2014 9th International Conference on Availability, Reliability and Security

The SMM Rootkit Revisited: Fun with USB

Joshua Schiffman and David Kaplan
Security Architecture Research and Development
Advanced Micro Devices, Inc.
Austin, TX, USA
Email: {josh.schiffman, david.kaplan}@amd.com

***Abstract*—System Management Mode (SMM) in x86 has enabled a new class of malware with incredible power to control physical hardware that is virtually impossible to detect by the host operating system. Previous SMM rootkits have only scratched the surface by modifying kernel data structures and trapping on I/O registers to implement PS/2 keyloggers. In this paper, we present new SMM-based malware that hijacks Universal Serial Bus (USB) host controllers to intercept USB events. This enables SMM rootkits to control USB devices directly without ever permitting the OS kernel to receive USB-related hardware interrupts. Using this approach, we created a proof-of-concept USB keylogger that is also more difficult to detect than prior SMM-based keyloggers that are triggered on OS actions like port I/O. We also propose additional extensions to this technique and methods to prevent and mitigate such attacks.**

***Keywords*-Computer security; Embedded software; Universal Serial Bus;**

I. Introduction

Spyware is a class of malware that logs sensitive inputs and exfiltrates it to unauthorized parties. For example, keyloggers record user input without user awareness that captures content, including passwords or credit card numbers, that hackers use to impersonate victims. While there are a number of attack vectors for implanting loggers, the most important factor is the logger's ability to remain hidden to collect the most data. As security experts continue to design new techniques to root out spyware, so too do attackers find new ways to push the limit of detectability.

In recent years, System Management Mode (SMM) exploits have become popular because they give attackers an unparalleled level of access and stealth. In particular, code executing in SMM has direct control of physical resources including system physical memory and devices that otherwise are shielded by numerous software- and hardware-protection mechanisms. Moreover, SMM suspends normal code execution, which allows anything running in this mode to remain hidden from the operating system (OS) kernel without risk of being pre-empted by interrupts.

While SMM offers great potential to manipulate physical hardware, its utility is limited by its constrained memory footprint and execution window. Despite this, early attacks on SMM have demonstrated the design of SMM malware to modify kernel variables to escalate privilege [1], subvert secure launch procedures [2], and reprogram device firmware [3], [4]. More recently, SMM keyloggers have been published that capture PS/2 keyboard inputs through ACPI trapping [5] and I/O trapping [6]. More sophisticated malware designed by national governments has been discovered that target specific BIOS firmware and network appliances [7]. Once captured, this data can be transmitted stealthily via cooperating userspace programs or even compromised network cards [8] and radios [9].

In this paper, we present a novel SMM-based rootkit that intercepts and controls communication between Universal Serial Bus (USB) devices and the OS kernel. Unlike previous malware that required the kernel to trap to SMM when reading or receiving interrupts, our custom SMM rootkit can intercept USB events before they are delivered to the OS kernel. It does this by reconfiguring the USB host controller (HC) to route all interrupts to a special SMM handler normally intended for PS/2 emulation of USB devices. Using this technique, we designed and implemented a proof-of-concept USB keylogger and tested it on a Linux system using recent hardware. During our experiments, we successfully intercepted, replaced, and even injected keystrokes with an average overhead per keystroke of only 61 μs.

In this paper, Section II describes SMM functions and how rootkits can take advantage of the environment. We describe the USB protocol and delve into our design for hijacking the channel between the HC and the OS kernel in Section III. We detail our implementation in Section IV and discuss expansions to its capabilities in Section V. We then offer possible mitigations in Section VI and conclude with future work in Section VII.

II. Background

This section provides a brief overview of SMM including how it works, what security mechanisms are used to protect it, and previous attacks on and from within SMM. As background for our attack and example keylogger, we describe the USB protocol and give an example of how a generic USB keyboard driver functions.

A. System Management Mode

SMM is a special operating mode on x86 processors in all processors since the Intel 486 [10]. It was designed for power management, BIOS flash updates, PS/2 keyboard emulation, bug fixes, and other miscellaneous tasks that need to be executed by the BIOS. Hence, it is also a highly privileged mode in which normal execution is suspended and special code stored in SMRAM is executed, all hidden from the main OS. SMM is similar to a hypervisor mode in that the CPU hardware saves the processor state on

978-1-4799-4223-7/14 $31.00 © 2014 IEEE
DOI 10.1109/ARES.2014.44

entering SMM, similar to a world switch, and restores it after leaving SMM. Code executing in SMM is usually short because it takes control of all processor cores and masks off all interrupts. As a result, it is nearly impossible for users to know when their system is in this mode or for the OS to observe what is occurring.

SMM is entered via an unmaskable system management interrupt (SMI) to the processor. On receipt, the CPU saves the processor state to a special region of DRAM, called SMRAM [11]. It then begins executing the SMI-handler code also stored in SMRAM. While in this mode, the processor is set automatically to 16-bit real mode, paging is disabled, and all interrupts are masked. When the handler finishes, it executes the resume from system management mode (RSM) mode instruction, which restores the CPU state. Due to the complexities of resource management, modern CPUs typically enter SMM on all cores when an SMI arrives, effectively suspending the entire system until the SMI handler has finished executing [12].

1) Protecting SMRAM: SMM operates at a higher privilege level than normal ring 0 x86 code, and thus has access to everything ring 0 code does and more. In particular, code executing in SMM has exclusive access to SMRAM [11] and the ability to write to BIOS flash [13]. This is an important restriction because code typically is loadable only at boot time by the BIOS, which is stored in the BIOS flash.

Access to SMRAM is fenced off by hardware mechanisms (e.g., northbridge) and allows access only when the processor is in SMM mode. SRAM is divided further into two regions. The first, called **ASEG**, is always located at the memory address range 0xA0000-0xBFFFF. This region is known also as the legacy VGA card space and typically is mapped to the framebuffer. The second region, called **TSEG**, is a programmable region for BIOS and programmable registers specify its size and location.

When the processor is not in SMM and software tries to access either ASEG or TSEG, the accesses are directed to MMIO space instead of DRAM. Thus, while address 0xA0000 normally maps to the VGA card, in SMM mode, it instead maps to DRAM at the same address (taking advantage of the so-called “VGA memory hole”). If software outside of SMM tries to set up a DRAM mapping for any SMRAM address, the hardware will redirect it to MMIO instead.

As a protection mechanism against malicious uses of SMM, write access to SMRAM can be locked by setting a bit in the SMM configuration registers; this bit is called SmmLock in AMD CPUs [12] and D_LCK in Intel chips. The intention is that once BIOS has set up ASEG/TSEG, loaded the appropriate SMI handler, and locked the configuration, the SMRAM cannot be modified until a reset, thereby maintaining SMRAM integrity.

2) Previous Attacks on SMM: Many of the properties that make SMM secure and difficult to modify also make it an excellent location for nearly undetectable malware that cannot be seen by normal processes like anti-virus software or the OS.

In theory, many SMM-based attacks should be prevented by the BIOS’ ability to lock SMRAM. However, early work by Duflot et al. [1] demonstrated that it was possible to enable write access to SMRAM and thus insert a custom SMI handler. An attacker then could disable privilege separation for the superuser in OpenBSD without detection by the OS kernel. This attack was possible in part because the SMRAM lock bit never was set by some BIOS firmware, thereby permitting the attacker to enable write access. Subsequently, BIOS vendors began setting the lock bit to prevent such exploits. In addition to unlocked SMRAM, Duflot’s attack relied on writes to the VGA card space that first was unmapped and thus exposed the SMRAM address range. Later, Heasman [4] presented additional method of attacking SMM and BSDaemon, coideloko, and D0nand0n [14] published several additional techniques for circumventing SMM protections via a set of libraries for developing portable SMM rootkits.

More sophisticated SMM rootkits were designed by Embleton and Sparks [5] that used APIC redirection to trigger SMI on keyboard IRQs. Their custom SMI handler then would function as a keylogger to capture keystrokes on PS/2 keyboards and send them out over the network. More recently, Wecherowski published an SMI keylogger that uses I/O trapping to raise SMIs when the OS reads from a PS/2 keyboard (or one in PS/2 emulation mode) [6]. In both of these scenarios, the OS was notified that some interrupt had occurred. As a result, timing-based detection schemes could identify unusual response times due to the SMI interposing on keystrokes. Our approach avoids this issue by trapping to SMI before the OS ever receives the interrupt. Moreover, it is not limited to PS/2 keyboard input; instead, it could interpose on all USB traffic.

Wojtczuk and Rutkowska later showed how to use SMM to subvert Intel’s TXT technology [2] and how cache poisoning could lead to writes to SMRAM after the lock bit was set [15]. Duflot also presented an attack using cache poisoning to change the SMBASE value to point SMIs to a malicious SMI handler [16]. This attack would leave the normal handler untouched, which could fool integrity monitors that check the SMRAM for changes.

The key take-away from these attacks is that despite efforts to lock down SMRAM access, SMM remains a viable vector for launching attacks on the rest of the system. Even more dangerous is the possibility of BIOS flash with malicious code through vulnerabilities in the upgrade process or direct hardware access [7].

B. USB

We now detail the USB standard as background for our SMM-based rootkit. USB is nearly ubiquitous in modern computer systems and commonly used for human interface devices like keyboards and pointers. USB devices are controlled through the four main components shown in Figure 1.

At the hardware level, a USB HC, consisting of several physical USB ports, is connected to one or more USB *endpoint devices* (e.g. a keyboard). The USB HC is pre-

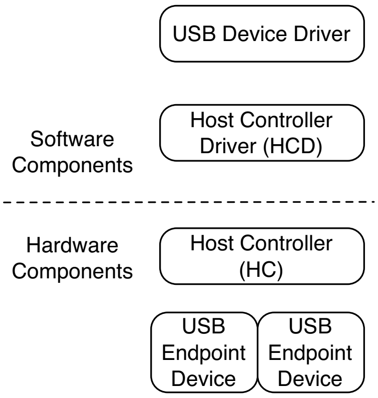

Figure 1. **USB uses four components.** The endpoint device communicates at the physical layer over USB to the USB HC. The HCD sends requests to the HC at the request of the USB device driver.

sented as a PCI device to the OS kernel, which uses a USB HC driver (HCD) to communicate with it. A higher-level device driver (e.g., an HID driver) communicates with the specific endpoint device via the HCD and hardware.

Most HCs use the Open Host Controller Interface (OHCI) standard [17], which supports USB 1.1 devices. USB 2.0 is an addition on top of OHCI, called the Enhanced Host Controller Interface (EHCI), but HID devices including keyboards/mice always use USB 1.1 regardless of what the system supports. For this white paper, we will focus on OHCI.

The USB HC hardware is controlled via a set of memory-mapped I/O (MMIO) registers. These contain a number of pointers to various linked lists that track attached devices and map requests to devices. The HC periodically polls the attached devices for new information in response to requests and, on receiving information, writes that data to main system memory.

USB device drivers talk to devices using the OHCI specification protocol to issue request packets defined by the specific HCD used by the OS. On Linux, for example, a device driver may send a USB request block (URB) [18] to query information from the device. The HCD creates a transfer descriptor (TD) for this request and links it to the appropriate list for the device. Once this request is completed, the HC hardware will unlink the TD from the list it was on, and link it to the DoneList. It also writes any result data into memory via DMA as defined by the TD, and optionally sends an interrupt indicating it has new data on the DoneList. On receiving this interrupt, the HCD walks the DoneList and processes the TDs found. It then notifies the original driver (typically via an asynchronous callback) that its URB is complete and the data is available.

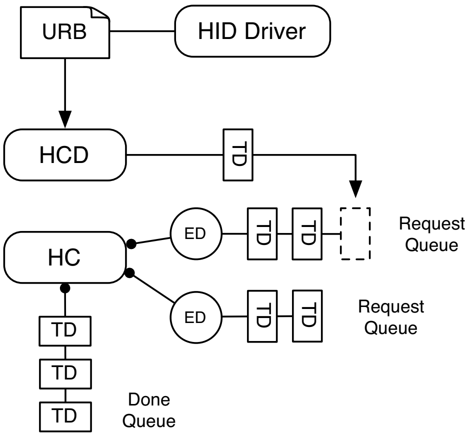

Figure 2. **USB keyboard example.** The human interface device (HID) driver sends a request (e.g., read active LEDs) to the host control driver (HCD) using an USB request block (URB). The HCD appends a transfer descriptor (TD) to the appropriate endpoint device's request queue. The HC then unlinks a TD, services the request via the OCHI protocol, and appends the result to done queue.

0 7	8 15
Modifiers	Reserved
Keystroke 1	Keystroke 2
Keystroke 3	Keystroke 4
Keystroke 5	Keystroke 6

Figure 3. **Format of the data buffer returned by keyboard as specified by the OHCI specification. The first byte indicates any modifier keys (e.g. Shift, Ctrl, Alt). Bytes 2-7 are the pressed or released keys.**

C. A Typical USB Keyboard Driver

As an example, this section describes the basic flow of a USB keyboard driver, which is applicable on Linux, and Windows systems, and likely other OSes as well. On Linux, the USB keyboard driver[1] interacts with the OHCI USB HCD[2]. Once loaded, the USB keyboard driver submits a URB to the USB keyboard asking for information about a keystroke. The resulting TD is configured such that it will stay pending until there is new keyboard information, at which point it will be completed and an interrupt is generated. Typically, the USB HC hardware is set to poll the keyboard every millisecond for any new keystrokes.

On a keyboard event (key press or release), the data from the keyboard is transferred via DMA to memory the TD points to. The TD packet defines where the DMA request will point; this is important because any change to the TD will alter the expected behavior of the HC issuing the DMA request. For keyboards, this result consists of 8 bytes as defined by the USB HID specification and shown in Figure 3. The first byte is a bit-field that indicates which modifier keys are pressed. Bits 0-3 indicate the left CTRL, SHIFT, ALT, and WINDOWS keys respectively. Bits 4-7 indicate the right-side versions of those keys. Bytes 2-7 indicate the (up to six) keys pressed at that time. For a table of the key codes for these fields, see [19].

Once completed, the TD is unlinked from the device's Request queue and placed in the Done queue. After this URB is completed, an interrupt is raised to alert the HCD and thus the keyboard driver to processes the data. Finally, a new URB is sent to restart this process and request the

[1] Located in the kernel source under drivers/hid/usbhid/usbkbd.c

[2] Located in drivers/usb/host/ohci-hcd.c

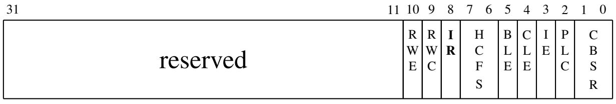

Figure 4. **HcControl register.** Bit 8 is the interrupt routing (IR) bit that determines how interrupts generated by events are routed. When clear, interrupts are routed normally on the host bus. When set, an SMI is triggered instead. The HCD uses this bit to indicate HC ownership by the BIOS.

next keyboard event.

III. Design of a USB-controlling Rootkit

This section describes the high-level design of our SMM rootkit for controlling USB devices. First, we explain how to hijack the USB HC so it redirects communication to a custom SMI handler. We then describe the process for setting up the attack on the system.

A. Interrupt Rerouting to SMI

Because we want to interpose on USB events *before* they arrive at the OS, we need a way to prevent the normal hardware interrupts from being triggered by the HC. Fortunately, the OHCI specification [17] defines a feature that does just this. In the HC's control register (Figure 4), bit 8, called the interrupt routing (IR) bit, controls whether events trigger normal hardware interrupts or instead are diverted to SMM. If the bit is set, an SMI is raised and the SMI handler processes the USB event.

The IR bit is intended to give BIOS control over USB devices during boot when no OS is present in memory to handle the device. It also enables PS/2 emulation of USB keyboard and pointer devices through the SMI handler. In this way, USB events can be read using normal PS/2 I/O registers. Normally, the HC sets this bit to 1 at boot and the OS HCD clears the bit after the kernel is loaded. However, our experience has shown that default OS HCDs do not check if the bit is later set.

B. High-level Attack Flow

With the IR bit enabled, a custom SMI could receive all USB events, modify them as desired, and raise an event as normal (or not) when done. If we are repurpose this handler, we must first replace the default SMI handler provided by the BIOS. On most desktop systems, default SMI handlers do very little (if anything) and we found that replacing them resulted in no noticeable effect on our test system. For systems that use advanced power-saving features (e.g., laptops and tablets), more care must be taken to avoid disabling battery management, power stepping, or the ability to enter lower power states. Given the small size of the rootkit payload, as will be demonstrated by our example in Section IV, it should be possible to fit it within the available memory space. Moreover, existing SMI code can be accessed from the BIOS flash directly and disassembled to reengineer the existing handler. The fact that many devices share common BIOS code makes this doable before the attack.

Now that we have a custom SMI handler and the ability to redirect events, our attack flow works as follows:

1) Inject malicious SMI handler into running system. Alternatively, overwrite BIOS flash to use malicious SMI handler (see Section IV-B).
2) Set IR bit to 1 after OS load.
3) On a USB event, the SMI handler parses the HCD device queues to find the triggering event. If it is the target of the attack, perform the attack-specific functions.
4) *Optional.* Raise the USB event interrupt so the OS sees the event as normal.
5) Restore the system from SMM mode with an RSM operation.

Under this scenario, the SMI handler has full control of all USB devices attached to that HC. The handler can inspect all events, choose which ones to forward to the OS, and even communicate to the devices independent of the OS. However, the data structures managed by the OS to communicate with the endpoint devices are fairly complex and an overly lengthy SMI handler could result in noticeable delay. Also, the SMRAM region is relatively small, thereby limiting the sophistication of the handler code. Nevertheless, Section IV will provide an example of how to implement a USB keylogger in this environment. Section V discusses additional extensions.

IV. Implementing an SMM Keylogger

We created a USB keylogger to demonstrates how an attacker could leverage the USB redirection described in Section III to capture keystrokes stealthily. In this section, we first describe the implementation of our rootkit's SMI handler and how it parses the various USB data structures to detect keyboard data. We then discuss possible techniques for acquiring access to SMRAM to install the SMI handler. Finally, we detail our experiments with the keylogger and its performance during these tests.

A. Crafting the SMI Handler

Our USB keylogger's goal is to intercept keystrokes without the host system noticing. We designed our handler to process USB events as illustrated in Figure 5. The custom SMI handler performs three main steps:

1) Parse SMIs for potential keyboard data.
2) Log keystrokes for later exfiltration.
3) Raise an interrupt and restore control to the OS.

Once a USB event raises an SMI, control will pass to the SMI handler. First, the handler puts the processor into either 32- or 64-bit mode (depending on our target), but does not reenable paging that was disabled on entering SMM. This gives our code easy access to the physical address space of the system. Next, the handler determines if the SMI came from a USB device by finding the HC, which can be anywhere on the PCI bus. However, many modern systems have integrated southbridges, and the HCs are at fixed locations depending on CPU manufacturer and version. Simple CPUID checks can point us to where the USB HC is if the hardware is unknown, but a known system will be easier to handle ahead of time. As with

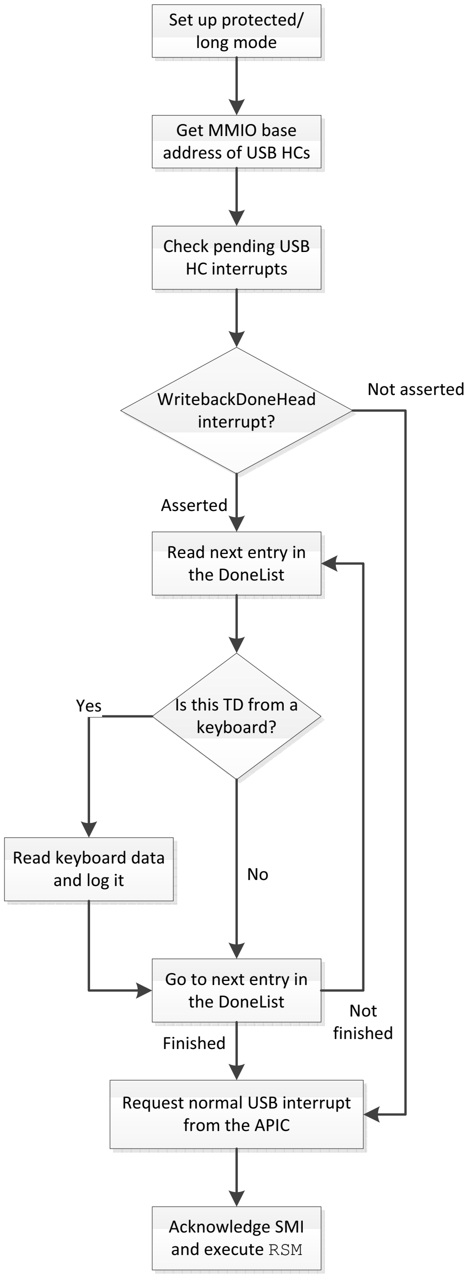
Set up protected/ long mode
Get MMIO base address of USB HCs
Check pending USB HC interrupts
WritebackDoneHead interrupt?
Not asserted
Asserted
Read next entry in the DoneList
Is this TD from a keyboard?
Yes
No
Read keyboard data and log it
Go to next entry in the DoneList
Not finished
Finished
Request normal USB interrupt from the APIC
Acknowledge SMI and execute RSM

Figure 5. **SMI Handler Flow.**

laptops, the keyboard is often on a dedicated bus, which makes identification trivial.

The OHCI USB HC is controlled by a set of memory-mapped registers, and the base pointer of these is located at offset 0x10 in the PCI configuration space [13]. For example, to get this base address from a USB HC at Device `0x12` and Function 0, we can use the I/O ports CF8/CFC as shown in Listing 2. Our handler pushes this value on the stack to pass it as a parameter (`ohci_base`) to a C function defining the rest of the keylogger.

The function in Listing 1 first checks if the HC is asserting `WritebackDoneHead`, indicating it has finished a packet. This check is done in line 2 by masking the `0x2` bit of the `HcInterruptStatus` register. If the bit is set, the handler then looks for the base of the DoneList, `DoneListHead`. This points to the start of a linked list of completed packets. Our code then walks this list and looks for a packet from the keyboard.

The OHCI spec does not specify that the TD in the DoneList must contain any specific information about the device it came from. The HCD typically includes this information, and this data format has remained consistent in Linux. We use the Linux kernel headers to cast this pointer as a `struct td` (transfer descriptor) in line 5.

We then can determine the classcode of the device as follows: The `UsbDeviceDescriptor` structure includes the `bDeviceClass`, `bDeviceProtocol`, and other relevant information. Keyboards have a device class of 3, and a device protocol of 1.

Once it has been determined the TD corresponds to a keyboard, the data is read as on line 9. The HC increments the buffer end (BE) pointer by 8 as it writes the data. Because keyboard data is always 8 bytes long, we can get the pointer to the start of the region by subtracting 7. Alternatively, Linux keeps a copy of this pointer in `td_dma` in this structure [20].

Our keylogger is now able to read and manipulate the keystrokes. We choose a location in physical memory and write the results to that memory buffer. One easily imagines more complex exfiltration techniques such as writing to a cooperating process or even sending data out through a compromised network card [3].

The remaining step for our keylogger is to pass the keystroke to the OS. This maintains the illusion that nothing has intercepted the input. To do this, we need to send a normal interrupt to the CPU with the same vector that the HC uses. The vector assigned to the HC is dependent on the system configuration, but can be obtained easily in Linux by looking at `/proc/interrupts` while the system is running. The mapping of devices to interrupt vectors generally is static, so the same vector will be used on every boot. On our test system, the HC in which we were interested used interrupt vector `0x29`.

To send this interrupt, the handler writes to the interrupt command request (ICR) register in the APIC. The address of this register generally is fixed on systems, and we can use the ICR register to ask for specific interrupts [12]. In Listing 3, we create a pointer to the ICR location and write to it to request vector `0x29`.

Finally, the handler must acknowledge the SMI interrupt, allowing future SMIs to be set. This is done by setting an end-of-SMI (EOS) bit. While the exact mechanism varies between Intel and AMD platforms, example code can be found in the SMI handler of the open-source BIOS, Coreboot [21]. Because our SMI handler did not modify any HC state, on leaving SMM (via `RSM`), the standard Linux interrupt routine is triggered and the USB keyboard driver still sees the keystroke as normal.

```
HcInterruptStatus = *(int*)(ohci_base+0xC);
if (!(HcInterruptStatus & 0x2)) {
Hcaa_base = *(unsigned char**)(ohci_base+0x18); \\Host Communication Area
DoneListHead = *(unsigned char**)(Hcaa_base+0x8);
TdPointer = (struct td*)(DoneListHead);
UrbPointer = TdPointer->urb;
UsbDevice = UrbPointer->dev;
UsbDeviceDescriptor = UsbDevice.descriptor;
DataPtr = (TdPointer->hwBE)-7;
```

Listing 1. **SMI handler code.** This snippet shows how the handler finds keystrokes in the HC data structures.

```
1    movw $0xCF8, %dx
2    movl $0x80009010, %eax
3    outl %eax, %dx
4    movw $0xCFC, %dx
5    inl %dx, %eax
```

Listing 2. Assembler code in AT&T syntax that locates the MMIO base address [12]. In line 2, specifies Bus=0, Device=0x12, Function=0, Offset=0x10.

```
unsigned int* APIC_ICR = NULL;
APIC_ICR = (unsigned int*) 0xFEE00300;
*APIC_ICR = 0x4C029; // Set vector to 0x29
```

Listing 3. Code to raise a normal USB HC interrupt vector through APIC. The interrupt is requested to be asserted and sent to the requesting processor.

Without our handler complete, we wanted to measure how much delay our handler would impose on normal USB devices. During our initial tests, we could not visually observe any delays. Given how small this SMI handler is, and that USB keyboards report events only every millisecond, the handler should not surprisingly go beyond the polling frequency. To confirm this, we instrumented the handler to record average execution times using the CPU's time-stamp counter. After 200 invocations, we found the handler ran 61 μs with very minor variance. This was in line with our performance expectations.

B. Installing the SMI Handler

Once the SMI handler is complete, an attack must install it into the SMRAM at runtime. Because modern systems lock the SMRAM configuration during boot, installing an SMM rootkit or other malware requires more creativity than a simple `memcpy`. We discuss two main approaches to installing custom SMM code: exploiting bugs and flashing a new BIOS image.

Exploiting Bugs: Bugs that grant access to SMRAM are not unheard of. Previous work has documented a cache-poisoning bug in Intel processors that allowed writing to TSEG and redirecting the SMM handler [15]. In addition, bugs may be present in BIOS code (including the SMM handler) that allow unauthorized access to SMM mode. This type of vulnerability is made worse because users rarely (if ever) update their BIOS code, so security vulnerabilities are likely to stick around in the wild longer than typical security bugs. In fact, many BIOS prior to 2005 did not lock SMRAM at all [5]. Furthermore, failure by OEMs to lock access to BIOS flash storage or prevent unauthenticated updates may enable root privileged code to modify the existing SMI handler.

Flashing a New BIOS: Because BIOS code sets up the SMRAM and SMM handler, installing a custom BIOS that is loaded on every reset is a persistent method to compromise a system. Although many modern systems allow for preventing writes to the BIOS flash when not in SMM, we have found that this feature is not enabled on the systems we tested.

Writing to flash requires ring 0 access, but that is hardly a burden in many scenarios. For example, an "evil maid" attack can occur if someone is permitted to gain physical access to the host machine. This person may then run Linux Live-CD and boot the system into that, thereby allowing full root access. Utilities like flashrom[3] then can be used to write to the BIOS, allowing for installation of a malicious BIOS. This malicious BIOS could be a modified version of Coreboot, which has broad support for many systems [21]. Once installed, the Live-CD can be removed and the system will return to its normal state, with SMM malware installed and ready to go.

Another attack vector is to compromise the system before it is installed, such as in the case of a compromised equipment supplier, boarder inspection, or delivery service [7]. A rogue agent (whether a company or an individual) could ship a brand new system with a compromised BIOS containing the SMM malware. Alternatively, the malicious party could intercept the device and exploit known (or unknown) weaknesses in the firmware or hardware to install the payload. This malware could sit dormant on a machine, unlikely to be detected, until it is chosen to be activated.

V. Extending the Payload

In this section, we present two modifications to the SMM payload. This shows how the rootkit can go beyond keylogging to disable security features and hide physical devices from the OS.

A. Man in the Middle Attacks

A simple extension of our keylogger is to use it for a man-in-the-middle attack. Because our keylogger runs before the OS sees the keystroke, the SMM handler code could change the keystroke seen by the OS. A prime target for this type of attack would be something like the

[3] http://flashrom.org

Sequence	Keystrokes
05 00 4C	LEFT_CTRL+LEFT_ALT+DEL
41 00 4C	LEFT_CTRL+RIGHT_ALT+DEL
14 00 4C	RIGHT_CTRL+LEFT_ALT+DEL
50 00 4C	RIGHT_CTRL_RIGHT_ALT+DEL

Table I
Possible secure attention sequences.

Windows CTRL+ALT+DEL command. Software generally assumes this sequence is secure, and applications cannot intercept this command. However, our SMM keylogger can intercept this by checking if the first 3 bytes of the keyboard data seen is one of the following values indicating CTRL+ALT+DEL.

An example attack might be to conspire with a user application to trick users into thinking they changed their password. When they press CTRL+ALT+DEL, the SMM keylogger replaces the keystroke, which the user application uses to display a fake password change dialog. An alternative attack would be to fake a CTRL+ALT+DEL sequence, potentially allowing malware to bypass the security implied by the secure attention command.

B. Hidden Devices

Another potential attack using a custom SMM handler would involve hiding a physical device from the OS. When the IR bit is set, all USB interrupts go through the SMM handler, including interrupts about devices being plugged in or removed. By removing the final step of our keylogger (sending the interrupt to the OS), our malware can hide all device activity from the OS.

For example, a USB thumb drive could be attached covertly to a compromised system. It even might be installed inside the chassis where no one is likely to look, perhaps directly connected to a motherboard USB header. While the system is running, data (such as keystrokes, or other sensitive data) may be written to that USB thumb drive without any knowledge of the OS. All data and interrupts related to this device are handled by the SMM code, and it would not show up in the Windows device manager or in the Linux `lspci` command.

This sort of attack might be useful for a system that otherwise is not connected to the external world. A rogue agent could install this USB drive, and later remove it to analyze the data. A worrisome scenario for many companies is one in which a newly purchased server is compromised with a custom BIOS, and a secret USB drive attached inside. When the server is later serviced, the drive could be removed and customer information stolen.

A similar attack could involve a hidden listening device, such as a microphone or webcam. With such a device physically hidden and no information about the device appearing in the OS, a user could be recorded secretly. The recorded information could be stored to a hidden flash device, sent over the network, or otherwise exfiltrated.

VI. Possible Mitigation

We now discuss some possible defenses against this class of malware and how to detect its presence.

A. Protecting SMRAM

The most straightforward defense is to prevent any unauthorized changes to SMRAM. If the malicious handler cannot be installed, then SMI redirection either will have no effect or, more likely, will hang the system. In addition to setting the lock on SMRAM after boot, BIOS also should ensure that writes to flash can be performed only from within SMM. While some BIOS permit writes from ring 0 to facilitate easy BIOS flashing, this is a potential security risk. Instead, the SMI handler should perform the BIOS flash update after first performing an authenticity check via a signature check.

B. UEFI Secure-Boot

More recently, motherboard vendors have started to replace legacy BIOS with a newer standard, Unified Extensible Firmware Interface (UEFI) [22]. UEFI supports a secure-boot mechanism that performs code-signing checks as each piece of the firmware is loaded. A move towards this standard should help close loopholes in legacy BIOS that enables installation of malicious SMM rootkits. The ROM performs the signature check using a key stored in the ROM. This check prevents unauthorized code from overwriting the flash. However, bugs in signature checks or leaked encryption keys still could enable attackers to bypass this defense.

C. Detecting SMI Redirection

The IR bit in our attack enables SMI handlers to get full control over USB devices. Normally, the OS clears this bit after boot, so there should be very little reason for it to be set in the future (except for PS/2 emulation). It might be prudent for the OS HCD or some other malware detection service to check this bit periodically to see if it was set. This would indicate something unusual is happening and that additional steps should be taken to check for compromise.

D. Stand-alone Security Processor

A more complete solution would be to have a dedicated security processor in the system that can verify SPI-ROM storage (flash) integrity and monitor SMRAM for unauthorized code or access. The AMD Platform Security Processor^{TM} (PSP) is an example of such a processor. This core, independent of the normal x86 cores on the SoC, implements a form of hardware validated boot by verifying the first block of the BIOS in flash before permitting the CPU to continue the boot processes [23]. The keys used to verify the BIOS code's signature are stored in an immutable ROM within the PSP itself.

In addition to validating the boot process, such a dedicated security processor could be used to establish access control protections that require writes to SMRAM to come from the security process. This would give the system more flexibility to update SMRAM during runtime, while still providing authorization checks on the updated contents. Moreover, the security processor functions independently of x86 cores during SMM, which lets the security processor detect when an unusual number of SMIs are raised or inspect critical memory locations and registers that our attack would modify.

This protection scheme goes beyond the capabilities of existing hardware security modules like the Trusted Platform Module (TPM), which can only passively store hashes of data provided to it or perform basic cryptographic operations and key management [24].

VII. Future Work

In this paper, we presented an SMM-based exploit for interposing on normal USB event-triggered hardware interrupts. Our proof-of-concept keylogger was able to capture keystrokes before the OS without any noticeable delay; it introduced only a 61 μs delay per keystroke on average. This exploit leverages the IR bit in the HC's configuration register, which never was intended to wrest legitimate control of USB devices from the OS. This makes it possible for rootkits to monopolize the USB HC completely and undermine secure input mechanisms or hide malicious peripheral devices. As a precaution, we recommend administrators check for suspicious HC settings that might indicate the presence of such attacks.

In future work, we plan to refine our rootkit to include exfiltration methods that take advantage of the privileged access that SMM code provides. We also will study additional techniques for intercepting a broad range of USB device information such as mass storage and pointer devices. Finally, we will explore the limitations of maintaining stealthiness while running in SMM, which includes execution overhead and memory footprint.

References

[1] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," in *"CanSecWest"*, 2006.

[2] R. Wojtczuk and J. Rutkowska, "Attacking intel trusted execution technology," http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20slides.pdf, Feb 2009.

[3] S. Sparks, S. Embleton, and C. C. Zou, "A chipset level network backdoor: bypassing host-based firewall & ids," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, 2009.

[4] J. Heasman, "Hacking the extensible firmware interface," https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf, Jul 2007.

[5] S. Embleton, S. Sparks, and C. Zou, "SMM rootkits: a new breed of OS independent malware," in *Proceedings of the 4th International Conference on Security and privacy in Communication Networks.* ACM, 2008.

[6] F. Wecherowski, "A real smm rootkit: Reversing and hooking bios smi handlers," http://www.phrack.com/issues.html?issue=66&id=11, Nov 2009.

[7] J. Appelbaum, J. Horchert, and C. Stöcker, "Shopping for spy gear: Catalog advertises nsa toolbox," www.spiegel.de/international/world/catalog-reveals-nsa-has-back-doors-for-numerous-devices-a-940994.html, December 2013.

[8] A. Triulzi, "The jedi packet trick takes over the deathstar," *CanSecWest* 2010.

[9] D. Goodin, "Spy software's bluetooth capability allowed stalking of iranian victims," arstechnica.com/security/2012/06/spy-softwares-bluetooth-capabilty-allowed-stalk-of-iranian-victims.

[10] Wikipedia, "System management mode," en.wikipedia.org/wiki/System_Management_Mode.

[11] Advanced Micro Devices, Inc., "Amd64 architecture programmer's manual volume 2: System programming," http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf, September 2012.

[12] I. Advanced Micro Devices, "Bios and kernel developer's guide (bkdg) for amd family 15h models 00h-0fh processors," http://support.amd.com/us/Processor_TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf, January 2013.

[13] Advanced Micro Devices, Inc., "Amd sb700/710/750 register reference guide," http://developer.amd.com/wordpress/media/2012/10/43009_sb7xx_rrg_pub_1.00.pdf, July 2009.

[14] "System management mode hacks," http://www.phrack.org/issues.html?issue=65&id=7#article, March 2008.

[15] R. Wojtczuk and J. Rutkowska, "Attacking SMM Memory via Intel CPU Cache Poisoning," http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf, March 2009.

[16] L. Duflot, O. Grumelard, O. Levillain, and B. Morin, "Getting into the smram: smm reloaded," in *Proceedings of the 10th CanSecWest Conference*, March 2009.

[17] Compaq, Microsoft, and National Semiconductor, "Openhci - open host controller interface specification for usb," ftp://ftp.compaq.com/pub/supportinformation/papers/hcir1_0a.pdf, October 1996.

[18] Free Electrons, "Urb documentation," http://lxr.free-electrons.com/source/Documentation/usb/URB.txt.

[19] Microsoft, "Usb hid to ps/2 scan code translation table," http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/translate.pdf, April 2004.

[20] Free Electrons, "Linux cross reference," http://lxr.free-electrons.com/source/drivers/usb/host/ohci.h, September 2002.

[21] Coreboot, "Coreboot source," http://review.coreboot.org/p/coreboot, 2014.

[22] U. E. Forum, "UEFI Specifications," http://www.uefi.org/specs/, February 2014.

[23] R. Lai, "Amd security and server innovation," http://www.uefi.org/sites/default/files/resources/UEFI_PlugFest_AMD_Security_and_Server_innovation_AMD_March_2013.pdf, March 2013.

[24] T. C. Group, "Tpm main specification," www.trustedcomputinggroup.org/resources/tpm_main_specification.