System Isolation Beyond BIOS using the Unified Extensible Firmware Interface

Vincent J. Zimmer
System Software Division
Intel Corporation
DuPont, Washington, USA

Abstract - *This paper describes a means by which a platform containing Intel Trusted eXecution Technology (TXT), including CPU SMX leaf instructions, virtualization extensions VT-x & device virtualization VT-d, along with an implementation of Unified Extensible Firmware Interface (UEFI)-based platform code, can be isolated from pre-OS malware, bus-master DMA devices, non-host platforms such as system service processors (SSP's), baseboard management controllers (BMC's), and platform service processors. With service processors always facing the network and running a full web services stack, the non-host/embedded platform isolation is even more imperative to guard against. This is in addition to the isolation of the UEFI DXE core implementation from 3rd party UEFI drivers, applications, and OS loaders.*

Keywords: BIOS, Trusted Computing, Virtualization, TXT, service processor

1 Introduction

The first several million instructions executed when a system is powered on are firmware, software stored in a chip. The firmware is responsible for initialization of much of the system, including important components such as RAM, video, and keyboards and mice. The firmware is responsible for finding and loading the operating system (such as Microsoft® Windows XP ® or Linux) from a number of different types of media, ranging from hard disks to LANs. Firmware then cooperates with the operating system to load further parts of the operating system before the operating system completely takes over. Today, on PC-class machines at least, that class of software is known as "BIOS."

What do we mean when we mention "UEFI" and "Beyond BIOS?" "UEFI" [1] is an industry group that is standardizing what were the EFI1.10 specification and the Framework PEI (Pre-EFI Initialization) and DXE (Driver Execution Environment) specifications [3][4]. Within UEFI, the main UEFI specification (beginning with UEFI2.0) is handled by the UEFI Specification Working Group (USWG) and the Platform Initialization (PI) Architecture PEI/DXE content are handled in the Platform Initialization (PI) Working Group (PIWG). The differences are that USWG is focused on the OS-to-platform interfaces, whereas the PIWG is focused upon platform initialization. The USWG parties of interest are the OS (operating system) vendors, pre-OS application writers, and independent hardware vendors who write boot ROM's for block or output console devices. The PIWG parties of focus include the platform builders, such as Multi-National Corporations/Original Equipment Manufacturers (MNC's/OEMs), chipset vendors, and CPU vendors. The PIWG work can accommodate both UEFI operating systems and today's conventional/Int19h-based OS's.

PIWG-based components provide one means of producing the UEFI interfaces. What is common across both UEFI and PI Arch is the extensibility of code and interfaces. For UEFI and PI DXE, the extensibility point is a loadable driver model and for PI PEI extensibility is through firmware files with PEI Modules. Although it is common to think of firmware as being stored in a ROM (Read-Only Memory), most firmware is stored in NOR flash devices, which act like a ROM but may be updated to add enhancements or fix bugs. The flash devices are divided into the equivalent of sectors on a disk.

Background

The primary problem being addressed is that that platform manufacturer (PM)/Original Equipment Manufacturer (OEM)/Multi-national corporation (MNC) brand value is based upon the expected behavior of the platform in the field. OEM's spend substantial sums validating their platform, etc. So isolating the pre-OS firmware implementation (i.e., DXE core) from non-OEM code is imperative in order to have assurance of the OEM factory validated behavior. We don't do things like DXE isolation for the OS and it's Digital Rights Management (DRM) [32] applications, etc, to a first order. This is done to vet for the OEM platform behavior and provide a basis of integrity upon which other features can be based, such as trustworthy launch of OS and pre-OS applications.

Another problem being addressed is that any firmware-based security feature, such as the cryptographic loading and checking of UEFI application signatures using digital signature technology like Authenticode, can only be guaranteed to operate as designed in the field if the implementation of the codes are isolated from untrusted content. To-date, pre-OS isolation has been effected via ad hoc, incomplete mechanisms like System Management Mode (SMM) [28], but in a world where SMM may not be available or to meet the cross-architecture requirements of UEFI, another solution is required.

Solution

The solution described in this design is the use of Intel®'s Trusted Execution Technology (TXT) ®[12][27], which has the SMX leaf instructions to allow for authorizing which code enables Virtualization Technology, such as VT-x ® [17][18][19] for x64; TXT is employed in order to isolate the standards-based implementation of UEFI interfaces, namely the UEFI Platform Initialization (PI) Architecture Driver Execution Environment (DXE) components from the basic PEI and DXE core in the systemboard. Because of space constraints in today's ROM's, the implementation is really more of an isolation kernel that maps the machine memory in a 1:1 virtual-to-physical without device emulation, versus a full-up hypervisor (Hv) or Virtual machine monitor (VMM) that provides non1:1 memory mapping and rich device models and inter-guest/operating system separation. The layout of the page tables for this architecture is shown in below. The SINIT module used during the Secure Launch is a stripped down version that admits to inclusion within the overall platform ROM.

Since DXE is the preferred UEFI implementation, we refer to this separate are as "IsoDxe." This design includes TXT-enhanced IsoDxe with DMA protections.

Why is this important?
The threats on the platform for the pre-OS abound, including the PCI [30] and ACPI [7] exploits described by Heasman [24] or the pernicious virtualization virus's like Blue Pill [25].

In fact, UEFI attacks have been described in a recent track at the BlackHat (BH) conference [26].

How does this work?
This design works by launching a platform isolation kernel from the UEFI firmware. The important aspect of this design is that the isolation barriers are erected prior to launching any untrusted, 3rd party code. Since the reset vector and early firmware flows are all under control of the OEM, the IsoDxe driver can be launched here.

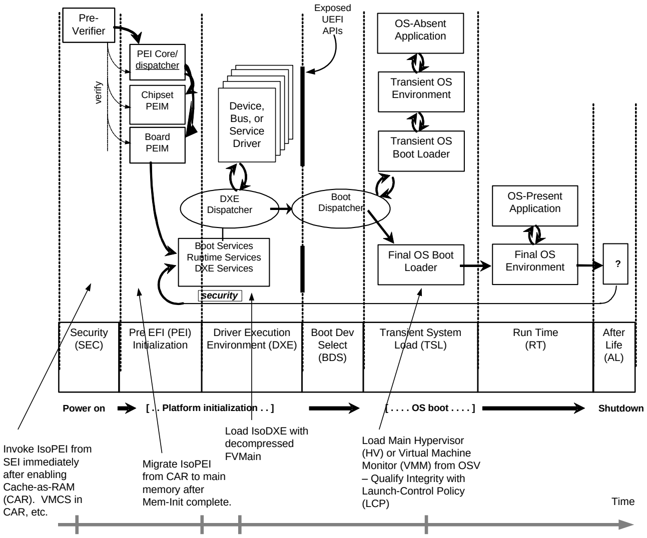

Pre-Verifier
PEI Core/ dispatcher
Chipset PEIM
Board PEIM
verify
Device, Bus, or Service Driver
Exposed UEFI APIs
DXE Dispatcher
Boot Dispatcher
Boot Services Runtime Services DXE Services
security
OS-Absent Application
Transient OS Environment
Transient OS Boot Loader
OS-Present Application
Final OS Boot Loader
Final OS Environment
?
Security (SEC)
Pre EFI (PEI) Initialization
Driver Execution Environment (DXE)
Boot Dev Select (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on
[. . Platform initialization . .]
[. . . . OS boot . . . .]
Shutdown
Invoke IsoPEI from SEI immediately after enabling Cache-as-RAM (CAR). VMCS in CAR, etc.
Migrate IsoPEI from CAR to main memory after Mem-Init complete.
Load IsoDXE with decompressed FVMain
Load Main Hypervisor (HV) or Virtual Machine Monitor (VMM) from OSV – Qualify Integrity with Launch-Control Policy (LCP)
Time

Figure 1-1 Launch of IsoDxe

This work is in contrast to conventional hypervisor (Hv) or Virtual Machine Monitor (VMM) isolation technologies in that all known deployments treat the Hv or VMM as an operating system with respect to the platform boot firmware. In order to accrue assurance and availability, though, the earliest launch is the most desirable. Thus this design pushes the launch phases into the most earliest practical, namely as a Driver Execution Environment (DXE) component of the UEFI Platform Initialization (PI) phase.

This design is motivated by performing a Clark-Wilson [8][14] integrity analysis [23] of the pre-OS. Certain controlled data items (CDI's), such as the UEFI System Table (uefi_system_table_data_t) and other internal state objects for the DXE implementation, were noted as not having appropriate protection. The "before" picture is shown in Figure 1-2, and the protected, or isolated picture is shown in Figure 1-3.

Figure 1-4 shows the flow of this design. Figures 1-6 and 1-7 show instances of the use of IsoDxe for protecting the UEFI service table from attacks by a bus-master DMA device (recall the Heaseman [24] PCI device rootkit attack – this is a countermeasure).

Below is some of the attack code in an errant 3rd party driver, for example.

```
  //
  // start hack
  //
  EfiCopyMem (SystemTableDataBuffer, mSystemTable, 512);
  ((EFI_SYSTEM_TABLE  *)SystemTableDataBuffer)->FirmwareRevision  =
0x12345678;
```

```
  Status = AtaUdmaWrite (IdeDev, SystemTableDataBuffer, StartLba,
NumberOfBlocks);
  if (EFI_ERROR (Status)) {
    DEBUG ((EFI_D_ERROR, "Write SystemTable buffer error - %r\n",
Status));
    return ;
  }
  Status    =    AtaUdmaRead    (IdeDev,    mSystemTable,    StartLba,
NumberOfBlocks);
  if (EFI_ERROR (Status)) {
    DEBUG ((EFI_D_ERROR, "Read SystemTable buffer error - %r\n",
Status));
    return ;
  }
```

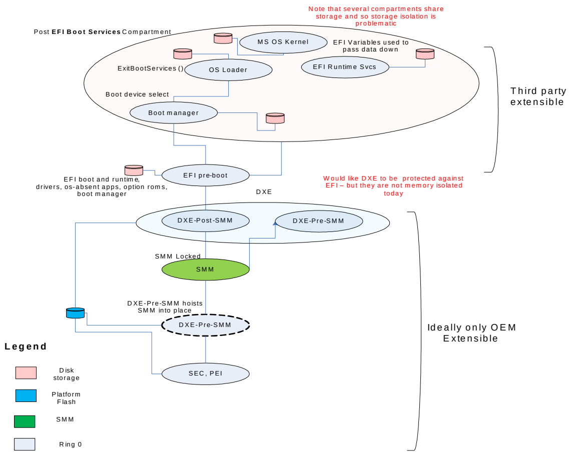
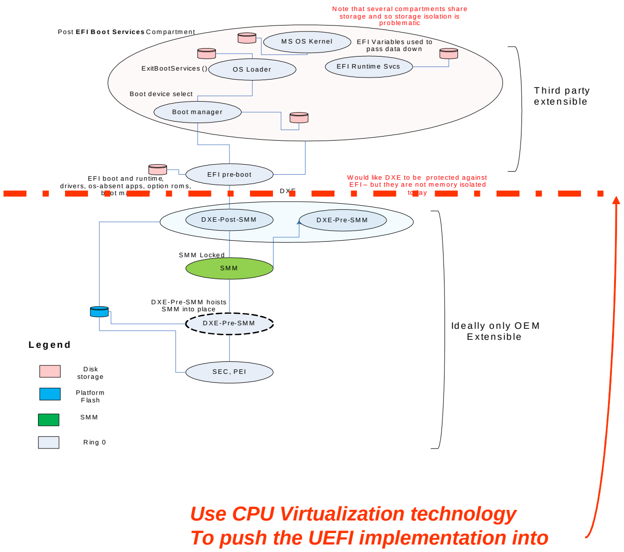

Note that several compartments share storage and so storage isolation is problematic
Post EFI Boot Services Compartment
MS OS Kernel
EFI Variables used to pass data down
ExitBootServices ()
OS Loader
EFI Runtime Svcs
Boot device select
Boot manager
Third party extensible
EFI boot and runtime, drivers, os-absent apps, option roms, boot manager
EFI pre-boot
Would like DXE to be protected against EFI – but they are not memory isolated today
DXE
DXE-Post-SMM
DXE-Pre-SMM
SMM Locked
SMM
DXE-Pre-SMM hoists SMM into place
DXE-Pre-SMM
Ideally only OEM Extensible
Legend
Disk storage
Platform Flash
SMM
Ring 0
SEC, PEI

Figure 1-2 System prior to DXE isolation


Note that several compartments share storage and so storage isolation is problematic
Post EFI Boot Services Compartment
MS OS Kernel
EFI Variables used to pass data down
ExitBootServices ()
OS Loader
EFI Runtime Svcs
Boot device select
Boot manager
Third party extensible
EFI boot and runtime, drivers, os-absent apps, option roms, boot m
EFI pre-boot
Would like DXE to be protected against EFI – but they are not memory isolated today
DXE
DXE-Post-SMM
DXE-Pre-SMM
SMM Locked
SMM
DXE-Pre-SMM hoists SMM into place
DXE-Pre-SMM
Ideally only OEM Extensible
Legend
Disk storage
Platform Flash
SMM
Ring 0
SEC, PEI

Use CPU Virtualization technology
To push the UEFI implementation into
Ring "-1" and isolate from 3rd party codes

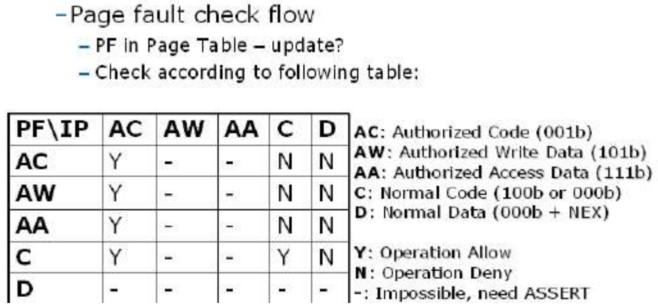
Figure 1-3 Figure after the isolation

Without this design, an errant 3rd party driver could usurp the UEFI services by doing things like patching API's in the UEFI System Table.

The design uses the Virtual Translation Lookaside Buffer (VTLB) algorithm and manages the access state of each page via AVAIL bits, viz.,

- Page type
  - Use AVAIL bits (9:11) to mark page type.
    - Bit 9: NEED AUTHORIZED
    - Bit 10: READ PROTECTED
    - Bit 11: WRITE PROTECTED
- Active Page table (1:1 mapping present)
  - For Authorized CODE (Check Write)
    - Not allow update
  - For Authorized DATA Write (Check Write)
    - Check AVAIL bit
  - For Authorized DATA Read/Write (Check Access)
    - Check AVAIL bit

The actual protection is implemented during a page fault by assessing the following algorithm



- Page fault check flow
  - PF in Page Table – update?
  - Check according to following table:

PF\IP	AC	AW	AA	C	D
AC	Y	-	-	N	N
AW	Y	-	-	N	N
AA	Y	-	-	N	N
C	Y	-	-	Y	N
D	-	-	-	-	-

AC: Authorized Code (001b)
AW: Authorized Write Data (101b)
AA: Authorized Access Data (111b)
C: Normal Code (100b or 000b)
D: Normal Data (000b + NEX)

Y: Operation Allow
N: Operation Deny
-: Impossible, need ASSERT

Again, the layout of the page tables for this architecture is shown below.

These same page-table for inter-guest (i.e., separation of drivers from DXE core) can be used for the bus-master DMA isolation, namely the programming of VT-d [22] viz.,

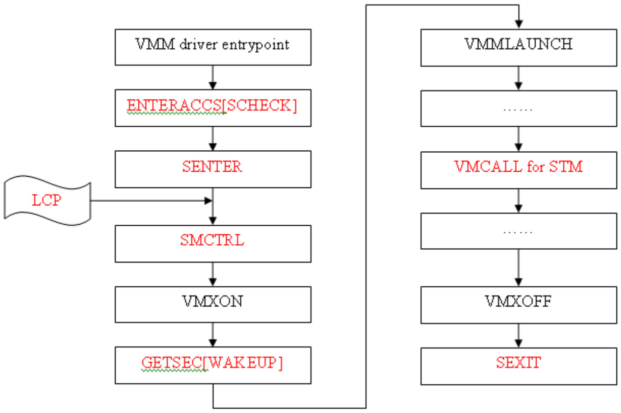
- DMAR ACPI table
  - The platform should report DMAR ACPI table earlier. Maybe gathering information in build HOB for VMM.
  - So that VMM can get DMAR information in its entrypoint.

```
typedef {
    ACPI_DMAR_TABLE  DmarTable;
} EFI_DMAR_HOB;
```

The UEFI PI PEI will create a hand-off block (HOB) [31] in order to describe the isolation of the IsoDxe from Direct Memory Access (DMA) busmaster devices.

The specifics of the DMA protection are as follows

- Guest use the same method to report what memory attribute as mentioned in DxeIsoByVT. (no update)
- VMM setup Context Entry table for all PCI device and map to one domain.
- VMM setup domain page table for all memory space.
- VMM update domain page table to create hole for critical memory for authorized code, authorized data.



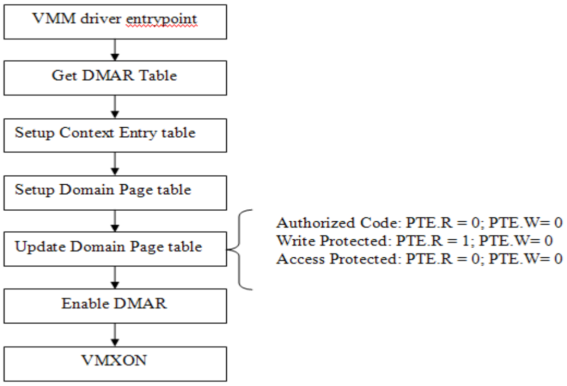
The TXT protection is enabled as follows via the interface:

- BIOS/SINIT ACM information
  - The platform should report BIOS/SINIT ACM information earlier. Maybe gathering information in build HOB for VMM.
  - So that VMM can get ACM information in its entrypoint, and run SCHECK and SENTER.

```
typedef {
            EFI_PHSICAL_ADDRESS  BiosACMAddress;
            UINTN                BiosACMLength;
            EFI_PHSICAL_ADDRESS  SinitACMAddress;
            UINTN                SinitACMLength;
} EFI_TXT_ACM_HOB;
```

BIOS/SINIT ACM information

- Before BIOS expose EFI_TXT_ACM_HOB, it should program all required chipset register to meet the SCHECK/SENTER execution requirement.
- BIOS should know VMM info so that no SCHECK will be performed again in BIOS code.


VMM driver entrypoint
Get DMAR Table
Setup Context Entry table
Setup Domain Page table
Update Domain Page table
Enable DMAR
VMXON
Authorized Code: PTE.R = 0; PTE.W= 0
Write Protected: PTE.R = 1; PTE.W= 0
Access Protected: PTE.R = 0; PTE.W= 0

Today, UEFI implementations co-locate the DXE implementation and the third party UEFI drivers/OS loaders in ring0. There is no isolation between the two classes of code (OEM and 3rd party). This design will erect an isolation barrier by pushing the DXE into “ring -1”, thus can avoid breaking compatibility with the 3rd party UEFI codes that still believe they have unfettered “ring 0” access.

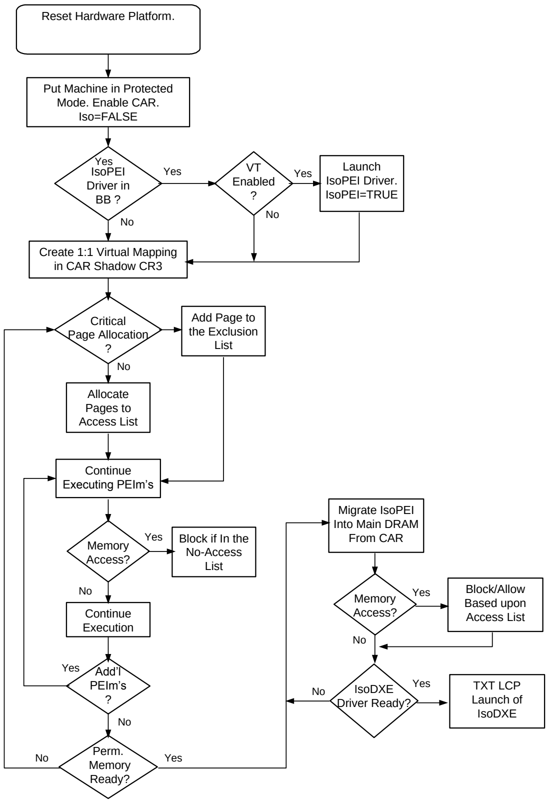
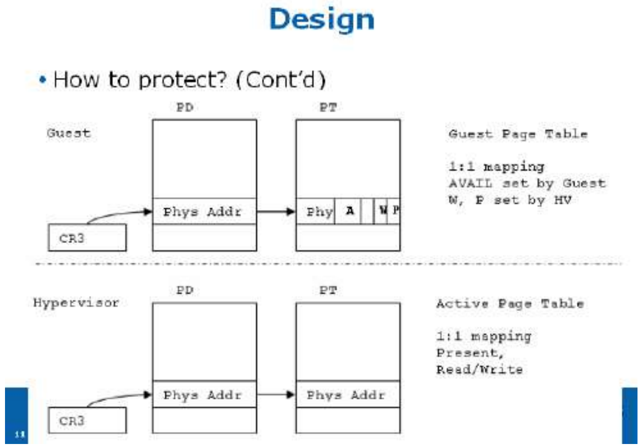

Reset Hardware Platform.
Put Machine in Protected Mode. Enable CAR. Iso=FALSE
IsoPEI Driver in BB ?
Yes
No
VT Enabled ?
Yes
No
Launch IsoPEI Driver. IsoPEI=TRUE
Create 1:1 Virtual Mapping in CAR Shadow CR3
Critical Page Allocation ?
No
Add Page to the Exclusion List
Allocate Pages to Access List
Continue Executing PEIm’s
Memory Access?
Yes
No
Block if In the No-Access List
Continue Execution
Add’l PEIm’s ?
Yes
No
Perm. Memory Ready?
No
Yes
Migrate IsoPEI Into Main DRAM From CAR
Memory Access?
Yes
No
Block/Allow Based upon Access List
IsoDXE Driver Ready?
No
Yes
TXT LCP Launch of IsoDXE

Figure 1-4 Flow of the design

The design of this approach is to essentially use spare bits in the page table to mark which entries in the UEFI memory map [1] are owned by IsoDxe and the DXE implementation, versus which pages are free or owned by 3rd party UEFI drivers and applications. This allows the page-table emulation algorithm of IsoDxe to transparently allow UEFI applications, including but not limited to diagnostics, operating system loaders, drivers, and applications, to believe that they are executing in 1:1 virtual-to-physical and their view of the page tables reflect the actual machine.



Design

- How to protect? (Cont'd)

Guest

PD

PT

Phys Addr

Phy A W P

CR3

Guest Page Table

1:1 mapping
AVAIL set by Guest
W, P set by HV

Hypervisor

PD

PT

Phys Addr

Phys Addr

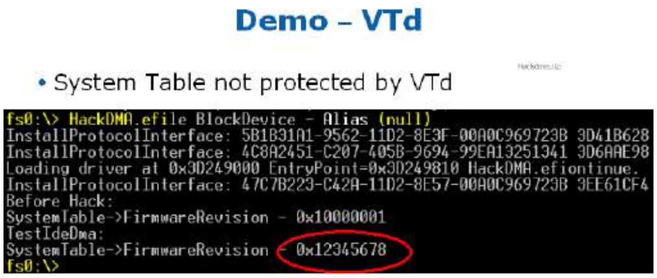
CR3

Active Page Table

1:1 mapping
Present,
Read/Write

Figure 1-5 Page table design for IsoDxe

The following pictures demonstrate the creation of attack code that attempt to hijack or corrupt the UEFI System Table [1]. The before picture shows the UEFI system table contents, as displayed by a UEFI shell application that has attacked a system without these protections. The latter picture shows the same attack that has been foiled by the presence of IsoDxe.

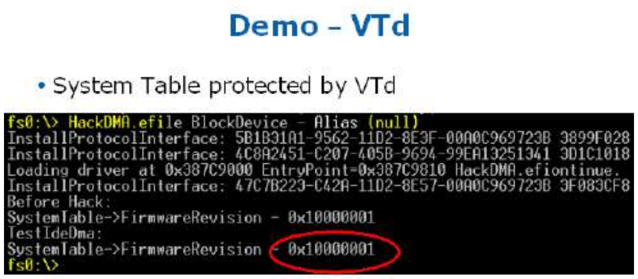


Demo – VTd

- System Table not protected by VTd

```
fs0:\> HackDMA.efile BlockDevice - Alias (null)
InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 3D41B628
InstallProtocolInterface: 4C8A2451-C207-405B-9694-99EA13251341 3D6AAE98
Loading driver at 0x3D249000 EntryPoint=0x3D249810 HackDMA.efiontinue.
InstallProtocolInterface: 47C7B223-C42A-11D2-8E57-00A0C969723B 3EE61CF4
Before Hack:
SystemTable->FirmwareRevision - 0x10000001
TestIdeDma:
SystemTable->FirmwareRevision - 0x12345678
fs0:\>
```

Figure 1-6 VT-d attack



Demo – VTd

- System Table protected by VTd

```
fs0:\> HackDMA.efile BlockDevice - Alias (null)
InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 3899F028
InstallProtocolInterface: 4C8A2451-C207-405B-9694-99EA13251341 3D1C1018
Loading driver at 0x387C9000 EntryPoint=0x387C9810 HackDMA.efiontinue.
InstallProtocolInterface: 47C7B223-C42A-11D2-8E57-00A0C969723B 3F083CF8
Before Hack:
SystemTable->FirmwareRevision - 0x10000001
TestIdeDma:
SystemTable->FirmwareRevision - 0x10000001
fs0:\>
```

Figure 1-7 VT-d attack mitigation

Going forward - More Coverage

In addition to having IsoDxe to protect the UEFI phase of execution when 3rd party option ROM's and drivers load, this art can be extended to protect the entire pre-OS from reset vendor up-to-and-including the UEFI phase.

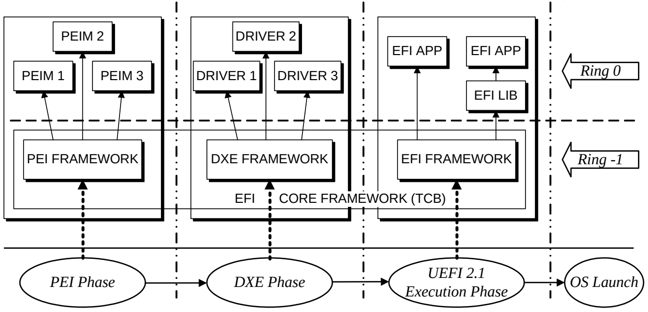


Figure 1-8 Isolation of entire UEFI PI boot flow

The motivation for this is that both the PEI and DXE phase of execution admit the loading of additional binary modules. Today the trust model is that PEI and DXE are OEM extensible only, but in a future of having the equivalent of 3rd party content in this phase, the PEI and DXE core components need to be protected. The earliest isolation kernel would run from PEI and we call it "IsoPei" as an analogous term to the earlier described IsoDxe. PEI has additional space constraint in the ROM and available memory, so it is much simpler than IsoDxe.

The process of one hardware virtualization-aware agent, such as IsoPei, invoking a successive one, such as IsoDxe, we refer to as "Hexec" (which stands for "hypervisor execute") since the process is akin to the Kexec usage model where one Linux kernel can invoke another. [34]

The policy-control of a given virtualization launch using Launch-Control Policy (LCP) [33] is shown in Figure 1-9.

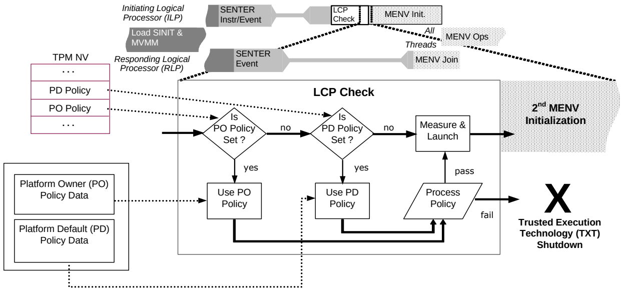


Figure 1-9 Launch control policy

Finally, this protect allows for fine-grain access control and inter-domain service invocation. Specifically, imagine isolating a particular UEFI option ROM that supplies the EFI_BLOCK_IO_PROTOCOL services from the rest of the system. Below is a diagrammatic view of such a protection scenario.

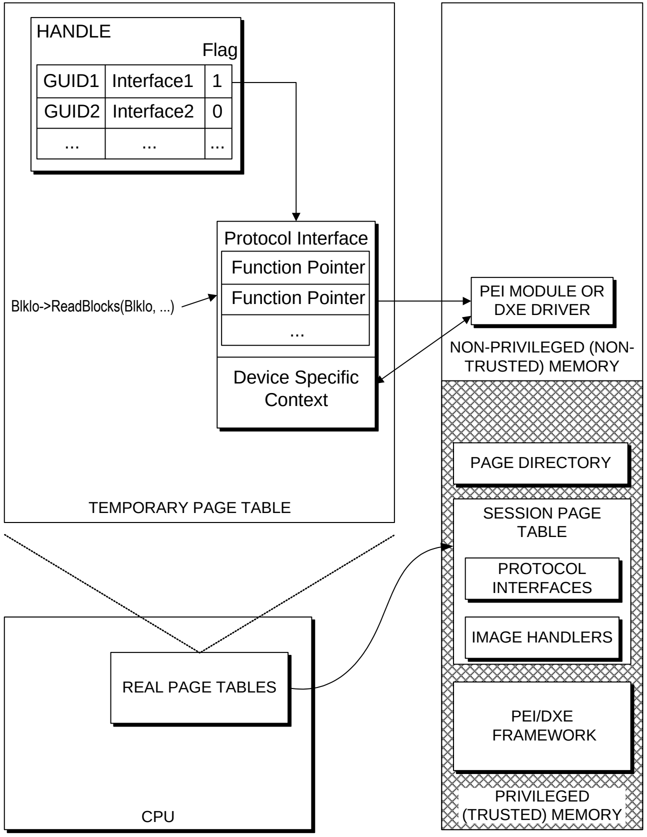


Figure 10 Isolation of driver w/ given API

Related work

This art complement implementations of secure boot, whether BIOS-based [2] or UEFI-based [16], but really entails use of virtualization in the pre-OS for isolation and assurance of construction for features like secure boot. Other trusted computing practice has been described, in general, by [20]. And the use of runtime VMM's for isolation [15][35]. But beyond some designs in [29], this is the first reduction to practice of virtualization [21] for pre-OS security.

Conclusion

Intel can use this design to allow UEFI platforms to be more malware/virus resistant than legacy BIOS platforms or ones that do not implement this isolation. This design will ensure that the promise and value of UEFI's extensibility does not become an issue in market deployment.

This design is novel in that it allows platforms to maintain compatibility with the EFI1.02 drivers that expect unfettered ring 0 execution and applications that have been shipping since 1999 and to harden emergent UEFI capabilities, such as UEFI secure boot [16].

2 References

[1] Unified Extensible Firmware Interface Specification Version 2.1, January 23, 2007. http://www.uefi.org.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith, ``A Secure and Reliable Bootstrap Architecture," in *Proceedings 1997 IEEE Symposium on Security and Privacy*, pp. 65-71, May 1997.

[3] Vincent Zimmer, Michael Rothman, Robert Hale, *Beyond BIOS: Implementing the Unified Extensible Firmware Interface Specification with Intel's Framework*. Intel Press, September 2006. ISBN 0-9743649-0-8 http://www.intel.com/intelpress/sum_efi.htm

[4] Vincent Zimmer, "Advances in Platform Firmware Beyond BIOS and Across all Intel® Silicon", Technology @ Intel Magazine, January 2004. http://www.intel.com/technology/magazine/systems/it01043.pdf.

[5] Trusted Computing Group EFI Protocol and Platform Specifications, Version 1.2. https://www.trustedcomputinggroup.org/specs/PCClient

[6] Frank Stajano, Ross Anderson, "The Resurrecting Duckling: Security Issues for Ad-Hoc Wireless Networks," *Lecture Notes in Computer Science*, Issue 1796. Springer-Verlag, 1999.

[7] Advanced Configuration and Power Interface (ACPI) Specification, Version 3.0b, http://www.acpi.info.

[8] D. Clark and D. Wilson, "A Comparison of Commercial and Military Security Policies," *IEEE Symposium on Security and Privacy*, 1987.

[9] NSA Suite B Cryptography. www.**nsa**.gov/ia/industry/crypto_**suite**_**b**.cfm

[10] X. Wang, Y.L. Yin, and H. Yu. Finding Collisions in the Full SHA-1, Advances in Cryptology -- Crypto'05.

[11] Integrity Measurement Architecture. http://domino.research.ibm.com/comm/research_projects.nsf/pages/ssd_ima.index.html

[12] Intel ® Trusted Execution Technology. http://download.**intel**.com/**technology**/**security**/downloads/31516803.pdf

[13] Secure Hash Standard. csrc.**nist**.gov/publications/fips/fips180-2/fips180-2.pdf

[14] K. J. Biba, “Integrity Considerations For Secure Computer Systems,” ESD-TR-76-372, NTIS#AD-A039324, Electronic Systems Division, Air Force Systems Command, April 1977

[15] T. Garfinkel, B. Pfaff, J. Chow, M., Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 193–206, 2003.

[16] Vincent Zimmer, “Platform Trust Beyond BIOS Using the Unified Extensible Firmware Interface,” in *Proceedings of the 2007 International Conference on Security And Management, SAM’07*, CSREA Press, June 2007

[17] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy Santoni, C.M. Martins, Andrew Anderson, Steven Bennet, Alain Kagi, Felix Leung, Larry Smith, “Intel Virtualization Technology,” *IEEE Computer*, May 2005

[18] Intel Corp., "Intel Virtualization Technology Specification for the IA-32 Architecture," at www.intel.com/technology/vt

[19] Intel Corp., "Intel Virtualization Technology Specification for the Intel Itanium Architecture" at www.intel.com/technology/vt/

[20] P. England, B. Lampson, J. Manferdelli, M. Peinado, B. Willman, "A Trusted Open Platform," *IEEE Computer*, pp. 55–62, July 2003.

[21] R. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, pp. 34–45, June 1974

[22] Intel Corp., "Intel Virtualization Technology Specification for Directed I/O Specification," at www.intel.com/technology/vt/

[23] Trent Jaeger and Reiner Sailer and Xiaolan Zhang. **Analyzing Integrity Protection in the SELinux Example Policy.** in *Proceedings of the 11th USENIX Security Symposium*, pages 59--74. August, 2003

[24] Heasman PCI ACPI http://www.ngssoftware.com/research/papers/Implementing_And_Detecting_A_PCI_Rootkit.pdf

[25] Rutkowska Blue Pill http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

[26] Heaseman EFI attack discussion https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf

[27] David Grawrock, *The Intel Safer Computing Initiative* Intel Press, 2006 http://www.intel.com/intelpress/sum_secc.htm

[28] **Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide** http://www.intel.com/products/processor/manuals/

[29] Vincent J. Zimmer, “A Method For Providing System Integrity And Legacy Environment Emulation”, US Patent #7,103,529, Issued 9/5/2006.

[30] Peripheral Component Interconnect (PCI) Specification www.pcisig.org

[31] UEFI Platform Initialization Specification, Version 1.0, Volumes 1 -5 www.uefi.org

[32] Biddle, England, Peinado, Willman, “The Darknet and the Future of Content Distribution” http://crypto.stanford.edu/DRM2002/darknet5.doc

[33] Joseph Cihula. “Trusted Boot: Verifying the Xen Launch.” http://xen.org/files/xensummit_fall07/23_JosephCihula.pdf

[34] Hariprasad Nellitheertha. “Reboot Linux faster using kexec. “ http://www.ibm.com/developerworks/linux/library/l-kexec.html

[35] Seshadri, Luk, Qu, Perrig, “SecVisor: a tiny hypervisor to provide lifetime kernel code protection,” ACM Symposium on OS Principles, Stevenson, WA 2007