An Introduction to Hardware-Assisted Virtual Machine (HVM) Rootkits

Michael Myers, Crucial Security
Stephen Youndt, Crucial Security
http://crucialsecurity.com/

August 7, 2007

Abstract

Since mid-2006, a number of security researchers have made public claims about a theoretical rootkit technique based on the hardware-assisted virtualization capabilities of the most recent x86 processors from AMD and Intel. The presentations on the topic generated a great deal of excitement and private research. Until recently, however, there were no publicly-available source code or implementation details available. This paper and its associated source code presents a minimal hypervisor framework for a rootkit. Supporting code to insert the hypervisor into a running Windows XP system is also provided.

1 Introduction

Hypervisor Virtual Machine (*HVM*) rootkits were first described in 2006 [1, 2], but few implementation details were available. Recently, source code for an Intel-specific implementation of an HVM rootkit was contributed to Rootkit.com [3] and an AMD implementation followed soon after at BluePillProject.org [4].

In this paper, we provide an introduction and overview of currently-available knowledge on the topic of HVM rootkits, as well as implementation details to complement previously-discussed research on the method. We will consider the detection techniques that have been proposed to date and evaluate the viability of each.

In addition, we will provide the components necessary to create an HVM rootkit on an AMD multi-core x86 processor. Although not presenting a complete solution, this paper will describe the unique features and provide sufficient source code for a programmer knowledgeable in contemporary rootkits to begin working with virtualization technology.

2 AMD Virtualization

In our hypervisor example, only AMD Virtualization (AMD-V^{TM}) technology, formerly known by the code name "Pacifica," will be implemented. Although Intel VT-x^{TM} (code-named "Vanderpool") technology is similar, a multi-platform implementation would require significant additional code and complexity. Nonetheless, the discussions in this paper should serve as a road map to implementing similar functionality on Intel^{TM} hardware.

AMD-V, also referred to as "AMD Secure Virtual Machine" (SVM) technology in the AMD programmer's manuals [5], is implemented as a set of exceptions which intercepts any instructions or CPU events that may affect the hypervisor or its guests. AMD-V defines several new instructions, registers and control flags to implement a more privileged mode, or "ring minus one." Previously, ring zero was the most privileged mode implemented in the x86 architecture. Data structures known as Virtual Machine Control Blocks (**VMCB**) allow exceptions to be controlled. VMCBs are not shared amongst processor cores.

The instructions defined to implement SVM include those to execute a guest (`VMRUN`) and manage guest state information (`VMSAVE` and `VMLOAD`). The `VMMCALL` instruction allows guests to explicitly communicate with the hypervisor while `STGI` and `CLGI` are used to control global interrupts. `INVLPGA` is used to invalidate entries in the memory controller's Translation Lookaside Buffer (TLB) while `SKINIT` provides the ability to implement a secure hypervisor loader. In addition, the `MOV` instruction has been extended to directly read and write the CR8 control register (Task Priority Register), as an SVM-related performance speedup. Lastly, enabling SVM affects the behavior of a number of existing AMD64 instructions. Overall, the newly defined instructions are used primarily for creating the hypervisor and switching between a guest and the host (known as a "world switch").

The VMCB is described in the AMD64 Architecture Programmer's Manual.[5] The VMCB consists of two areas. The first area contains control bits including the intercept enable mask. This mask determines which conditions cause a `#VMEXIT`. The second area maintains the guest state. This save state area preserves the segment registers and most of the virtual memory and entry point control registers, but not the general purpose or floating point registers. Most of the VMCB (2564 bytes in size) is currently defined as unused bytes and is reserved for future expansion. The VMCB must be allocated as page-aligned contiguous physical memory.

The central functionality in any AMD-V hypervisor is a loop of `VMRUN` followed by `#VMEXIT` processing. The hypervisor initiates guest operation by executing the `VMRUN` instruction, providing the appropriate VMCB. Execution of the guest continues until an enabled `#VMEXIT` condition occurs. Control returns to the hypervisor at the instruction following `VMRUN`. The guest state, including the reason for the `#VMEXIT`, is placed in the save-state area of the VMCB. The hypervisor may emulate, deny, or alter the execution of the intercepted instruction by making changes to the VMCB. Table 2 shows the logical layout of the VMCB[5]. Listing 1 is the C language listing for the VMCB.

The functionality of primary interest in an HVM rootkit is the set of enabled exception conditions in the VMCB and the method chosen for handling the `#VMEXIT` conditions. Although nearly any operation can be intercepted by the hypervisor, a minimal AMD-V hypervisor need only trap execution of the `VMRUN` instruction.

3 Intel Virtualization

Intel^{TM} Virtualization Technology for IA-32 processors is known as VT-x. These extensions are conceptually the same as those implemented in AMD-V, but many of the details differ significantly. Some of these differences stem from the fact that while the AMD64 memory controller is on-die with the CPU, Intel implementation is on a separate chip.

The primary VT-x data structure is the VMCS. It fills the same role of the AMD-V VMCB, but has a completely different structure. The VMCS is of varying length and indicates that Intel may produce incompatible revisions of VT-x since the structure starts with a revision number and states that software should not use a VMCS of one revision on a processor of another. Table 3 shows the initial fields of the VMCS. The VMCS data is made up of varying length sections. These are described briefly in Table 4.

The instructions that make up the VT-x extensions include `VMPTRLD` and `VMPTRST` which respectively load to and save from the *current-VMCS pointer*. The `VMCLEAR` instruction ensures that data for the VMCS specified are copied to the VMCS region in memory and initializes parts of the VMCS region. The `VMWRITE` and `VMREAD` instructions copy components of the VMCS to and from the contents of a register operand. `VMCALL` is used in non-root operation to allow the guest to interact with the HVM. The `VMLAUNCH` instruction begins execution of a guest, while `VMRESUME` continues execution of the current guest. `VMXON` and `VMXOFF` enable and disable HVM operations on the current core.

A minimal Intel VT-x hypervisor must implement exit handling for CPUID, INVD, and MOV from CR3. The hypervisor must also handle exits for instructions introduced with VT-x. These include VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON. In addition there are required exits that the chipset implements.

4 Rootkits

The first rootkits were collections of user-mode programs for UNIX and UNIX-like operating systems that replaced key system binaries in order to present a false view of the system to administrators and other users [6]. They generally filtered out processes, files and network connection information whose presence would betray the existence of malicious software running on the system. Although initially effective, these rootkits were generally trivial to detect by using alternate user mode tools to retrieve the same information.

As user-mode detection techniques evolved, rootkit authors moved to using shared libraries and kernel modules. This allowed the rootkits to generalize, affecting the system view of not only chosen programs, but all programs on the system that use the subverted library or system service. Kernel-mode rootkits are the most prevalent kind today, most likely because they are well-situated to hide from both user-mode programs and system-level security services. User-mode and shared library rootkits may see a resurgence, however, in light of Microsoft Windows Vista^{TM}'s driver-signing requirement and kernel patch protection. Regardless of whether they are implemented in a shared library or as a device driver, contemporary rootkits do their work by hooking or rewriting selected services that provide information about the current state of the system (i.e. processes, files, and network objects). The only real drawback of user-mode rootkits is that they are vulnerable to detection by other services running in kernel mode.

HVM rootkits, theoretically at least, are not vulnerable to any action the operating system can take since the rootkit runs in a more privileged state than the OS. A hypervisor need not even exist in memory that is accessible to the operating system. When a hypervisor bootstraps itself, it makes any instructions or memory accesses which could betray its existence more privileged than the operating system. When these instructions occur, the processor traps to the hypervisor, allowing it to modify the results. Thus, the hypervisor need not make any changes to the operating system to hide its own presence. The hypervisor can, however, make any such changes with impunity and make them undetectable by the operating system.

In 2006, two researchers (Dino Dai Zovi and Joanna Rutkowska) worked in parallel (but not collaboratively) on proving the concept of an HVM rootkit. They presented within hours of each other at the 2006 Black Hat Briefings in Las Vegas; Zovi presented a skeleton framework for an HVM rootkit implemented with Intel VT, and Rutkowska presented similar materials for an HVM rootkit implemented with AMD-V. Neither presented source code, but both claimed to have working prototypes named Vitriol and Blue Pill, respectively. Of the two, only Rutkowska claimed to have designed a fully undetectable rootkit, and that claim became the center of controversy within the research community due to lack of proof made available and inability to test her claims.

Researchers interested in creating their own HVM rootkits had to start more or less from scratch. For nearly a year after the Black Hat presentations, there still were not any publicly-available HVM examples to build upon, except in the code contributed to the Xen project by engineers from Intel and AMD. The Xen code, unfortunately, is far more complex than what is required to implement an HVM rootkit. Xen is designed to support multiple guest VMs and multiple CPU architectures and abstracts architecture-specific virtualization schemes with its own set of meta-structures. The AMD and Intel virtualization technology documentation is complete as a specification, but it is far from being a roadmap to implementation.

Unlike the typical hypervisor which loads before it starts its guest VM, an HVM rootkit loads into an already loaded operating system and turns it into its guest VM. Therefore the rootkit must initialize the hypervisor functionality and dynamically take over as host, turning the running operating system into a guest. This process has been repeatedly referred to as "forking" or "migrating" the OS to a guest state, but these terms are misleading. The hypervisor and OS do not run concurrently (as they would if they were "forked" in the UNIX sense) nor does the OS need to be altered or moved in any way ("migrated"). The process of installing an HVM hypervisor in a running OS is more like that of installing a "shim," in the software engineering sense of the word. A shim is a piece of code between two layers, in this case between the CPU and all other software running on the system.

5 Prerequisites

As mentioned in Section 2, an AMD-V hypervisor initiates guest execution with `VMRUN`. The `VMRUN` instruction must be issued in ring-0, i.e. from a kernel-mode driver. An HVM rootkit, then, must begin with a driver. But when the hypervisor is installed, that driver would now be in the "guest" and may reveal the presence of the hypervisor to the guest. Rather than hiding the driver in the normal manner of rootkits, we have chosen to have the driver allocate a non-paged memory region for containing the hypervisor, copy the hypervisor code there, and then unload the driver from the kernel. Therefore the driver that installs the hypervisor must act only as a loader for the hypervisor and not as a container for it. After the hypervisor is installed, the driver file on disk could even be deleted, perhaps from the driver's unload routine.

Our code requires the Windows DDK to compile (available online from Microsoft). We supply a Visual Studio 2005 project for convenience (compatible with the freeware Visual C++ Express). The project includes the source for a user-mode executable that checks your system for AMD-V support, registers and loads the driver, then unregisters and unloads the driver.

6 Implementation

The following steps are required to install an HVM rootkit under AMD-V:

(a) load a driver (requires administrator privilege);

(b) turn on the flag enabling AMD-V functionality;

(c) allocate non-paged contiguous physical memory space for a VMCB structure;

(d) allocate a non-paged area in kernel memory and copy the hypervisor code there;

(e) allocate a non-paged contiguous physical memory space for a host save area, and store the physical address to this area in the VM_HSAVE_PA register;

(f) initialize the control area of the VMCB with a set of intercept conditions that will cause execution to transfer out of the guest and back to the hypervisor;

(g) initialize the guest area of the VMCB with the entire current state of the operating system, in order to seamlessly turn the current state into a guest VM;

(h) transfer execution to the hypervisor code outside the driver;

(i) issue a `VMRUN` instruction from the hypervisor to "launch" the operating system, thereby performing the seamless switch to guest mode;

(j) transfer execution back to the driver, and unload the driver.

In our implementation of the technique, driver code begins execution, like all Windows kernel-mode drivers, in its `DriverEntry()` routine. Because multi-core and multi-CPU support is desired, the next step is to iterate across all CPUs in the system and perform the hypervisor setup steps on each of them in turn. In order to do this, we to use a constant set by the Windows kernel called `KeNumberProcessors` and an undocumented kernel API called `KeSetAffinityThread()` to immediately reschedule the current thread onto the processor/core specified. A routine we define called `EnableAMDV()` performs the steps to enable AMD-V functionality on each processor/core: read the Model-Specific Register (MSR) called "VM_CR" and check for the bit signifying that AMD-V has been disabled in the BIOS (bit 4). If it is clear, `EnableAMDV()` proceeds to set the appropriate bit in the Extended Features MSR to turn on AMD-V functionality (EFER.SVME). Next, a region of physically contiguous, page-aligned, non-pageable memory is allocated by the routine `AllocateVMCB()` using two kernel APIs: `MmAllocateContiguousMemory()` and `MmProbeAndLockPages()`. The next routine to be called is `StartHypervisor()`, which allocates another region of memory for holding the hypervisor code (defined by the routine `HypervisorCode()`), by calling our routine `CopyHypervisorCodeToMemory()`. Only one instance of the hypervisor is necessary, even if there are multiple processors/cores in the system. Next, the host-state save area is allocated, and its physical address is written to the required location, the VM_HSAVE_PA MSR. Whenever physical addresses of data are required, we use the kernel API `MmGetPhysicalAddress()`. The chosen set of intercepts to be implemented by the hypervisor is written into the appropriate field of the VMCB, as is the address where guest execution should begin.

Finally, `StartHypervisor()` calls the `HypervisorCode()` routine so that the hypervisor can actually begin. In order to ensure exclusivity on each CPU, the processor's IRQ priority level (IRQL) is raised to DISPATCH_LEVEL (high priority) and the CLGI instruction is executed to disable global interrupts. Then we fill in the all of the initial guest's state with the current values of the host (copy the values of all of the current registers into the corresponding locations in the VMCB guest state save area), load the physical address of the VMCB into EAX, and call VMRUN. Execution begins in guest mode now, at the end of the `HypervisorCode()` routine, which returns back to the driver code. The driver code proceeds to switch CPU cores and load hypervisors on the other processors/cores in the same fashion. When all processors/cores have entered guest mode, the driver is free to unload and go away.

The preceding discussion is a complete description of how to implement an HVM hypervisor using AMD-V. However, to go one step further and create an HVM rootkit, methods by which the hypervisor may be detected or attacked must be taken into account. There are many avenues of attack which the HVM rootkit must defend. These include, but may not be limited to,

(a) emulating all of the AMD-V feature extensions,

(b) emulating access to all AMD-V related MSRs,

(c) intercepting I/O to off-CPU local timers to cheat timing analysis,

(d) "rewinding" the TSC counter after emulating instructions (again to cheat timing analysis),

(e) implementing Shadow Paging or using the Nested Page Table feature of AMD-V to avoid detection by advanced page table caching (TLB) analysis,

(f) using External Access Protection to cloak the memory region containing the hypervisor as seen by peripheral devices with DMA access to system memory,

(g) intercepting access by the guest OS's software to the memory region containing the hypervisor,

(h) detecting "anomalous detector behavior" that cannot be defeated, and having an unload/reload scheme to dodge such detection.

In the next section we will discuss in detail the reasons for the rootkit needing each of these defenses. The official AMD documentation makes no mention of any "hypervisor present" bit flag, but unofficial documentation at Sandpile.org claims that when a hypervisor is present and `CPUID` function 8000_000A is called, that RAX bit 8 is set to 1 [7]. We were not able to confirm this from any other source, nor in testing on first-generation AMD-V capable processors.

7 Detection

One might be tempted to suggest that rootkits are only expected to hide from less security-savvy portions of the population in order to be successful. However, contemporary kernel-mode rootkits are already sufficiently hidden from a casual investigation. The goal of an HVM hypervisor should be to avoid detection even under expert analysis.

There are two primary approaches to hiding the existence of a hypervisor. The first is to emulate a less capable CPU, i.e. one without the AMD-V instructions. The second is to emulate the AMD-V instructions in the hypervisor.

The first approach is by far the easier to implement, but makes the rootkit considerably less stealthy. Incomplete emulation of the target system will be immediately obvious to a user expecting a processor with virtualization functionality. *Xen* 3.1.0, contrary to some claims [8], takes this simpler approach.

The second, and preferred method is to completely emulate all instructions intercepted by the hypervisor. This would in theory allow any hypervisor, including the rootkit itself, to execute in the guest exactly as it would directly on the hardware. An HVM rootkit should give the appearance of full AMD-V functionality. In order to support nested hypervisors in this fashion, it is only necessary to intercept and emulate the `VMRUN` instruction. The "BluePill" rootkit takes this approach [1].

In our code, the simpler approach was taken – it does not attempt to support nested hypervisors. For the sake of discussion, we will nevertheless consider all of the theoretical measures and countermeasures for detecting a "complete" HVM rootkit.

Current research indicates that performance of current hardware-supported virtualization is significantly worse than that implemented completely in software [9, 10, 11]. If a user is familiar with the normal operation of his system, the difference will be noticed. The more limited virtualization necessary for a simple rootkit will, under most circumstances, have much smaller performance impact on the system. We can also assume that hardware implementations will only get better (e.g. with Intel moving to an on-die memory controller [12] and the industry adopting IOMMU to virtualize peripheral devices [13]) such that performance will eventually be nearly indistinguishable from native execution of the operating system.

Even so, there is and always will be a measurable performance hit incurred whenever a hypervisor intercepts any instruction. In tests we performed with an AMD Athlon64 X2 4000+ processor, the difference between the time required to perform a `RDMSR EFER` natively and with a hypervisor interception was profound. Native execution took only 50 processor cycles, while intercepted and emulated execution took around 10,000 cycles.

Checking a local time stamp before and after the execution of certain instructions suspected to be hypervisor-intercepted seems like an obvious attack. However, because successful virtualization has historically depended on a coherent and predictable time stamp counter (TSC), both AMD and Intel implemented methods by which the hypervisor can account for trap-related delays. The hypervisor can trap on the `RDTSC` instruction, or simply use a "TSC Offset" feature. In practice, though, this feature is not reliable enough to use as a hypervisor stealth mechanism. An HVM rootkit needs to read the TSC before and after emulating any instruction, and manually "fix up" the value in the TSC as seen by the guest.

Yet another problem that complicates any attempt to use TSC-based timing measurements as a detection scheme is the power/heat-saving features of modern CPUs. Modern CPUs, especially those designed for laptop use, can dynamically vary their clock speed to meet the current processing load. A 2GHz AMD processor might scale down to a mere 800MHz when AMD Cool N' Quiet^{TM} is enabled and supported by the motherboard. This is a documented cause for TSC unreliability [14]. Dynamic clocking is not an feature unique to AMD processors. Many chips have an unreliable TSC when features like speed-stepping, various power modes, and multiple processors or cores are active [15]. Alternate types of local system hardware timers may work in place of TSC for timing-analysis based HVM rootkit detection schemes. VMWare has written a white-paper specifically on the challenges of virtualizing hardware timers for VM guests, and it may be of help when studying this issue [16]. A detector could make use of any of the various timers present on the system to do timing-analysis detection, and an HVM rootkit would need to intercept access to all of those timers in order to maintain hidden. No rootkit to date is capable of this, but it is perhaps theoretically possible through interception of the I/O instructions used to access these off-CPU timers.

VMWare engineer Keith Adams and Symantec researcher Peter Ferrie have both independently proposed another class of detection method based on profiling resource usage by the hypervisor [17, 8]. With the assumption that an HVM rootkit must intercept the `CPUID` instruction, and that doing so will inherently flush part or all of the TLB, a virtualized guest could detect this change made to the TLB. Normal execution of the `CPUID` instruction does not flush the TLB. The guest detects a change to the TLB by measuring the time required to access a certain page of memory, with and without first explicitly flushing the TLB. This produces a known difference: the difference between accessing a cached page table entry (PTE) and accessing the same PTE when it is not already cached in the TLB. Then the guest executes a `CPUID` instruction. If a subsequent attempt to access the same PTEs (that ought be cached) takes the amount of time to access a non-cached page, then it is known that the TLB has been flushed during the `CPUID` instruction and this would indicate the presence of an HVM hypervisor. Using Intel VT, an HVM rootkit cannot avoid this detection scheme since `CPUID` generates an exit unconditionally. However, when using AMD-V, the `CPUID` instruction does not indicate the presence of a hypervisor, and thus it need not be intercepted. This does not mean the TLB profiling method will not work against an AMD-V rootkit, it just means the detector must choose a different instruction as the test case, such as `RDMSR EFER`.

Building upon the idea of trying to indirectly observe the hypervisor's cache usage, Adams suggested another TLB profiling approach that does not require the use of a timer. He suggested a detector which directly edited Page Table Entries (PTEs) to invalidate the TLB's cached copies, then executed an HVM-intercepted instruction, then attempted to access the memory locations that were known to have been cached in the TLB directly beforehand. It would be apparent whether the memory locations were accessed using the mappings cached in the TLB, or the ones in the actual Page Table. If it were the former, the detector could assume that the TLB had been flushed and refilled, signing the presence of an HVM hypervisor.

HVM rootkit proponents have pointed out that AMD-V, unlike Intel VT, supports a feature whereby every TLB entry is tagged with an Address Space Identifier (ASID) marking whether it is an entry from the host or from the guest. The purpose of this is to allow the hypervisor to avoid flushing the guest's TLB during any intercepted instruction. However, it is a performance feature, not a stealth feature. It will not entirely prevent alterations to the TLB, and does not enable an HVM rootkit to fully hide from TLB-profiling detection.

It has also been suggested that the Nested Page Tables feature of an upcoming revision of AMD-V may defeat TLB profiling-based detection [18]. The HVM rootkit would use Nested Page Tables to monitor accesses to the page table entries for detector-like behavior. If such anomalous behavior were observed, the rootkit would unload and return some time later.

Another detection scheme that has been suggested is to exploit buggy behavior ("errata") specific to certain CPU models [19]. Most modern x86 processors have known and documented errata, which is any behavior that deviates from the processor architecture specification. If an HVM hypervisor is not designed to accurately emulate the bugs in the native behavior of the CPU it is running on, an HVM hypervisor detector could test for these inconsistencies. One such example is AMD Erratum 140, which documents that VMCB.TSC_OFFSET is not added to the timestamp counter when it's being read via MSR 0x10. As a detection scheme, this approach is unattractive to some because it relies on processor model-specific bugs rather than a generically applicable technique.

The simplest and yet most effective detection method is also one of the most recent [20]. A multicore processor, as most of today's systems contain, allows for true concurrent execution of two threads. An HVM rootkit detector can use this capability to execute a simple loop counter in a thread on one CPU core, while on another CPU core a second thread attempts an instruction that would be intercepted by an HVM hypervisor. The threads are synchronized to start and stop at the same time, thus the timer thread is able to reliably measure the execution time of the instruction thread. Results will vary greatly between a system with and one without a hypervisor present.

At this stage, the practical detectability of HVM rootkits is still hotly contested. The trend is clear, however. Researchers are continually suggesting new detection methods, and the code required in the HVM rootkit to evade each one is successively layering more and more complexity into the rootkit's design. Paradoxically, the more complicated an HVM rootkit must be to avoid known detection schemes, the more presence it has within the system to have to work to hide. With that said, the popular consensus has been that HVM rootkits will eventually become too difficult or costly to implement, or even that they are ultimately unfeasible. One might point to the fact that there are no known HVM rootkits in the wild. But neither are there any HVM rootkit detectors to find them if there were.

Proponents of the HVM rootkit threat make the point that it is not enough to simply detect the presence of an HVM hypervisor. If we cannot ascertain anything about the nature of the hypervisor, then we cannot make any decisions on behalf of the user. With the increasing adoption and utilization of HVM technology for legitimate purposes, simply knowing that it is being actively used on a particular system may not be useful information at all.

8 Prevention

Regardless of the effectiveness of HVM rootkit detection schemes, we ought to consider how to prevent the rootkit's installation in the first place. Obviously, most of the effort in defending against HVM rootkits should be in preventing malicious code from executing at all, specifically code with administrator privileges required to stage a rootkit installation. However, assuming malicious code does execute with administrator privilege and enter the kernel, there are still two preventative practices that should defend against an HVM rootkit installation.

The first method is to preemptively load a protective hypervisor which denies any subsequent attempts to load other (potentially malicious) hypervisors. The preventative hypervisor need only intercept `VMRUN`, since hiding its presence is not one of its goals.

The second method is a brand new security feature that has only very recently been introduced by AMD, likely in response to the concerns raised in the security research community. In the July 2007 revision of the AMD64 Programmer's Manual, AMD documents a "revision 2" of the SVM architecture, which has the ability to lock the SVM Enable flag, and optionally, to provide a 64-bit key to unlock it again. If a key is not set before the lock is activated, the SVM Enable flag cannot be changed without a processor reset. We were not able to obtain a processor supporting this feature at the time we wrote this paper, but expect such CPUs to be widely available soon.

Either one of these methods ought to provide full protection against unwanted HVM hypervisors. If the lock and key method is used, however, one should be aware of the importance of securing the key value against access by malicious software on the local machine.

9 Memory forensics

If detection of an HVM rootkit is difficult and impractical, actually procuring a forensic copy of an active HVM rootkit may very well be impossible. The HVM rootkit only needs to exist in RAM, meaning a forensic examiner is tasked with obtaining a copy of all of physical memory in order to search for the presence of an HVM hypervisor. There are two common approaches to imaging physical memory. One approach is to run software on the local system which accesses physical memory through an operating system hardware interface layer. In this case, an HVM hypervisor is able to intercept and control access to the physical memory by way of its exception conditions. It could present the pages of memory containing its hypervisor code to appear to contain all zeros or all ones, for instance. This subversion of memory would go unnoticed by the forensic capture process. The second approach to imaging physical memory is to take advantage of the abilities of certain hardware peripheral devices that can perform Direct Memory Access (DMA), that is, devices which can read and write system memory without intervention by the CPU. Normally, this would be the most reliable and forensically sound method for capturing the contents of physical memory. However, AMD-V introduces a feature to secure the system memory against any unwanted access from DMA devices. It is called External Access Protection (EAP), and it allows the hypervisor to define "Device Exclusion Vectors," or maps of memory which should be secured against DMA access by certain devices on the system. EAP is implemented in and performed by the Northbridge host bridge controller, and is set up by way of a pair of control registers in PCI configuration space, DEV_OP and DEV_DATA. The use of this feature is documented in the AMD64 Architecture Programmer's Manual [5].

Note that a similar technique was first introduced by researcher Joanna Rutkowska at a presentation at Black Hat Briefings DC 2007 [21]. Rutkowska's technique is more general; it does not require an HVM hypervisor nor the EAP feature to implement, but it works similarly by programming the memory controller to present a different view of memory as seen from DMA device access than by local system software access. However, without the HVM hypervisor's ability to secure access to the memory controller, a clever forensic DMA device could undo (re-program) the subverted state of the memory controller and achieve unimpeded access to the true values of system memory [22]. No such "clever" forensic device yet exists, however. Also, it is not clear whether evidence gathered via such an approach is legally admissible in a court of law, because the approach requires tampering with the contents of memory in order to image it.

10 Other uses

The concept of an undetectable HVM hypervisor is worrisome, and naturally we should be concerned with the potential that it is a technology that will be used against the user. It has been demonstrated how an HVM rootkit can be used to hide evidence of a security breach from its owner. In the same manner, digital rights management (DRM) seems a natural application for the technology to hide cryptographic secrets from reverse engineers.

Obviously though, there are benign uses for HVM hypervisors, even when implemented as the HVM rootkit is, loadable in a running OS. For one, it could be used as an undetectable debugger to aid in malware analysis. There are many powerful anti-debugger techniques [23] that malware authors can employ to frustrate or even defeat reverse engineering of their executables. A debugger implemented with the use of an HVM hypervisor to trap execution, and a hypervisor-guest control scheme, using VMMCALL perhaps, would give reverse engineers a powerful time-saving tool. Another use for an HVM hypervisor, mentioned previously, is as a security monitor to prevent subsequent attempts to load unwanted hypervisors. The concept might be extended to monitor and/or log access to other resources on the system. This leads to a third use of an undetectable HVM hypervisor, which is as a honeypot, i.e. a system set up to lure malicious attackers and monitor and study their behavior [24].

11 Terminology and Operation

Table 1: Comparison of common terms

AMD	**Intel**	**Description**
VMCB	VMCS	Data structure describing the behavior of the hypervisor and the guest and host save state areas
IOPM		Input/Output permissions map
MSRPM		Model Specific Register permissions map
RAX	current-VMCS	Register containing physical address of current VM control data structure
VM_CR	VM_HSAVE_PA	VM control register.

EFER.SVME := 1
CR4.VMXE := 1
VMXON
	IA32_FEATURE_CONTROL.bit_5 := 1	Enable VM feature on processor.

MOV RAX, vmcb_pa VMRUN RAX	MOV RAX, vmcs_pa VMPTRLD VMLAUNCH	Load VM control structure and start a guest.
VMSAVE	VMWRITE	Write guest state information to hypervisor "save-state" area.
VMLOAD	VMREAD	Read guest state information from hypervisor "save-state" area.
VMMCALL	VMCALL	Explicitly exit to hypervisor.
AMD-V^{TM}	VT-x^{TM}	Official trade names for virtualization technology
Pacifica	Vanderpool	Development code names for virtualization technology
#VMEXIT	VM exit	Event causing return to hypervisor
SVM	VMX	Generic term used for extensions associated with virtual machines.
guest	non-root	The cpu mode for operating systems running under a hypervisor
host	root	The cpu mode for a hypervisor when HVM is enabled.

Table 2: Logical layout of the VMCB[5]

Offset	Bits	Description
000h	0–15	Intercept reads of CR015.
	16–31	Intercept writes of CR015.
004h	0–15	Intercept reads of DR015.
	16–31	Intercept writes of DR015.
008h	0–31	Intercept exception vectors 031.
00Ch	0	Intercept INTR.
	1	Intercept NMI.
	2	Intercept SMI.
	3	Intercept INIT.
	4	Intercept VINTR.
	5	Intercept CR0 writes other than CR0.TS or CR0.MP.
	6	Intercept reads of IDTR.
	7	Intercept reads of GDTR.
	8	Intercept reads of LDTR.
	9	Intercept reads of TR.
	10	Intercept writes of IDTR.
	11	Intercept writes of GDTR.
	12	Intercept writes of LDTR.
	13	Intercept writes of TR.
	14	Intercept RDTSC.
	15	Intercept RDPMC.
	16	Intercept PUSHF.
	17	Intercept POPF.
	18	Intercept CPUID.
	19	Intercept RSM.
	20	Intercept IRET.
	21	Intercept INTn (software interrupt).
	22	Intercept INVD.
	23	Intercept PAUSE.
	24	Intercept HLT.
	25	Intercept INVLPG.
	26	Intercept INVLPGA.
	27	IOIO_PROT–Intercept IN/OUT accesses to selected ports.
	28	MSR_PROT–intercept RDMSR or WRMSR accesses to selected MSRs.
	29	Intercept task switches.
	30	FERR_FREEZE: intercept processor freezing during legacy FERR handling.
	31	Intercept shutdown events.
010h	0	Intercept VMRUN.
	1	Intercept VMMCALL.
	2	Intercept VMLOAD.
	3	Intercept VMSAVE.
	4	Intercept STGI.
	5	Intercept CLGI.
	6	Intercept SKINIT.
	7	Intercept RDTSCP.
	8	Intercept ICEBP.
	9–31	RESERVED, MBZ
014h-03Fh	all	RESERVED, MBZ
040h	0–63	IOPM_BASE_PA–Physical base address of IOPM (bits 11:0 are ignored).
048h	0–63	MSRPM_BASE_PA–Physical base address of MSRPM (bits 11:0 are ignored).
050h	0–63	TSC_OFFSET–To be added in RDTSC and RDTSCP.

Table 2: Logical layout of the VMCB (continued)

Offset	**Bits**	**Description**
058h	0–31	Guest ASID.
	32–39	TLB_CONTROL (0–Do nothing, 1–Flush TLB on VMRUN, Others-Reserved)
	40–63	RESERVED, MBZ
060h	0–7	V_TPR–The virtual TPR for the guest;
		currently bits 3:0 are used for a 4-bit virtual TPR value; bits 7:4 are MBZ.
	8	V_IRQ–If nonzero, virtual INTR is pending.
	9–15	RESERVED, MBZ
	16–19	V_INTR_PRIO–Priority for virtual interrupt.
	20	V_IGN_TPR–If nonzero, the current virtual interrupts ignores the (virtual) TPR.
	21–23	RESERVED, MBZ
	24	V_INTR_MASKING–Virtualize masking of INTR interrupts.
	25–31	RESERVED, MBZ
	32–39	V_INTR_VECTOR–Vector to use for this interrupt.
	40–63	RESERVED, MBZ
068h	0	INTERRUPT_SHADOW–Guest is in an interrupt shadow;
	1–63	RESERVED, MBZ
070h	0–63	EXITCODE
078h	063	EXITINFO1
080h	063	EXITINFO2
088h	063	EXITINTINFO
090h	0	NP_ENA–Enable nested paging.
	163	RESERVED, MBZ
098h–0A7h		RESERVED. MBZ
0A8h	063	EVENTINJ–Event injection.
0B0h	063	H_CR3–Host-level CR3 to use for nested paging.
0B4h–3FFh		RESERVED, MBZ

Table 3: Intel VMCS Region [25]

Byte	Contents
0	VMCS revision identifier
4	VMX-abort indicator
8	VMCS data (implementation-specific format)

Table 4: Data Region in VMCS [25]

Section	Description
guest-state area	Guest processor saved on VM exits and loaded upon VM entry. This includes guest registers and other ancillary data such as interrupt state.
host-state area	CPU state is loaded on VM exit. This includes segment registers, descriptor table registers, as well as system call table information.
VM-execution control fields	Control processor behavior in non-root operation. This includes bit-maps for which guest execution conditions cause a VM exit such as interrupts, reserved instructions and register accesses.
VM-exit control fields	Controls how certain VM exits occur. This contains information on guest address space size and whether interrupts are acknowledged on exit.
VM-entry control fields	Determine mode of operation for the guest after VM entry. This includes controls for the loading and saving of model specific registers and for the injection of exceptions into the guest environment.
VM-exit information fields	This area recieves information on the cause and nature of a VM exit.

Listing 1: Amd-v.h – definition of VMCB

```
#pragma pack(1) // In GCC, use __attribute__ ((packed))
typedef struct
{
  u32 interceptCR0;
  u32 interceptDR0;
  u32 interceptExceptionVectors;
  u32 interceptsGeneralPurpose1;
  u32 interceptsGeneralPurpose2;

  u8 reserved_space_01[44];
  u64 iopmBasePA;
  u64 msrpmBasePA;
  u64 tscOffset;
  u32 guestASID;
  u8 tlbControl;
  u8 reserved_space_02[3];
  vintr_t vintr;

  u64 interruptShadow;
  u64 exitcode;
  u64 exitinfo1;
  u64 exitinfo2;
  u64  exitintinfo;

  u64 np_enable;
  u8 reserved_space_03[16];

  eventinj_t  eventinj;
  u64 host_level_cr3;
  u8 reserved_space_04[840];
  segment_selector_t es;
  segment_selector_t cs;
  segment_selector_t ss;
  segment_selector_t ds;
  segment_selector_t fs;
  segment_selector_t gs;
  segment_selector_t gdtr;
  segment_selector_t ldtr;
  segment_selector_t idtr;
  segment_selector_t tr;
  u64 reserved_space_05[5];
  u8 reserved_space_06[3];
  u8 cpl;
  u32 reserved_space_07;
  u64 efer;                    // offset 1024 + 0xD0
  u64 reserved_space_08[14];
  u64 cr4;                     // loffset 1024 + 0x148
  u64 cr3;
  u64 cr0;
  u64 dr7;
  u64 dr6;
  u64 rflags;
  u64 rip;
  u64 reserved_space_09[11];
  u64 rsp;
  u64 reserved_space_10[3];
  u64 rax;
  u64 star;
  u64 lstar;
  u64 cstar;
  u64 sfmask;
  u64 kernel_gs_base;
  u64 sysenter_cs;
  u64 sysenter_esp;
  u64 sysenter_eip;
  u64 cr2;
  u64 pdpe0;
  u64 pdpe1;
  u64 pdpe2;
  u64 pdpe3;
  u64 guest_pat;  //used only if nested paging is enabled
  u64 reserved_space_11[50];
  u64 reserved_space_12[128];
  u64 reserved_space_13[128];
} amdv_vmcb_t;
```

References

[1] J. Rutkowska, Subverting Vista^{TM} Kernel for Fun and Profit. [Online]. Available: http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf

[2] D. A. D. Zovi, Hardware Virtualization Rootkits. [Online]. Available: http://www.theta44.org/software/HVM_Rootkits_ddz_bh-usa-06.pdf

[3] S. Embleton, Hooking CPUID – A Virtual Machine Monitor Rootkit Framework. [Online]. Available: http://rootkit.com/newsread.php?newsid=758

[4] J. Rutkowska and A. Tereshkin, Blue Pill Project. [Online]. Available: http://bluepillproject.org/

[5] AMD64 Architecture Programmer's Manual Volume 2: System Programming. [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

[6] G. Hoglund and J. Butler, Rootkits: Subverting the Windows Kernel.

[7] IA-32 architecture – CPUID. [Online]. Available: http://www.sandpile.org/ia32/cpuid.htm

[8] P. Ferrie, Attacks on Virtual Machine Emulators. [Online]. Available: http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf

[9] K. Adams and O. Agesen, A Comparison of Software and Hardware Techniques for x86 Virtualization. [Online]. Available: http://www.vmware.com/pdfs/asplos235_adams.pdf

[10] J. Munro, Virtual Machines & VMware, Part II. [Online]. Available: http://www.extremetech.com/article2/0,1697,1156372,00.asp

[11] A. Kivity, Subject: [ANNOUNCE] kvm-22 release. [Online]. Available: http://lkml.org/lkml/2007/5/6/39

[12] Intel Developer Forum Day 1 News Disclosures From Beijing. [Online]. Available: http://www.intel.com/pressroom/archive/releases/20070416supp.htm

[13] AMD I/O Virtualization Technology (IOMMU) Specification. [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf

[14] Microsoft Knowledge Base Article 895980 - Programs that use the QueryPerformanceCounter function may perform poorly in Windows Server 2003 and in Windows XP. [Online]. Available: http://support.microsoft.com/kb/895980/

[15] Linux kernel development discussion [patch 13/19] GTOD: Mark TSC unusable for highres timers. [Online]. Available: http://readlist.com/lists/vger.kernel.org/linux-kernel/55/277594.html

[16] Timekeeping in VMWare Virtual Machines. [Online]. Available: http://www.vmware.com/pdf/vmware_timekeeping.pdf

[17] K. Adams, Blue Pill Detection in Two Easy Steps. [Online]. Available: http://x86vmm.blogspot.com/2007/07/bluepill-detection-in-two-easy-steps.html

[18] J. Rutkowska and A. Tereshkin, IsGameOver(), anyone? [Online]. Available: http://bluepillproject.org/stuff/IsGameOver.ppt

[19] T. Garfinkel and K. Adams, Compatibility is Not Transparency: VMM Detection Myths and Realities. [Online]. Available: http://www.cs.cmu.edu/~jfrankli/hotos07/vmm_detection_hotos07.pdf

[20] E. Barbosa, Detecting Blue Pill. [Online]. Available: http://rapidshare.com/files/42452008/detection.rar.html

[21] J. Rutkowska, Beyond the CPU: Defeating Hardware Based RAM Acquisition Tools (Part I: the AMD case). [Online]. Available: http://blackhat.com/presentations/bh-dc-07/Rutkowska/Presentation/bh-dc-07-Rutkowska-up.pdf

[22] T. Ptacek, Rutkowska Concedes: Clever Forensics Devices Can Beat DMA Tricks. [Online]. Available: http://www.matasano.com/log/870/rutkowska-concedes-clever-forensics-can-beat-DMA-tricks

[23] N. Lawson, Anti-debugger techniques are overrated. [Online]. Available: http://rdist.root.org/2007/04/19/anti-debugger-techniques-are-overrated

[24] Honeypot (computing) - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Honeypot_(computing)

[25] Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide. [Online]. Available: http://www.intel.com/design/processor/manuals/253669.pdf