CanSecWest 2014

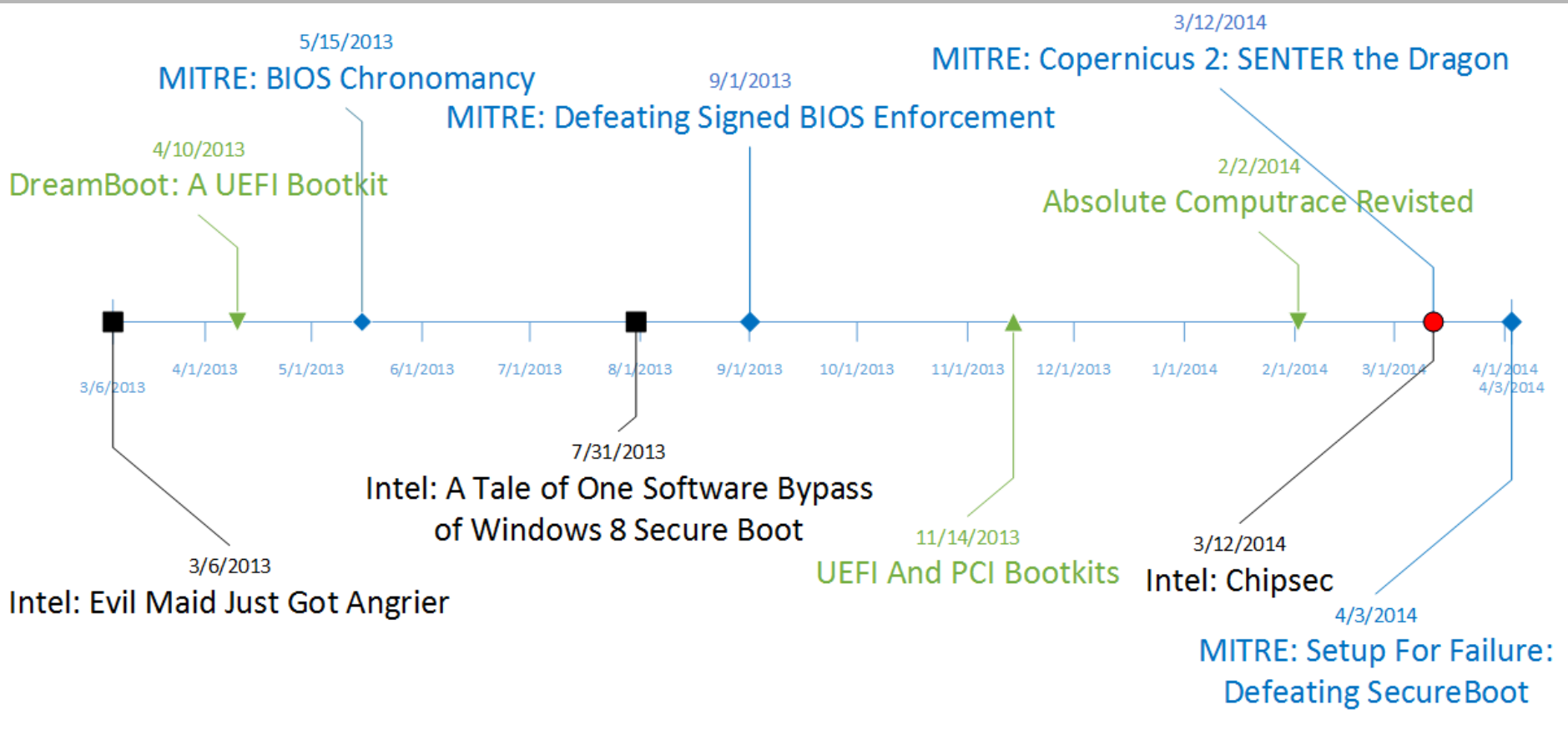
ALL YOUR BOOT ARE BELONG TO US

MITRE Corp
Corey Kallenberg
Xeno Kovah
John Butterworth
Sam Cornwell

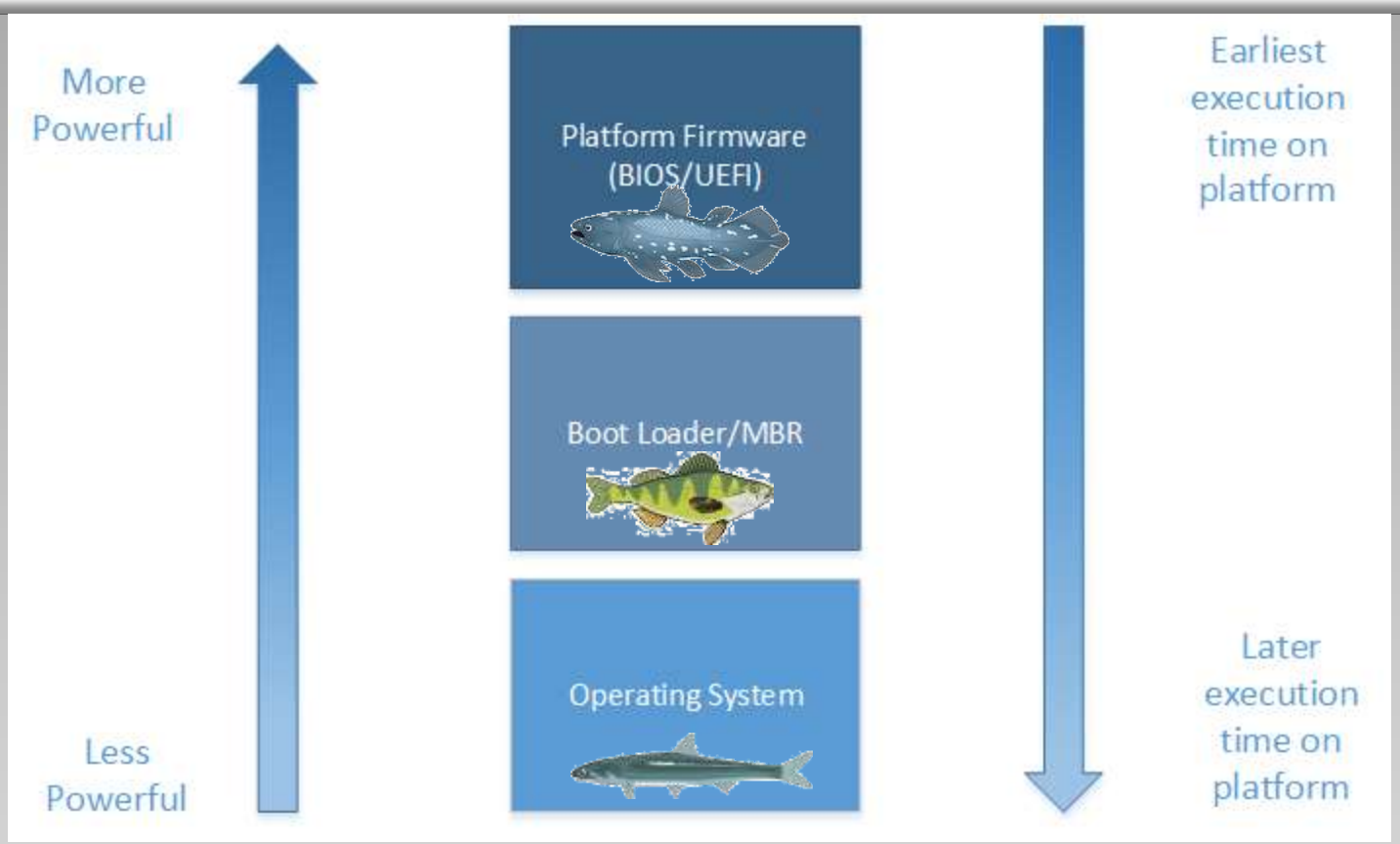
- History and UEFI Bootkits
- OS Secure Boot
- Attacking Securely Booted OS From User Land
- Introducing CHIPSEC

Agenda

Intel Security and MITRE have been working on Platform Firmware security for some time now



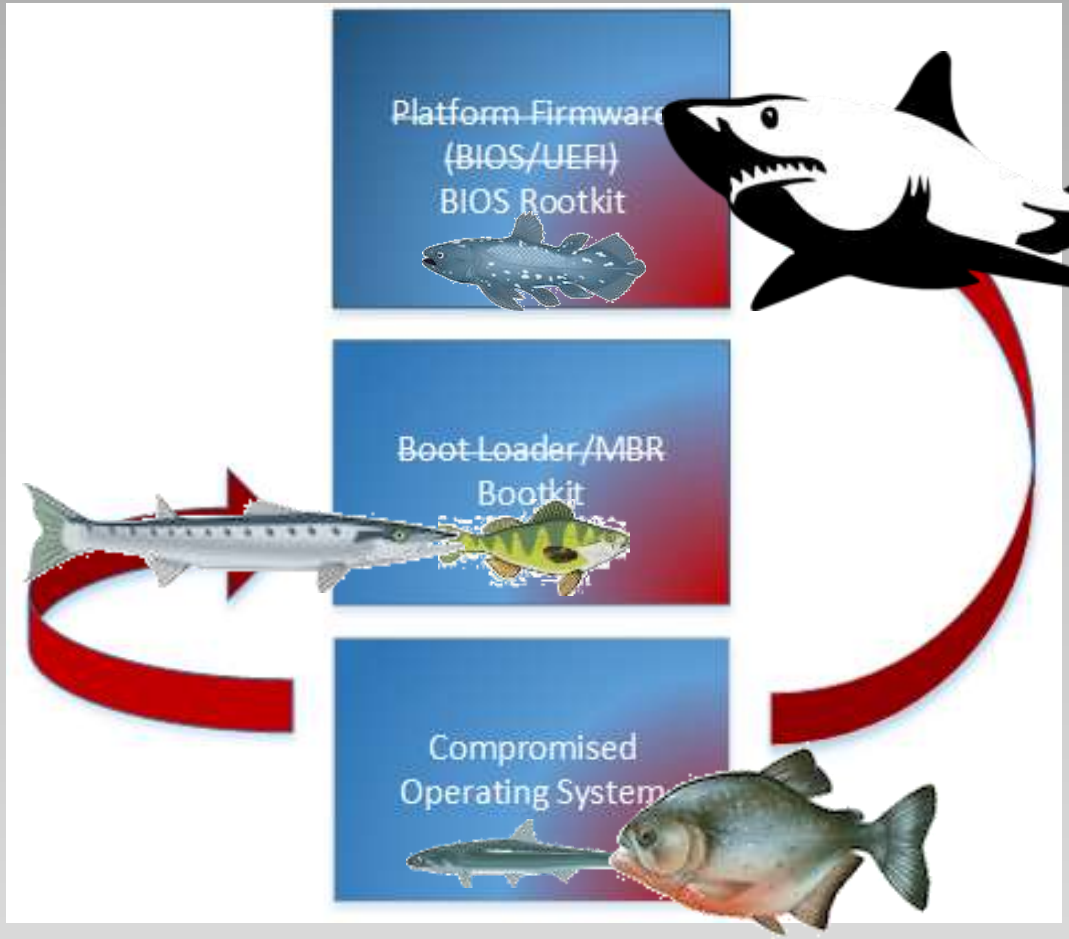
The intersection of our research was inevitable



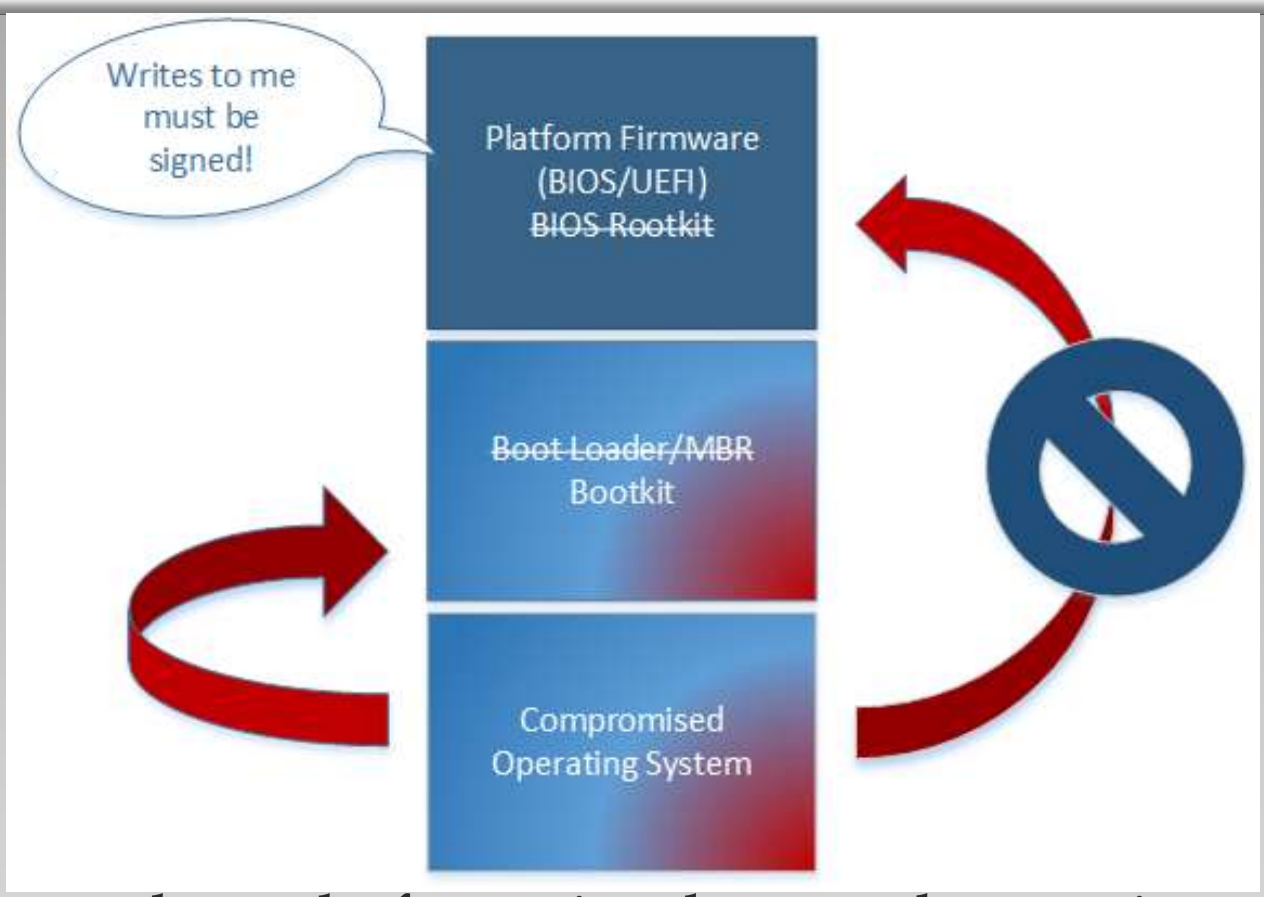
- Rootkits that execute earlier on the platform are in a position to compromise code that executes later on the platform, making earliest execution desirable

The Malware Food Chain

- 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



Blood in the Water



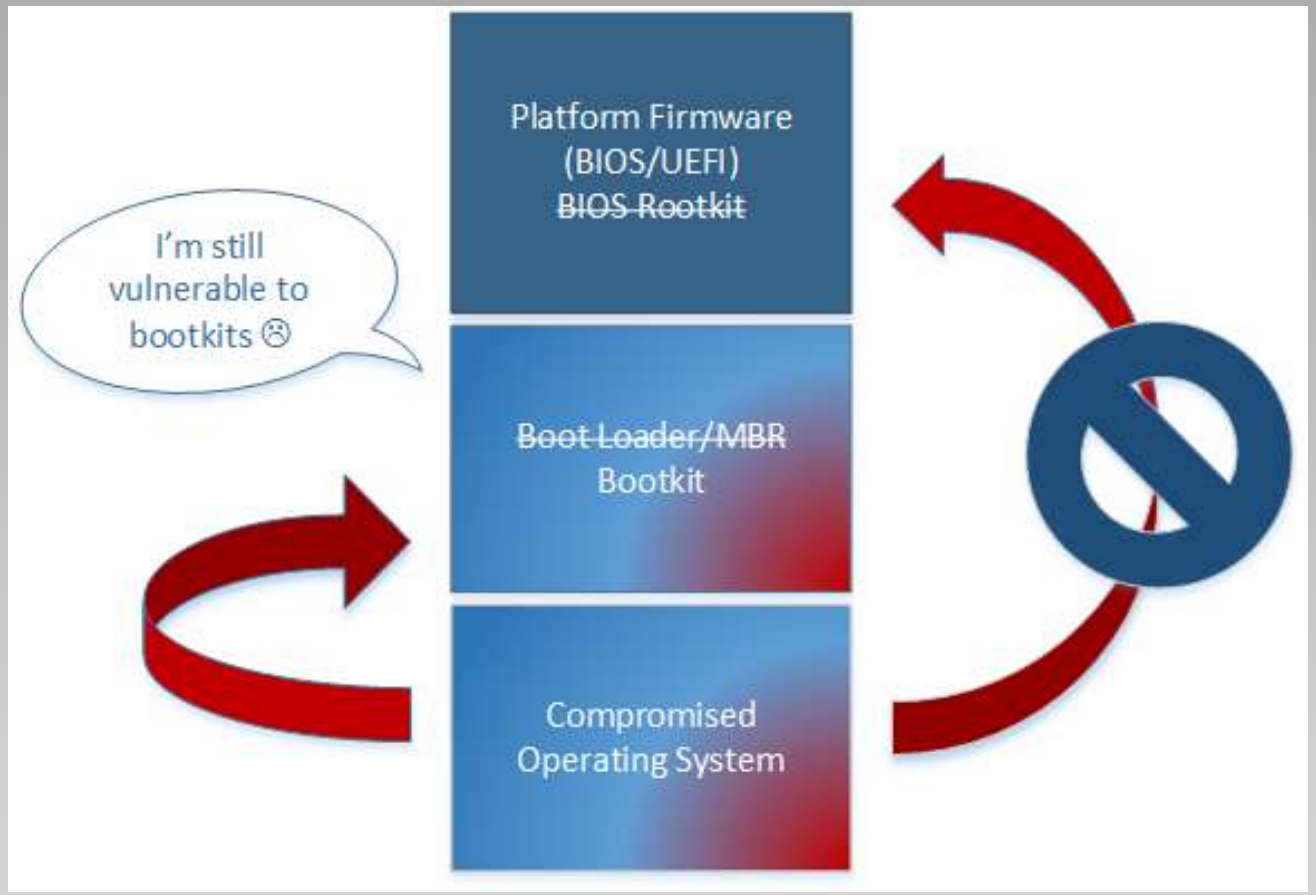
- Many modern platforms implement the requirement that updates to the firmware must be signed. This makes compromising the BIOS with a rootkit harder.

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

- MITRE has 3 more talks we are currently working on which show new attacks that defeat the signed firmware update requirement… even on the latest UEFI systems
- "Just when you thought it was safe to go back in the water…"
- Duh dun…duh dun…

More on Signed BIOS Requirement



- Signed BIOS requirement did not address malicious boot loaders, leaving the door open for Bootkits /Evil Maid attacks

UEFI Secure Boot

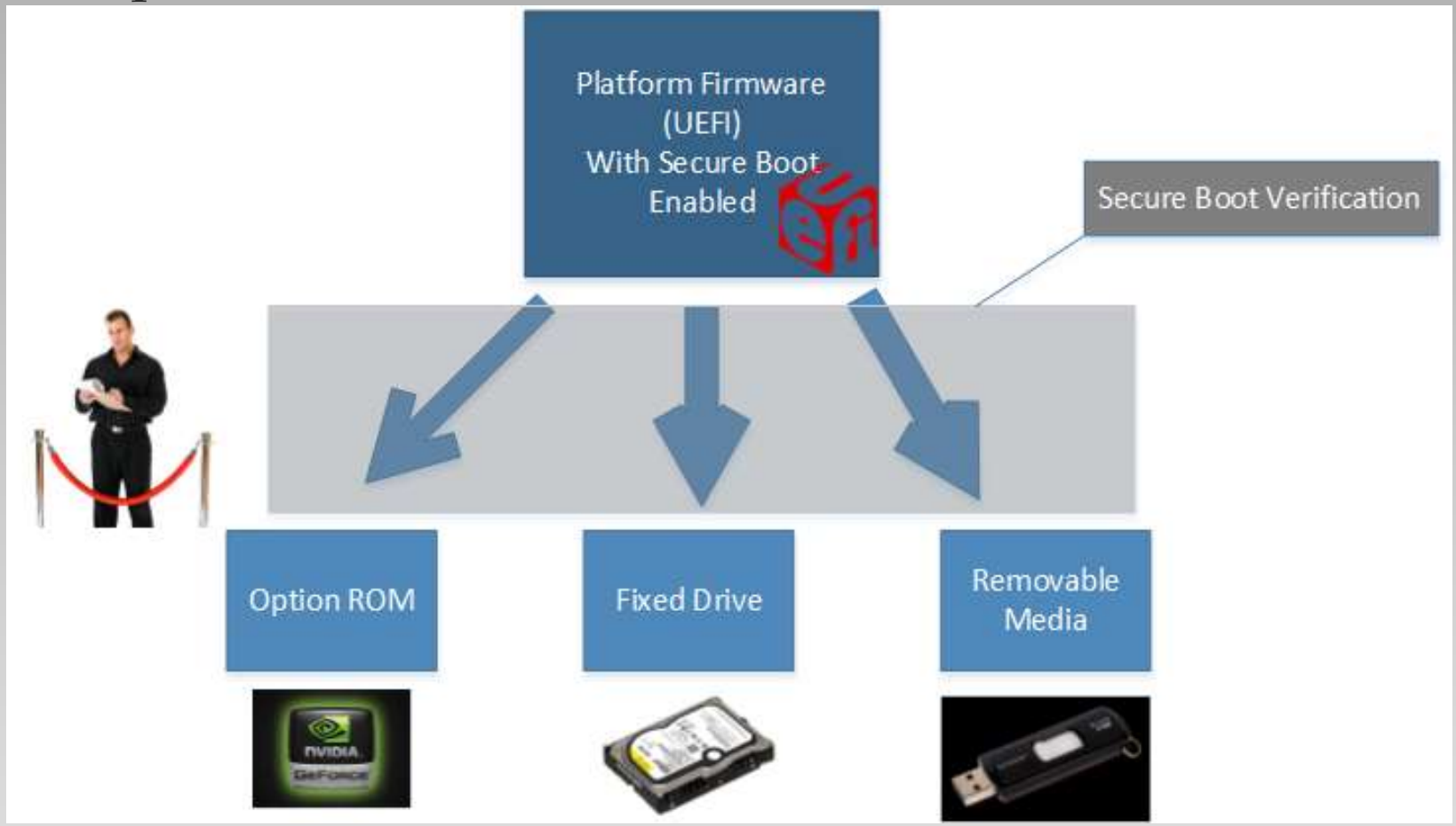
switch

Intel presents secure boot

Attacking Secure Boot Part 2

Defining Secure Boot Image Verification Policies
a.k.a. “Violating the Policy”

Secure Boot will attempt to verify any EFI executable that it attempts to transfer control to. Sort of..



Secure Boot Verifies More Than Bootloader

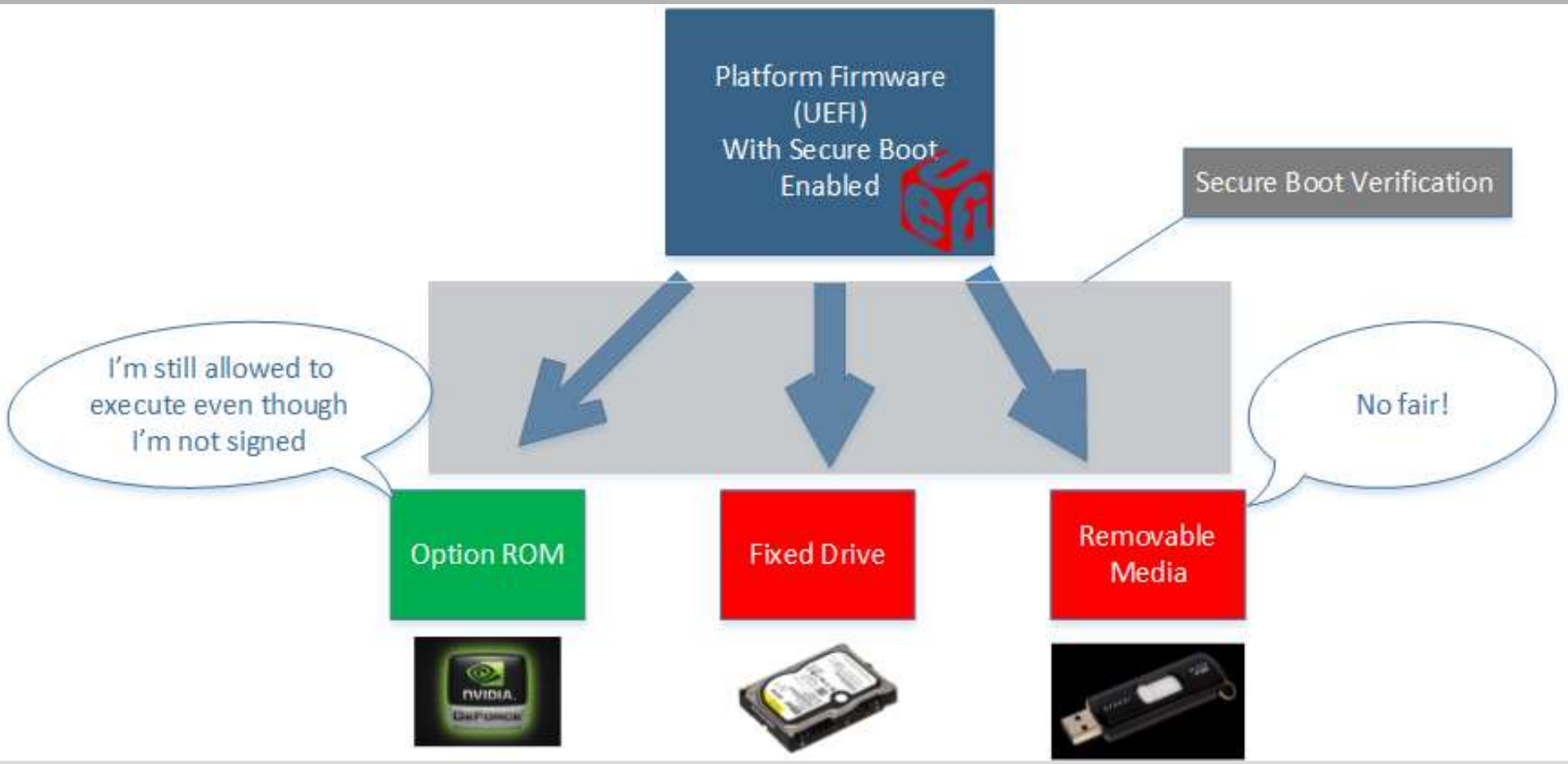
Image Verification Policies

- 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

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

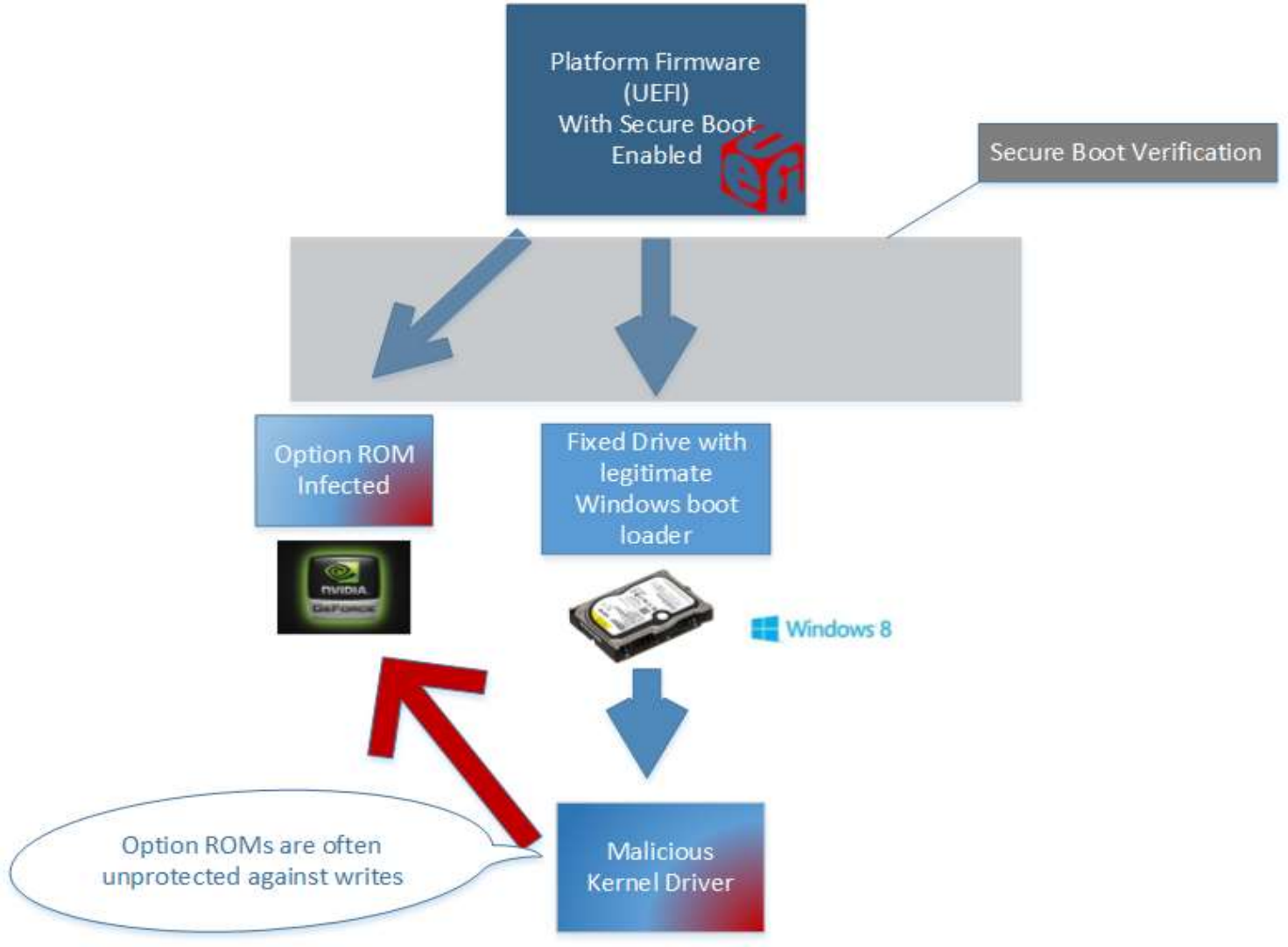
Code from EDK2 open source reference implementation available at:
https://svn.code.sf.net/p/edk2/code/trunk/edk2

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



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

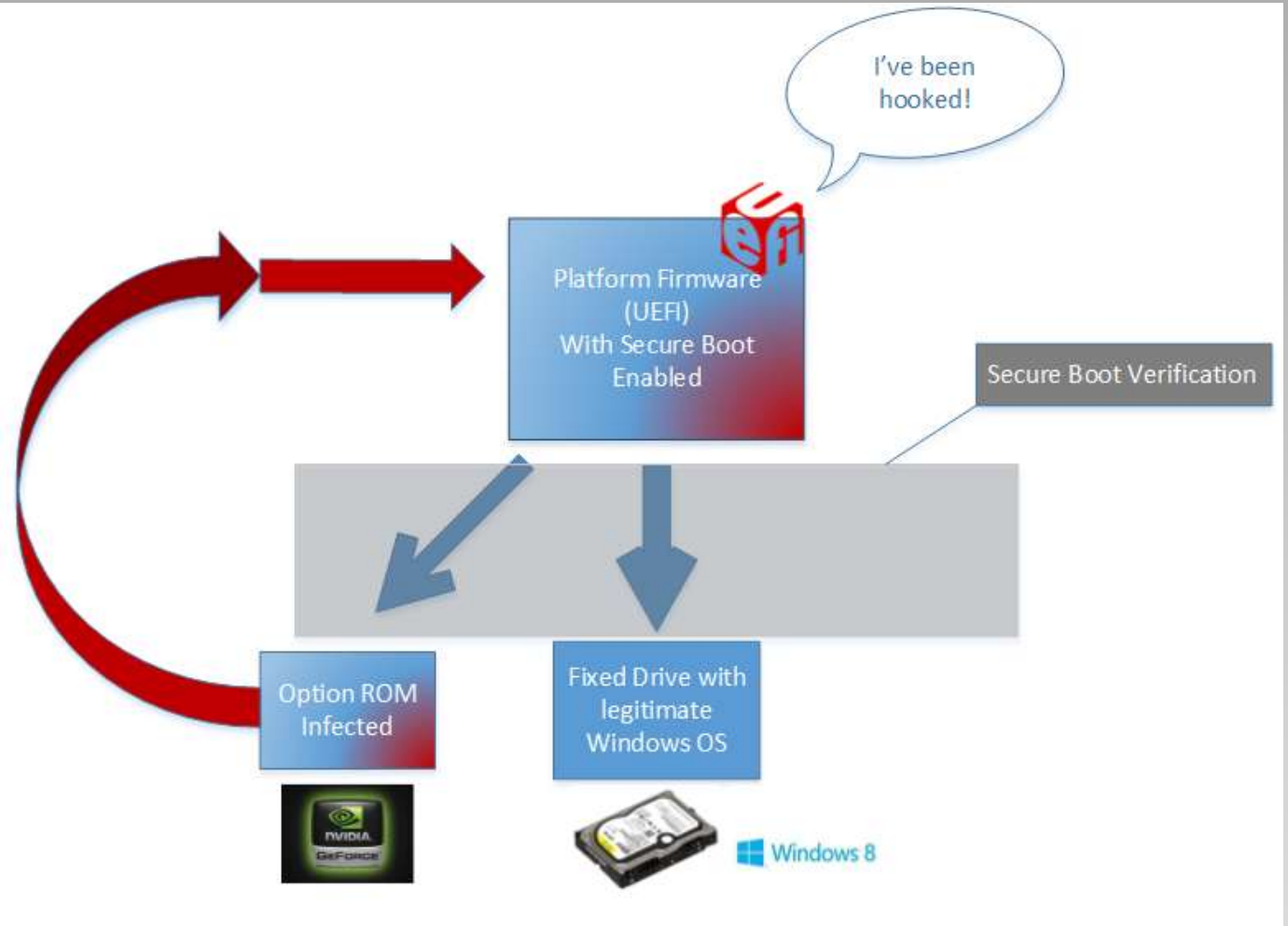
other legitimate EFI executables can be compromised later.



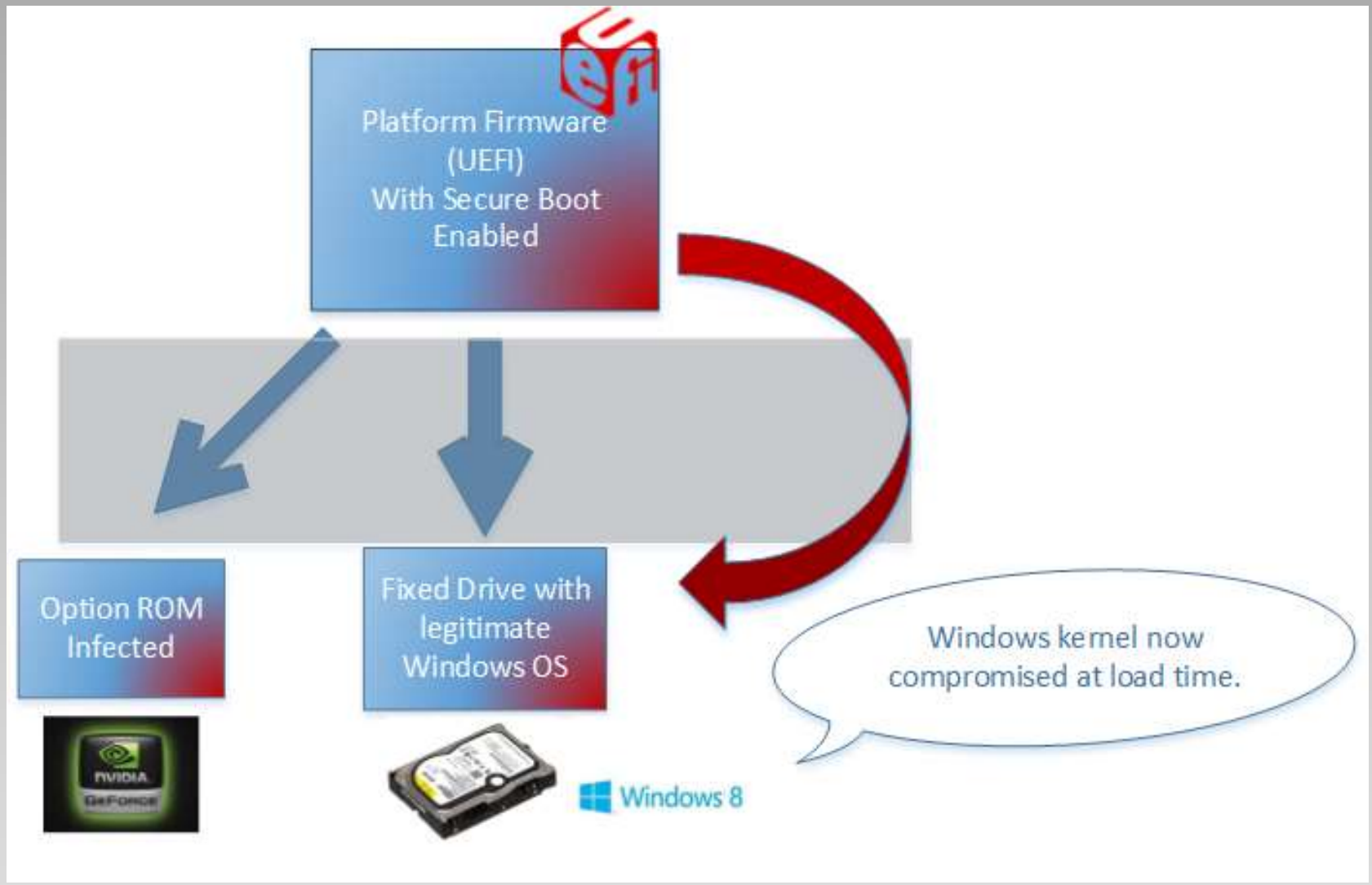
Malicious OROM will run before the legitimate boot loader

Think old school BIOS rootkit IVT hooking

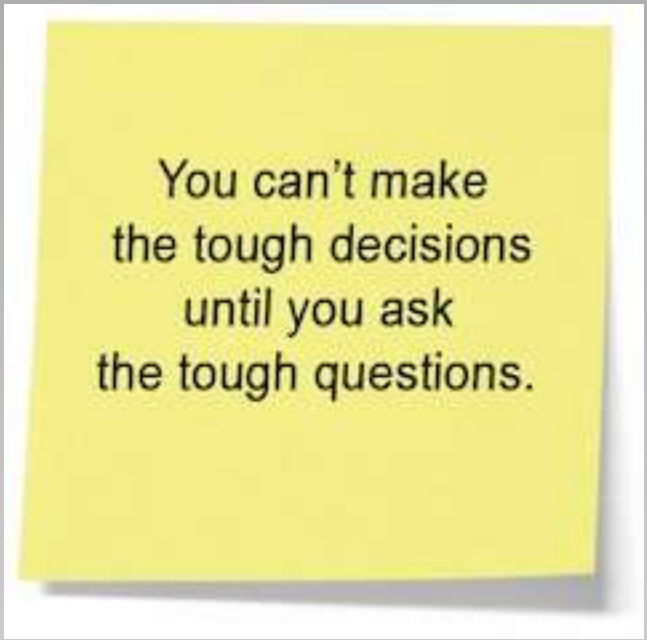
* The actual flash chip contents aren't modified here, only in-memory copies of relevant FW code/structures



Malicious OROM hooks some code that legitimate boot loader will call later

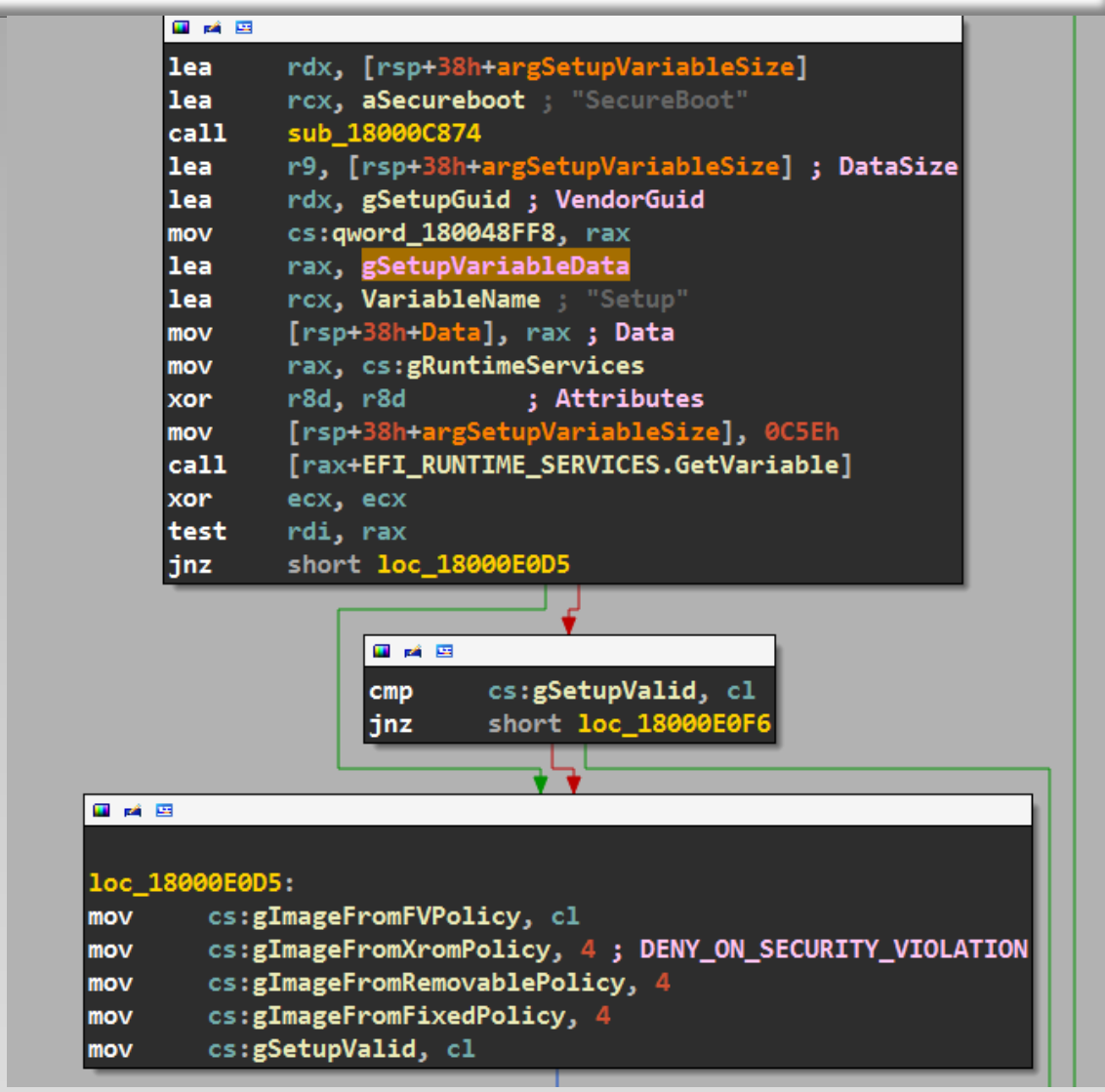


Boot loader is compromised by BIOS code.
OS is then later compromised



- 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?

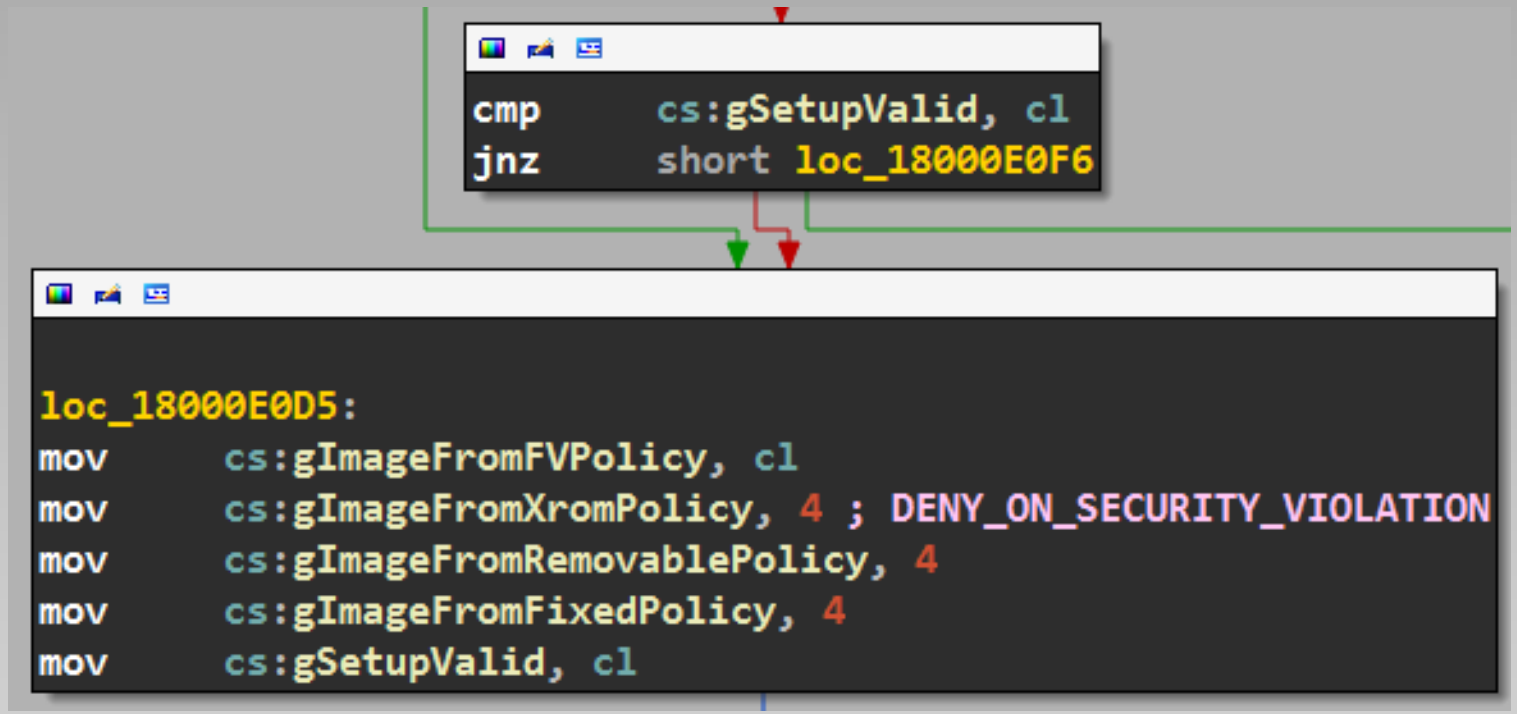
Determining the Secure Boot policy of a “real” system.



Real Policies

```
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]
```

- The firmware attempts to read the EFI non-volatile “Setup” variable
- Setup variable size is 0xC5E



- The Secure Boot policy can be either hardcoded, or derived from the Setup variable

```
.data:000000018014E0C0 gSetupVariableData db    0
.data:000000018014E0C1                    db    0
.data:000000018014E0C2                    db    0
.data:000000018014E0C3                    db    0
```

```
000000018014EC09 gImageFromFVPolicy db 0
000000018014EC09
000000018014EC0A gImageFromXromPolicy db 0
000000018014EC0A
000000018014EC0B gImageFromRemovablePolicy db 0
000000018014EC0B
000000018014EC0C gImageFromFixedPolicy db 0
```

```
.data:000000018014ED16 gSetupValid      db 0
.data:000000018014ED16
```

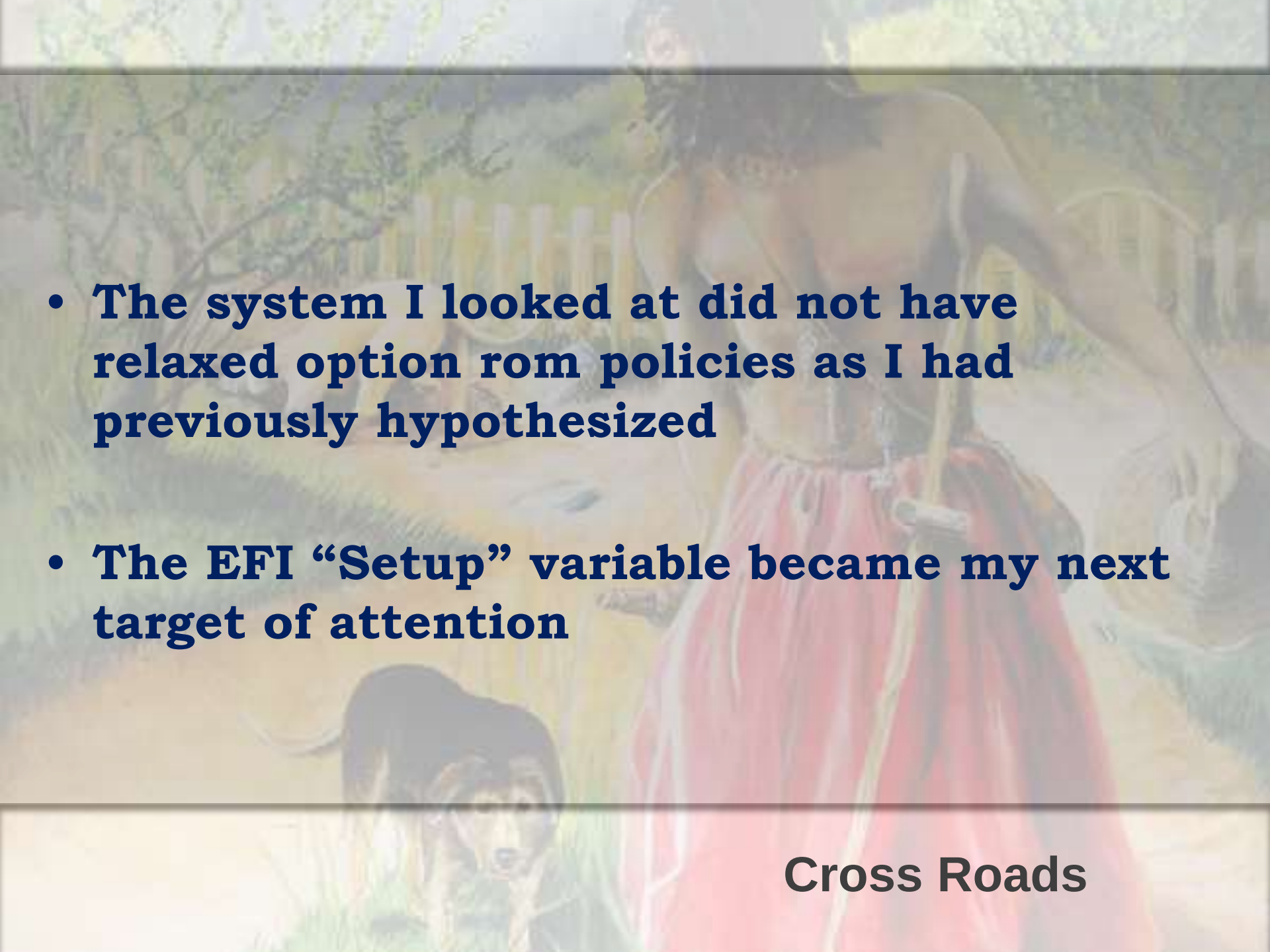
- Setup variable data is read in at 0x18014E0C0. (Size of variable was 0xC5E)
- The gSetupValid byte at 0x18014ED16 determines whether to use the hardcoded secure boot policy, or if the policy embedded in the Setup variable should be used instead
- Secure Boot policy data located at 0x18014EC09

```
.data:000000018014E0C0 gSetupVariableData db    0
.data:000000018014E0C1                    db    0
.data:000000018014E0C2                    db    0
.data:000000018014E0C3                    db    0
```

```
000000018014EC09 gImageFromFVPolicy db 0
000000018014EC09
000000018014EC0A gImageFromXromPolicy db 0
000000018014EC0A
000000018014EC0B gImageFromRemovablePolicy db 0
000000018014EC0B
000000018014EC0C gImageFromFixedPolicy db 0
```

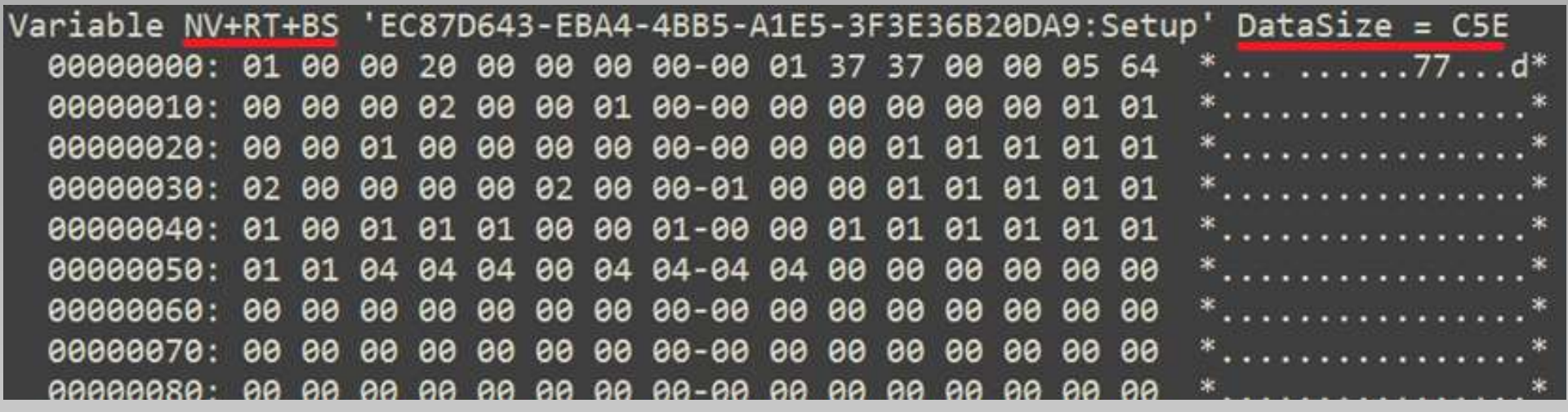
```
.data:000000018014ED16 gSetupValid      db 0
.data:000000018014ED16
```

- Policy valid byte located at offset gSetupValid – gSetupVariableData = 0xC56 in Setup variable data
- Secure Boot policy data located at offset gImageFromFvPolicy – gSetupVariableData = 0xB49 in Setup variable data

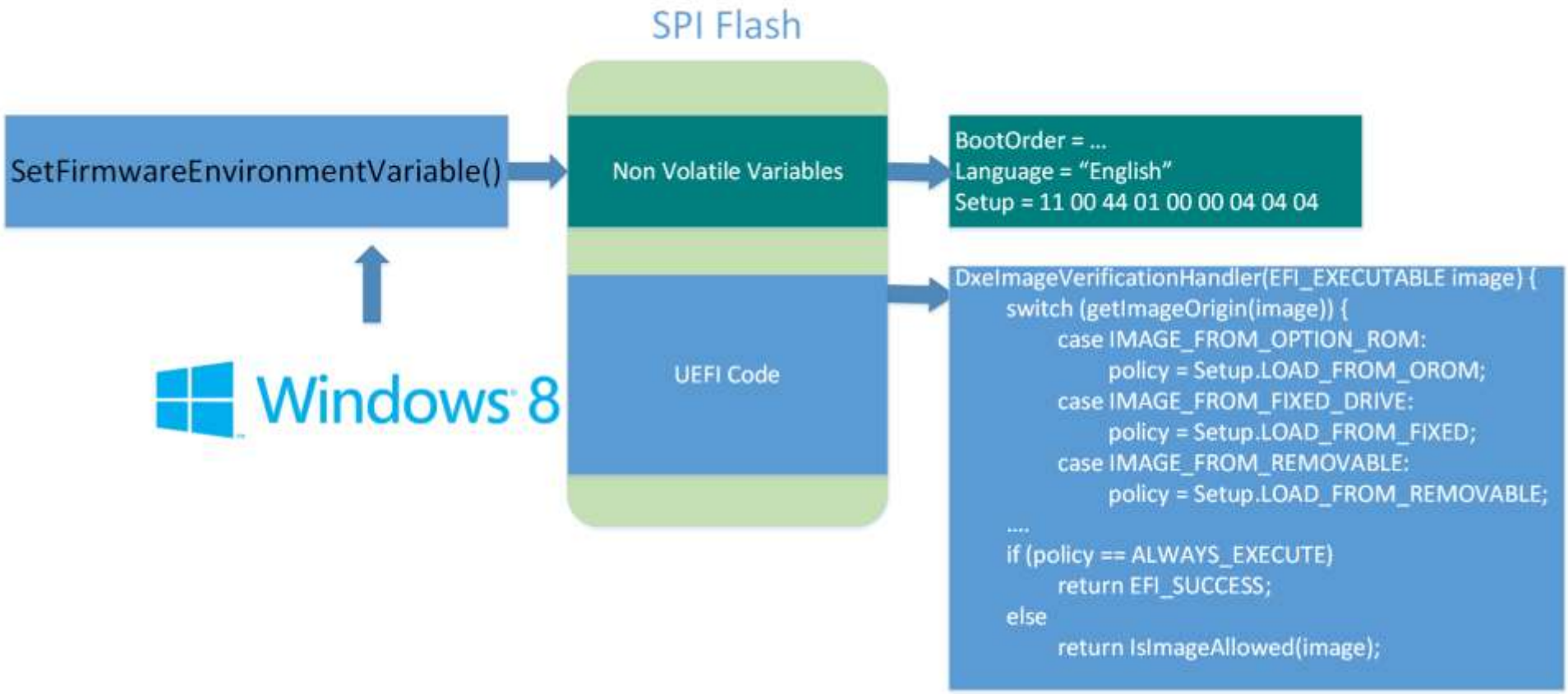


- **The system I looked at did not have relaxed option rom policies as I had previously hypothesized**
- **The EFI "Setup" variable became my next target of attention**

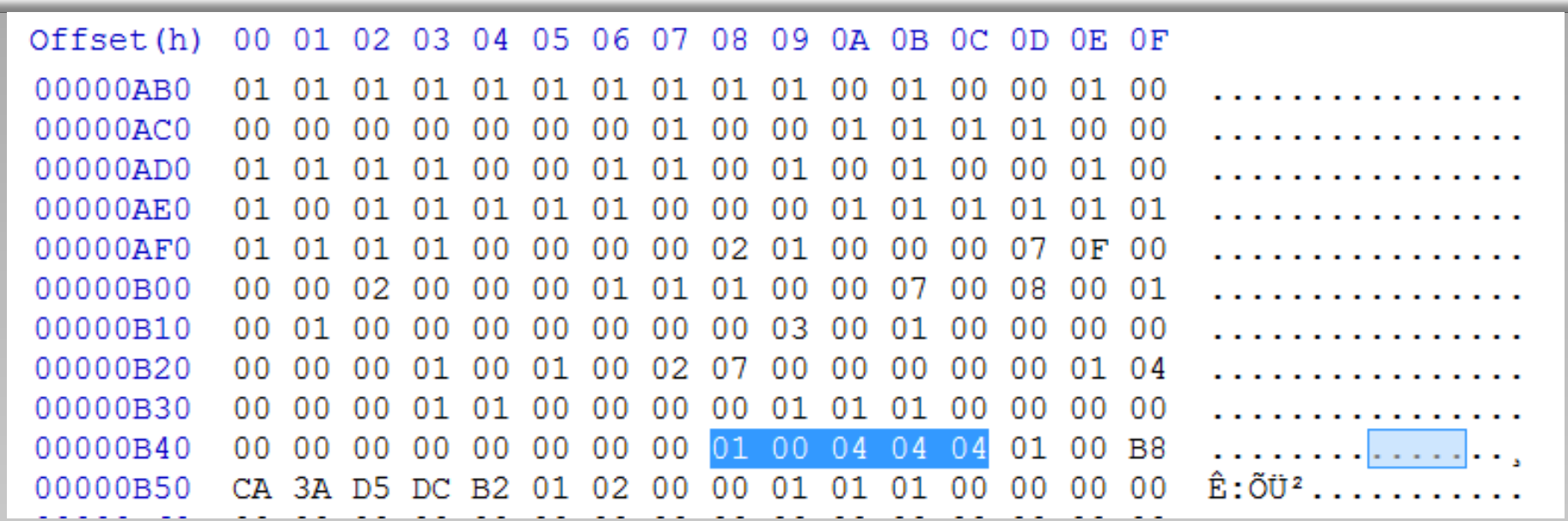
Cross Roads



- 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!

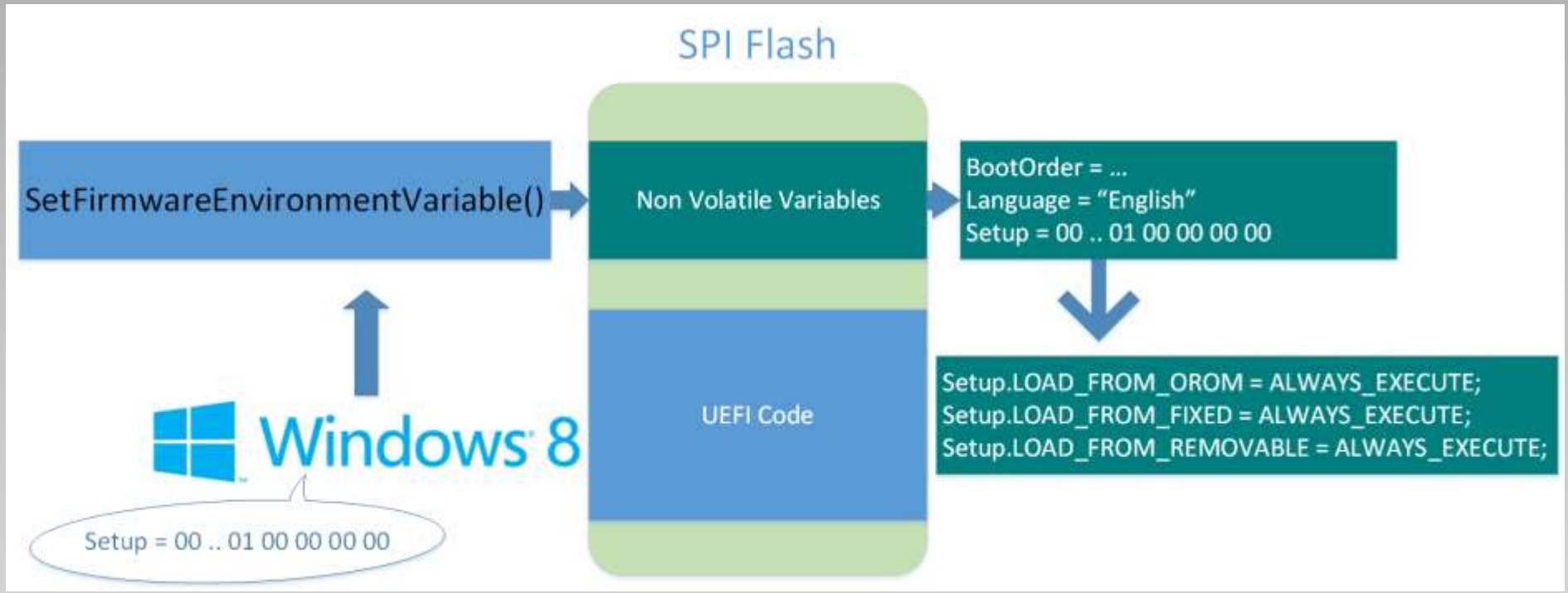


- Not all variables are arbitrarily modifiable from the operating system, such as authenticated variables
- Luckily for us, the Setup variable has no protections on this platform

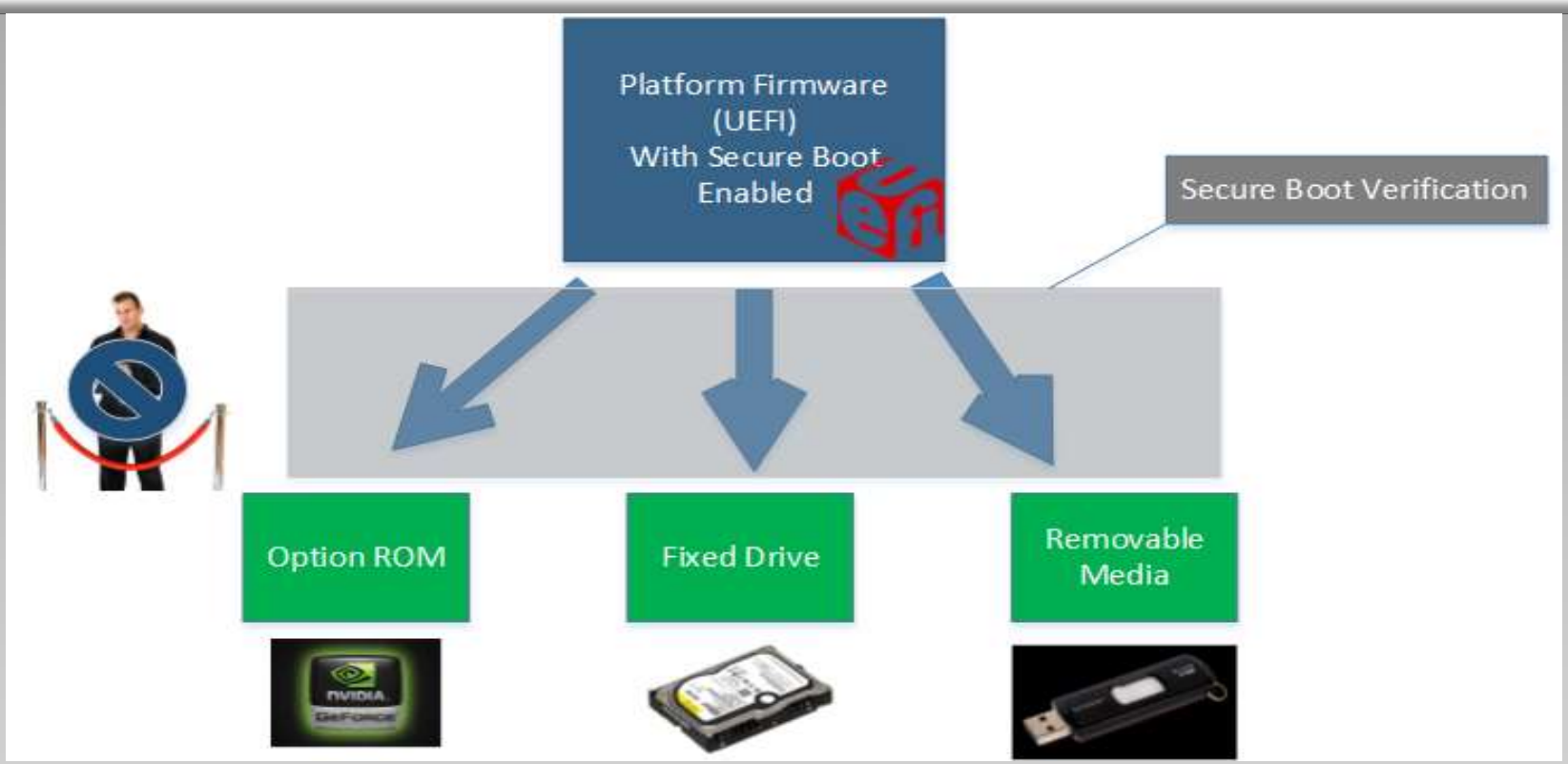


```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

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.....
```

- Offset B48 is the secure boot on/off byte (currently on).
- Offset B49 is the policy byte for “IMAGE_FROM_FV” which is set to ALWAYS_EXECUTE (0x00).
- B4A-B4C are the policy bytes for removable media, fixed media and option rom. All are set to “DENY_EXECUTE_ON_SECURITY_VIOLATION.”
- Let’s use Win8 API to set all of these policies to ALWAYS_EXECUTE.

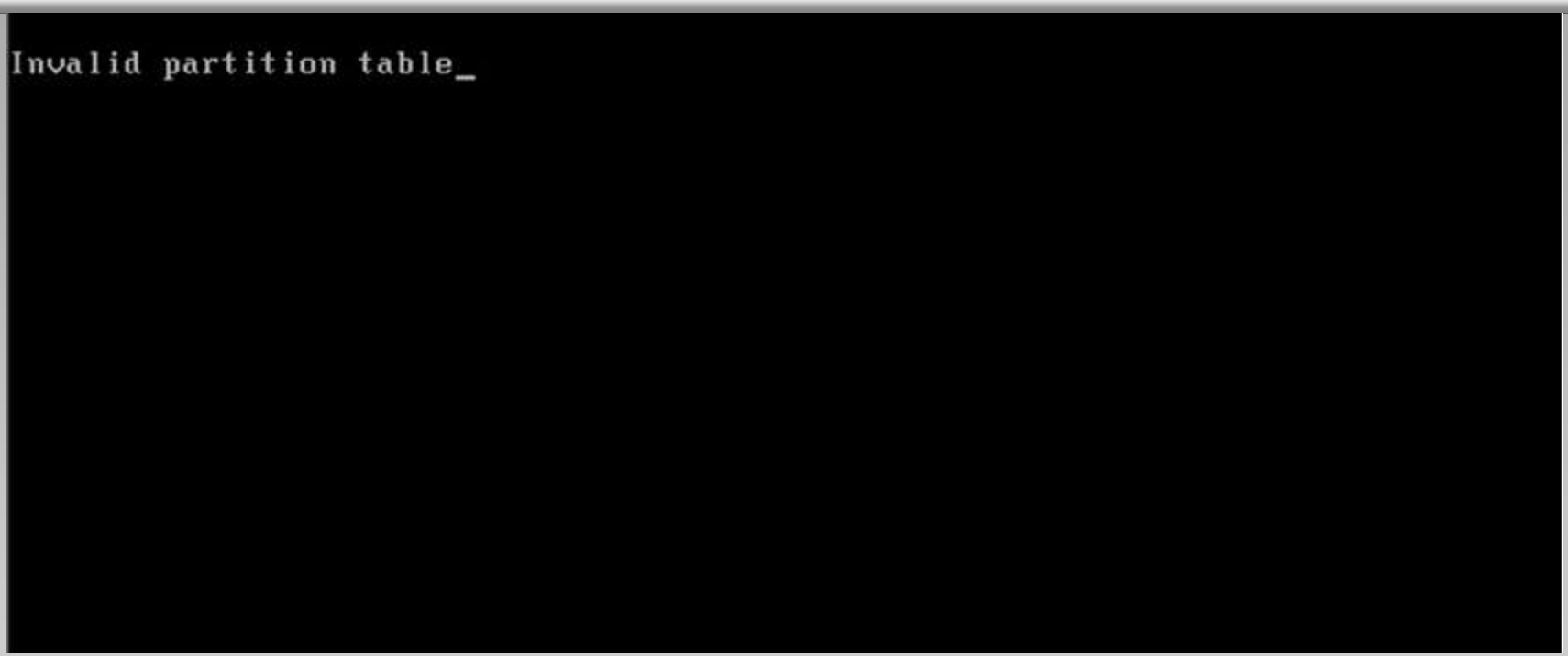


Attack 1 Execution



- 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 Result

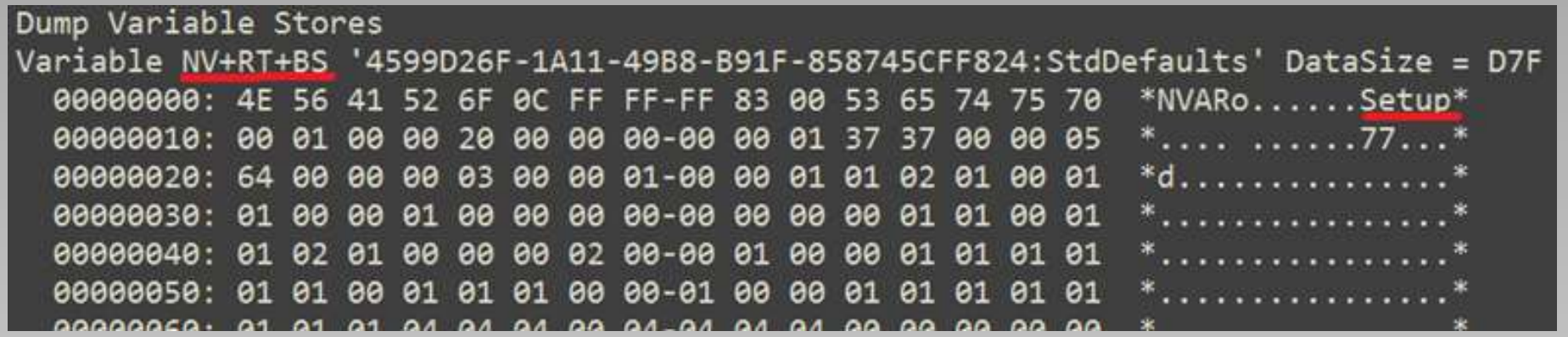


- Deleting the Setup variable reverts the system to a legacy boot mode with secure boot disabled
- This is also effectively a secure boot bypass, as it will force the firmware to transfer control to an untrusted MBR upon next reboot

Attack 2

- Attack 1
  - **Malicious Windows 8 process can force unsigned executables to be allowed by Secure Boot**
  - **Bootkits will now function unimpeded**
  - **Secure Boot will still report itself as enabled although it is no longer “functioning”**
- Attack 2
  - **Malicious Windows 8 process can truly disable Secure Boot by deleting “Setup” variable**
  - **Legacy MBR bootkits will now be executed by platform firmware**
  - **Secure Boot would report itself as “disabled” in this case**

Summary of Attacks



- Actually, when the firmware detects the “Setup” variable has been deleted, it attempts to restore it’s contents from the “StdDefaults” variable
- This variable is also modifiable from the operating system, thanks to its non-authenticated and runtime permissions
- An attacker could modify the StdDefaults variable such that even if an administrator restored the BIOS settings to default, the insecure “allow everything” secure boot policy would remain

StdDefaults Variable

- Malicious Windows 8 process can change the “system defaults” for important BIOS configuration data
- Firmware would restore vulnerable Secure Boot policy whenever firmware configuration reverted to defaults
- This could make life difficult for Administrators

Attacks Corollary

Protect Image Verification Policies

- Don't store them in places writeable by malware (like RUNTIME_ACCESS UEFI Variables)
- Use Pcd (Platform Configuration Database) for the platform specific policies rather than UEFI variables

Set Image Verification Policies to Secure Values

- Using `ALWAYS_EXECUTE,ALLOW_EXECUTE_ON_SECURITY_VIOLATION` in `Pcd[OptionRom|RemovableMedia|FixedMedia]ImageVerificationPolicy` is a bad idea
- Especially check `PcdOptionRomImageVerificationPolicy`
- Default should be `NEVER_EXECUTE` or `DENY_EXECUTE`..

Recommendations

Intel

Related Issues/Guidance