1-1-2013

SMM rootkit: a new breed of OS independent malware

Shawn Embleton
University of Central Florida

Sherri Sparks
University of Central Florida

Cliff C. Zou
University of Central Florida

SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks (2010)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/sec.166

SPECIAL ISSUE PAPER

SMM rootkit: a new breed of OS independent malware

Shawn Embleton, Sherri Sparks and Cliff C. Zou*

School of Electrical Engineering and Computer Science, University of Central Florida, USA

ABSTRACT

The emergence of hardware virtualization technology has led to the development of OS independent malware such as the virtual machine-based rootkits (VMBRs). In this paper, we draw attention to a different but related threat that exists on many commodity systems in operation today: The system management Mode based rootkit (SMBR). System Management mode (SMM) is a relatively obscure mode on Intel processors used for low-level hardware control. It has its own private memory space and execution environment which is generally invisible to code running outside (e.g., the Operating System). Furthermore, SMM code is completely non-preemptible, lacks any concept of privilege level, and is immune to memory protection mechanisms. These features make it a potentially attractive home for stealthy rootkits used for high-profile targeted attacks. In this paper, we present our development of a proof of concept SMM rootkit. In it, we explore the potential of system management mode for malicious use by implementing a chipset level keylogger and a network backdoor capable of directly interacting with the network card to send logged keystrokes to a remote machine *via* UDP and receive remote command packets stealthily. By modifying and reflashing the BIOS, the SMM rootkit can install itself on a computer even if the computer has originally locked its SMM. The rootkit hides its memory footprint and requires no changes to the existing operating system. It is compared and contrasted with VMBRs. Finally, techniques to defend against these threats are explored. By taking an offensive perspective we hope to help security researchers better understand the depth and scope of the problems posed by an emerging class of OS independent malware. Copyright © 2009 John Wiley & Sons, Ltd.

KEYWORDS

system management mode; rootkit; malware; hardware security; operating system security

***Correspondence**

Cliff C. Zou, School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL, USA.
E-mail: czou@eecs.ucf.edu

1. INTRODUCTION

A *rootkit* consists of a set of programs that work to subvert control of an operating system from its legitimate users [1]. If one were asked to classify viruses and worms by a single defining characteristic, the first word to come to mind would probably be *replication*. In contrast, the single defining characteristic of a rootkit is *stealth*. Viruses reproduce, but rootkits hide. They hide by compromising the communication conduit between an operating system and its users. Secondary to hiding themselves, rootkits are generally capable of gathering and manipulating information on the target machine. They may, for example, log a victim user's keystrokes to obtain passwords or manipulate the system state to allow a remote attacker to gain control by altering security descriptors and access tokens.

Since the user's view of the computer system and its resources is strictly mediated by the information the operating system provides to it *via* hardware and software interfaces, a malicious program that controls the interfaces controls the entire system. A rootkit hides its presence by intercepting and altering the interface communications of various operating ystem or hardware components to hide files, processes, and network connections on the computers that it is installed upon. This hiding may be achieved either directly or indirectly using code modifications, data modifications, or a combination of both.

It is important to emphasize, however, that the primary nature of the rootkit is not to infect or compromise a machine, but rather to hide an attacker's presence on an already compromised system. The initial security breach that allows installation of the rootkit may arise from social engineering attacks that trick an unsuspecting user into

running a malicious application or from the exploitation of unpatched vulnerabilities in the operating system and other critical software.

Early rootkits relied upon system *file masquerade* to hide their presence. An attacker would replace a system file with a subversive file that "masqueraded" as the original [1]. The login program was a common target for this type of attack as it could be replaced by a malicious version which captured the passwords of users as they attempted to log into a system. This motivated the development of file system integrity checkers like Tripwire [1]. Rootkit authors quickly developed execution path redirection, also known as *hooking*, techniques to counter detection by integrity checkers. Hooking encompasses a class of techniques whereby a program's normal control flow is altered to execute a block of malicious code. It is important to note that execution path redirection is impervious to traditional integrity checkers like Tripwire which typically only check files stored on the hard disk for modifications. This is because they make their changes to the loaded images in memory rather than to the disk images. Though more difficult to detect than system file masquerade, hooking remains detectable by memory-based integrity checkers and other heuristic techniques. Eventually, rootkit authors figured out how to evade hook detection by using direct kernel object manipulation (DKOM) to modify dynamic kernel data structures for which it is impossible to establish reliable heuristics or trusted baseline values [2]. The idea is that by controlling the data used in a function, a rootkit can indirectly control the execution path.

It is clear that rootkit development has exhibited an adaptive, co-evolutionary pattern in response to security software advancements. The result has been an ongoing, sophisticated game of 'hide and seek' between rootkit developers and detectors. As rootkits seek ever better methods to hide their presence on infected systems, defenders must develop newer, more advanced techniques to find them. With the emergence of hardware virtualization technology, the rootkit battle field has changed dramatically. Previous rootkits co-existed with the operating system (OS). They exerted their influence by redirecting control flow within the OS to their own malicious code [3]. This was accomplished by making modifications to either static or dynamic OS data structures in memory. Security researchers responded by developing integrity checkers and heuristics to detect these changes [4].

Unfortunately, these techniques are useless against virtual machine based rootkits (VMBRs) which have the ability to exist independently of any OS. Such rootkits are able to exert an alarming degree of control without modifying a single byte in the operating system [5]. A VMBR hoists the operating system into a virtual machine and exerts its controls over the machine from an external virtual machine monitor (VMM). This process is invisible to the guest OS. Once installed, the VMM is capable of transparently intercepting and modifying states and events occurring in the virtualized OS. It can observe and modify keystrokes, network packets, memory, and disk I/O. If the VMBR has virtualized memory, its code footprint will also be invisible. These things make a VMBR extremely difficult to detect.

In this paper, we draw attention to another, similar threat that exists on many commodity systems in operation today: the system management mode (SMM) based rootkit (SMBR). SMM is an abbreviation for intel's system management mode, a processor mode which has existed since the i386, yet still remains largely obscure. Unlike the other processor modes, (e.g., protected, real, virtual 8086) which are designed for running operating systems or user applications, SMM was developed exclusively for managing low-level hardware operations like power and thermal regulation. SMM has its own private memory space and execution environment which is invisible to code running outside. Furthermore, SMM code is completely non-preemptible, lacks any concept of privilege level, and is immune to memory protection mechanisms [6]. These features make it an attractive home for malicious rootkits.

In this paper, we present our development of a proof of concept of an SMBR. In it, we explore the potential of System Management Mode for malicious use. By taking an offensive perspective we hope to help security researchers better understand the depth and scope of the problems posed by an emerging class of OS independent malware.

Our SMM rootkit provides a high degree of stealth and control. We demonstrate the construction of a chipset level keylogger by redirecting the keyboard interrupt request (IRQ) to system management mode in the advanced programmable interrupt controller (APIC). Logged keystrokes are then encapsulated into UDP packets and sent out *via* the chipset LAN interface. This is all accomplished without making any visible changes to the target system. By modifying and reflashing the BIOS, the SMM rootkit can install itself on a computer even if the computer has originally locked its SMM. We also show that, once installed, the rootkit remains hidden in memory making it difficult to detect or remove.

Because of the many similarities, we also compare and contrast VMBRs with SMBRs on several key characteristics including operating environment, size, complexity, stealth, and control. Finally, we discuss countermeasures to detect and defend against these threats.

It should be noted that SMM rootkit is mainly useful for advanced targeted attack. First, there is no operating system driver support for the SMBR to rely upon. Hardware access therefore requires implementation of rudimentary low-level drivers inside the SMM handler. Second, the handler must be written in 16 bit assembly [7]. It is at least mildly reassuring that writing chipset level hardware device drivers in 16 bit assembly is beyond the reach of all but the most sophisticated attackers. As a result, it is unlikely that SMM will appear in common malware, but will instead remain limited to sophisticated, targeted attacks.

The rest of this paper is organized as follows. In Section 2, we discuss some related work. In Section 3, we give

an overview of system management mode. We cover the design and implementation of our proof of concept SMBR in Section 4. We evaluate it in Section 5 and provide a comparison and contrast with virtual-machine based rootkits in Section 6. Defense is discussed in Section 7. Finally, we conclude in Section 8.

2. RELATED WORK

Our research on SMM rootkits (SMBRs) is related to three areas of existing rootkit technology: memory management subversion, virtualization, and BIOS exploitation.

Once a rootkit is publicly known, anti-virus software can develop a signature for it. Furthermore, rootkit changes to the OS are detectable using heuristic memory scans. It is, therefore, advantageous for a rootkit to be able to hide its memory footprint. Memory subversion was first implemented in the shadow walker rootkit [8]. The Shadow Walker rootkit demonstrated that it was possible to control the view of memory regions seen by the operating system and other processes by hooking the paging mechanism and exploiting the intel split TLB architecture. Using these techniques, it was capable of hiding both its own code and changes to other operating system components. This enabled it to fool both signature and heuristic-based scans. Memory virtualization support on intel and AMD platforms with hardware virtualization extensions can also be exploited to hide the memory footprint of malicious code. The general idea behind memory virtualization is that the virtual machine monitor (VMM) maintains its own set of page tables in addition to the virtualized guest OS's paging structures. The guest OS is free to manage its own page tables, however, physical translation occurs using the VMM's page tables rather than the guest OS's. Furthermore, the VMM page tables are inaccessible to the guest. As a result, the VMM has complete control over all of the physical memory the guest is allowed to access. Instructions which affect paging structures and the cache are also virtualized to cause traps to the VMM. The Blue Pill II rootkit demonstrated this capability [9]. A SMM rootkit also has the ability to hide its code footprint, but it does not require the implementation of complex memory virtualization code.

Virtual-machine based rootkits have many characteristics in common with the system management mode-based rootkit presented in this paper. They both operate at a layer below the operating system and they both are capable of intercepting and emulating low-level system events without needing to modify any existing OS code or data structures. The VMBR threat was analyzed by Reference [5]. Using Vmware and Virtual PC, authors in [5] implemented several malicious VMBR services to subvert both Windows and Linux. Their implementation, however, was primarily theoretical. This is due to the fact that real world operating systems run on native hardware, not in software virtual machines like Vmware. As real world attackers are unlikely to implement their malicious code in Vmware, the malicious services implemented by Reference [5] are primarily simulations of real world scenarios. Joanna Rutkowska took the VMBR into the practical domain with her development of the Blue Pill rootkit [9,10]. The Blue Pill rootkit exploits AMD hardware virtualization extensions to migrate a running windows operating system into a virtual machine. It hides its code footprint using memory virtualization, supports nested virtual machine monitors, and implements countermeasures against timing-based detections. Reference [11] implemented a similar proof of concept rootkit for MacOS X on the Intel virtualization platform. This rootkit was code named Vitriol. On the other hand, there has been very little research on SMM-based rootkits.

Finally, BIOS rootkits are related to SMM rootkits. The BIOS is the first code that runs when a system is powered on. It performs diagnostics and initializes the chipset, memory, and peripheral devices. A rootkit that infects the BIOS is capable of controlling hardware at a level similar to an SMBR with the additional benefit of being able to survive reboots and reinstallations of a new OS. John Heasman developed a proof of concept BIOS rootkit that acts as a simple Windows NT backdoor [12]. He used the Advanced Configuration and Power Interface (ACPI) to patch a kernel API in system memory. Because his rootkit changed code in the OS it was detectable using existing rootkit detection tools like VICE, Blacklight, or Rootkit Revealer [4,13,14]. For more advanced BIOS rootkits, suggested countermeasures include disabling ACPI in the BIOS and auditing the ACPI tables. Further hardware mitigations include preventing BIOS reflashing or requiring that the BIOS is signed [12]. These countermeasures, however, cannot defend against an SMM-based rootkit.

Using SMM to escalate privilege was first discussed by Loic Duflot [15]. On OpenBSD, the superuser is granted limited privileges. Duflot demonstrated an exploit against OpenBSD that allowed an attacker to arbitrarily extend superuser privileges. Because SMM code has unrestricted access to physical memory, Duflot demonstrated that if attacker can run code in system management mode and locate the internal variable in memory that the OS uses to determine the current privilege level, then he she can modify it to circumvent the operating system's built in security and obtain full privileges. To perform this exploit, the attacker must have the ability to read and write the programmed I/O registers and the legacy video memory range. Duflot's exploit, however, was not a rootkit. His stated goal was privilege escalation, not stealth. The ability to read and write physical memory is only one system management mode capability of interest to a rootkit author. A potentially more advanced and interesting capability lies in the ability of SMM code to exert unrestricted control over peripheral hardware. The fact SMM code is non-pre-emptible and communicates directly with the hardware makes it stealthy and relatively immune to detection. In this paper, we build upon Duflot's work to explore some of the advanced capabilities of system management mode. The ability to control

peripheral hardware could make SMM-based malware, like rootkits, a formidable security threat. Our successful construction of a SMM chipset level, rootkit keylogger, and network backdoor shows that SMM is a practical threat that could be exploited by real world malware authors.

3. OVERVIEW OF SMM

This section gives an overview of system management mode (SMM) and discusses how its features make it an ideal execution environment for stealthy malware.

The intel architecture defines four processor modes of operation: real mode, virtual-8086 mode, protected mode, and system management mode [6]. Real mode and virtual-8086 mode are legacy modes dating back to the 80286/80386 CPU. Real mode is characterized by a segmented 20 bit addressable memory space and the lack of hardware memory protection. MS-DOS and early windows OS versions ran in real mode. Current operating systems run in either 32 or 64 bit protected mode. Protected mode overcomes the limitations of real mode by extending the addressable memory space to 32/64 bits and adding support for paging, memory protection, and multi-tasking. Virtual 8086 mode was designed to allow real mode and protected mode programs to co-exist; however, it is seldom used by modern operating systems. In contrast to the other modes, system management mode (SMM) was not designed for running operating systems or user programs. Rather, it was intended for managing low-level hardware operations (e.g., power management and thermal regulation) and is usually installed by the BIOS. SMM has its own memory space and execution environment which is invisible to code running outside of SMM. Furthermore, SMM code is completely non-preemptible, lacks any concept of privilege level, and is immune to memory protection. These things clearly make SMM a potentially attractive home for stealthy rootkits. System management mode is entered when the processor receives a system management interrupt (SMI) [6].

3.1. SMRAM–the SMM memory space

The system management memory space (SMRAM) is used to hold the processor state information saved upon an entry to SMM, the SMI handler, and its associated data [6]. The intel chipset documentation defines three locations for SMRAM: compatible, high memory segment (HSEG), and top of memory segment (TSEG) [16]. The compatible region overlaps the legacy VGA memory range from 0xA0000 to 0xBFFFF and is the default location for SMRAM. Normally, the contents of SMRAM are only visible to code executing in system management mode. This isolation is ensured by the chipset's rerouting of any non-SMM memory accesses to the VGA frame buffer. Compatible SMRAM is also limited to 128K. The

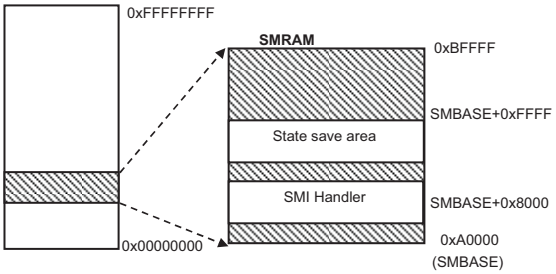

Figure 1. The physical memory map for the intel 845 chipset showing location of the compatible SMRAM region and its layout on a 32-bit system.

HSEG and TSEG regions provide an extended, write-back, cacheable SMM memory space up to 256 MB in size.

Structurally, the SMRAM space consists of a state save area and the system management interrupt (SMI) handler. The remaining space is available for use by the handler for data and stack storage. An internal processor register, called SMBASE, holds the physical address pointer to the start of the SMRAM space. The SMBASE value is also stored in the state save area. Furthermore, the state save area is located at an offset from the beginning of SMRAM in physical memory. This area is used to store the register context when a system management interrupt (SMI) occurs. The SMI handler is also located at an offset from the start of SMRAM. Figure 1 illustrates the location and layout of compatible SMRAM.

3.2. Entering & exiting SMM

The processor enters system management mode when it receives a system management mode interrupt (SMI) [6]. When an SMI is received, execution context is saved into the SMRAM state save map and execution of the SMI handler is commenced. The saved state information includes the processor's control registers, segment registers, task register, general purpose registers, flags, instruction, and stack pointers. The SMM execution environment is similar to 16 bit real mode, with the difference that the full 32 bit flat physical address space is accessible. Code executing in SMM is non-pre-emptible because SMIs have greater priority than both processor exceptions and external interrupts, including non-maskable interrupts (NMI). When the SMI handler wishes to exit system management mode, it executes the resume from system management mode (RSM) instruction [7]. The RSM instruction restores the previous execution context by copying the saved state information in SMRAM back into the processor's registers and then returns control back to the interrupted code. The I/O controller chipset documentation defines a variety of events capable of triggering an SMI. A few of them include: a power button press, real time clock (RTC) alarm, USB wake events, advanced configuration and power interface (ACPI) timer overflows, periodic timer expiration, and a write to the advanced power management control (APM) register, 0xB2

[17]. In the next section, we detail how some of these events might be exploited by a stealthy rootkit.

4. SMBR DESIGN & IMPLEMENTATION

A successful SMBR must overcome two obstacles. First, it must write its code into the SMM handler portion of the SMRAM memory space. This process should be capable of occurring from within a protected mode environment (e.g., Windows or Linux operating system) in order to give the rootkit its maximum infection potential. Second, the rootkit must have some means of intercepting events in the host system and gaining control of execution.

In this section, we discuss the design and implementation of an SMBR. We take a similar approach to Reference [5] with our design and development; however, we opt to design a practical rootkit that can be implemented on native hardware, as opposed to a simulated virtualization platform like Vmware. Section 4.1 describes how the SMBR can be installed on a running operating system. We discuss our implementation of a SMM handler that functions as a chipset-level keylogger and network backdoor in Section 4.2. Finally, we discuss the potential for other, related forms of malicious hardware subversion at the chipset level.

4.1. Rootkit installation

The rootkit can install a new SMM handler when it has I/O port access privileges, the ability to map physical memory, and when the SMRAM region has not been locked by the BIOS or other system software. We used a Windows kernel driver to install the SMBR. The intel chipset documentation defines a system management RAM control register (SMRAMC) which controls the accessibility and visibility of SMM space from other processor modes [17]. The two relevant bits in this register are the D_LCK bit and the D_OPEN bit. D_OPEN controls the visibility of SMRAM. If D_OPEN is clear, SMRAM is only visible to code executing in SMM mode. Non-SMM mode memory reads / writes are diverted by the chipset to the VGA frame buffer. Figure 2 illustrates this process. D_LCK controls the accessibility of SMRAM by controlling access to the SMRAMC register. If D_LCK is set, the SMRAMC register becomes read-only and remains that way until a reset occurs. Assuming that the D_LCK bit is clear, the rootkit is installed as follows:

(1) On a host machine, an attacker makes SMRAM visible from protected mode for reading and writing by setting the D_OPEN bit.
(2) Once D_OPEN is set, the attacker copies the rootkit SMM handler code to the handler portion of SMRAM as defined by the Intel documentation [6].
(3) Finally, the attacker clears the D_OPEN bit and sets the D_LCK bit. This has the effect of making SMRAM invisible to everything other than the subverted (rootkit) SMI handler and of locking the SMRAMC register so that it can no longer be modified. The addressing of the SMRAMC register is chipset specific.

4.2. Rootkit SMM handler implementation

In the following section, we discuss the implementation of our proof of concept rootkit SMM handler. Our rootkit functions as a chipset-level keylogger and network backdoor. We use intel chipsets as opposed to other chipsets because intel provides extensive documentation for them, and hence, it is easier for us to come up with the prototype. Besides, intel chipsets are one of the most popular chipsets used in personal computers. For other chipsets that do not have detailed documentation, advanced attackers can conduct reverse engineering in order to figure out how to program those chipsets.

First, we give an overview of the intel APIC architecture. This is followed by a description of the APIC redirection technique that we use to trap key presses and the procedure used to exfiltrate the key data over the chipset LAN interface. The intel advanced programmable interrupt controller (APIC) is used to manage communication between the CPU, chipset, and external peripheral devices. It consists of two components: The I/O APIC and the local APIC (LAPIC) [18]. The I/O APIC is located on the motherboard while the local APIC is integrated into the CPU. There is typically one I/O APIC for each peripheral

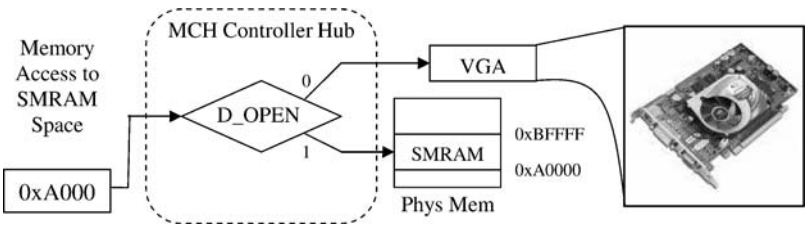

Figure 2. SMRAM memory accesses are filtered by the chipset based on their origin and the state of D_OPEN in the SMRAMC register. SMM accesses are normally directed to SMRAM while non-SMM accesses are directed to VGA memory.

bus and one local APIC per CPU. The primary job of the I/O APIC is to route the interrupts it receives from peripheral buses to one or more local APICs on the system. In turn, each local APIC is responsible for receiving and managing the external interrupts for the CPU that it belongs to. When it receives interrupts, the LAPIC dispatches them to the processor, one at a time, based upon their priorities.

The processor looks up the handler for the interrupt in the interrupt descriptor table (IDT) [7]. Each interrupt is assigned a unique identifier, called a vector. The processor uses this value as an index into the IDT. The interrupt descriptor table is a processor specific data structure containing one entry for each of 255 defined vectors. Kernel rootkits often use IDT *hooking* to intercept processor interrupts and exceptions [19]. This involves replacing the operating system handler contained in the IDT with a pointer to a malicious *hook* routine. Fortunately, such blatant modifications of the IDT are easily detectable. Detection simply involves enumerating each of the handler pointers and validating that the address is within the range of either the OS kernel or a legitimate system driver. If the address falls outside one of these known ranges, it is flagged as suspicious and a security analyst can conduct further investigations.

Differing from the kernel rootkit described above, a rootkit operating in system management mode does not need to make any detectable changes to the IDT in order to intercept interrupts. Rather than intercepting an interrupt at the processor handling level, the SMM rootkit can intercept it directly at the chipset level by rerouting the interrupt in the APIC. We demonstrate this technique in our rootkit by implementing a chipset keylogger. There are three steps in this process. First, we must be able to intercept the keyboard interrupt. Second, we must be able to sniff the keystrokes from the keyboard's internal buffer. Finally, we should forward the interrupt to the CPU for normal handling.

We accomplish the first step by rerouting the keyboard IRQ to system management mode. Thus, whenever a user presses a key, our SMM handler is called. In the handler, we are able to sniff the key. Finally, we manually forward the interrupt to the CPU for normal handling by taking advantage of the local APIC's inter processor interrupt (IPI) mechanism. We outline the implementation details in the following section.

As mentioned previously, the I/O APIC's primary function is to receive and route peripheral hardware interrupts to the local APIC for delivery to the CPU. For this purpose, the I/O APIC architecture defines a *redirection* table [17]. The redirection table contains a dedicated entry for each interrupt pin. It is used to translate the physical, hardware signal into an APIC message on the APIC bus. This table can be used to specify the destination of the interrupt, the vector, and the delivery mode.

The delivery mode is the primary field of interest for our rootkit. Most interrupts use the *fixed* delivery mode. This mode automatically forwards the interrupt to the LAPICs for all processors specified in the destination. Our rootkit changes the delivery mode of the keyboard IRQ from *fixed* to *SMI*. Now, rather than automatically forwarding the interrupt, it will be redirected to our SMM handler. In our handler, we are free to sniff the contents of the keyboard buffer and send it out in network packets.

We can accomplish the second step of extracting the keyboard data by reading the keyboard's internal hardware registers. The key press information is extracted by reading from the keyboard *data* register. Unfortunately, this read is destructive. Therefore, after the key data has been read, it must be replaced so that it is accessible to other system software. We replace it by writing a specific command byte to the keyboard *command* register. This byte instructs the keyboard that the next byte written to the data register should remain there as if placed there by a physical key press [20].

Once we have extracted the keyboard data, it is necessary to forward the interrupt to the CPU for normal user input handling. Otherwise, the keyboard will no longer function. We use the local APIC's ability to issue inter processor interrupts (IPI) for this purpose. The LAPIC documentation defines an interrupt command register (ICR) [7]. Using this register it is possible to send an interrupt to one or more processors, including self. As in the I/O APIC's redirection table, the destination, vector, and delivery mode are all specifiable. When the lower 4 bytes of the ICR are written to, the LAPIC generates the IPI message and sends it out over the system bus. From within our SMM handler, we re-issue the interrupt with a destination of self and a fixed delivery mode by writing to the ICR. Therefore, the keyboard interrupt is delivered to the processor in the normal manner as soon as we exit from SMM mode. Figure 3 illustrates how the SMBR intercepts a keystroke signal and forwards it to the CPU.

4.3. Data exfiltration

After we have captured the keyboard data, we use the chipset LAN controller to transmit the key data collected by our SMM keylogger to an external IP address. Thus, our SMM handler has two functions: it logs keystrokes and then sends the logged data out over the chipset LAN interface. The transmit action is performed periodically in the SMM handler when a defined keyboard data storage buffer becomes full. Using a buffer as opposed to sending the keystrokes immediately as they are received allows more variability in when to send the data and could be exploited by a rootkit wishing to use traffic shaping techniques to stealthily blend in with existing network activity. This simulates the behavior of a malicious attacker attempting to exfiltrate sensitive material from a compromised system.

The LAN controller acts as both a master and a slave on the PCI bus. In the role of master, it interacts with system memory to access transmit and receive data buffers. As a slave, the host processor accesses the LAN controller's internal structures to read and write information to its on-chip registers. These registers may be either I/O mapped

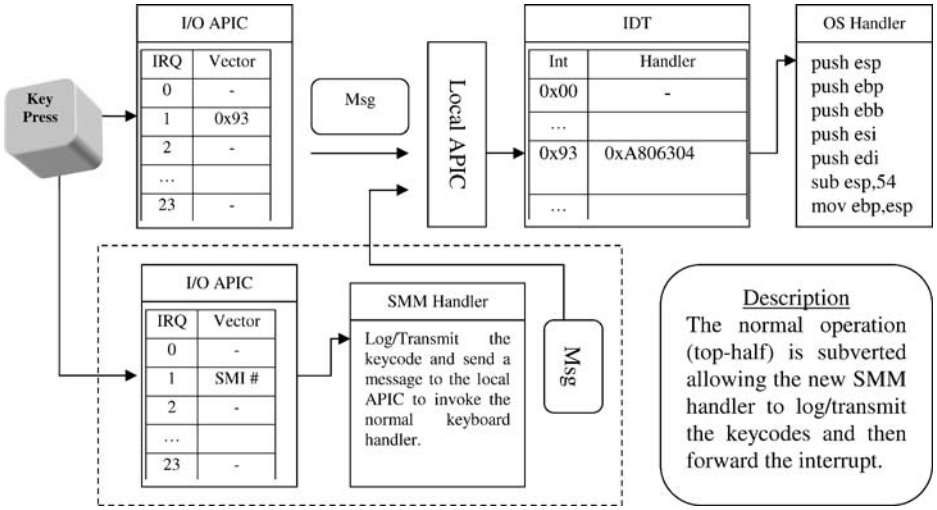

Figure 3. Normal and SMBR keystroke handling paths.

or memory mapped. The method to use is determined by system software.

The data exfiltration component was designed for the Intel 8255x chipset. Therefore, in the following section, we give a brief overview of the Intel 8255 frame transmission architecture.

The two primary hardware components of the intel 8255x chipset are the command unit (CU) and the receive unit (RU). For this paper, we will focus on the command unit since it controls the function of frame transmission. Software issues commands to the CU by writing to the command word field of a memory mapped data structure called the system control block (SCB). Various commands cause the device to transmit, suspend, resume, or idle. The layout of this structure is illustrated in Figure 4.

The CU's frame transmission function operates upon another data structure called the Command Block List (CBL). The CBL is a linked list data structure in shared system memory consisting of command blocks containing command parameters and status information. These blocks include diagnostic and configuration commands in addition to the transmit command. Figure 5 shows the layout of the command block list.

In order to transmit a packet, we need to construct the data packet and initialize a transmit command block (TCB). The TCB is a special type of command block used for transmit

Offset	Upper Word	Lower Word
0x00	SCB Command Word	SCB Status Word
0x04	SCB General Pointer	

Figure 4. 8255x system control block (SCB).

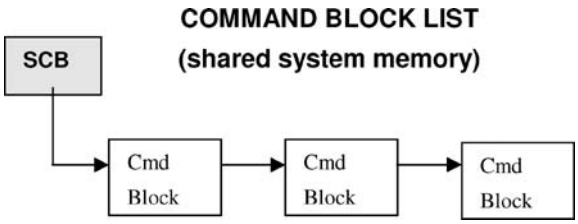

Figure 5. 8255x command block list.

commands. The steps for building and sending a data packet are outlined below:

(1) First, we construct the data packet. Because we don't have access to the upper level NDIS or TDI drivers, this process must be performed manually. For simplicity, we chose to use the UDP protocol in our proof of concept implementation. Thus, the basic packet structure consists of an Ethernet header followed by an IP header, followed by a UDP header followed by the payload.
(2) Second, we build a Transmit Command Block. The exact format of this data structure is contained in the Intel 82558 chipset documentation. Typically, the Transmit Command Block is followed in memory by the transmit data buffer.
(3) After the data packet and Transmit Command Block are defined, we check the LAN controller to ensure that it's in an idle state and load its System Control Block's General Pointer field with the physical address of the Transmit Command Block.
(4) Finally we initiate execution of the LAN controller by sending it a CU Start command. This causes it to begin executing the Transmit Command Block that will send the data packet out over the network.

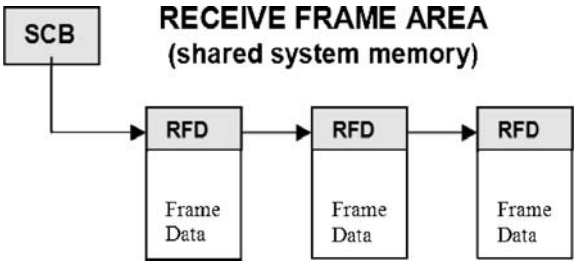

Figure 6. 8255x receive frame area.

4.4. Optional data infiltration

Since SMM rootkits are usually used for high-profile targeted attacks, attackers in most situations would like to keep their SMM rootkits running as long as possible to obtain more and long-time sensitive data from compromised targets. As their objectives change along the time, most attackers would like to be able to control what operations their remote rootkits should conduct. For this purpose, SMM rootkits could build with a data infiltration capability so that they could receive further commands from their owners.

Packet reception on the 8255x is based upon the concept of a receive frame area (RFA). The layout of RFA is shown in Figure 6. The RFA is a region of physical memory that is shared between the NIC and the CPU. It is subdivided into blocks called receive frame descriptors (RFDs). The receive frame descriptor is a data structure consisting of two parts: a header followed by a data buffer capable of holding the maximum ethernet packet size. Every frame received by the NIC controller is described by one RFD. The RFD layout is shown in Figure 7. The NIC's RFA can be located by reading the "general pointer" field from the NIC's status control block. The last RFD in the list is indicated by setting the EL bit.

Frame reception occurs when the device detects a frame on the link with an address that matches either the individual address, a multicast address, or broadcast address. It transfers the frame to the receive FIFO which in turn causes the NIC's receive DMA unit to transfer the frame to main memory on the host machine. Successful frame reception, in turn, causes the NIC to raise a frame receive (FR) interrupt on the host machine. The FR interrupt handler is responsible for extracting the RFD data, setting the appropriate status bits in the RFD header, and ensuring that it is passed to kernel and user components higher in the networking stack.

Offset	Command Word							Status Word			
0x00	EL	S	00000000	H	SF	000	C	0	OK	Status	
0x04	Link Address										
0x08	Reserved										
0x0C	0		0		Size		EOF	F	Count		

Figure 7. 8255x receive frame descriptor.

On Windows, during normal operating, the RFA is co-operatively managed between the Windows NDIS driver and the intel bus driver (e100b325.sys). A malicious driver can circumvent the normal operation of packet arrival by inserting itself between the physical hardware interface and the operating system. This is in contrast to previous stealthy network backdoors like Joanna's DeepDoor rootkit [21] which inserted themselves in NDIS, deep in the OS networking stack, yet still above the physical hardware interface. Our backdoor operates one level lower. By intercepting the NIC's FR interrupt that indicates packet arrival, we can inspect arriving frames prior to the OS or any firewall software running on the host machine.

When the LAN controller receives an interrupt, the APIC dispatches it to the CPU where it is looked up in the Interrupt Descriptor Table. Normally, the interrupt handler for the network card is managed by the Windows NDIS driver. The upper half of Figure 8 illustrates this process. We can intercept it by replacing the pointer with our own. Thus, when a packet arrives, we will receive the first notification and will be able to inspect the receive buffer prior to any operating system software.

This direct hooking technique, however, can be detected by security software by checking if the NIC interrupt in the IDT points to the OS where it should. To improve the stealthiness of our network backdoor, we can redirect the NIC's interrupt to another interrupt that is not being currently used by the OS. As mentioned previously, the I/O APIC's primary function is to receive and route peripheral hardware interrupts to the Local APIC for delivery to the CPU. For this purpose, the I/O APIC architecture defines a redirection table. The redirection table contains a dedicated entry for each interrupt pin. It is used to translate the physical, hardware signal into an APIC message on the bus. This table can be used to specify the destination of the interrupt, the vector, and the delivery mode. We can therefore, change interrupt vector for the NIC and redirect it to a different, unused entry in the IDT. From this handler, after we inspect the incoming frame we can pass it on to the OS handler. The bottom half of Figure 8 illustrates this redirection technique.

The implementation of the process for monitoring incoming traffic can be described as follows:

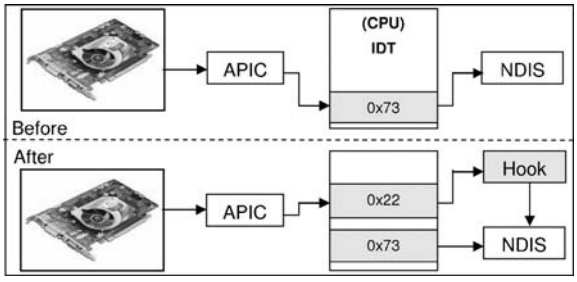

Figure 8. Interception of packet receive using IOAPIC interrupt redirection.

(1) Identify the interrupt for 8255x compatible network card.
(2) Look up the OS handler for that interrupt in the CPU interrupt descriptor table (IDT) and save the pointer.
(3) Locate an unused interrupt in the IDT and *hook* it by replacing the handler address with the address of our backdoor's handler.
(4) Redirect the NIC interrupt to our new, hooked IDT vector by modifying the chipset's APIC Redirection Table.

When an interrupt from the NIC is received, the following steps occur:

(1) Determine if the interrupt is due to a frame arrival (check bit in Status Control Block). If it's for some other reason, call the OS handler.
(2) If the interrupt is due to frame arrival, locate the start of the receive frame area (RFA) from the SCB general pointer field.
(3) Scan the data portion of the receive frame descriptors in the RFA. This scan is used to identify a "special" ICMP packet.
(4) If a "special" packet belonging to the backdoor is identified, then erase it.
(5) Else, pass control to the OS handler and let it process the packet normally.

As illustrated in Figure 8, our method of redirecting the interrupt at the I/O APIC redirection table increases the SMM rootkit's stealthiness because we are not directly hooking the OS interrupt handler for the network. Instead, we take an unused interrupt and reprogram the chipset to interrupt on the new vector. This technique makes the rootkit undetectable by current rootkit detection software that relies on IDT table scanning for possible rootkit hooking. However, data exfiltration function will more or less increase the SMM rootkit exposure chance since it changes the low-level APIC redirection table–it can be detected once rootkit detection software checks chipset level data structures for suspicious modifications. Therefore, we believe data exfiltration is an optional design for SMM rootkit; attackers may implement it when they feel it is necessary compared to the exposure risk.

4.5. Real world SMM rootkits

Although we have only implemented a proof of concept keylogger and network backdoor, a real-world SMM rootkit could implement an unlimited number of malicious services. Virtually every peripheral hardware device can be subverted using these techniques. Some of these devices include the USB ports, Mouse, and Hard Disks. We can envision an extended version of our rootkit that not only transmits exfiltrated data, but also receives malicious commands from an attacker and relays all manner of sensitive materials stealthily out over the network. An SMM rootkit can also gain control on non-hardware events like periodic timer expiration. This would allow for SMIs to be generated at regular intervals, a potentially useful feature for a malicious rootkit wishing to periodically gain control to inspect the state of the system.

Furthermore, such malicious activities are difficult to detect. The SMM handler code is completely inaccessible to the host system and there are no changes to processor or operating system data structures. Indeed, the only potentially detectable changes are the modification to the I/O APIC redirection table and network activity. As there are legitimate reasons to change the delivery mode to SMI, the modification of I/O APIC redirection table is not a sufficient heuristic to identify a stealthy rootkit. One such legitimate use is to provide legacy keyboard and mouse support for USB devices [22]. Finally, the network transmission, which occurs inside of SMM at the chipset level will bypass any host-based intrusion detection systems or firewalls. The network activity could be further concealed by using traffic shaping techniques.

4.6. Limitations

Our proof of concept rootkit has several limitations. It currently works with PS/2 keyboards and a subset of network cards, and it is limited to single processor systems. However, all of these limitations could be addressed. First, it is possible to extend our PS/2 implementation to intercept events from USB keyboards. The chipset I/O controller hub documentation defines a legacy keyboard handling mechanism for USB keyboards which may be exploitable. This legacy operation is performed through SMM space and provides an area for future research.

"When a USB keyboard is plugged into the system, and a standard keyboard is not, the system may not boot, and DOS legacy software will not run, because the keyboard will not be identified. The ICH4 implements a series of trapping operations which will snoop accesses that go to the keyboard controller, and put the expected data from the USB keyboard into the keyboard controller. This legacy operation is performed through SMM space [17]."

Second, network card support could be extended provided that chipset documentation is available. Intel provides developer documentation for most of their LAN cards. Finally, our rootkit could probably be extended to work on the newer multi-processor and multi-core systems. We don't have a multi-core test machine with SMM unlocked, however, the documentation indicates that any processor in a multiprocessor system can respond to an SMI event and that two processors can be executing in SMM at the same time. Furthermore, the manual states that SMM is not re-entrant and that each processor should have its own dedicated SMRAM space. Based upon this documentation, it should be possible to extend our rootkit to handle SMI's on more than processor; however, it will require additional research and development.

In general, the architectural limitations that will apply to an SMM-based rootkit include whether or not the SMRAMC register is locked, the chipset specific nature of an SMBR, and the size limitation of the SMM memory space. Clearly, the biggest limitation is the fact that an SMBR can be installed only if the SMRAMC register is unlocked. The hardware specific nature of the SMBR is probably the second biggest limitation. Because many of the register offsets vary based on chipset, an attacker would need to both know the hardware of the target machine or hardcode a table of register offsets for every chipset and do detection on the fly. There may also be other subtle discrepancies in the chipset and/ or hardware implementation that would require additional code to detect and handle. This coupled with the fact that SMRAM is limited in size may render a generic approach impractical. Finally, an SMBR is non-persistent [23]. It exists only in volatile memory and must be re-installed after a system reboot.

5. TESTING

We conducted four different tests. The first one was a vulnerability assessment. We wanted to get an idea of how wide-spread the SMBR threat might be and the types of systems that were most likely to be affected. Our other experiments involved testing our proof of concept SMM rootkit on a live system. We sought to validate its invisibility to other system software and its functionality as a keylogger network backdoor capable of exfiltrating sensitive data.

5.1. Vulnerability assessment

The goal of our first experiment was to perform a system vulnerability assessment. We wrote a Windows device driver to query the SMRAMC register for the values of the D_OPEN, D_CLOSED, and D_LCK bits. We ran this program on 14 different systems and recorded the manufacturer, chipset, BIOS version, BIOS date, and whether or not the system was locked. Figure 9 shows debug output from the test driver we wrote. Out of these 14 systems, we found six were unlocked and vulnerable to the SMBR threat. Because a majority of the unlocked systems had BIOS revision dates 4 years old or greater and most of the locked systems had more recent BIOS revision dates, we concluded that newer BIOS were locking system management mode. Nevertheless, a substantial percentage of commodity hardware in use today is at least 4 years old. This still makes SMBRs a significant threat. Figure 10 summarizes our preliminary results. We will conduct more comprehensive vulnerability assessment in the near future, especially testing a variety of older intel-based machines.

5.2. Live testing–hiding in memory

Our next experiment involved testing our proof of concept SMBR. We installed it on an unlocked DELL dimension 2400 running an intel 845 chipset. The system was installed with the Microsoft Windows XP operating system. We first sought to verify the invisibility of the installed rootkit. This was accomplished by using the WinDbg kernel debugger to view the physical memory region where we loaded the rootkit code [24]. As expected, we were unable to read the code from this area because one of the functions of the rootkit installer is to close and lock system management mode by writing to the SMRAMC register. As shown in Figure 2, this will cause the access to be routed to VGA memory. This result is unsurprising when one considers that the chipset's memory controller hub (MCH) functions as a gatekeeper for all physical memory accesses. All memory accesses, regardless of whether they originate from software or hardware must pass through the MCH logic. The MCH logic snoops physical addresses on the bus and blocks unauthorized access to certain ranges like the SMM memory space.

5.3. Live testing–key logging

Next, we validated the operation of the keylogger. Our proof of concept code is currently limited to keyboards with a P/S 2 interface. Because it is impossible to read SMRAM once

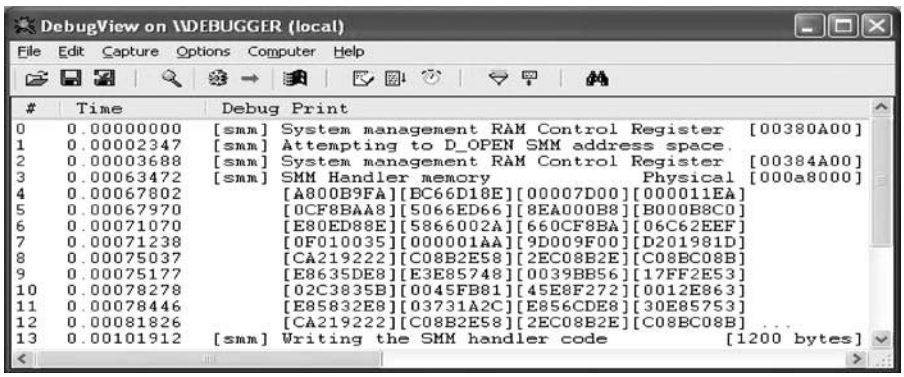

Figure 9. Our test driver opening SMRAM space and displaying the original SMM handler.

Manufacturer	Model	Chipset	Purchase Date	Locked?
DELL	Inspiron 8100	i815EP	2001	NO
DELL	Dimension 4500	i845E	2002	NO
DELL	Inspiron 1100	i845GL	2003	NO
DELL	Dimension 2400	i845GV	2003	NO
DELL	Dimension 4600	i865PE	2004	NO
Custom Built	N/A	i845PE	2004	NO
IBM	T42	i855PM	2005	YES
DELL	Precision 390	i975X	2006	YES
DELL	Dimension 9200	i965P	2006	YES
DELL	Dimension 9150	i945P	2006	YES
DELL	Inspiron 9400	i945GM	2006	YES
DELL	Inspiron 530	iP35	2007	YES
Sony	VAIO	i945GM	2007	YES
Custom Built	N/A	i945X	2007	YES

Figure 10. System vulnerability assessment.

the rootkit is installed and the size of the SMRAM space is limited, we needed a way to save and verify the logged keystrokes. We implemented two different output methods: the serial port and system physical memory.

In the first method, we output the logged keystrokes over the serial port from inside the SMM handler. We use the Windows hyperterminal program to capture the serial output and verify it against our key presses. This method is primarily useful for debugging the SMM rootkit code.

In the second method, the SMM handler writes the keyboard data to an allocated page of physical memory. Since this page is outside the SMRAM space, we were able to attach the WinDbg kernel debugger and read the recorded keyboard scan codes from the page. An attacker could use system memory in this manner as a temporary storage for the key log file. To make it even stealthier, the attacker could encrypt the data in SMM mode before writing it out to system memory. Because SMRAM is not accessible outside SMM, it would be impossible to obtain a copy of the key to decrypt the stored data, even if one knew where to look. To an outsider the encrypted keyboard data would simply appear as random bytes and would be unlikely to raise suspicion.

It should also be mentioned that our SMBR implementation doesn't adversely affect the performance of the target system. That is, from a subjective, user's perspective, our SMBR key logger does not introduce any noticeable slow down or latency in keyboard input at the GUI level. We validated this at different typing rates, but did not quantify the SMBR's performance using objective measures. This is an area of future research.

5.4. Live testing–data exfiltration

We validated that our network backdoor was able to both log keystrokes and transmit packets containing the logged data successfully from inside the SMM rootkit handler. We used an intel Pro 100B network card for development and tested it using two machines connected to an Ethernet network *via* a DLINK router. The first machine was the aforementioned Dell dimension 2400. We installed the SMBR on it. The second machine was a Dell Precision 390 running Windows XP. We installed Microsoft Network Monitor 3.1 on it so that we could sniff incoming network traffic. We were able to validate that the key press data was successfully received by the second machine by examining the sniffer output. The SMM network code, however, is card specific and would require modification to run on other network cards. Figure 11 provides a screenshot showing that the SMM keyboard data was packaged into a TFTP packet and sent out to the remote machine using UDP over IP. The data payload following the TFTP header is highlighted.

To illustrate the stealth of the SMBR, we tested our rootkit's data exfiltration capability against the Zone alarm intrusion detection system. Zone alarm provides both inbound and outbound intrusion detection. It also has a LOCK feature which allows the user to lock his/her computer so that applications can neither send nor receive data over the network. We chose to test under the strictest possible conditions with the LOCK feature enabled.

Our test consisted of sending out logged keyboard data in specially crafted UDP packet. As expected, Zone alarm did not detect any access attempt. Indeed, detecting this type of rootkit behavior is very difficult from within a compromised system. This is due to the fact that the rootkit exists in an isolated environment, does not require any changes to the OS or other architectural data structures, and communicates directly with the hardware. If one were able to detect and validate reads and writes on the shared memory region of the card, it might be possible to monitor outgoing traffic. Unfortunately, both SMM and the card hardware address memory physically rather than virtually and the x86 does not support monitoring physical memory accesses.

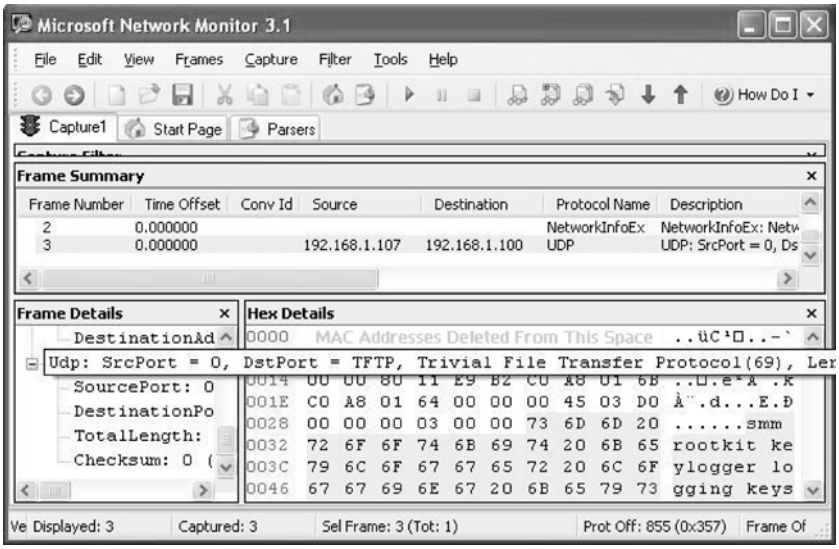

Figure 11. Capturing the key logger packet (TFTP header is [00,03,00,00] for a data packet).

This test serves to highlight the seriousness of a weaponized SMBR that has a built-in data exfiltration capability. Unrestricted, difficult to detect outbound data exfiltration will impose a serious and long-term information exposure to some high-profile compromised targets, such as financial institutes, government agencies and military sections.

5.5. Live testing–data infiltration

Finally, we used specially crafted ICMP packets containing the data payload "r00t was here before this!" to conduct data infiltration test. This string serves as a form of "signature" in the packet receive interrupt handler to indicate that the packet is destined for the backdoor rather than the operating system. In order to test data infiltration, we used a secondary laptop running the network packet generator (NPG) program to craft these special packets. NPG is a free GNU GPL Windows packet injector. It uses WinPcap to send packet out the network interface. The packets are defined in a packet file and it is possible to craft any kind of packet, regardless of headers or payload.

The approach we tested was packet erasing. In this method, the rootkit zeroes out the data portion of the receive frame descriptor including the MAC, IP, TCP, and ICMP headers. In this case, the OS drops the packet without sending it up the network stack, and hence, incoming packets cannot be observed by any OS-based security software. When we compared the backdoor's Debug output with Microsoft network monitor, we saw that Microsoft network monitor failed to report any kind of network activity, ICMP or otherwise. Even when we set the Windows XP firewall to block all outside sources from connecting to the computer, the network backdoor could still successfully receive the incoming crafted ICMP packets. In addition, we tested the data infiltration by running security software zone alarms and snort; none of them can detect the incoming Internet access activities.

5.6. Exploiting PCI expansion ROMs to install SMBRs on "locked" systems

As mentioned previously, one of the SBMR's biggest limitations is the fact that it can be installed only if the SMRAMC register is unlocked. Based upon our research (as shown in Figure 10), SMM rootkits are more likely to exist on older processors containing older BIOS versions (more than 2 years old). This is due to the fact that many newer BIOS have set the D_LCK bit in the SMRAM control register rendering SMRAM inaccessible outside the BIOS.

Despite this limitation, an attacker may still be able to install an SMBR with an additional step. An attacker may be able to modify and reflash the BIOS such that the BIOS leaves SMM unlocked. Disassembling the BIOS to identify the code that needs to be modified is, however, a tedious and difficult task. Furthermore, the mechanism for reflashing the system BIOS is typically vendor specific and poses a danger to the system if there is any possibility of error or power failure.

Alternatively, the attacker can install the rootkit before the BIOS locks SMRAM, then he/she will have control of the system. It may be possible to do this by exploiting an expansion ROM. In Reference [25] Heasman describes an approach a malware author may use to exploit an expansion ROM. When a system boots, the system BIOS is copied from flash memory to RAM and begins executing. One of the first tasks performed by the BIOS is a scan of the PCI bus to detect installed devices. During this scan, the BIOS

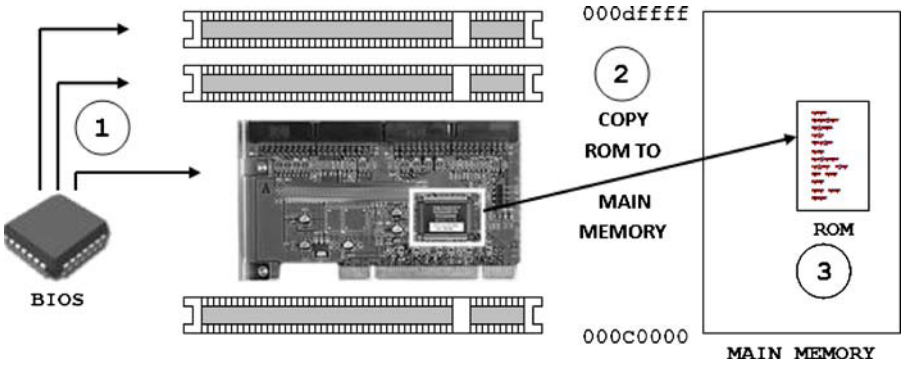

Figure 12. Illustration of the execution of BIOS during system boot. (1) The BIOS begins executing. (2) The BIOS enumerates the PCI bus, copies any expansion ROMs to main memory, and hands off control to them. *Note*: SMM has not been locked yet so it is possible for a malicious ROM to install a malicious SMM handler at this point. (3) After executing the expansion ROMs, the BIOS regains control, and completes system initialization. The BIOS locks SMM at some point after the expansion ROMs have run, but before it turns control over to the Operating System's boot loader.

will identify whether or not any installed devices contain expansion ROMs. Expansion ROMs hold device specific initialization code that needs to run during execution of the system BIOS. If it finds any, it copies them to memory, and then executes them by calling into the ROM at offset 0x03. This process is illustrated in Figure 12.

We were curious whether or not BIOS writers initialized and locked SMM before or after calling the expansion ROM's. If the BIOS locked SMM after calling them, it might be possible to re-flash a PCI card to install a malicious SMM handler and lock it into place before the BIOS locked SMRAM. To test this theory, we re-flashed the BIOS on an old promise fast track TX2 raid controller card with code to query the value of the SMRAMC register at the time the boot ROM was executed. Although we only tested it on three systems, we discovered that at the time our code ran, the BIOS had not yet locked SMM. If this is indicative of a general trend, a malicious boot ROM might provide a viable option for installing SMBRs on locked systems. The drawbacks, however, would be the need for the target system to have a flashable boot ROM and the need for the attacker to have prior knowledge of the hardware configuration of the target system. Malware authors may partially avoid these limitations by targeting popular, widely deployed PCI hardware devices that are flashable.

6. EVALUATION & DISCUSSION

SMM and VMM-based rootkits both operate at a level outside an existing operating system. Therefore it makes sense to compare and contrast them. We compare and contrast the SMM and VMM rootkits based on four characteristics: operating environment, complexity and size, control, and stealth. Figure 13 summarizes the comparisons between SMBRs and VMBRs.

6.1. Operating environment

SMBR and VMBR rootkits each have their own optimal target environment. Both types of rootkits are hardware specific. Virtualization rootkits can only exist on processors supporting virtualization extensions. This limits them to newer processors mostly less than 2–3 years old. In contrast, SMM rootkits are more likely to exist on older processors containing older BIOS versions (greater than 4 years old as shown in Figure 13). This is due to the fact that many newer BIOS have set the D_LCK bit in the SMRAM control register rendering SMRAM inaccessible outside the BIOS.

Additionally, while virtualization rootkits are processor-specific, system management mode-based rootkits are chipset specific. This makes them best suited for a sophisticated, targeted attack rather than a vector for widespread malware distribution. The operating environments are also very different because VMBRs operate in protected mode with paging enabled while SMBRs operate in a 16 bit environment similar to real mode without paging. Finally, both VMBR's and SMBR's can be classified as non-persistent rootkits. Non-persistent rootkits exist only in memory and lack the ability to persist across reboots on the machine they are installed on. Although on the surface this seems like a significant disadvantage, when one considers that many server systems run for weeks or months at a time between reboots, it becomes less of an issue. Due to the complex nature of the SMBR, it is unlikely that such a rootkit will appear on the more frequently rebooted systems (e.g., home user machines) anytime soon.

6.2. Complexity & size

Compared with VMBRs, SMBRs have an advantage in terms of size and complexity. While they have the added developmental complexity of having to deal with writing

Category	SMBR	VMBR
Vulnerable System Space	• Most pre 2005 Systems • Post 2005 Systems with BIOS modification	Post 2006 Systems
Operational Environment	16 bit Mode w/o Paging	32 bit Protected Mode w/ Paging
OS Independent	YES	YES
Memory Footprint Hiding	YES	YES (with memory virtualization)
Control	**Chipset IRQs** ▪ Keyboard ▪ Mouse ▪ Network Card ▪ USB ▪ Disk	**CPU** ▪ Processor Interrupts ▪ Debug Register R/W ▪ Control Register R/W ▪ Privileged Instructions ▪ Memory Access
Defense	Set D_LCK in SMRAMC register either in BIOS or during early boot of OS.	OS or BIOS should install a secure virtual machine that prevents installation of 3rd party virtual machines.

Figure 13. Comparison of VMBRs and SMBRs.

the SMM handler in legacy 16 bit assembly, they expand little effort to conceal their memory footprint as the chipset handles the memory access redirection once SMRAM has been closed and locked by the handler. On the other hand, in order to provide similar stealth, a VMBR will likely have a larger code footprint. This is due to its need to include complex paging code for memory virtualization support.

6.3. Control

Both VMBR and SMBR rootkits are capable of efficiently exerting control over the system and neither needs to modify the target operating system in order to obtain that control. With that said, VMM rootkits might have the upper hand where flexibility is concerned. They can intercept a greater number of higher level events like interrupts, memory access, debug, and control register reads/writes, and execution of specific privileged instructions. Although SMBRs have considerable control over peripheral hardware as we demonstrated in our proof of concept keylogger, in general, they tend to intercept lower level hardware events like power management, thermal regulation, and bus errors and will have limited control over processor specific events like memory access and instruction execution.

6.4. Stealth

Compared with VMBRs, SMBRs are stealthier. Several detections based upon cache and TLB discrepancies have been proposed for detecting virtualization rootkits [26]. Because a VMBR operates in protected mode with paging enabled, there is no easy way for it to prevent its execution from affecting the TLB. SMBRs are immune to these types of detection because they operate in an environment without caching or paging enabled. Therefore, they should not have any detectable effects upon either the cache or the TLB.

SMM rootkits may provide greater stealth with less overhead. A VMM rootkit is not hidden in memory unless it implements memory virtualization. In contrast, the SMBR is hidden by default due to the MCH redirection of non-SMM originated memory accesses to the SMRAM region.

7. DEFENSE

In this section, we consider the detection and prevention of both OS dependent and OS independent rootkits. We feel that the emergence of OS independent rootkits necessitates a shift in focus from detection to prevention.

7.1. OS dependant rootkits

To date, most rootkit defense has focused on rootkit detection. This is possibly because detecting an OS dependent rootkit may, in fact, be easier than preventing its installation. Prevention is difficult and there are several reasons for this. These include difficulty controlling end user behavior, multiple attack entry points, and the presence of unpatched vulnerabilities.

It is difficult to control the behavior of an end user. End users are subject to social engineering attacks which may lead to them inadvertently install or run a malicious application. Assuming that the malicious application has gotten past the user, security software may attempt to prevent further damage by preventing or limiting access to the kernel. Unfortunately, there are many entry points into the kernel and it is difficult to guard them all. Additionally,

there are often undocumented entry points into a system. These usually take the form of exploits for unpatched vulnerabilities in the operating system or critical software.

While prevention is a difficult problem, detection may be slightly easier. OS dependent malware can be detected fairly reliably using signature or heuristic based scans. Clearly, malware that coexists with an OS must make changes within the environment in order to exercise control over it and/or hide itself. Heuristics have been developed to detect many of these changes [4,27].

7.2. OS independent rootkits

OS independent rootkits present a new dilemma. Both virtualization and SMM rootkits are considerably more difficult to defend against than OS dependent malware. First, it is not necessary for them to make any visible changes to the OS. Thus, heuristics are not useful. Second, they have the capability to conceal their memory footprints making signatures useless. As a result, indirect detection measures like timing or cache discrepancies have been suggested for virtualization rootkits [26].

Timing attacks may provide a method of detecting an SMM rootkit. We have validated that the processor's timestamp counter is updated, even while executing in SMM. Thus, it may be possible to devise a detection that reads the timestamp counter before and after an SMI and compares it with the normal time taken by a machine's SMM code. However, if an attacker knows the existence of this form of detection on target machine, he can develop the SMM rootkit to counter-attack this defense. This is because the rootkit itself has access to the counter and is capable of modifying it before returning control back to the host operating system.

Another class of detection for OS independent malware relies on cache or TLB discrepancies. For example, VMM rootkits may be detected by their effects on the cache or TLB because they must exist in cacheable, pageable memory. Unfortunately, this kind of timing attack is not valid against SMM rootkit. SMM rootkit does not influence the cache or TLB because it can exist in uncached memory and does not use paging.

It has also been suggested to move detection off the CPU onto another hardware device that has access to physical memory [28]. The problem with this approach is that the chipset arbitrates all external device communication/physical memory access through the memory controller hub (MCH). Therefore, SMRAM will remain inaccessible to any devices residing on the system bus.

As mentioned previously, one could check the IOAPIC redirection table for interrupts that have been routed to SMIs. A rerouted interrupt may be considered a "red flag", but even that may not be a sufficient heuristic. There are, in fact, legitimate reasons to route an interrupt to an SMI. One such legitimate use is to provide legacy keyboard and mouse support for USB devices [22]. Therefore, lacking other rootkit indicators, it may be difficult to determine the illegitimacy of a rerouted interrupt and state with certainty that it was installed by a rootkit. It may, however, be possible to detect VMBR and SMBR malware during installation if a signature is known. This is possible because an anti-malware kernel module can scan third party drivers and processes as they are being loaded. On the other hand, if a signature for the malware is not known or the malware installs itself through an undocumented interface (e.g., exploit), it is unlikely to be detected.

We suggest that the emergence of OS independent malware like SMM and virtualization rootkits necessitates a shift in emphasis from detection to prevention. Virtualized rootkits may be prevented by installing a secure virtual machine monitor (VMM) that prevents the installation of other virtual machines [29]. SMBRs can be prevented by locking down the SMRAM register in the BIOS. Therefore, chipset manufacturers should be encouraged to release BIOS updates to address this problem and system administrators of older machines should ensure that their BIOS are up to date. In the interim, the operating system could greatly mitigate this problem by locking the register during early boot before third party drivers are loaded. This would prevent such rootkits from being installed by user mode applications or kernel drivers on a running system. Failing both of the aforementioned suggestions, a third party anti -virus or host based intrusion prevention (HIPS) software application could write a driver to lock the SMRAM control register that the OS installs during early boot. Unfortunately, it is difficult to guarantee that the protection driver will be loaded before another malicious kernel driver.

8. CONCLUSION

In this paper, we have exposed a potential threat that has not been widely recognized. We have established that a SMM rootkit has chipset level control over peripheral hardware including the network controller, USB ports, mouse, keyboard, and disk. It has control of both the I/O and local APIC, is able to easily conceal its memory footprint, and read/write indiscriminately to the 32 bit physical address space. Practical development of an SMM rootkit, however, is constrained by the following limitations: the need for SMRAM to be unlocked, the need to write the handler in assembly, and the lack of operating system support. As a result, it is likely that SMM rootkits will remain limited to sophisticated, targeted attacks.

Finally, we note that the SMM rootkit can be viewed as a new breed of OS independent malware related to VMBR and BIOS rootkits and that a significant number of older systems (>2 years old) remain vulnerable to this threat. Newer systems could also be vulnerable to the SMM rootkit if it can modify and reflash the BIOS such that the BIOS leaves SMM unlocked. Furthermore, the SMBR provides a method of implementing an OS

independent rootkit on processors that don't support the new virtualization extensions. Thus, it may contribute to an effective multi-vector rootkit attack capable of targeting a large subset of current systems on the market. We suggest that the emergence of such malware necessitates a shift in perspective from detection to prevention and that a closer relationship between security researchers and hardware developers should be fostered.

ACKNOWLEDGEMENTS

This work was supported by NSF Grant CNS-0627318 and Intel Research Fund.

REFERENCES

1. Thimbleby H, Anderson S, Cairns P. A Framework for Modeling Trojans and Computer Virus Infections. *The Computer Journal*, 1998; **41**(7): 444–458.
2. Fuzen Op. The FU rootkit. www.rootkit.com/project.php?id=12
3. Hoglund G. A *REAL* NT Rootkit, patching the NT Kernel. In *Phrack Magazine,* Vol. 9, No 55 1999.
4. Butler J. VICE–Catch the Hookers. Presented at *Black Hat USA*. August 2004.
5. King ST, Chen PM, Wang Y-M, Verbowski C, Wang HJ, Lorch JR. Subvirt: Implementing malware with virtual machines. In *Proceedings of IEEE Symposium on Security and Privacy (S&P'06)* pages 314–327, Washington, DC, USA, 2006; IEEE Computer Society.
6. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume3B: System Programming Guide, Part 2. May 2007.
7. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume3A: System Programming Guide, Part 1. May 2007.
8. Sparks S, Butler, Shadow Walker: J. Raising the Bar for Windows Rootkit Detection. In *Phrack Volume 0x0B, Issue 0x3D, Phile #0x08 of 0x14* 2005.
9. Rutkowska J. New Blue Pill. www.bluepillproject.org/stuff/nbp-0.11.zip 2007.
10. Rutkowska J. Subverting Vista Kernel for Fun and Profit. Presented at Black Hat USA, August 2006.
11. Zovi. DA. Hardware Virtualization Rootkits. Presented at Black Hat USA, Aug 2006; www.theta44.org/software/HVM_Rootkits_ddz_bh-usa-06.pdf
12. Heasman J. Implementing and Detecting an ACPI BIOS Rootkit. Presented at Black Hat Federal, 2006.
13. Cogswell B, Russinovich M. RootkitRevealer v1.71. November 1, 2006; Http://www.microsoft.com/technet/sysinternals/Utilities/RootkitRevealer.mspx
14. F-Secure Black Light. www.f-secure.com/blacklight
15. Duflot L, Etiemble D, Grumelard O. Using CPU system management mode to circumvent operating System security functions. In *DCSSI 51 bd. De la Tour Maubourg 75700 Paris Cedex, France* 2007.
16. Intel Corporation. Intel 845GE/845PE Chipset Datasheet. October 2002.
17. Intel Corporation. Intel 82801DB I/O Controller Hub 4 (ICH4). May 2002.
18. Heasman J. Implementing and Detecting an ACPI BIOS Rootkit. Presented at Black Hat, Federal
19. Butler J, Hoglund G. Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, 2005.
20. [8042] Keyboard Controller. heim.ifi.uio.no/<stanisls/helppc/ 8042.html
21. Rutkowska J. "Rootkits vs. Stealth by Design Malware", Presented at Black Hat, Europe 2006.
22. Support for USB and Legacy Keyboards and Mouse Devices. December 2001; microsoft.com/whdc/device/input/usbhost.mspx
23. www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf
24. Windbg. en.wikipedia.org/wiki/WinDbg 2007.
25. Heasman J. Implementing and detecting a PCI Rootkit. Presented at Black Hat Federal 2007.
26. Garfinkel T, Adams K, Warfield A, Franklin J. Compatibility is Not Transparency: VMM Detection Mythis and Realities. In *HotOS XI: 11*th *Workshop on Hot Topics in Operating Systems* 2007; USENIX.
27. Rutkowska J. System Virginity Verifier–Defining the Roadmap for Malware Detection on Windows System. Presented at Hack In The Box. September 2005.
28. Petroni NL, Fraser T, Molina J, Arbaugh WA. Copilot–A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceeding Usenix Security Symposium* August 2004.
29. Butler J, Sparks S. Windows rootkits of2005, Part 2. www.securityfocus.com/infocus/1851 2005.