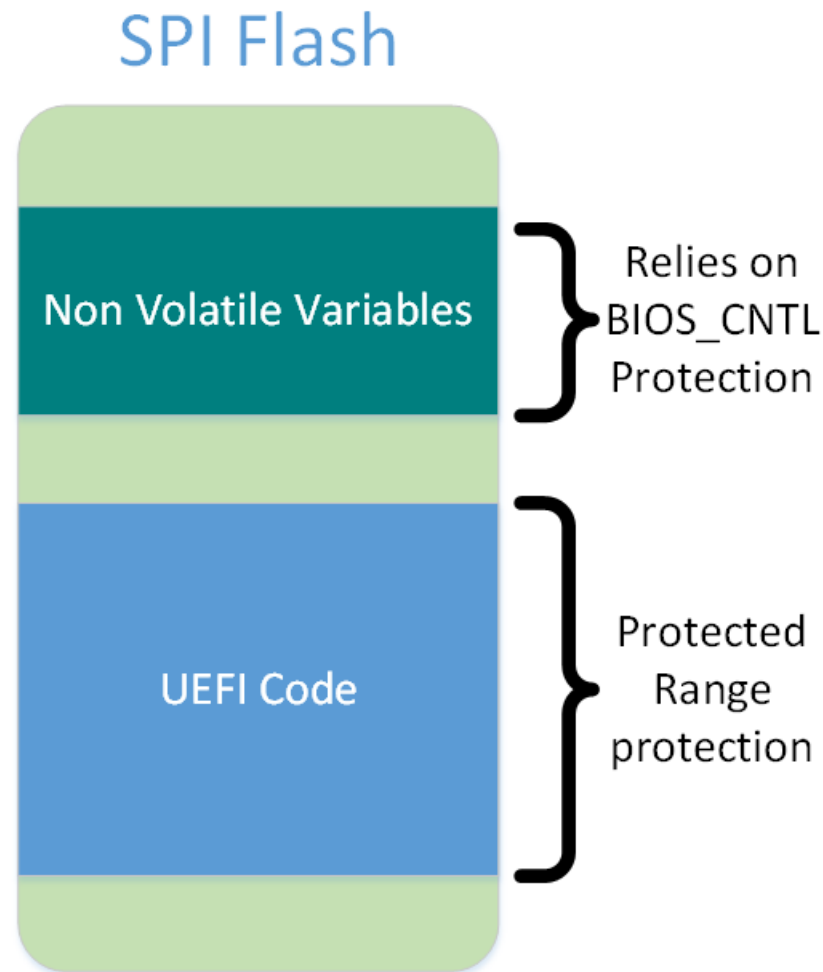
# Intel Protections Summary

- The Protected Range and BIOS\_CNTL registers provide duplicative protection of the SPI flash chip that contains the platform firmware.
  - Protected Range registers can be configured to block all write access to ranges of the SPI Flash.
  - BIOS\_CNTL protection puts SMM in a position to decide who can write to the SPI Flash.

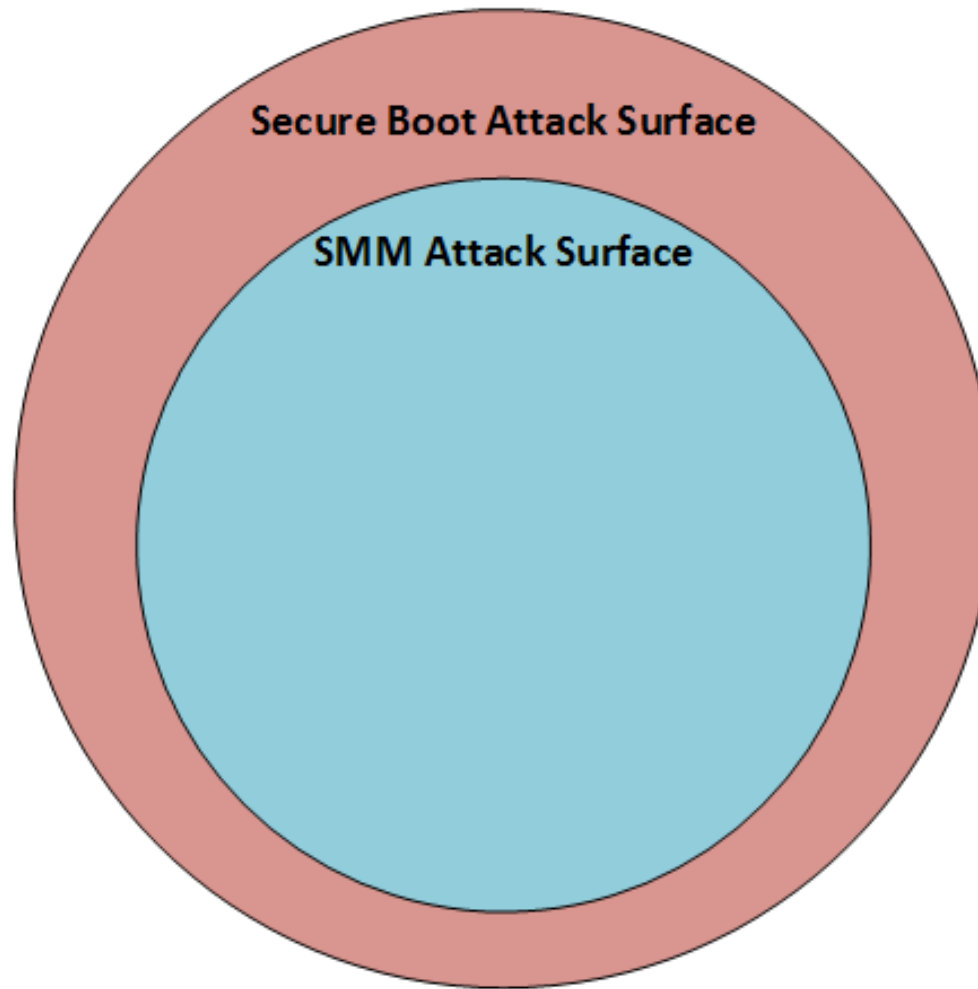


- The architecture of the UEFI variable implementation requires SMM to perform the actual writing.
- The alternative would be to allow Ring0 code write access to the variable region of the SPI Flash, which would allow malicious Ring0 code to bypass Secure Boot





- The key observation is that the variable region of the flash cannot use protected range protection, as it has to be writeable by *someone* at runtime.
- Instead, the variable region has to rely on the “strength” of BIOS\_CNTL protection.



- The Authenticated Variable dependency on BIOS\_CNTL protection dictates that the attack surface against Secure Boot is a superset of the attacks against SMM.

# SMM is bullet proof right?

- SMM Cache Poisoning vulnerabilities
  - Dufлот and Invisible Things Lab
- “Getting into the SMRAM: SMM Reloaded” by Dufлот
- “Attacking Intel BIOS” by Invisible Things Lab
- “Defeating Signed BIOS Enforcement” by MITRE

# SMM and UEFI

```
csc\fvsc\fv3\43172851-cf7e-4345-9fe0-d7012bb17b88\csc iFfsSmm
csc\fvsc\fv3\5552575a-7e00-4d61-a3a4-f7547351b49e\csc SmmBaseRuntime
csc\fvsc\fv3\59287178-59b2-49ca-bc63-532b12ea2c53\csc PchSmbusSmm
csc\fvsc\fv3\6869c5b3-ac8d-4973-8b37-e354dbf34add\csc CmosManagerSmm
csc\fvsc\fv3\753630c9-fae5-47a9-bbbf-88d621cd7282\csc SmmChildDispatcher
csc\fvsc\fv3\77a6009e-116e-464d-8ef8-b35201a022dd\csc DigitalThermalSensorSmm
csc\fvsc\fv3\7fed72ee-0170-4814-9878-a8fb1864dfaf\csc SmmRelocDxe
csc\fvsc\fv3\8d3be215-d6f6-4264-bea6-28073fb13aea\csc SmmThunk
csc\fvsc\fv3\921cd783-3e22-4579-a71f-00d74197fcc8\csc HeciSmm
csc\fvsc\fv3\9cc55d7d-fbff-431c-bc14-334eaea6052b\csc SmmDisp
csc\fvsc\fv3\ab0bad9f7-ab78-491b-b583-c52b7f84b9e0\csc SmmControl
csc\fvsc\fv3\abb74f50-fd2d-4072-a321-cafc72977efa\csc SmmRelocPeim
csc\fvsc\fv3\acaeaa7a-c039-4424-88da-f42212ea0e55\csc PchPcieSmm
csc\fvsc\fv3\bc3245bd-b982-4f55-9f79-056ad7e987c5\csc AhciSmm
csc\fvsc\fv4\025b3ec4-28dc-44ae-8c94-d07563be743f\csc DellFnUsbEmulationSmm
csc\fvsc\fv4\0369cd67-fa74-45a3-bdcb-d25675d5ffde\csc DellO3A30CtrlSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\08abe065-c359-4b95-8d59-c1b58eb657b5\csc IntelLomSmm
csc\fvsc\fv4\099fd87f-4b39-43f6-ab47-f801f99209f7\csc DellDcpRegisterSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\09d2cb46-c303-42c2-9726-5704a1fdfbbd\csc DellVariableSmmWrapper
csc\fvsc\fv4\0d28c529-87d4-4298-8a54-40f22a9fe24a\csc DellDaHddProtectionSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\0d81fdc5-cb98-4b9f-b93b-70a9c0663abe\csc DellDccsSmmDriver
csc\fvsc\fv4\0dde9636-8321-4edf-9f14-0bfca3b473f5\csc DellIntrusionDetectSmm
csc\fvsc\fv4\1137c217-b5bc-4e9a-b328-1e7bcd530520\csc DellThermalDebugSmmDriver
csc\fvsc\fv4\1181e16d-af11-4c52-847e-516dd09bd376\csc DellCenturyRolloverSmm
csc\fvsc\fv4\119f3764-a7c2-4329-b25c-e6305e743049\csc DellSecurityVaultSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\12963e55-5826-469e-a934-a3cbb3076ec5\csc SmmSbAcpi
csc\fvsc\fv4\1478454a-4584-4cca-b0d2-120ace129dbb\csc DellMfgModeSmmDriver
csc\fvsc\fv4\166fd043-ea13-4848-bb3c-6fa295b94627\csc DellVariableSmm-Edk1_06-Pi0_9-Uefi2_1
csc\fvsc\fv4\16c368fe-f174-4881-92ce-388699d34d95\csc SmmGpioPolicy
csc\fvsc\fv4\1afe6bd0-c9c5-44d4-b7bd-8f5e7d0f2560\csc DellDiagsSbControlSmm
csc\fvsc\fv4\26c04cf3-f5fb-4968-8d57-c7fa0a932783\csc SbServicesSmm
csc\fvsc\fv4\2a502514-1e81-4cda-9b50-8970fa4ac311\csc R5U242Smm
csc\fvsc\fv4\2aeda0eb-1392-4232-a4f9-c57a3c2fa2d9\csc BindingsSmm
```

- Of the 495 individual EFI modules on my Dell Latitude E6430, 144 appear to contribute code to SMM...

# Disturbing Trend

- Although SMM should be treated as a “trusted code base” due to its security critical nature, the amount of code executing in SMRAM appears to be on an upward trajectory
- Expect more vulnerabilities here in the future, any of which could be used to bypass Secure Boot.

# Today's Result

- BIOS\_CNTL protection of the SPI Flash can be defeated on a large number of systems by temporarily suppressing SMM.
  - Attack does not require arbitrary code execution in SMM.
- But how can we suppress SMM?



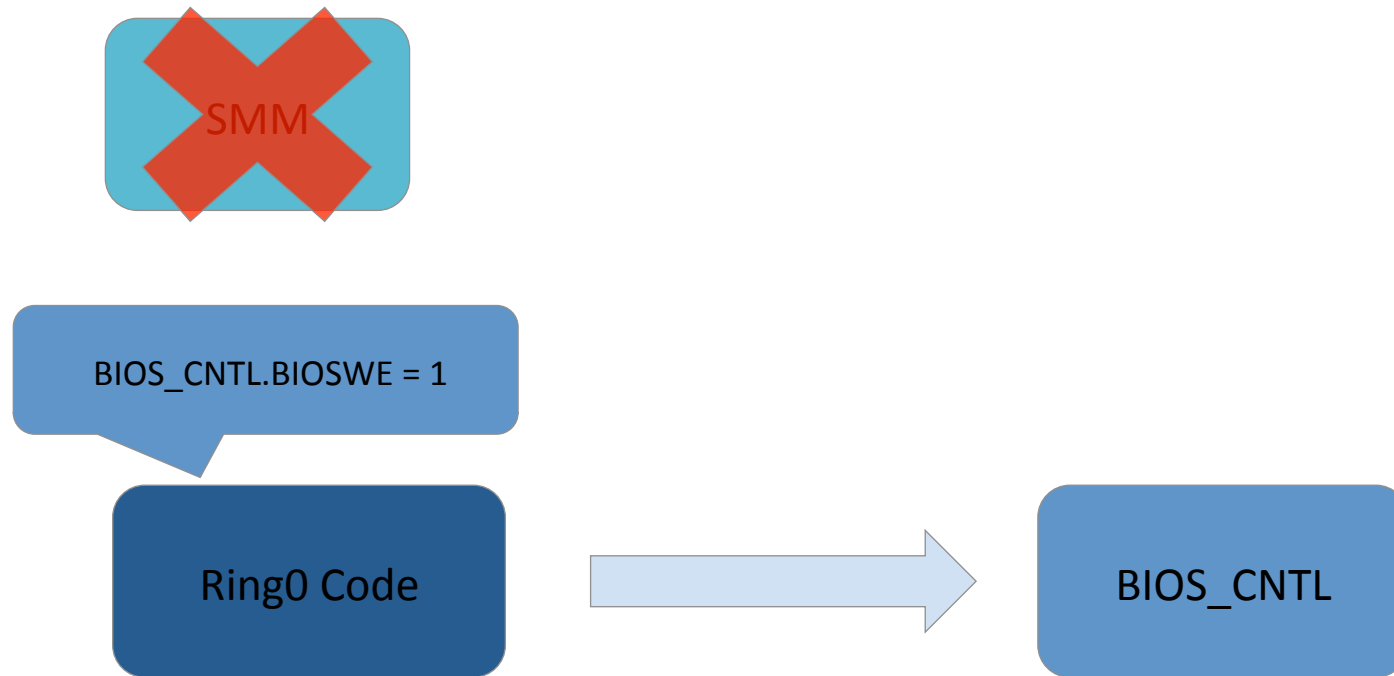
### 13.8.3.7 SMI\_EN—SMI Control and Enable Register

I/O Address:	PMBASE + 30h	Attribute:	R/W, R/WO, WO, R/WL
Default Value:	00000002h	Size:	32 bit
Lockable:	No	Usage:	ACPI or Legacy
Power Well:	Core		

0	<p><b>GBL_SMI_EN</b> — R/WL.</p> <p>0 = No SMI# will be generated by PCH. This bit is reset by a PCI reset event.                  1 = Enables the generation of SMI# in the system upon any enabled SMI event.  <b>NOTE:</b> When the SMI_LOCK bit is set, this bit cannot be changed.</p>
---	---

- On systems without SMI\_LOCK set, we can temporarily disable SMIs and write to flash regions relying on BIOS\_CNTL protection, like the EFI variable region.
- 3216 of 8005 (~40%) systems surveyed did not have SMI\_LOCK set.
  - A greater number could probably be made vulnerable by downgrading the BIOS to a vulnerable revision, which is usually allowed.

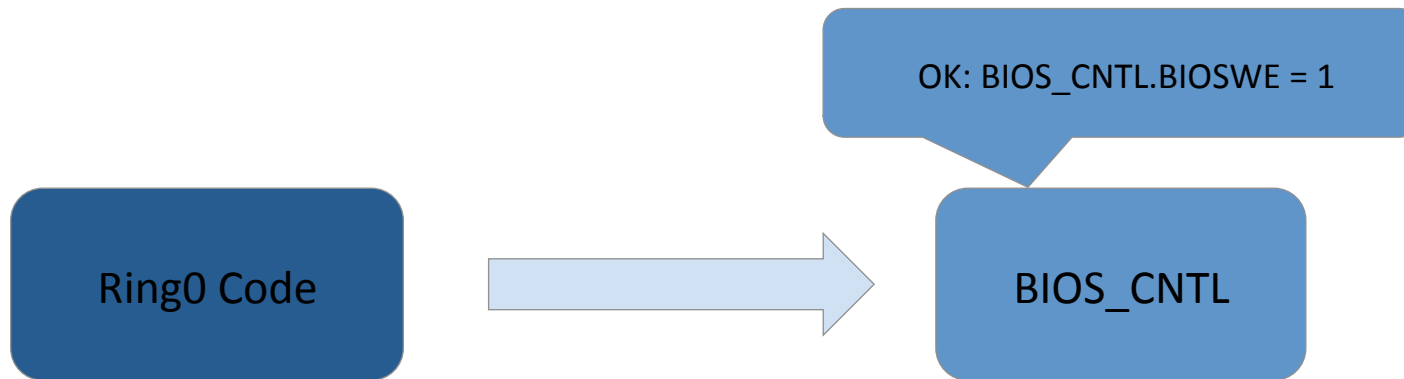
# Disabled BIOSWE protection (1 of 2)



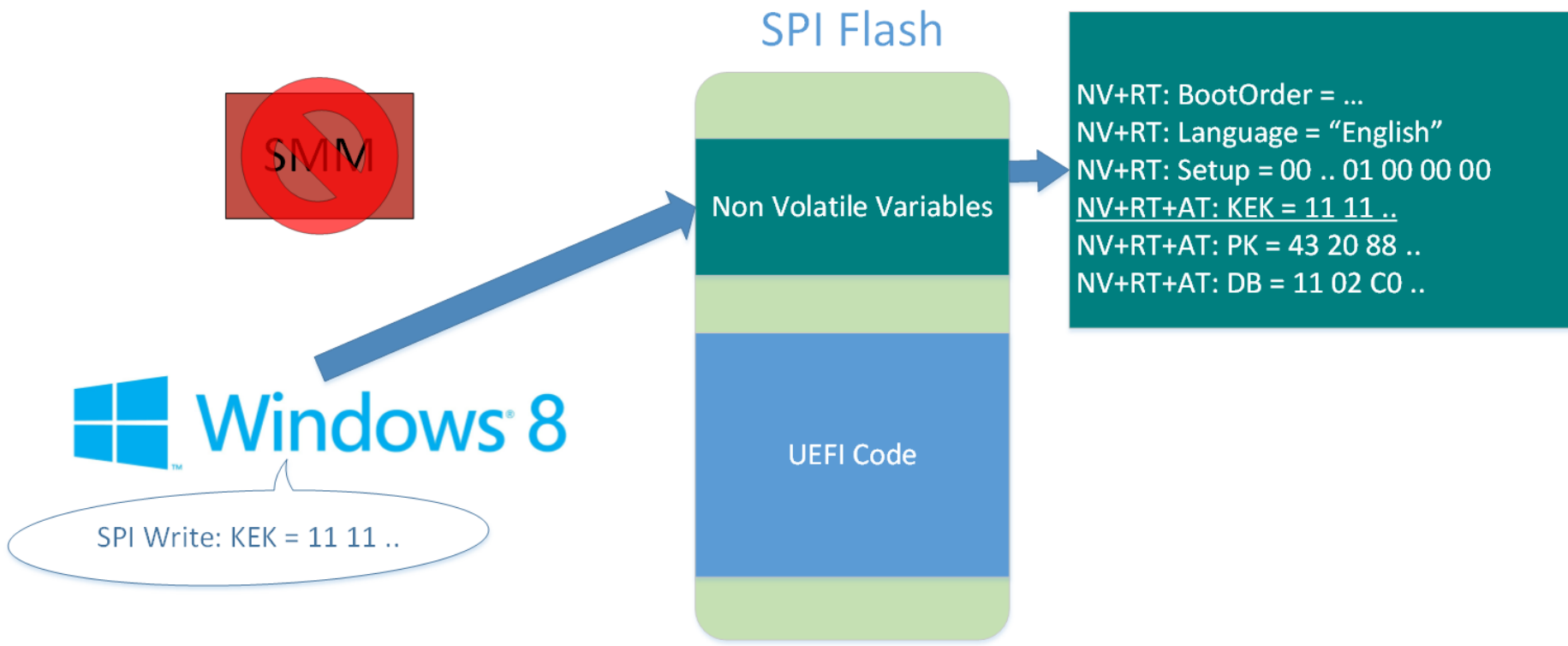
- Again the attacker tries to set the BIOS Write Enable bit to 1 to allow him to overwrite the BIOS chip.



# Disabled BIOSWE protection (2 of 2)



- This time the SMI that protects BIOSWE fails to fire.



- Ring0 can now modify authenticated EFI Variables, which allows trivial bypassing of Secure Boot.

# Demo

- Demo video

# Other ways to suppress SMM?

- Yes.
- Chipset dependent, read your chipset documentation ;)





# Attack 2 Summary Part 1

- Many OEMs are misconfiguring the Intel provided flash protection mechanisms
- On these systems, Secure Boot can be bypassed by a compromised operating system that is able to temporarily suppress SMM and then make direct writes to the authenticated variable region of the flash chip.
- Requires Ring0 code execution in Windows

# Attack 2 Summary Part 2

- It is required that the SPI Flash region associated with EFI variables be writable to at least SMM, thus the protections applied to this region are fundamentally weaker.
- This weakness can often be exploited through SMI suppression, leading to a Secure Boot break.
- OEMs/IBVs could improve the security of this region by setting SMM\_BWP, but most are not doing so currently.



# Another UEFI Attack Surface

- We've now talked about defeating Secure Boot by:
  - Abusing Non-Authenticated EFI Variables that are being used in security critical ways
  - Abusing misconfigured Flash protection mechanisms to overwrite security critical authenticated variables
- Now let's talk about memory corruption attack surface against UEFI...

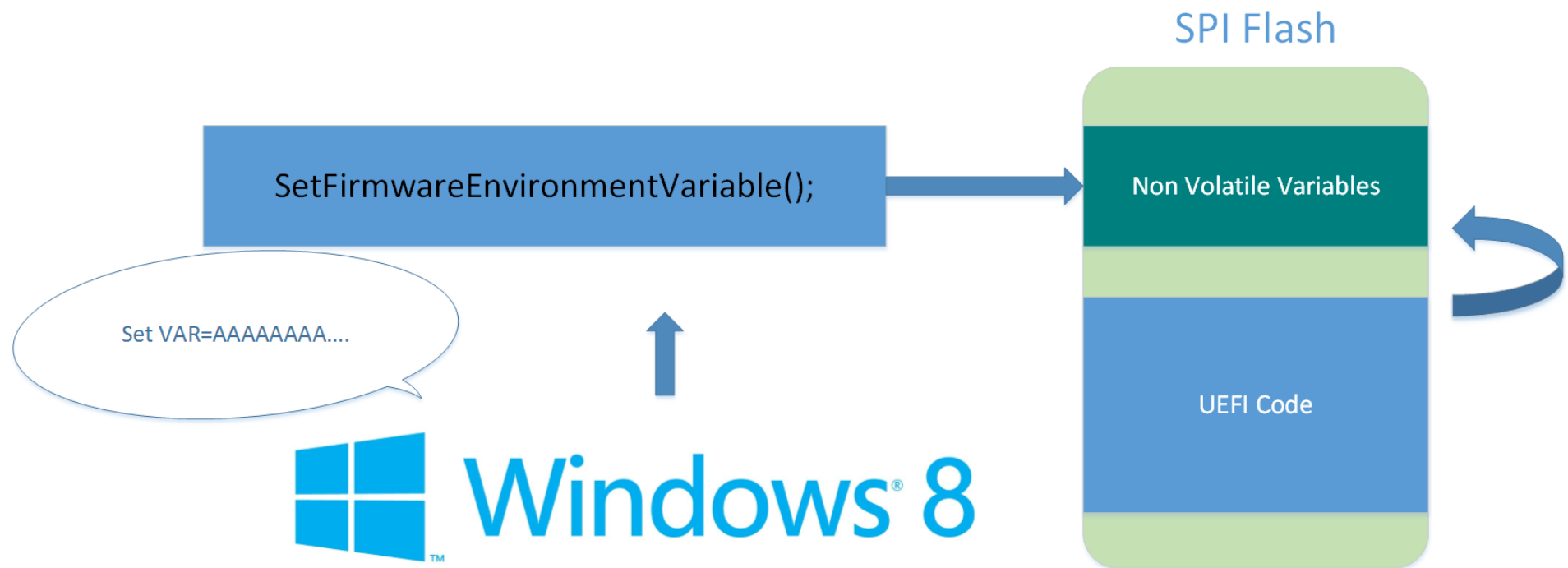
# Remember this?

CVE-ID	
<b><u>CVE-1999-0046</u></b>	<a href="#">Learn more at National Vulnerability Database (NVD)</a> • Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings
Description	
Buffer overflow of rlogin program using <u>TERM environmental variable</u> .	
References	
<b>Note:</b> <a href="#">References</a> are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.	
<ul style="list-style-type: none"><li>• CERT:CA-97.06.rlogin-term</li><li>• XF:rlogin-termbo</li></ul>	
Date Entry Created	
<b>19990827</b>	Disclaimer: The entry creation date may reflect when the CVE-ID was allocated or reserved, and does not necessarily indicate when the vulnerability was discovered, shared with the affected vendor, publicly disclosed, or updated in CVE.
This is an entry on the <a href="#">CVE list</a> , which standardizes names for security problems.	
<b>SEARCH CVE USING KEYWORDS:</b> <input type="text"/> <input type="button" value="Submit"/>	
You can also search by reference using the <a href="#">CVE Reference Maps</a> .	
<b>For More Information:</b> <a href="mailto:cve@mitre.org">cve@mitre.org</a>	

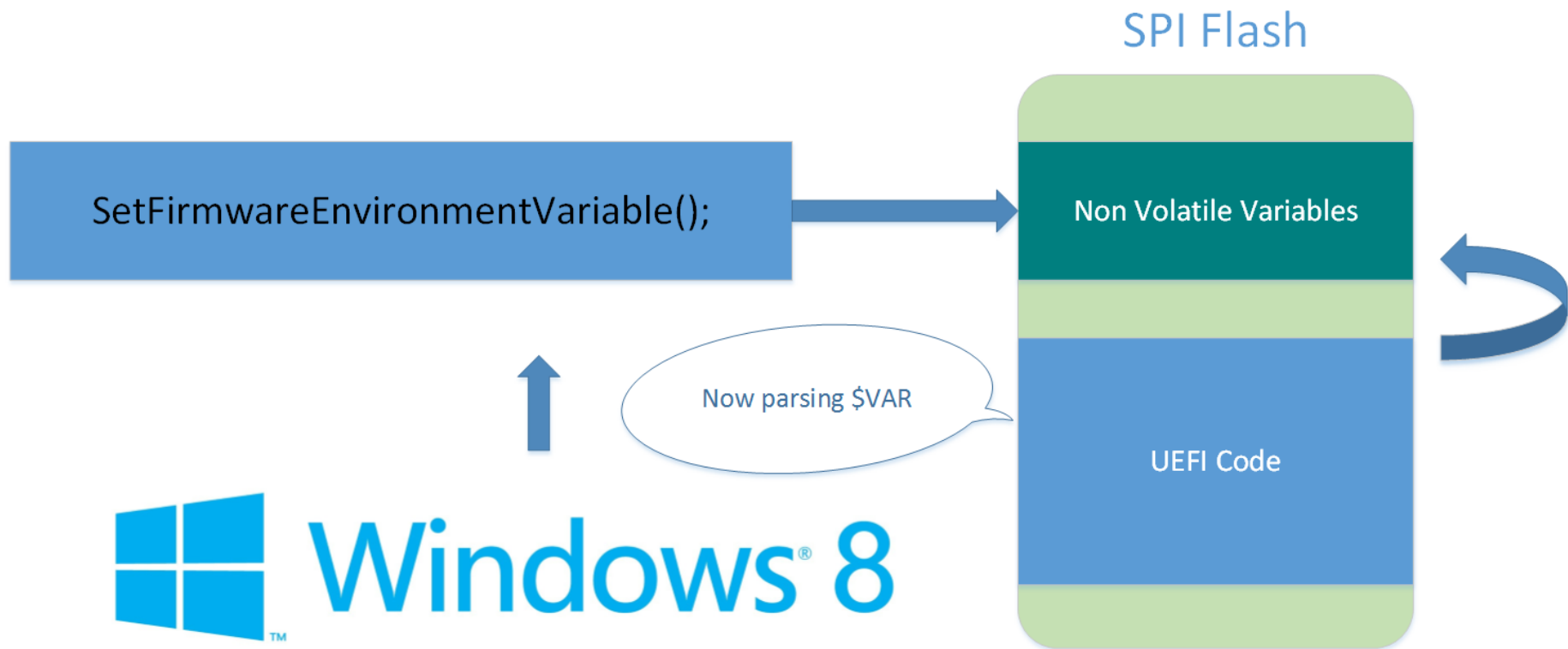
- Long ago there were many privilege escalations in \*nix environments associated with suid programs using and parsing environment variables in unsafe ways.
- Set TERM=AAAAAAAAAAAAAAAAAAAAAAAAA.....



- You thought they were gone/defeated/never to return...
- IT's back.



- Instead of guest to root privilege escalation, this time we are possibly looking at Ring0 to Secure Boot bypass, or beyond...



- There is room for memory corruption vulnerabilities in the UEFI firmware's parsing of attacker controlled variables.
- Vulnerabilities in here would allow an attacker to hijack EIP and circumvent Secure Boot... or worse.
- But how complicated can the parsing of these EFI variables be?

```

Variable NV+RT+BS '78CE2354-CFBC-4643-AEBA-07A27FA892BF:WdtPersistentData' DataSize = 1
00000000: 00                                     *.*
Variable NV+RT+BS 'E6C2F70A-B604-4877-85BA-DEEC89E117EB:PchInit' DataSize = 4
00000000: 00 90 F9 77                             *...w*
Variable NV+RT+BS 'EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9:SetupDptfFeatures' DataSize = 2
00000000: 00 00                                     *..*
Variable NV+RT+BS 'Efi:MonotonicCounter' DataSize = C
00000000: BE FF FF FF 00 00 00 00-00 00 00 FE   *.....*
Variable NV+RT+BS '87F22DCB-7304-4105-BB7C-317143CCC23B:MvcS3Resume' DataSize = BE8

```

```

Variable NV+RT+BS '87F22DCB-7304-4105-BB7C-317143CCC23B:ScramblerBaseSeed' DataSize = 8
00000000: AC 38 00 00 63 DF 00 00-             *.8..c...*
Variable NV+RT+BS 'EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9:SetupSnbPpmFeatures' DataSize = A
00000000: 00 01 00 00 00 00 01 01-00 00       *.....*
Variable NV+RT+BS 'C4975200-64F1-4FB6-9773-F6A9F89D985E:SaPegData' DataSize = 18
00000000: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....*
00000010: 00 00 00 00 00 01 FF 00-           *.....*
Variable NV+RT+BS 'Efi:GnvsAreaVar' DataSize = 8
00000000: 18 4E 44 77 00 00 00 00-           *.NDw....*
Variable NV+RT+BS '01F33C25-764D-43EA-AEEA-6B5A41F3F3E8:SbAslBufferPtrVar' DataSize = 4
00000000: 18 55 55 77                         * ^ w*

```

- Quite complicated, many variables are:
  - large
  - Proprietary, undocumented, vendor specific
  - Used in weird and complicated ways

# Coming Summer 2014

- Extreme Privilege Escalation in Windows 8/  
UEFI
  - Will be using exploits against this EFI variable attack surface to hijack control of EIP very early in the system, allowing an attacker to pivot to even more privileged parts of the system
  - Appearing in Blackhat USA and DEF CON

# Final Thoughts

- UEFI and Secure Boot are good things ultimately for computer security
  - Attack vectors that were previously open are being addressed
  - An open source reference implementation should allow us to eventually have something like a trusted code base. This is superior to the “everyone roll’s their own proprietary voodoo” that was prevalent in conventional BIOS.
- UEFI still has growing pains it will have to endure



# Related Work

- “A Tale of One Software Bypass of Windows 8 Secure Boot”
  - Black Hat USA 2013
  - First published attack against Secure Boot
  - If SPI Flash is writable, malicious code with IO access can defeat Secure Boot.
- “Defeating Signed BIOS Enforcement”
  - EkoParty, HITB, PacSec 2013
  - BIOS\_CNTL protection of the SPI Flash can be defeated by SMM present malware.

# Acknowledgements

- Intel PSIRT team
- Rafal Wojtczuk
- Rick Martinez
- EFIPWN developers

ANY  
QUESTIONS  
?