Access Control Beyond BIOS Using the Unified Extensible Firmware Interface

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

Abstract *- This paper describe access control for a platform containing an implementation of Unified Extensible Firmware Interface (UEFI)-based platform code. This access control entails full mediation using a fine-grain policy mechanism. Given the different market segments and platform types for which UEFI solutions can be composed, this policy mechanism is flexible enough to allow for high-assurance, MAC deployments all the way to open platforms, DAC usage models.*

Keywords: BIOS, UEFI, Access Control, Type enforcement

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 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 associated 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.

This document describes a method Type-Enforcement (TE) for mandatory access control (MAC) under Unified Extensible Firmware Interface (UEFI) environment (www.uefi.org). The problem being addressed is that the "extensibility" of UEFI creates a larger attack surface and opportunity for injecting malware into the platform.

The UEFI 2.2 specification introduced User Identity concept, and we have stack introspection, and name space management to enforce the security policy. However, they are still discretionary access control (DAC)- see Figure 0-11. The mechanisms have weakness in that they fail to recognize a fundamental difference between human users and computer programs. They assume a benign environment where all programs are trustworthy and without flaws. So for high security system consideration, we use a Type-Enforcement (TE) [33] model for mandatory access control under UEFI environment. This design extends the application of MAC from a monolithic OS like Linux into the platform/Pre-OS/OS-absent space.

And given UEFI / BIOS is a fundamental technology that ships with the platform and should admit to management of its capabilities, this design can be extended to other technologies that ship with the platform. This is important because there is a single platform administrator for the OS (whether it's a hypervisor as the "OS" or a monolithic/static OS) and the other manageable elements on the machine. Having a single holistic way to express policy eases management tasks, but to the stronger, allows for expressing policy that spans the various execution regimes that cannot be done today.

This design works hand-in-hand with other guards, such as code-signing or TXT LCP [27], that allow for image invocation and isolation technology for reference monitor, such as VT-guarding of DXE core, EFI Byte Code (EBC) [1] sandboxing, and 3rd part bare-metal-executable isolation. This access-control infrastructure composes with these image authorization and isolation techniques by providing fine-grain, pre-OS runtime access control infrastructure for the rich, extensible Protocols of UEFI (www.uefi.org) and tables like the Advanced Configuration and Power Interface (ACPI) [7]. More importance, this design provides platform owner an access control policy engine that spans heterogeneous compute domains within a single platform.

Why is this important?
This design will allow platform manufacturers to create MAC (Mandatory-Access Control)-based UEFI deployments, for example. This is something that Multi-Level Security (MLS) or Multi-lateral Security deployments, such as government computer deployments, may require in the solution. Moreover, there is already MAC-based secure OS, such as Security-Enhanced Linux [23]. On the firmware side, UEFI should also have such kinds of support to co-work with OS.

And armoring the platform is an imperative business need. The Black Hat conference featured several attacks against the pre-OS and UEFI [24][25][26].

While many of the abuses have development items in place to address, the guarding or protection of API's is still an open item for the pre-OS. Again, recall the problem being addressed wherein UEFI & the ability to dynamically instantiate protocols creates a larger attack surface with respect to legacy BIOS. But the well-defined interfaces of UEFI protocols and the ability to name the UEFI executables as "subjects" that can be tracked via the loaded image protocol allows for binding "subject" rights to various API's. This allows for the platform supplier (PS), driver author, and/or local platform owner (PO)/administrator, to update these relationships. These relationships of subject/API can be thought of as a variant of an Access Control List (ACL).

How does it work?

The current set of interfaces in the UEFI environment is free to be called by all code subjects. In this design, we introduce two firmware TCBs (Trusted Computing Base): Type-Enforcement (TE) module for security policy, and a reference monitor to provide access control. A TE checker is part of reference monitor. It invokes the policy management interface to check the access right for each inter-module communication, such as service/protocol call, system resource (variable, file, system tale) access. See figure 0-1 for current API access, and figure 0-2 for API access with TE checker.

The reference monitor is used to monitor the whole system. Currently, we have prototyped an IsoBMX (Isolation for Bare Metal eXecutable), which is similar in scope to IsoDxe [12][29]. However, this design has no limitation on the reference monitor implementation.

The TE module is for Type-Enforcement policy management. It exposes policy management interface for TE checker TE checker. See figure 0-3 for the whole picture. As such, the DXE Core, TE module, and Reference Monitor are 3 major components in firmware Trusted Computing Base (TCB). Other UEFI drivers will run under Reference Monitor, and the access will be controlled by security policy.

The TE Module has 4 components: security service, access vector cache, audit module, and policy storage.
Security service provides support for policy check, policy load, system information registration, and context switch.

Access Vector Cache (AVC) contains the access policy information. Firmware policy is mandatory. OS policy information could also be used as a reference if needed.

Audit module provides a way to record the audit information, such as file system, , network, serial port, and etc.

Policy storage contains the policy. The policy binary should be stored in file system, UEFI flash variable, NAND partition, etc.

See figure 0-4, 0-5, 0-6, and 0-7.

How does TE work under UEFI?

Type and policy
In TE module, each component in UEFI environment, such as driver, protocol, system table, are assigned a type.

The policy rule has such format:
Rule SourceType TargetType : ObjectClass Permision;

Source type: Usually describes the domain type of a process attempting access

Target type: The type of an object being accessed by the process

Object class: The class of object that the specified access is permitted

Permission: The kind of access that the source type is allowed to the target type for the indicated object classes

Below are some types for UEFI:

DxeCore: DxeCore_t
SystemTable: EfiSystemTable_t
BootServiceTable: EfiBootServiceTable_t
RuntimeServiceTable: EfiRuntimeServiceTable_t
BootServices: EfiBootService_t
BootService->InstallProtocolInterface:
EfiBootService_InstallProtocol_t
AcpiSupport driver:
AcpiSupport_Code_t
SimpleFileSystem protocol:
SimpleFileSystem_Protocol_t

Below is some objects and permission for EFI:

Services: execute

Protocol: install, uninstall, reinstall, locate, execute

SystemTable: read, write

Image: load, start, unload, read, write, execute

Event: create, close, signal, check

EfiDriverBinding: start, stop, support

EfiServiceBinding: create_child, destroy_child

AcpiTable: create, delete, read, write

Variable: create, delete, read, write, get_name, get_attribut

- Type mapping

All the type information is determined at runtime. So we need an interface to register instance-type mapping. The TE module produces the interface and the reference monitor consumes the interface and is responsible for the registration.

For example, when IsoBMX initializes, it should register SystemTable address range as EfiSystemTable_t to TE module. So when TE module checks an access policy later, it can know what the EfiSystemTable_t is.

For a driver, when the DXE core create thunk for a BMX driver (e.g. AcpiSupport driver), the IsoBMX knows the address range for that, and registers the driver mapping as AcpiSupport_Code_t.

For a protocol instance, when a driver calls BS->InstallprotocolInterface (e.g. SimpleFileSystem protocol), the IsoBMX knows the meaning of this invocation, and it registers the protocol mapping as SimpleFileSystem_Protocol_t.

The same mechanism can be adopted for registration of Event services, Variable services, AcpiTable construction, etc.

- Domain transition

EFI is a flat environment where all components are co-located into the same protection domain. The DXE core will give a protocol access to another module. So the domain transition happens if an external function call happens AND TE policy marks it as allowed.
Here, we need IsoBMX as reference monitor to record the "CALL" action when the domain transition happens.

- UEFI Runtime

The OS could check the policy before invoking runtime services.
If the check passes, the OS should tell firmware the context.
When runtime services are invoked by OS, such as capsule services, the context is derived from OS.
The firmware could use this context to for TE check.

- System Management Mode (SMM) [28][31]

Use the same policy infrastructure as DXE core, the only difference is that DXE environment need to check SMM entry policy.

- Who gives the initial policy?

The initial policy storage should be built into flash or system processor (e.g.,ME [35]) firmware. So when firmware boot, it can use the default policy.
This is mandatory for MAC model, because MAC will deny all access except the allowed one.

- Who updates the policy?

When an owner is assigned to a platform, he/she could update the default policy.

A normal user should not be allowed to update the policy.

- How many code-base items need updating to support this design?

DXE Core, check BMX driver, and use of the BMX_PROTOCOL to handle accesses.
Add IsoBMX driver, as reference monitor.
Add PolicyEngine, as a TE module.
Add Audit support.
Add default policy binary into firmware.

This design can be extended into the OS runtime when the OS has an EFI virtual machine (i.e., runs EFI images during OS runtime inside of a container). Using this modality, the UEFI TE Policy could be mapped by the system administrator into native OS TE policies, such as Secure-Enhanced Linux (SELinux). Imagine a scenario where the EFI drivers from boot-time are re-invoked during OS runtime for some "safe-mode" scenarios. As such, the OS can allow for deep actions by the guest-hosted EFI drivers since it can ensure the rights of the sand-boxed drivers are consistent with the OS protection model (recall that an EFI VM at OS runtime has the OS as the EFI core, so in this scenario the TE module would actually be a main OS agency).

Below items show Policy example for AcpiTable: (See figure 0-8 for the ACPI table control)

Allow AcpiPlatform_Code_t {Acpi_Data_t, Acpi_Code_t}
: Acpi_table {read, write}
- AcpiPlatform driver can access ACPI table data, code.
Allow AcpiSupport_Code_t {Acpi_Data_t}
: Acpi_table {read, write}
- AcpiSupport driver can access ACPI table data.
Allow AcpiPlatform_Code_t AcpiSupport_Code_t
: protocol {execute}
- AcpiPlatform driver can execute AcpiSupport protocol.

Below items show Policy example for SMM:

Allow Smm_Code_t Smram_Data_t: Smram {read, write}
- Only Smm driver can read/write SMRAM.
Allow Smm_Code_t SmmReservedMemory_Data_t: ReservedMemory {read, write}
- Smm driver can read/write the memory reserved for SMM.
Allow Smm_Code_t {Smm_Protocol_t, Fv_Protocol_t}: protocol {execute}
- Smm driver can execute Smm protocol.
Allow Smm_Code_t Variable_Smm_t: Variable {read, write}
- Smm driver can access smm related variable.
Allow Smm_Platform_t Variable_Platform_t: Variable {read}
- SmmPlatform driver can only read Platform variable.

More details on the design can be found below. For example, below is the sample code for the policy check:

```
EFI_EXECUTION_POLICY_POLICY
BmxExecutionPolicyGetPolicy (
  IN  EFI_PHYSICAL_ADDRESS
SubjectAddress,
  IN  EFI_PHYSICAL_ADDRESS
ObjectAddress,
  IN  EFI_EXECUTION_ACCESS_TYPE
AccessType
  )
{
  EFI_TYPE_ENFORCEMENT_TYPE
SubjectTeType;
  EFI_TYPE_ENFORCEMENT_TYPE
ObjectTeType;
```

```
  SubjectTeType = BmxGetTeTypeFromAddress
(SubjectAddress);
  if (SubjectTeType == EfiTeTypeInvalid) {
    return EfiExecutionPolicyDeny;
  }

  ObjectTeType = BmxGetTeTypeFromAddress
(ObjectAddress);
  if (ObjectTeType == EfiTeTypeInvalid) {
    return EfiExecutionPolicyDenied;
  }

  return BmxGetPolicyEntryFromTeType
(SubjectTeType, ObjectTeType, AccessType);
}

EFI_STATUS
EFIAPI
BmxExecutionPolicyGetCallexPolicy (
  IN  EFI_EXECUTION_POLICY_PROTOCOL
*This,
  IN  EFI_PHYSICAL_ADDRESS
CodeAddress,
  IN  EFI_PHYSICAL_ADDRESS
FunctionAddress,
  IN  EFI_PHYSICAL_ADDRESS
StackAddress,
  OUT EFI_EXECUTION_POLICY_POLICY
*Policy
  )
{
  // ... ...
  if (FunctionAddress ==
(EFI_PHYSICAL_ADDRESS)(UINTN)gRT-
>GetVariable) {
    ObjectName = *(VOID
**)(UINTN)StackAddress;
    ObjectGuid = *(VOID
**)((UINTN)StackAddress + sizeof(UINTN));
    *Policy =
BmxExecutionPolicyGetVariablePolicy (
               CodeAddress,
               ObjectGuid,
               ObjectName,
               EfiExecutionAccessRead
               );
    return EFI_SUCCESS;
  }
  // ... ...

  *Policy = EfiExecutionPolicyAllow;
  return EFI_SUCCESS;
}
```

Below is sample code for external call in IsoBMX:

```
em_handler_error_status_t
emulator_call_handler(
  void
*caller_defined_state, // pointer to caller
defined state
  uint64               eip
// new eip virtual address
  )
{
  if (BmxCheckCallEx
((UINTN)EmulatorContext->em_eip,
(UINTN)eip)) {
    //
    // External call - thunk to native
    //

    //
    // Check policy for API invoke
    //
    Status = mExecutionPolicy-
>GetAccessPolicy (

mExecutionPolicy,

(EFI_PHYSICAL_ADDRESS)(UINTN)EmulatorContex
t->em_eip,

(EFI_PHYSICAL_ADDRESS)eip,

EfiExecutionAccessExecute,
                                    &Policy
                                    );
    if (!EFI_ERROR (Status)) {
      SendAuditItem (Policy,
AuditTypeExternalCall,
(UINTN)EmulatorContext->em_eip,
(UINTN)eip);
    }

    if (Policy == EfiExecutionPolicyAllow)
{
      //
      // Check policy for parameter check
      //
#ifdef EFIX64
      //
      // For x64 fill stack
      //
      *(UINTN *)EmulatorContext->em_esp =
EmulatorContext->em_ecx;
      *(UINTN *)(EmulatorContext->em_esp +
8) = EmulatorContext->em_edx;
      *(UINTN *)(EmulatorContext->em_esp +
16) = EmulatorContext->em_r8;
      *(UINTN *)(EmulatorContext->em_esp +
24) = EmulatorContext->em_r9;
#endif
      Status = mExecutionPolicy-
>GetCallexPolicy (

mExecutionPolicy,

(EFI_PHYSICAL_ADDRESS)(UINTN)EmulatorContex
t->em_eip,

(EFI_PHYSICAL_ADDRESS)eip,

(EFI_PHYSICAL_ADDRESS)(UINTN)EmulatorContex
t->em_esp,
                                    &Policy
                                    );
      if (!EFI_ERROR (Status)) {
        SendAuditItemEx (Policy,
AuditTypeCallFunction,
(UINTN)EmulatorContext->em_eip, (UINTN)eip,
(UINTN)EmulatorContext->em_esp);
      }
    }

    if (Policy == EfiExecutionPolicyAllow)
    {
      BmxCALLEX ((UINTN)eip,
EmulatorContext);

      SendAuditItem
(EfiExecutionPolicyAllow,
AuditTypeExternalRet,
(UINTN)EmulatorContext->em_eip,
(UINTN)eip);
    } else {
      EmulatorContext->em_eax =
EFI_SECURITY_VIOLATION;
    }
    return status;
  }
}
```

We also provide a demonstration of this capability on UEFI firmware.

Consider below TE Policy:

ALLOW {BmxDriver1_Code_t, BmxDriver2_Code_t} {SystemTable_t, BootServiceTable_t, RuntimeServiceTable_t}: SystemTable {read}

ALLOW {BmxDriver1_Code_t} {BootService_InstallProtocol_t, RuntimeService_SetVairblae_t} : Service {execute}
ALLOW {BmxDriver1_Code_t} {BmxTest_Variable_t}: Variable {read, write}

ALLOW {BmxDriver2_Code_t} {BootService_LocateProtocol_t, RuntimeService_GetVariable_t, RuntimeService_SetVariable_t} : Service {execute}
ALLOW {BmxDriver2_Code_t} {BmxDriver1_Code_t}: Protocol {read}
ALLOW {BmxDriver2_Code_t} {BmxTest_Protocol_Test_t}: Protocol {execute}
ALLOW {BmxDriver2_Code_t} {BmxTest_Variable_t}: Variable {read}

See the demo result on figure 0-9, 0-10.

For driver 1:

RT->GetVariable API is denied for Driver1, because it is not allowed.

For driver 2:

It is allowed to call driver1 BmxTestProtocol->Test(). In this API, writing to SystemTable is denied, because it is not allowed.

It is denied to call driver1 BmxTestProtocol->Test2(), because it is not allowed.

Although RT->SetVariable API is allowed, the EfiVariable is denied for Driver2.

Today, UEFI interfaces are exposed between all subjects, or programs. Going forward, there is a desire for inter-protocol isolation because some drivers may be signed, thus more privileged than others. Also, there may be business/competitive reasons to segregate information (e.g., elilo.conf for Linux boot subdirectory on ESP and BCD binary in the Windows® boot directory). This design allows for strict segregation of data and code accessors.

This design is different than Java class loader [34] in that the latter enforces the type-safety of the classes and has a different systems model. This design is customized for UEFI pre-OS and the extensible protocol and Boot/Runtime service capability. It leverages code-identity via the image loader and privilege via user settings or image signing to assign rights. This design complements code signing and hardware or sandbox-based isolation to guard the individual interface actions.

Today, all UEFI images are treated co-equal w.r.t. access rights. This creates a possible vector for malware. In this design, we can maintain compatibility by running today's drivers (perhaps unsigned) and simple limiting the possible API invocations, versus draconian measures of simply not running unsigned code and having a possible availability/reliability scenario (i.e., not booting).

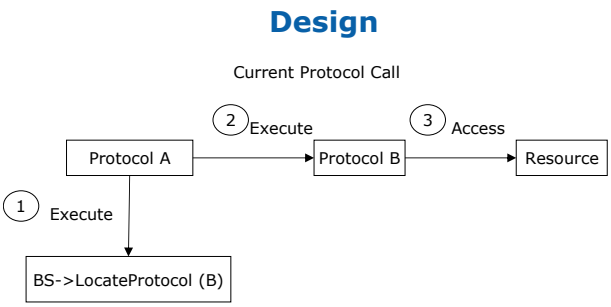


Figure 1-1 Current API access in UEFI

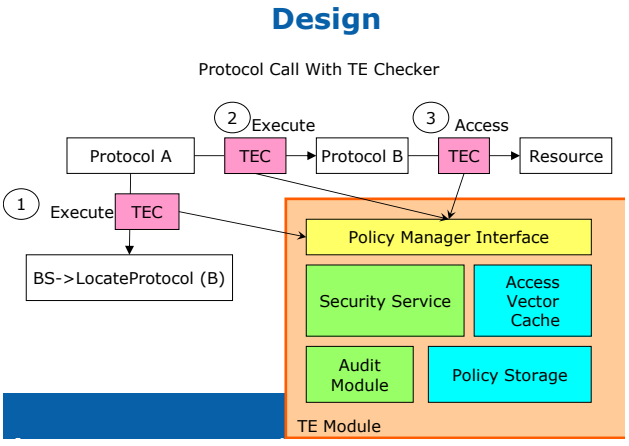


Figure 1-2 API access with TE Checker

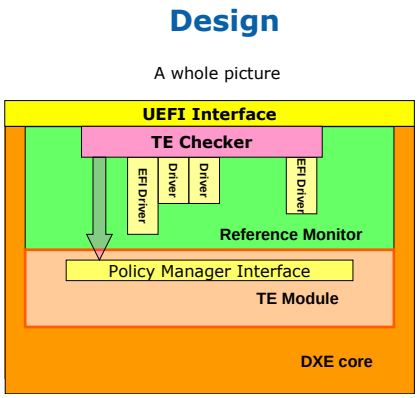


Figure 1-3 TE Module in UEFI Environment

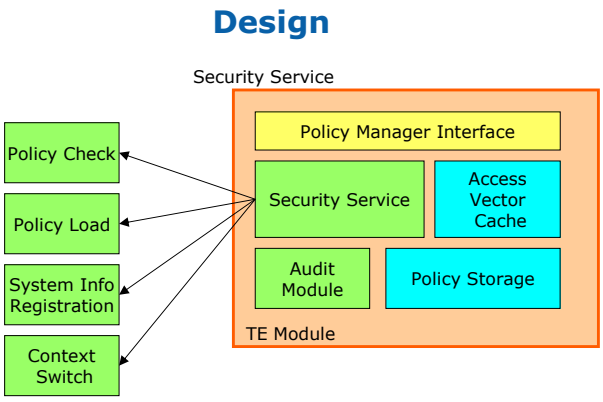


Figure 1-4 Security Service

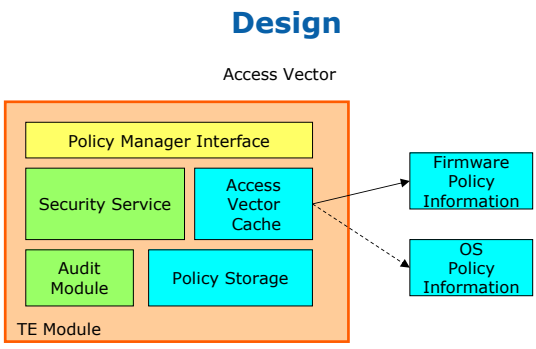


Figure 1-5 Access Vector Cache

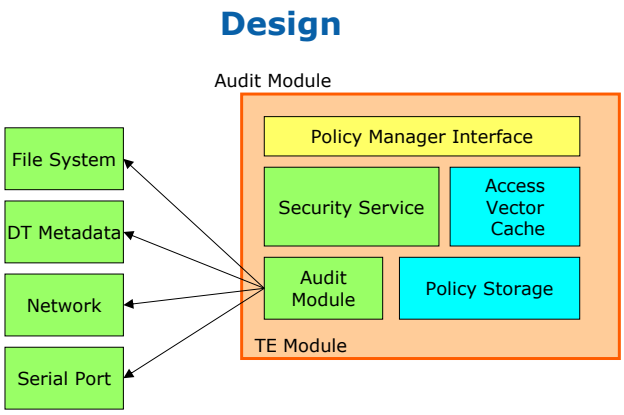


Figure 1-6 Audit Module

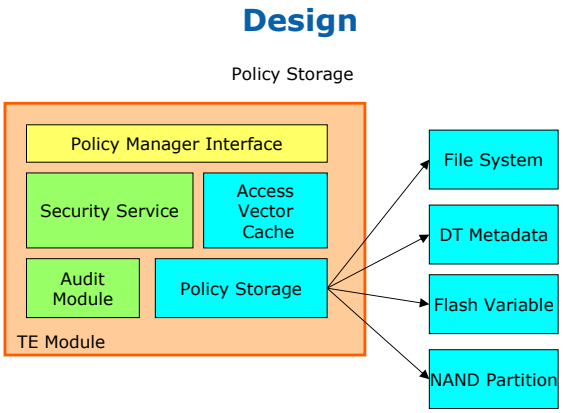


Figure 1-7 Policy Storage

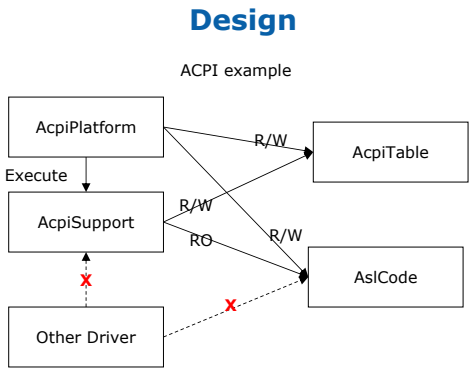


Figure 1-8 ACPI example

Demo

• Driver 1 load

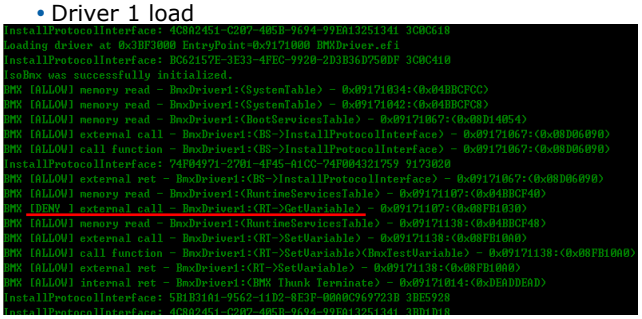
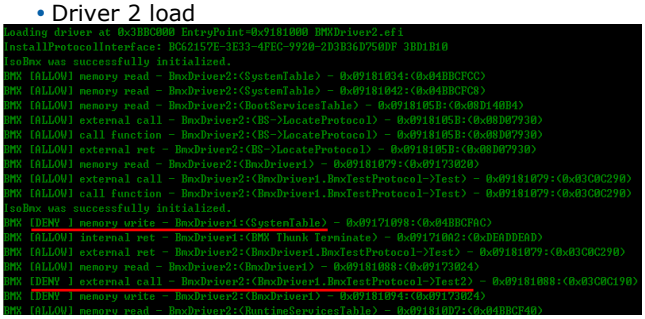


Figure 1-9 Demo – Driver1

Demo

• Driver 2 load



Demo

• Driver 2 load (Cont'd)

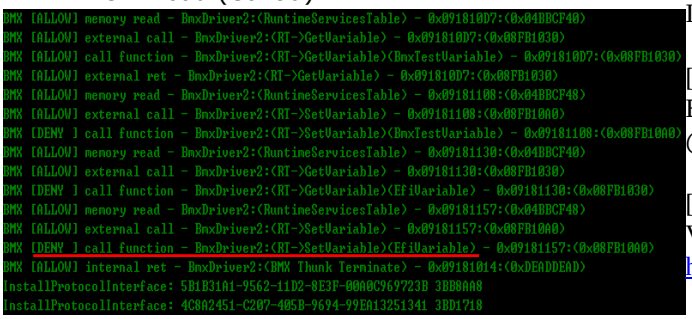


Figure 1-10 Demo – driver2

Related work

This design complements implementations of secure boot, whether BIOS-based [2] or UEFI-based [16], but really entails use of a reference monitor in the pre-OS for mediation and a pre-OS policy module. To-date, this type of work has been targeted for either hypervisor or static OS's, such as IBM's sHype additions to Xen [11] and SELinux, respectively.

Conclusion

UEFI platform manufacturers 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 policy-based access control. This design will ensure that the promise and value of UEFI's extensibility does not become an issue in market deployment.

This design is unique in that it allows platforms to maintain compatibility wifth 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

[4] Vincent Zimmer, "Advances in Platform Firmware Beyond BIOS and Across all Intel® Silicon", Technology @ Intel Magazine, January 2004

[5] Trusted Computing Group EFI Protocol Specification, 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] Sailer, et al. "Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor," Proceedings of the Annual Computer Security Applications Conference (ASAC), 2005

[12] Vincent Zimmer, "System Isolation Beyond BIOS Using the Unified Extensible Firmware Interface," in *Proceedings of the 2008 International Conference on Security And Manageability, SAM'08*, CSREA Press, June 2008

[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 Manageability, 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] Heaseman 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 www.pcisig.org

[31] UEFI Platform Initialization Specification, Version 1.1, 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] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments." In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, October 1998

[34] Li Gong, "Java Security: Present and Near Future," IEEE Micro, Volume 17, Issue 3, May 1997, Pages: 14 - 19

[35] Active Management Technology (AMT) http://www.intel.com/technology/platform-technology/intel-amt/index.htm