Implementing and Detecting a PCI Rootkit

John Heasman [john@ngssoftware.com]
15th November 2006

An NGSSoftware Insight Security Research (NISR) Publication

Introduction

In February 2006, the author presented a means of persisting a rootkit in the system BIOS via the Advanced Configuration and Power Interface (ACPI). It was demonstrated that the ACPI tables within the BIOS could be modified to contain malicious ACPI Machine Language (AML) instructions that interacted with system memory and the I/O space, allowing the rootkit bootstrap code to overwrite kernel code and data structures as a means of deployment[1]. Furthermore, the high level instructions processed by the AML interpreter (typically contained within the ACPI driver) meant that it was possible to interrogate the operating system and hardware in order to accurately determine the platform and specific version of the OS. This potentially allows a multi-platform future-proof rootkit to be created.

Whilst using ACPI as a means of persisting a rootkit in the system BIOS has numerous advantages for the rootkit writer over "traditional" means of persistence (that include storing the rootkit on disk and loading it as a device driver), there are several technologies that are designed to mitigate this threat. Both Intel SecureFlash and Phoenix TrustedCore motherboards prevent the system BIOS from being overwritten with unsigned updates. Furthermore, it is relatively easy to detect an ACPI rootkit by disabling ACPI in the CMOS setup program, or by booting from read-only media that does not load an ACPI device driver and auditing the ACPI tables located in system memory (in essence, this is the same cross-view detection method that is typically used to locate a rootkit on disk).

This paper discusses means of persisting a rootkit on a PCI device containing a flashable expansion ROM. Previous work in the Trusted Computing field has noted the feasibility of expansion ROM attacks (which is in part the problem that this field has set out to solve), however the practicalities of implementing such attacks has not been discussed in detail. Furthermore, there is little knowledge of how to detect and prevent such attacks on systems that do not contain a Trusted Platform Module (TPM). Whilst the discussion mainly focuses on the Microsoft Windows platform, it should be noted that the techniques are equally likely to apply to other operating systems. The following sections provide a concise overview of PCI, expansion ROM and BIOS boot technologies; for more detailed information the reader is advised to consult the relevant specification.

Introduction to the PCI Bus

The Peripheral Component Interconnect (PCI) specification, developed by Intel c.1990, describes a computer bus for attaching peripherals or other buses to the motherboard. PCI devices have become ubiquitous over the last fifteen years and a typical system is likely to contain several, including a graphics card, a network card and a storage controller (e.g. a SCSI card). These devices are connected to a bus; PCI buses are interconnected as follows:

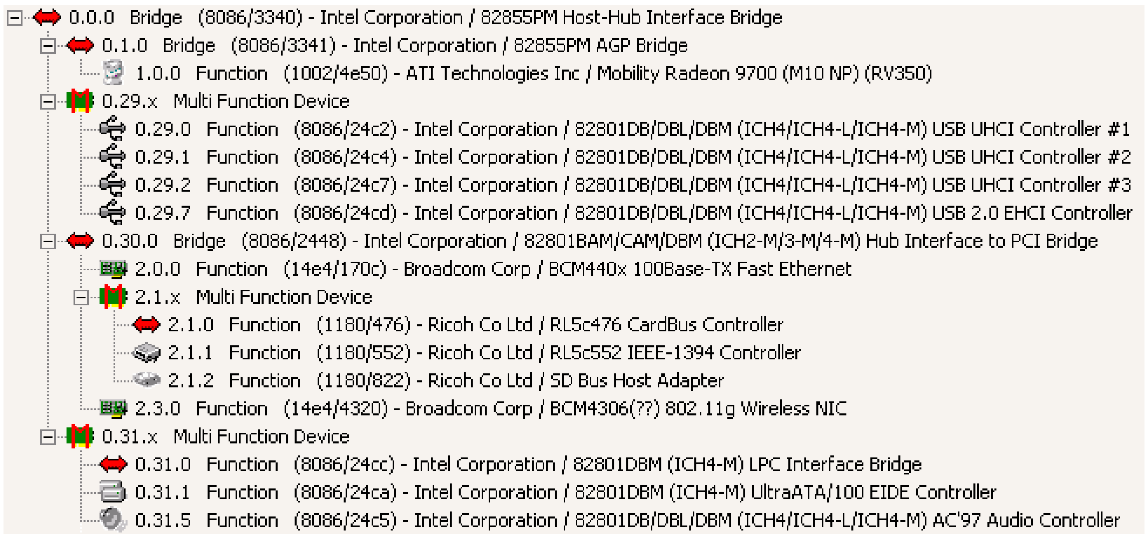
- The host/PCI bridge, often referred to as the North Bridge, connects the host processor bus to the root PCI bus.

- The South Bridge connects the root PCI bus to the ISA bus, and also typically incorporates the Interrupt Controller, the IDE controller, the USB Host Controller and the DMA Controller.

- The root PCI bus may also contain one or more PCI-to-PCI bridges.

- In PCI terminology, a “function” is a PCI device that performs a single, self-contained function, such as a video adapter or serial port. A single physical PCI component may actually contain up to eight functions.

A sample PCI bus from a notebook computer is shown in Figure 1. Devices such as graphics cards quickly approached the data transfer limits of the PCI standard prompting Intel to release the Accelerated Graphics Port (AGP) specification in 1997 and the PCI Express (PCIe) specification in 2002. PCIe is seen as a long term replacement to both PCI and AGP, with version 2.0 of the specification to be released in 2007. However, since the functionality discussed in this paper is common to PCI, AGP and PCIe, the term “PCI” is used hereafter to refer to any of the preceding technologies.



```
0.0.0   Bridge   (8086/3340) - Intel Corporation / 82855PM Host-Hub Interface Bridge
  0.1.0   Bridge   (8086/3341) - Intel Corporation / 82855PM AGP Bridge
    1.0.0   Function   (1002/4e50) - ATI Technologies Inc / Mobility Radeon 9700 (M10 NP) (RV350)
  0.29.x   Multi Function Device
    0.29.0   Function   (8086/24c2) - Intel Corporation / 82801DB/DBL/DBM (ICH4/ICH4-L/ICH4-M) USB UHCI Controller #1
    0.29.1   Function   (8086/24c4) - Intel Corporation / 82801DB/DBL/DBM (ICH4/ICH4-L/ICH4-M) USB UHCI Controller #2
    0.29.2   Function   (8086/24c7) - Intel Corporation / 82801DB/DBL/DBM (ICH4/ICH4-L/ICH4-M) USB UHCI Controller #3
    0.29.7   Function   (8086/24cd) - Intel Corporation / 82801DB/DBL/DBM (ICH4/ICH4-L/ICH4-M) USB 2.0 EHCI Controller
  0.30.0   Bridge   (8086/2448) - Intel Corporation / 82801BAM/CAM/DBM (ICH2-M/3-M/4-M) Hub Interface to PCI Bridge
    2.0.0   Function   (14e4/170c) - Broadcom Corp / BCM440x 100Base-TX Fast Ethernet
    2.1.x   Multi Function Device
      2.1.0   Function   (1180/476) - Ricoh Co Ltd / RL5c476 CardBus Controller
      2.1.1   Function   (1180/552) - Ricoh Co Ltd / RL5c552 IEEE-1394 Controller
      2.1.2   Function   (1180/822) - Ricoh Co Ltd / SD Bus Host Adapter
    2.3.0   Function   (14e4/4320) - Broadcom Corp / BCM4306(??) 802.11g Wireless NIC
  0.31.x   Multi Function Device
    0.31.0   Function   (8086/24cc) - Intel Corporation / 82801DBM (ICH4-M) LPC Interface Bridge
    0.31.1   Function   (8086/24ca) - Intel Corporation / 82801DBM (ICH4-M) UltraATA/100 EIDE Controller
    0.31.5   Function   (8086/24c5) - Intel Corporation / 82801DB/DBL/DBM (ICH4/ICH4-L/ICH4-M) AC'97 Audio Controller
```

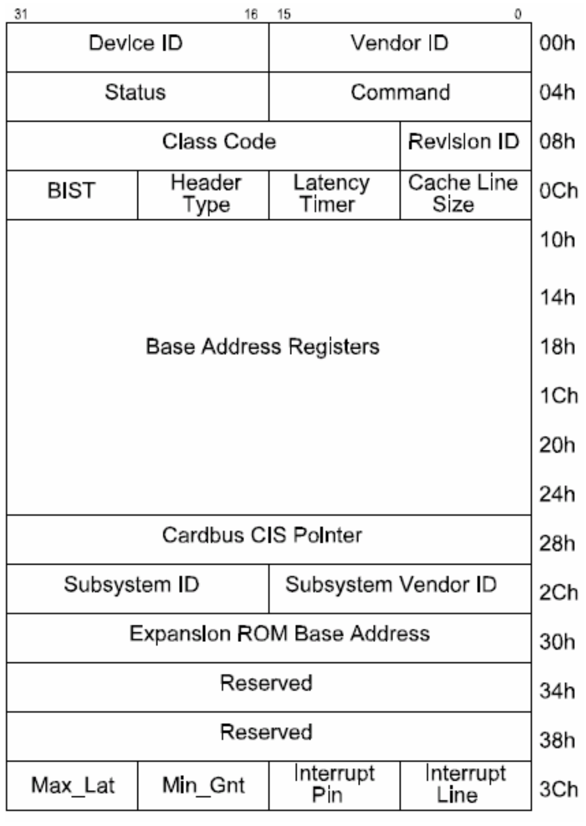
Figure 1: The logical representation of a typical PCI bus

The Power On Self Test

When a system is powered, a pre-boot sequence known as the Power On Self Test (POST) is performed during which time the system BIOS is typically copied from flash ROM to system memory. Execution of the system BIOS then begins to ready the system for boot. It is at this point that the PCI bus is scanned to determine what devices exist. In order to facilitate the walking of the bus, each PCI function implements a set of configuration registers as defined by the PCI specification. These can be used to determine the type of the device (known as the class code), the vendor (via the vendor ID register) and whether the device contains an expansion ROM (covered in more detail in the next section). The system BIOS accesses these registers, located in the PCI configuration space, via I/O.

Expansion ROMs

Many PCI cards contain an expansion ROM that holds additional code required to initialise the card during execution of the system BIOS. This code is also responsible for carrying out the device-specific self-test and hooking required interrupts. The presence of an expansion ROM is determined via the Expansion ROM Base Address Register within the PCI function’s Configuration Header, as shown in Figure 2.



31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	Cache Line Size	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved				34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

Figure 2: The PCI Configuration Header

The exact sequence of steps is as follows (taken from PCI System Architecture[2]):

- Assign a base address (representing physical system memory) to the register's Base Address field
- Enable its decoder by setting bit 0 in the register to 1
- Set the Memory Space bit in the function's Command Register
- Read the first two locations from the base address, to verify the presence of the expansion ROM signature (the word 'AA55h').

After enumerating the PCI bus and copying expansion ROMs to memory, the system BIOS scans the following regions in order to locate valid ROMs:

- 0xC0000 - 0xC7FFF, on 2KB boundaries for a VGA ROM
- 0xC8000 - 0xF0000, on 2KB boundaries, for non-VGA ROMs

It then executes from offset 03h in the ROM, which is typically a short jump to the ROM initialisation code required for the card to function correctly. It is recommended that the interested reader download a BIOS update for a graphics card from the manufacturer’s website. The download is likely to consist of a tool to perform the flash update and a binary ROM file. Disassembling the ROM as 16-bit code from offset 03h should reveal that the initialisation code installs itself as the handler for interrupt 10h in the Interrupt Vector Table (IVT). This is expected since interrupt 10h is the VGA interrupt used for outputting data to the screen.

It is worth noting that the expansion ROM does not necessarily hold x86 code nor does it have to contain a single ROM image. The code type field within the ROM data structure within the image specifies the presence of x86 code or OpenBoot interpretive code (documented in the Open Firmware standard).

The expansion ROM is stored on either an EPROM, or more commonly on an EEPROM. EPROMs require that the chip is removed from the card and erased via exposing it to strong ultraviolet light before it can be reprogrammed. EEPROMs, however, can be erased electrically, in-circuit, thus the card need not be removed from the system and can be re-flashed from the operating system.

Re-flashing a PCI Expansion ROM

Vendors of newer PCI cards often provide tools that can be used to flash the card from within Windows (provided the user has administrative privilege). Tools exist for older cards but these often require that the user boot off a DOS boot disk. Running the tool under the Windows Virtual DOS Machine Subsystem (NTVDM) is unlikely to work for reasons discussed shortly. Tools that can be run under Windows typically install a device driver that uses the Plug'n'Play Manager's IRP_MN_QUERY_INTERFACE and IRP_MN_READ_CONFIG/IRP_MN_WRITE_CONFIG requests or call HalGetBusData on Windows 2000 and earlier (this function has subsequently been made obsolete). Analysis of the IRP_MN_WRITE_CONFIG request reveals that ultimately I/O is performed to re-flash the card.

By default, only restricted I/O can be performed from user-mode since normal processes operate with an I/O privilege level (IOPL) of zero. This explains why running DOS-based tools to re-flash PCI cards under NTVDM does not work - the process is unable to perform unrestricted I/O. It is in fact possible to elevate a process to an IOPL of three on Windows, permitting unrestricted I/O from user-mode. In order to perform this, the user must have the SeTcbPrivilege and call the undocumented Native API function, NtSetInformationProcess with a process information class of ProcessUserModeIOPL. Once the user can perform unrestricted I/O, they can potentially re-flash the card without having to load a driver.

This raises the possibility of (1) a remote attack that yields LocalSystem privilege (such as the server service vulnerability patched in update MS06-040) being used to deploy a malicious expansion ROM, (2) a browser exploit, that, if the user is running under the administrative context, obtains SeTcbPrivilege and re-flashes a card.

Cards that contain an EPROM are not at risk from remote attacks that re-flash the expansion ROM since these require the chip to be removed from the card and erased by exposing it to a strong ultraviolet light. EPROMs could potentially be abused if an attacker has physical access to a system; alternatively an attacker could pre-infect a card if they know that it will be put in a specific system.

Subverting the NT Kernel from an Expansion ROM

A re-flashed expansion ROM permits the rootkit writer to execute code during the POST (as the system BIOS executes the ROM initialization code), after the POST and whenever a hooked real mode interrupt is called. However, controlling the operating system from this point is not necessarily a trivial task. A difficulty noted by the authors of BootRoot[3], a proof-of-concept boot sector rootkit written by Derek Soeder and Ryan Permeh, is that when the initial code executes after the POST, no part of the operating system has been loaded, and that during boot, the OS start-up brings about dramatic changes in system state (such as switching from 16-bit real mode to 32-bit protected mode) during which control must be maintained (or at least “scheduled to re-activate”). BootRoot hooks interrupt 13h, the interrupt that provides the low level disk services in order to patch the operating system loader so that it then patches key operating system files such as device drivers.

It is possible to analyse the interrupts called during Windows boot using Bochs, the open source x86 emulator[4]. Bochs contains an integrated debugger allowing the user to set a breakpoint on CPU mode switches as well as the execution of interrupts. It was noted that relatively late during the Windows boot, interrupt 10h is called. Further analysis revealed that this occurred though a kernel export, Ke386CallBios, whose prototype is given below:

```
NTSTATUS NTAPI Ke386CallBios(ULONG, PCONTEXT)
```

When this function is called, the operating system is in 32-bit protected mode, yet the first parameter represents the real mode interrupt to be called via the IVT. In order to carry out the interrupt, this function therefore results in a transition to Virtual 8086 mode, also called virtual real mode or VM86. The author determined that by hooking interrupt 10h via an expansion ROM, it is possible to explicitly detect the call from VM86 mode Ke386CallBios and subsequently locate and modify key kernel structures such that when the operating system returns to protected mode, arbitrary attacker code is executed with kernel privilege from the expansion ROM. This code could be used to subvert the kernel directly or bootstrap another rootkit component.

It should be noted that there are likely several other means of obtaining code execution in the operating system, given the ability to execute code during POST, immediately after POST, and when a specific real mode interrupt is called.

A Pre-Boot Means of Updating a Rootkit

Rootkits typically require some means of communication with a remote host, either to allow the controller to modify runtime settings or to retrieve data of importance. Indeed, the rootkit deployed on the infected host may only serve to bootstrap a rootkit that is loaded via network. This was the approach demonstrated by BootRoot, which patched the network driver so that it executed code directly out of specially crafted Ethernet frames. This approach is likely to require an intermediate period of building up the code since it is unlikely that the entire rootkit could be fitted into a single Ethernet frame. From the rootkit writer's perspective, the problem with a rootkit that updates itself via conventional means of communication such as Transport Dispatch Interface (TDI, the network abstraction layer presented by the NT kernel), is that the detection surface expands significantly. Personal firewalls that operate at the Network Driver Interface Specification (NDIS) level would be expected to trap this kind of communication unless the rootkit itself manipulates NDIS (e.g. as is the case with Joanna Rutkowska’s Deepdoor[5]).

As an alternative to updating once the operating system has loaded, communication with the network card could be performed pre-boot during execution of expansion ROM initialisation code. This would require low level knowledge of the network card in order to send and receive frames as well as a barebones implementation of a network stack. Depending on the card, this may not be especially difficult. Simple 16-bit real mode code to perform this for cards based on the NE2000 specification is readily available on the Internet, for example.

Introduction to PXE

PXE is the Preboot Environment developed by Intel as part of the "Wired for Management" initiative. The most recent version of the PXE specification is version 2.1, released in September 1999. PXE is commonly used for the following scenarios:

- Remote new system setup; the client machine downloads a Network Bootstrap Program (NBP) that automatically or interactively carries out installation of a new operating system.

- Remote emergency boot; the client machine downloads diagnostic tool or framework when a hardware or software component fails preventing normal boot.

- Remote network boot; the client machine may be set up as a diskless workstation, in which case PXE can be used to load an NBP that subsequently loads the entire operating system over the network.

PXE is built on industry standard protocols such as TCP/IP, DHCP and TFTP. In an environment that is PXE-enabled, the client sends an initial DHCP discover packet containing the PXE client extensions tag. The resulting DHCP offer will contain the PXE server extensions tag, providing the client with the IP address of the PXE server. The client is then able to retrieve the specified NBP via TFTP. This is obviously a simplified description of PXE; in reality there are numerous PXE related DHCP tags. For further details the reader should consult the PXE specification.

PXE is implemented as x86 code within an expansion ROM. The PXE ROM provides a set of API services corresponding to the layers two to five in the OSI reference model. In addition, the PXE ROM contains "base code", which represents the actual PXE program that is executed when network boot is initiated. The PXE APIs consist of:

- Pre-boot API; the pre-boot API provides functionality for initialising the UNDI ROM, described below, and starting execution of base code.

- Trivial File Transport Protocol (TFTP) API; the TFTP API provides, as one would expect, a set of functions for transferring an image into memory via TFTP. Functions include "TFTP OPEN", "TFTP CLOSE", "TFTP READ", "TFTP/MTFTP READ FILE", and "TFTP GET FILE SIZE".

- User Datagram Protocol (UDP) API; this API provides a set of function to send and receive UDP packets consisting of "UDP OPEN", "UDP CLOSE", "UDP READ" and "UDP WRITE".

- Universal Network Driver Interface (UNDI) API; The UNDI API is the lowest level PXE API, and is responsible for communicating with the card (i.e. performing the required I/O) in order to send and receive network frames. This part of the PXE ROM is therefore likely to be network card specific. The other APIs build on this in order to present useful services to the base code.

Later implementations of PXE often consist of a set of expansion ROMs rather than a monolithic ROM as is commonly found in old implementations. This has the advantage of allowing specific parts of the PXE implementation to be separated so that multiple devices can implement PXE without a linear increase in ROM storage requirements. It also has the advantage of allowing other components, such as the system BIOS, to make use of specific part of the PXE API set, such as UNDI, in order to perform its own pre-boot network communication. For the sake of brevity, this paper uses the term "PXE ROM" to mean to set of all PXE related ROMs present on the system.

If a PXE ROM is present on the network card or in the system BIOS, network boot can be selected as an Initial Program Load (IPL). The BIOS Boot Specification[6] details the structures that an expansion ROM must contain in order that the device is considered as an IPL and added to the IPL table. The user selects the IPL priority from the boot sequence screen within the BIOS Setup program, or dynamically during POST by pressing a specific hotkey key. The weakness in IPL that makes it open to abuse is support for the legacy cards that do not implement the required IPL structures. The BIOS Boot Specification states that:

"If a Legacy card’s option ROM code hooks INT 19h during its initialization call it controls the boot process."

Thus all a rogue option ROM is required to do in order to be assured of gaining control of initial execution after the POST, is to ensure that it hooks interrupt 19h (and that prior to leaving POST, a legitimate legacy ROM present on the system has not inserted its own hook).

Abusing PXE

With modifications to the base code, or by supplying alternate base code altogether, it is possible to subvert PXE in order to carry out a pre-boot update of a rootkit. Firstly, the rootkit's expansion ROM must hook interrupt 19h as described above, storing the existing address so that the normal boot sequence can be carried out post-update. Next, the PXE ROMs should be initialised and the rootkit base code should be executed. This uses the PXE APIs to obtain an IP address and contact the rootkit controller; the response from the rootkit controller indicates the presence of an update. The update is downloaded via TFTP and stored in memory, or re-flashed directly on to the appropriate card. The base code can implement features designed to reduce the chances of detection, such as request masking (out-bound requests are disguised as DNS), scheduled updates based on the real time clock, and compression and encryption of data sent from the rootkit controller to the client. Thus in the rootkit deployment phase, the attacker will seek to either replace the PXE ROM (containing the modified base code), or provide the new base code in another expansion ROM. If the attacker elects to replace the PXE ROM entirely, they may choose to replace it with a modified version of the original ROM, or a ROM created by a tool such as Etherboot[7].

Etherboot is open source network boot ROM creation tool that supports a huge number of network cards and actually goes beyond the requirements of the PXE specification to implement APIs that provide TCP and HTTP services. It is feasible that an attacker may attempt to re-flash a network card with an entirely new Etherboot ROM since on a stable system the user is unlikely to regularly select PXE as the IPL, and therefore will not notice the difference (assuming Etherboot has been modified to operate with stealth, i.e. by not outputting to the screen).

Detection Using Off-The-Shelf Tools

Whilst researching expansion ROM attacks, the author created several proof-of-concept rootkits based directly on the techniques described in this paper; these were tested on a popular model of graphics card. In attempting to detect such rootkits, one should remember that expansion ROMs are first copied into system memory, and that the code they contain is executed by the system processor, i.e. the rootkit does not use the processor on the graphic card, nor does it use memory on the PCI card that cannot be accessed from kernel mode. Thus in detecting a rootkit that has been persisted on a PCI card, the first step should be to detect how the rootkit has subverted the kernel (e.g. the hooks it has in place and the kernel structures it has modified). There are many Windows rootkit detection tools that operate a variety of different approaches; consequently, they often complement each other when hunting rootkits. A list of popular rootkit detection tools may be found at the Anti-Rootkit website[8].

Auditing Expansion ROMs

The ROM can be retrieved from system memory, or from the card itself. In order to carry out an audit, both should be retrieved. It is not uncommon for these to differ in size and content - the PCI specification elaborates that the initialisation code can be discarded, and that the ROM located in system memory can resize itself in order to conserve space. Expansion ROMs can be retrieved from user-mode pre-Windows 2003 SP1 via the \device\PhysicalMemory device, that can be opened for read access by administrators. From Windows 2003 SP1, access to \device\PhysicalMemory has been prevented from user-mode.

In order to retrieve the ROM from the card itself, it is necessary to follow the sequence of steps discussed earlier. This is likely only to be feasible from kernel mode since a region of physical memory must be allocated prior to enabling the Memory Space bit in the function's Command Register.

Having obtained the ROMs from system memory and the card itself, these should be compared to known good ROMs obtained from vendor websites. Ideally, vendors should carry downloads of all previously released ROMs and their corresponding hashes however, a brief survey suggested that majority currently only link to the most recent ROMs.

If the ROM differs from a known good ROM, it should be investigated in the same manner that malware is analysed. The following indicators suggest that the ROM may contain malicious functionality:

- The class code within the ROM data structure does not match that returned by the PCI configuration register within the device's configuration header. This indicates that a ROM has been placed on the device for non-device related purposes.

- The ROM hooks or calls interrupts not typically associated with the functionality of the card (e.g. whilst the PXE ROM is likely to contain interrupt 10h calls in order to display output to screen, it is extremely unlikely to hook interrupt 10h itself).

- The ROM hooks interrupt 19h. This may be indicative of a legacy ROM that does not implement the required IPL data structures, or it may indicate the presence of malicious code that requires control of execution after the POST.

- The presence of 32-bit code. If the code type in the ROM data structure indicates x86, the expansion ROM is expected to contain 16-bit real mode code, at least for the initialisation code. Some ROMs such as those generated by Etherboot contain 32-bit protected mode code, however upon disassembly one can clearly follow switches between real mode and protected mode. The presence of isolated 32-bit code that is seemingly not accessed by any other code in the ROM should be treated as suspicious.

- Operating system specific references. Evidence of strings or 32-bit addresses that lie within common NT kernel modules should be considered indicative of a rootkit.

- Code that appears to be heavily obfuscated. It is not uncommon for a ROM to contain compressed code that is decompressed during initialisation, however the presence of code that is heavily obfuscated and difficult to follow in a disassembler should be treated as suspicious.

Preventative Measures

General rootkit prevention steps typically involves keeping the system and all third party software fully patched as well as running a personal firewall and anti-virus software. As an additional step, the user can write protect the firmware of certain PCI cards via a physical switch or jumper, as can be seen in Figure 3.

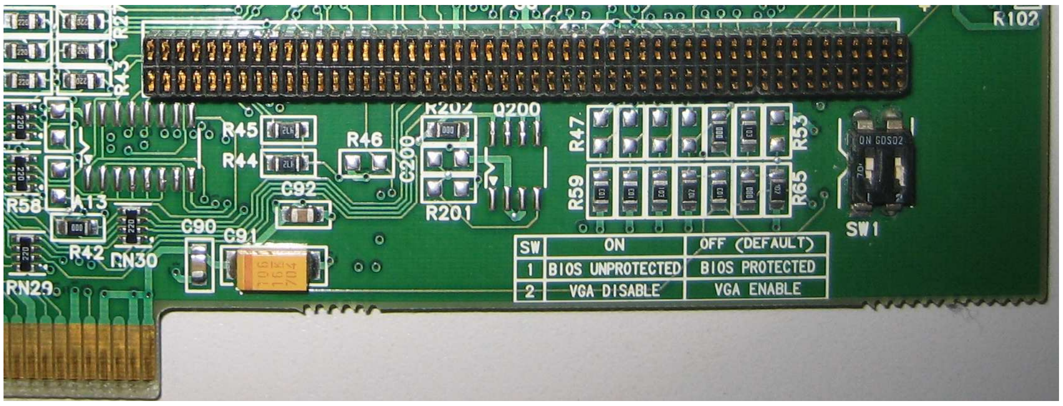


Figure 3: A PCI card that has BIOS protect switch

The author has noted that newer cards seem not to have this ability, perhaps because of the additional cost, or perhaps because user feedback indicated it was inconvenient or redundant. Furthermore, the author is not aware of any PCI cards that require a signed firmware update. Though support for this would increase the complexity and therefore the cost of a PCI device, it is an improvement that would potentially stop the techniques discussed in this paper outright.

Anti-virus software could also potentially monitor calls to NtSetInformationProcess, to prevent applications from changing their IOPL (thus preventing them from re-flashing the card without loading a driver), although the impact of blocking all calls to this function is unclear and if a single call is permitted, user-mode malware with the appropriate privilege could simply inject code into the relevant process.

The Impact of a Trusted Platform Module

The Trusted Platform Module is a microcontroller that stores secured information and provides facilities for generation of cryptographic keys as well as signing, verification, encryption and decryption. One role of the TPM is to provide the functionality required to implement a secure bootstrap. During the secure bootstrap, the core root of trust measurement (CRTM), also known as BIOS Boot Block, verifies the integrity of the system BIOS. The expansion ROMs present on devices on the system are then hashed and compared against known good values stored in the Platform Configuration Registers (PCRs). The sequence of PCR checks is illustrated in Figure 4 (taken from Hendricks and van Doorn[9]).

Implemented correctly, the secure bootstrap backed by the TPM prevents the rootkit persistence mechanism presented in this paper.

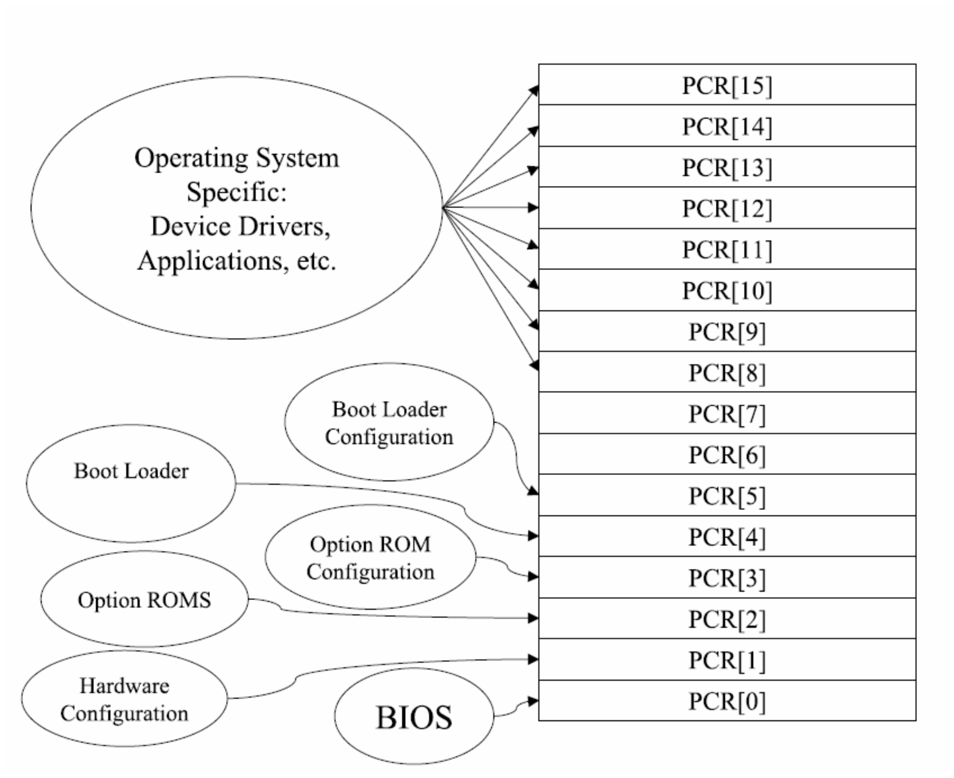


Figure 4: Sequence of PCR checks

The Feasibility of Malware Adoption

During the presentation of the author's ACPI research, the question of malware adoption of BIOS rootkit techniques was discussed. It is the author’s belief that the objective of most malware writers (such as those who target browser exploits as a means of deploying key loggers etc.) is to infect as many system as possible, whilst the objective of a rootkit writer is to infect a small set of systems for a specific purpose. The former therefore writes code that will run on most configurations whilst the latter is able to utilise the specific configuration of the target systems. If a malware writer wished to use BIOS rootkit techniques, the set of machines that their code would operate on successfully would likely decrease substantially. It is possible that malware could be designed to operate as it does currently, only using these techniques if a known hardware configuration is recognised, however increasingly it appears to be driven by economic factors. Whilst enough people do not regularly apply security patches to Windows and do not run anti-virus software, there is little immediate need for malware authors to turn to these techniques as a means of a deeper compromise – if a user detects the malware and removes it, there are plenty more unsuspecting targets on the Internet.

Conclusions

This paper has demonstrated that the PCI devices provide a viable means of persisting a rootkit on a system that does not contain a TPM. Whilst many new notebooks and desktop systems contain a TPM, it will take several years before their usage becomes widespread. Until then, developers of tasks that perform deep system analysis, such rootkit detection and digital forensics, should pay close attention to flashable devices.

References

[1] John Heasman: Implementing and Detecting an ACPI Rootkit, presented at BlackHat Federal, 2006.

[2] Tom Shanley and Don Anderson: PCI System Architecture, MindShare Inc., 1999.

[3] Derek Soeder and Ryan Permeh: eEye BootRoot, presented at BlackHat USA, 2005.

[4] Bochs IA-32 Emulator Project: http://bochs.sourceforge.net/

[5] Joanna Rutkowska: Rootkits vs. Stealth by Design Malware, presented at BlackHat Federal, 2006.

[6] BIOS Boot Specification, version 1.01: http://www.phoenix.com/NR/rdonlyres/56E38DE2-3E6F-4743-835F-B4A53726ABED/0/specsbbs101.pdf

[7] Etherboot: Open Source PXE implementation, http://www.etherboot.org/wiki/index.php

[8] Anti-Rootkit Software - Detection, Removal & Protection: http://www.antirootkit.com/software/index.htm

[9] James Hendricks and Leendert van Doorn: Secure Bootstrap is Not Enough: Shoring up the Trusted Computing Base, Proceedings of the Eleventh SIGOPS European Workshop, ACM SIGOPS, Leuven, Belgium, September 2004.

Acknowledgements

The author would like to thank his colleagues at NGSSoftware for their suggestions and support.