Race to bare metal:

UEFI and hypervisors

Agenda

1. Race to bare metal
2. Overview UEFI
3. UEFI in practice
4. Advantages of UEFI for anti/malware hypervisors
5. Some practical considerations

1. Race to bare metal

Importance of early loading

- Natural protection from hypervisor malware is installing another (security) hypervisor first
- After one of them is active, “game is over” for the other one
- That means, both should strive to get loaded as soon as possible

Importance of loading mechanism

- Weak point in preventing detection
- Weak point in security
- Early loading greatly increases complexity

2. Overview of UEFI

[Unified] Extensible Firmware Interface (EFI / UEFI)

- Pre-boot firmware interface (sort of BIOS replacement)
- Architecture-independent (portable: x86, x86-64, Itanium, ARM)
- C interface
- Big part of implementation available as open-source

Why replace BIOS?

- Outdated 16-bit real mode assembly interface
- Lack of functionality
- Lack of extensibility
- Short supply of real mode assembly programmers

History of UEFI

- mid 90s - First steps towards EFI
- 2000 - Itanium
- 2003 - x86-32
- 2005 - UEFI Forum
- 2006 - Intel-based Macs
- 2008 - x86-64
- 2008 - Vista SP1

UEFI Forum

- Unified EFI
- Intel, AMD, ARM, Dell, HP, Apple, IBM, Lenovo, Phoenix, AMI, Insyde, Microsoft
- UEFI Specification, Platform Initialization Specification
- See: http://www.uefi.org

Design

- Architecture-independent
- Modular
- Extensible

Features

- Console I/O
- Graphics
- Unicode
- Remote debugging
- Bytecode
- Networking (IPv4, IPv6, IPsec, TCP, UDP, FTP, PXE…)
- ACPI
- PCI bus support
- SCSI stack
- USB stack
- User management
- Filesystem access
- Secure boot, code validation

Implementation

- Intel Tiano / The Framework
- TianoCore (see http://www.tianocore.org)
- InsydeH2O, InsydeDIY
- Phoenix SecureCore, TrustedCore, AwardCore Tiano
- AMI Aptio

UEFI-BIOS relationship

- UEFI only (Non-x86 machines, Macs)
- Optional UEFI on top of legacy BIOS (non-Mac x86 PCs)

Current state of UEFI

- Pushed forward by all major players
- Standard on Itanium machines and Macs
- Available on non-Mac PCs as option
- Supported by 64-bit Windows Vista SP1
- Supported by grub and elilo (EFI lilo)

3. UEFI in practice

UEFI boot process

- Security phase (SEC)
- Pre-EFI Initialization (PEI)
- Driver Execution Environment (DXE)
- Boot device selection
- OS Boot loader / EFI application
- ExitBootServices()
- Run Time

UEFI Pre-boot (DXE) Environment

- Uniprocessor
- Protected mode in 32/64-bit mode
- Paging disabled, or identity-mapped
- Only timer interrupt

UEFI boot manager

- Controlled by NVRAM variables:
- Boot####, BootOrder, BootNext
- Driver####, DriverOrder
- When UEFI Boot is enabled in BIOS, this has priority over legacy BIOS boot from MBR

UEFI Drivers

- Boot Drivers

  unloaded on ExitBootServices() call

- Runtime Driver

  persists until shutdown, preserved and respected by OS

4. Advantages of UEFI for anti/malware hypervisors

Loading

- Earliest possible without reflashing BIOS
- Secure boot (code validation)
- Legal loading mechanism - no hacks needed
- Very easy to implement and install

Stealth

- Untouched by OS
- No “missing” resources
- Novel technology omitted by current security products
- Loading mechanism easy to conceal

Code complexity reduction

- Common routines available
- Transitional modes virtualization
- Minimal code required for loading mechanism

Pre-boot features

- Limited disk access
  - Data gathering
  - Validation of system boot
- Full network access
  - Data sending
  - Self-updating
  - Remote activity logging

5. Some practical considerations

Adding NVRAM variable

- SetVariable() EFI runtime service
- Undocumented APIs on Vista:

  NtAddDriverEntry, NtSetDriverEntryOrder, NtQuerySystemEnvironmentValue, etc…
- /dev/nvram on linux?

NtAddDriverEntry()

Prototype:

```
NTSTATUS NtAddDriverEntry(
        EFI_DRIVER_ENTRY *DriverEntry,
        DWORD Id
    );
```

Returns:

STATUS_SUCCESS (zero) when okay,

NTSTATUS error code upon failure

NtAddDriverEntry()

```
struct EFI_DRIVER_ENTRY {
    DWORD Version;          // must be 1
    DWORD Length;           // must be at least 20
    DWORD Id;               // at most 0xFFFF
    DWORD FriendlyNameOffset;  // from beginning of struc
    DWORD DriverFilePathOffset; // detto
    BYTE data[1];
}
```

DriverFilePathOffset holds offset to FILE_PATH structure

NtAddDriverEntry()

```
struct FILE_PATH {
    DWORD Version;          // must be 1
    DWORD Length;
    DWORD Type;
    BYTE path[1];
}
```

Type can be: 2=ARC Path, 3=NT path, 4=EFI path.

Real mode with VMX

- Intel VMX doesn’t yet (?) support real mode virtualization
- 64-bit Vista emulates real mode code - no problem
- UEFI still calls some real mode code, by switching back to real mode - problem
- On Intel CPUs, we still need real-mode emulation :(

Enabling virtualization

- Virtualization may be disabled by BIOS
- Once disabled, it can’t be re-enabled programatically
- However, on some BIOSes we can overwrite this BIOS setting programatically

Alternative: PCI Option ROM

- UEFI (like legacy BIOS) automatically loads Option ROM code for every PCI device which has some
- No traces of loading left (no NVRAM modification needed)
- No extra file on disk
- PCI Option ROM is easier to virtualize than file on disk

Links

- www.uefi.org
- www.tianocore.org
- Also see my articles "*Introduction to UEFI*" and "*UEFI programming: First steps*" at www.x86asm.net.

Q&A

Email: vid512@gmail.com