SubVirt: Implementing malware with virtual machines

Samuel T. King Peter M. Chen
University of Michigan
{kingst,pmchen}@umich.edu

Yi-Min Wang Chad Verbowski Helen J. Wang Jacob R. Lorch
Microsoft Research
{ymwang,chadv,helenw,lorch}@microsoft.com

Abstract

Attackers and defenders of computer systems both strive to gain complete control over the system. To maximize their control, both attackers and defenders have migrated to low-level, operating system code. In this paper, we assume the perspective of the attacker, who is trying to run malicious software and avoid detection. By assuming this perspective, we hope to help defenders understand and defend against the threat posed by a new class of rootkits.

We evaluate a new type of malicious software that gains qualitatively more control over a system. This new type of malware, which we call a virtual-machine based rootkit (VMBR), installs a virtual-machine monitor underneath an existing operating system and hoists the original operating system into a virtual machine. Virtual-machine based rootkits are hard to detect and remove because their state cannot be accessed by software running in the target system. Further, VMBRs support general-purpose malicious services by allowing such services to run in a separate operating system that is protected from the target system. We evaluate this new threat by implementing two proof-of-concept VMBRs. We use our proof-of-concept VMBRs to subvert Windows XP and Linux target systems, and we implement four example malicious services using the VMBR platform. Last, we use what we learn from our proof-of-concept VMBRs to explore ways to defend against this new threat. We discuss possible ways to detect and prevent VMBRs, and we implement a defense strategy suitable for protecting systems against this threat.

1. Introduction

A battle is taking place between attackers and defenders of computer systems. An attacker who manages to compromise a system seeks to carry out malicious activities on that system while remaining invisible to defenders. At the same time, defenders actively search for successful attackers by looking for signs of system compromise or malicious activities. In this paper, we assume the perspective of the attacker, who is trying to run malicious software (malware) and avoid detection. By assuming this perspective, we hope to help defenders understand and defend against the threat posed by a new class of rootkits (tools used to hide malicious activities) [24].

A major goal of malware writers is *control*, by which we mean the ability of an attacker to monitor, intercept, and modify the state and actions of other software on the system. Controlling the system allows malware to remain invisible by lying to or disabling intrusion detection software.

Control of a system is determined by which side occupies the lower layer in the system. Lower layers can control upper layers because lower layers implement the abstractions upon which upper layers depend. For example, an operating system has complete control over an application's view of memory because the operating system mediates access to physical memory through the abstraction of per-process address spaces. Thus, the side that controls the lower layer in the system has a fundamental advantage in the arms race between attackers and defenders. If the defender's security service occupies a lower layer than the malware, then that security service should be able to detect, contain, and remove the malware. Conversely, if the malware occupies a lower layer than the security service, then the malware should be able to evade the security service and manipulate its execution.

Because of the greater control afforded by lower layers in the system, both security services and rootkits have evolved by migrating to these layers. Early rootkits simply replaced user-level programs, such as *ps*, with trojan horse programs that lied about which processes were running. These user-level rootkits were detected easily by user-level intrusion detection systems such as TripWire [29], and so rootkits moved into the operating system kernel. Kernel-level rootkits such as FU [16] hide malicious processes by modifying kernel data structures [12]. In response, intrusion detectors also moved to the kernel to check the integrity of the kernel's data structures [11, 38]. Recently, researchers have sought to hide the memory footprint of malware from kernel-level detectors by modifying page protections and intercepting page faults [43]. To combat such techniques, future detectors may reset page protections and examine the code of the page-fault handler.

Current rootkits are limited in two ways. First, they have not been able to gain a clear advantage over intrusion detection systems in the degree of control they exercise over a system. The battle for control is evenly matched in the common scenario where attackers and defenders both occupy the operating system. If both attackers and defenders run at the most-privileged hardware level (kernel mode), then neither has a fundamental advantage over the other; whichever side better understands and anticipates the design and actions of the other will win.

Second, current rootkits are faced with a fundamental tradeoff between functionality and invisibility. Powerful, general-purpose malware leaves more traces of its activity than simple, single-purpose malware. E.g., a web server used for phishing leaves numerous signs of its presence, including open network ports, extra files and processes, and a large memory footprint.

Our project, which is called SubVirt, shows how attackers can use virtual-machine technology to address the limitations of current malware and rootkits. We show how attackers can install a virtual-machine monitor (VMM) underneath an existing operating system and use that VMM to host arbitrary malicious software. The resulting malware, which we call a virtual-machine based rootkit (VMBR), exercises qualitatively more control than current malware, supports general-purpose functionality, yet can completely hide all its state and activity from intrusion detection systems running in the target operating system and applications.

This paper explores the design and implementation of virtual-machine based rootkits. We demonstrate that a VMBR can be implemented on commodity hardware and can be used to implement a wide range of malicious services. We show that, once installed, a VMBR is difficult to detect or remove. We implement proof-of-concept VMBRs on two platforms (Linux/VMware and Windows/VirtualPC) and write malicious services such as a keystroke sniffer, a phishing web server, a tool that searches a user's file system for sensitive data, and a detection countermeasure which defeats a common VMM detection technique. Finally, we discuss how to detect and defend against the threat posed by VMBRs and we implement a defense strategy suitable for protecting systems against this threat.

2. Virtual machines

This section reviews the technology of virtual machines and discusses why they provide a powerful platform for building malware.

A virtual-machine monitor (VMM) manages the resources of the underlying hardware and provides an abstraction of one or more virtual machines [20]. Each virtual machine can run a complete operating system and its applications. Figure 1 shows the architecture used by two modern VMMs (VMware and VirtualPC) [1]. Software running within a virtual machine is called *guest* software (i.e., guest operating systems and guest applications). All guest software (including the guest OS) runs in user mode; only the VMM runs in the most privileged level (kernel mode). The host OS in Figure 1 is used to provide portable access to a wide variety of I/O devices [44].

VMMs export hardware-level abstractions to guest software using emulated hardware. The guest OS interacts with the virtual hardware in the same manner as it would with real hardware (e.g., `in/out` instructions, DMA), and these interactions are trapped by the VMM and emulated in software. This emulation allows the guest OS to run without modification while maintaining control over the system at the VMM layer.

A VMM can support multiple OSes on one computer by multiplexing that computer's hardware and providing the illusion of multiple, distinct virtual computers, each of which can run a separate operating system and its applications. The VMM isolates all resources of each virtual computer through redirection. For example, the VMM can map two virtual disks to different sectors of a shared physical disk, and the VMM can map the physical memory space of each virtual machine to different pages in the real machine's memory.

In addition to multiplexing a computer's hardware, VMMs also provide a powerful platform for adding ser-

[1]The ideas in this paper apply equally well to the other architectures used to build VMMs, which are called Type I and Type II [19].

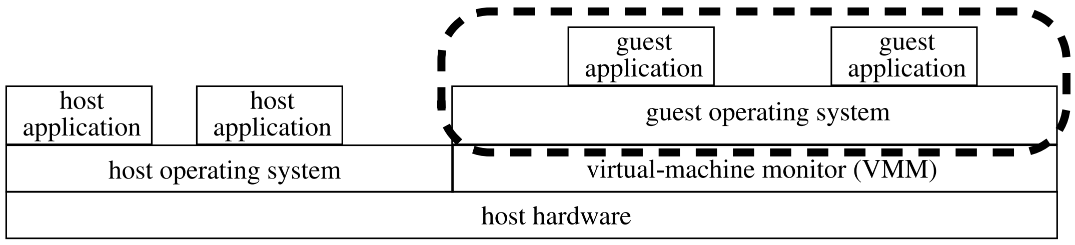

Figure 1. This figure shows a common structure used in today's virtual machine monitors. The VMM provides the abstraction of a virtual machine (dashed lines), each of which can run a complete *guest operating system* and a set of *guest applications*. The *host operating system* and its *host applications* are used to provide convenient access to I/O devices and to run VM services.

vices to an existing system. For example, VMMs have been used to debug operating systems and system configurations [30, 49], migrate live machines [40], detect or prevent intrusions [18, 27, 8], and attest for code integrity [17]. These *VM services* are typically implemented outside the guest they are serving in order to avoid perturbing the guest.

One problem faced by VM services is the difficulty in understanding the states and events inside the guest they are serving; VM services operate at a different level of abstraction from guest software. Software running outside of a virtual machine views low-level virtual-machine state such as disk blocks, network packets, and memory. Software inside the virtual machine interprets this state as high-level abstractions such as files, TCP connections, and variables. This gap between the VMM's view of data/events and guest software's view of data/events is called the semantic gap [13].

Virtual-machine introspection (VMI) [18, 27] describes a family of techniques that enables a VM service to understand and modify states and events within the guest. VMI translates variables and guest memory addresses by reading the guest OS and applications' symbol tables and page tables. VMI uses hardware or software breakpoints to enable a VM service to gain control at specific instruction addresses. Finally, VMI allows a VM service to invoke guest OS or application code. Invoking guest OS code allows the VM service to leverage existing, complex guest code to carry out general-purpose functionality such as reading a guest file from the file cache/disk system. VM services can protect themselves from guest code by disallowing external I/O. They can protect the guest data from perturbation by checkpointing it before changing its state and rolling the guest back later.

A virtual-machine monitor is a powerful platform for malware. A VMBR moves the targeted system into a virtual machine then runs malware in the VMM or in a second virtual machine. The targeted system sees little to no difference in its memory space, disk space, or execution (depending on how completely the machine is virtualized). The VMM also isolates the malware's state and events completely from those of the target system, so software in the target system cannot see or modify the malicious software. At the same time, the VMM can see all state and events in the target system, such as keystrokes, network packets, disk state, and memory state. A VMBR can observe and modify these states and events—without its own actions being observed—because it completely controls the virtual hardware presented to the operating system and applications. Finally, a VMBR provides a convenient platform for developing malicious services. A malicious service can benefit from all the conveniences of running in a separate, general-purpose operating system while remaining invisible to all intrusion detection software running in the targeted system. In addition, a malicious service can use virtual-machine introspection to understand the events and states taking place in the targeted system.

3. Virtual-machine based rootkit design and implementation

In this section, we discuss the design and implementation of a VMBR. Section 3.1 describes how a VMBR is installed on an existing system. Section 3.2 describes the techniques VMBRs use to implement malicious services, and Section 3.3 discusses the example malicious services we implemented. Section 3.4 explains how VMBRs maintain control over the system.

To explore this threat, we implemented two proof-of-concept VMBRs for the x86 platform using Virtual PC and VMware Workstation VMMs. Our proof-of-concept VMBRs both use the VMM architecture in Figure 1, which leverages a host OS to access the underlying hardware devices. The Virtual PC VMBR uses a minimized version of Windows XP [35] for the host OS and the VMware VMBR uses Gentoo Linux. To implement the proof-of-concept VMBRs, we modify the host Windows XP kernel, Virtual PC, and the host Linux kernel. We did not have source code for VMware, but our modifications to the host Linux kernel were sufficient to support our proof-of-concept VMware-based VMBR.

3.1. Installation

In the overall structure of a VMBR, a VMBR runs beneath the existing (target) operating system and its applications (Figure 2). To accomplish this, a VMBR must insert itself beneath the target operating system and run the target OS as a guest. To insert itself beneath an existing system, a VMBR must manipulate the system boot sequence to ensure that the VMBR loads before the target operating system and applications. After the VMBR loads, it boots the target OS using the VMM. As a result, the target OS runs normally, but the VMBR sits silently beneath it.

To install a VMBR on a computer, an attacker must first gain access to the system with sufficient privileges to modify the system boot sequence. There are numerous ways an attacker can attain this privilege level. For example, an attacker could exploit a remote vulnerability, fool a user into installing malicious software, bribe an OEM or vendor, or corrupt a bootable CD-ROM or DVD image present on a peer-to-peer network. On many systems, an attacker who attains root or Administrator privileges can manipulate the system boot sequence. On other systems, an attacker must execute code in kernel mode to manipulate the boot sequence. We assume the attacker can run arbitrary code on the target system with root or Administrator privileges and can install kernel modules if needed.

After the attacker gains root privileges, he or she must install the VMBR's state on persistent storage. The most convenient form of persistent storage suitable for VMBR state is the disk. An attacker can either use the target OS to allocate disk blocks (e.g., through the file system) or can parse on-disk structures to find unused blocks. When the target system is Windows XP, we store the VMBR state in the beginning of the first active disk partition. We relocate the data that was in these disk blocks to unused blocks elsewhere on the disk. When the target system is Linux, we disable swapping and use the swap partition to store persistent VMBR state. Both these installation procedures leave most of the target's data in its original location on disk.

The next step in installing a VMBR is to modify the system's boot sequence to ensure our VMBR loads before the target OS. The most convenient way for a VMBR to manipulate the system's boot sequence is to modify the boot records on the primary hard disk. Many current anti-malware applications detect modifications to the hard disk's boot blocks. Our implementation attempts to avoid this type of detection by manipulating the boot blocks during the final stages of shutdown, after most processes and kernel subsystems have exited.

When targeting Windows XP systems, we use a kernel module which registers a `LastChanceShutdown Notification` event handler that is invoked late in the shutdown sequence, after the file systems have been flushed and most processes have exited. When Windows invokes our event handler, our kernel module copies the VMBR boot code into the disk's active partition, which will cause the system to load the VMBR at the next system boot. Since our attack code runs within the OS, we have enough control over the system to avoid anti-malware software, even if it runs during the shutdown process. For example, we use the low-level disk driver to copy our VMBR boot code. Using the low-level disk driver bypasses the file system layer, which is where many anti-malware applications run. Furthermore, we interpose on the low-level disk controller's `write` function to ensure that only our VMBR is allowed to store disk blocks once installation begins.

When targeting Linux systems, we modify the boot sequence using user-mode code. We modify the shutdown scripts so that our installation code runs after all processes have been killed but before the system shuts down. We overwrite the disk master boot record using the Linux hard-drive block-device so that our VMBR loads at system boot instead of the target OS.

After installation, the target system's disk space is contained in a virtual disk. After rebooting, the VMM translates the target's virtual disk accesses to the corresponding location on the physical disk. To implement our disk redirection support for our Virtual PC-based VMBR, we modified the VMM's disk virtualization module. For our VMware-based VMBR, we modified the host Linux's hard-drive block device.

3.2. Malicious services

After a VMBR is installed, it can run malicious services. This section describes the techniques VMBRs use to implement various types of malicious services.

Traditional malware often trades-off ease of implementation against ability to avoid detection. Tradi-

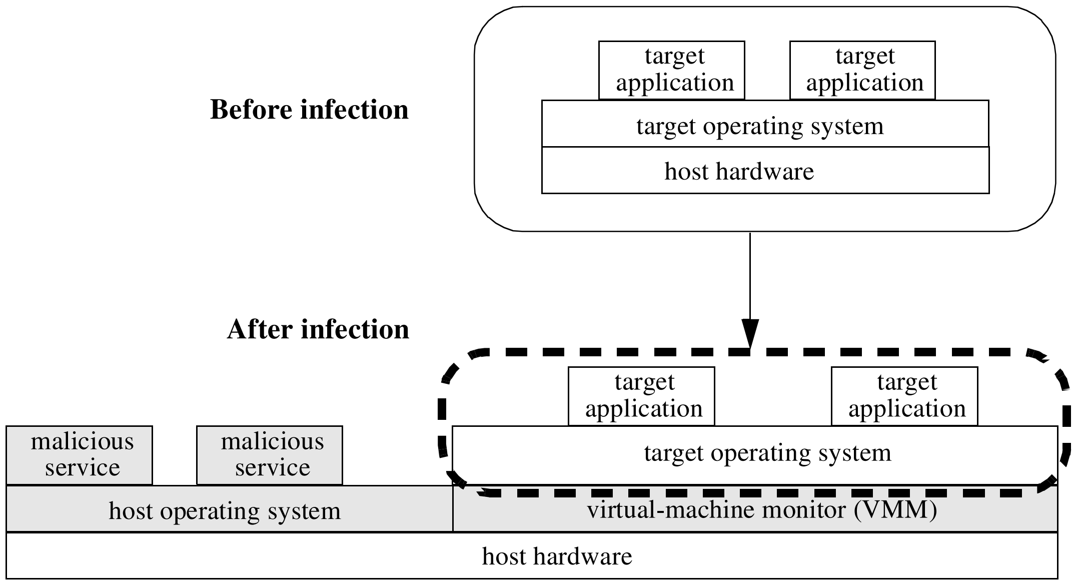

Figure 2. This figure shows how an existing target system can be moved to run inside a virtual machine provided by a virtual-machine monitor. The grey portions of the figure show the components of the VMBR.

tional user-mode malware that runs within the target OS tends to be easy to implement because malware authors can use any programming language to write these malicious services. Also, user-mode malware has access to all libraries and OS-level resources which makes it easy to provide a rich set of functionality. However, user-mode malware can be detected by security software running within the target OS because all malicious states and events are visible to the target operating system.

VMBRs use a separate *attack* OS to deploy malware that is invisible from the perspective of the target OS but is still easy to implement. None of the states or events of the attack OS are visible from within the target OS, so any code running within an attack OS is effectively invisible. The ability to run invisible malicious services in an attack OS gives intruders the freedom to use user-mode code with less fear of detection.

We classify malicious services into three categories: those that need not interact with the target system at all, those that observe information about the target system, and those that intentionally perturb the execution of the target system. In the remainder of this section, we discuss how VMBRs support each class of service.

The first class of malicious service does not communicate with the target system. Examples of such services are spam relays, distributed denial-of-service zombies, and phishing web servers. A VMBR supports these services by allowing them to run in the attack OS. This provides the convenience of user-mode execution without exposing the malicious service to the target OS.

The second class of malicious service observes data or events from the target system. VMBRs enable stealthy logging of hardware-level data (e.g., keystrokes, network packets) by modifying the VMM's device emulation software. This modification does not affect the virtual devices presented to the target OS. For example, a VMBR can log all network packets by modifying the VMM's emulated network card. These modifications are invisible to the target OS because the interface to the network card does not change, but the VMBR can still record all network packets.

VMBRs can use virtual-machine introspection to help observe and understand the software-level abstractions in the target OS and applications. Virtual-machine introspection enables malicious services to trap the execution of the target OS or applications at arbitrary instructions. When these traps occur, a malicious service can use use virtual machine introspection to reconstruct data and abstractions from the target system. For example, if a target application uses an encrypted socket, attackers can use virtual-machine introspection to trap all SSL socket `write` calls and log the clear-text data before it is encrypted. This logging is transparent to the target OS and applications since the malicious code runs outside of the target and also because virtual-machine introspection does not perturb the state of the target system.

The third class of malicious service deliberately modifies the execution of the target system. For example, a malicious service could modify network communication, delete e-mail messages, or change the execution of a target application. A VMBR can customize the VMM's device emulation layer to modify hardware-level data. A VMBR can also modify data or execution within the target through virtual-machine introspection.

3.3. Example malicious services

Using our proof-of-concept VMBRs, we developed four malicious services that represent a range of services a writer of malicious software may want to deploy. We implemented a phishing web server, a keystroke logger, a service that scans the target file system looking for sensitive files, and a defense countermeasure that defeats a current virtual-machine detector. To develop these services, we use the host OS as our attack OS (Figure 2). For some services, we also modify the VMM.

Using our VMware-based VMBR, we developed a phishing web server which represents malware that has no interactions with the target OS. Phishing web servers are used to deploy web sites that look like legitimate businesses and fool users into entering personal information like credit card numbers or passwords. Attackers commonly use compromised systems to deploy these malicious web sites. To implement our phishing site, we use a *thttpd* web server running within our attack OS. We modified our virtual networking settings so most network traffic is forwarded to the target, but any incoming TCP requests on port 8080 are forwarded to our phishing server. This phishing web server did not require any new code since we leveraged an existing web server and adjusted our existing virtual-network configuration settings, and yet we were able to run a full-blown web server within our VMBR environment that does not have any states or events which are visible within the target OS.

Using our Virtual PC-based VMBR, we implemented a keystroke logging malicious service. This represents malware that observes hardware-level data about the running target system. Attackers use keystroke loggers to obtain sensitive information, like passwords. To implement our keystroke logger, we modify the keyboard controller emulation module within the Virtual PC VMM so that all keystrokes are sent to a program in the attack OS before being forwarded up to the target OS. The program within the attack OS analyzes the keystrokes to extract passwords used to login to the target system. The amount of extra code needed to support keystroke logging is modest. Our modifications to the keyboard controller emulation module require 60 lines of code. Our analysis program takes 254 lines of code, which includes a GUI used to display results. The keystroke logger is transparent to the target OS because the target OS's interface to the virtual keyboard controller remains the same.

Using our VMware-based VMBR, we wrote a malicious service that scans the target OS file system looking for sensitive data. This type of malicious service represents malware which records software-level data about the target OS. When attackers break into a system, they sometimes search the file system looking for sensitive data like passwords and encryption keys. We implemented a malicious service that uses virtual-machine introspection to scan the target OS's file system to copy the password file (`/etc/shadow`) and search all user directories looking for private ssh keys (`*user_home*/.ssh/id_dsa`). Our scanning malicious service is a 24 line Perl script that runs within the attack OS. All malicious code runs outside of the target OS, and the scanning activities are invisible to the target system.

Using our Virtual PC-based VMBR, we implemented a defense countermeasure service. This represents malware that actively modifies the execution of the target OS. We implemented a countermeasure against an existing program that detects x86 virtual machines. The *redpill* [39] virtual-machine detection technique detects the presence of a VMM by using the `sidt` instruction. The `sidt` instruction reads the address of the processor's interrupt descriptor table. This address is different for an operating system running directly above hardware than for an operating system running above a VMM. VMMs emulate the `sidt` instruction when it is called from kernel-mode, but for performance reasons this instruction is not virtualized when called from user-mode. Thus, user-mode code can execute the `sidt` instruction to detect the presence of a VMM. To defeat this detection technique, we use virtual-machine introspection to emulate the `sidt` instruction when it is called by the *redpill* application. To emulate this instruction, we interpose on the Windows XP kernel function that maps executable files into the address space of a process. When this function executes, we determine if the *redpill* executable is being loaded. If it is, we set a breakpoint at the `sidt` instruction and emulate it when the target traps to the VMM. This countermeasure could be defeated by a program that generates the `sidt` instruction dynamically. Continuing the arms race, a more advanced countermeasure could trace the execution of guest software, using binary translation techniques, to detect and emulate all `sidt` instructions, but this might add overhead to the system which could be detected. Implementing the *redpill* detection countermeasure required adding 104 lines of code to the Virtual PC VMM.

3.4. Maintaining control

To avoid being removed, a VMBR must protect its state by maintaining control of the system. As long as the VMBR controls the system, it can thwart any attempt by the target to modify the VMBR's state. The VMBR's state is protected because the target system has access only to the virtual disk, not the physical disk.

The only time the VMBR loses control of the system is in the period of time after the system powers up until the VMBR starts. Any code that runs in this period can access the VMBR's state directly. The first code that runs in this period is the system BIOS. The system BIOS initializes devices and chooses which medium to boot from. In a typical scenario, the BIOS will boot the VMBR, after which the VMBR regains control of the system. However, if the BIOS boots a program on an alternative medium, that program can access the VMBR's state.

Because VMBRs lose control when the system is powered off, they may try to minimize the number of times full system power-off occurs. The events that typically cause power cycles are reboots and shutdowns. VMBRs handle reboots by restarting the virtual hardware rather than resetting the underlying physical hardware. By restarting the virtual hardware, VMBRs provide the illusion of resetting the underlying physical hardware without relinquishing control. Any alternative bootable medium used after a target reboot will run under the control of the VMBR.

In addition to handling target reboots, VMBRs can also emulate system shutdowns such that the system appears to shutdown, but the VMBR remains running on the system. We use ACPI sleep states [3] to emulate system shutdowns and to avoid system power-downs. ACPI sleep states are used to switch hardware into a low-power mode. This low-power mode includes spinning down hard disks, turning off fans, and placing the monitor into a power-saving mode. All of these actions make the computer appear to be powered off. Power is still applied to RAM, so the system can come out of ACPI sleep quickly with all memory state intact. When the user presses the power button to "power-up" the system, the computer comes out of the low-power sleep state and resumes the software that initiated the sleep. Our VMBR leverage this low-power mode to make the system appear to be shutdown; when the user "powers-up" the system by pressing the power button the VMBR resumes. If the user attempts to boot from an alternative medium at this point, it will run under the control of the VMBR. We implemented shutdown emulation for our VMware-based VMBR.

ACPI sleep states provide a fairly good illusion of system shutdown, but some systems have visible differences while in low-power mode. Specifically, some power LEDs behave differently while in low-power mode than when the system is shut down. In all of the systems we surveyed, the power LED was turned off when the system was shutdown. When in low-power mode, some LEDs turned off by default, some power LEDs could be turned off using BIOS functions, and others blinked or changed colors. Astute computer users might notice a difference in the power LED after an emulated shutdown, but average computer users probably would not. Furthermore, many computer users rarely shutdown their systems.

4. Evaluation

This section evaluates the impact of a VMBR on a system. We evaluate the disk space used by a VMBR, the time to install a VMBR, the effect of a VMBR on the time to boot the target OS, the impact of a VMBR as viewed by a user, and the effect of the memory space used by a VMBR.

All experiments for the VMware-based VMBR run on a Dell Optiplex Workstation with a 2.8 GHz Pentium 4 and 1 GB of RAM. All experiments for the Virtual PC-based VMBR run on a Compaq Deskpro EN with a 1 GHz Pentium 4 and 256 MB of RAM. Our VMware-based VMBR compromises a RedHat Enterprise Linux 4 target system, and our Virtual PC-based VMBR compromises a Windows XP target system.

We first measure the disk space required to install the VMBR. Our Virtual PC-based VMBR image is 106 MB compressed and occupies 251 MB of disk space when uncompressed. Our VMware-based VMBR image is 95 MB compressed and occupies 228 MB of disk space uncompressed. The compressed VMBR images take about 4 minutes to download on a 3 Mb/s cable modem connection and occupy only a small fraction of the total disk space present on modern systems. We made some effort to minimize the amount of persistent storage needed for both VMBRs, but we believe that the images could be made much smaller with additional effort. For example, `ttylinux` [5] is a Linux

	Installation	Target Boot Without VMBR	Target Boot After Emulated Reboot	Target Boot After Emulated Shutdown	Host Boot After Power-Off	Host Boot + Target Boot After Power-Off
VMware-Based VMBR (Linux Target)	24	53	74	96	52	145
Virtual PC-Based VMBR (Windows XP Target)	262	23	54	N/A	45	101

Table 1. VMBR installation and boot times (all times are in seconds). This table shows the installation and boot times for our Virtual PC-based and VMware-based VMBRs. As a reference point for our boot time measurements, we boot the system when it was not infected with a VMBR. We then measure the boot time after installing a VMBR for a number of different scenarios. All measurements are the average of three runs and have a variance of less than 3%, except for our Virtual PC-based VMBR install time which had a high variance because the system memory was almost completely used by the VMBR image. We did not implement shutdown emulation for the Virtual PC-based VMBR.

distribution that occupies only a few megabytes of disk space, yet still includes device support for a range of hardware.

We next measure the time it takes to install a VMBR and the effect of the VMBR on the time to boot a target system. Table 1 summarizes the results from our evaluation.

The installation measurements include the time it takes to uncompress the attack image, allocate disk blocks, store the attack files, and modify the system boot sequence. Installation time for the VMware-based VMBR is 24 seconds. Installation for the Virtual PC-based VMBR takes longer (262 seconds) because the hardware used for this test is much slower and has less memory. In addition, when installing a VMBR underneath Windows XP, we swap the contents of the disk blocks used to store the VMBR with those in the beginning of the Windows XP disk partition, and these extra disk reads/writes further lengthen the installation time.

We next measure boot time, which we define as the amount of time it takes for an OS to boot and reach an initial login prompt. Booting a target Linux system without a VMBR takes 53 seconds. After installing the VMware-based VMBR, booting the target system takes 74 seconds after a virtual reboot and 96 seconds after a virtual shutdown. It takes longer after a virtual shutdown than after a virtual reboot because the VMM must re-initialize the physical hardware after coming out of ACPI sleep. In the uncommon case that power is removed from the physical system, the host OS and VMM must boot before loading the target Linux OS. The VMware-based VMBR takes 52 seconds to boot the host OS and load the VMM and another 93 seconds to boot the target Linux OS. We speculate that it takes longer to boot the target OS after full system power-down than after a virtual reboot because some performance optimizations within the VMware VMM take time to warm up.

Booting a target Windows XP system without a VMBR takes 23 seconds. After installing the Virtual PC-based VMBR, booting the target system takes 54 seconds after a virtual reboot. If power is removed from the physical system, the Virtual PC-based VMBR takes 45 seconds to boot the host OS and load the VMM and another 56 seconds to boot the target Windows XP OS.

Once the target system runs, a user will see few differences between a target OS running directly on hardware and a target OS running above a VMBR. Virtual machines have been shown to have little performance impact on the system [10], and the video resolution, user interface, and interactivity of the system are unaffected by the VMBR. To achieve this level of performance, we used specialized drivers within the target OS. Specifically, we installed Virtual PC Guest Additions [34] and VMware Guest Tools [4] as part of our VMBR installation. Despite using specialized guest drivers, our current proof-of-concept VMBRs use virtualized video cards which may not export the same functionality as the underlying physical video card. Thus, some high-end video applications, like 3D games or video editing applications, may experience degraded performance.

The physical memory allocated to the VMM and attack OS is a small percentage of the total memory on the system (roughly 3%) and thus has little performance impact on a target OS running above the VMBR. Our VMware-based VMBR presents the target OS with the same amount of memory as is present on the physical system, and it uses paging to compensate for any memory allocated to the attack OS and VMM. To measure the overhead of this paging mechanism, we use a benchmark that randomly touches pages within the target OS. To avoid content-based page sharing [47] between the attack OS and the target OS, we use different kernel versions for both, and we fill unused pages within the target OS with random data to avoid sharing *zero pages*[2]. We then randomly access pages within the target OS, using working set sizes ranging from 10 MB to the entire physical memory of the computer. Our tests found that the average access time was equal across all working set sizes; thus we conclude that paging target OS memory results in minimal performance impact for our VMware-base VMBR. Our Virtual PC-based VMBR reserves approximately 100 MB of physical memory for the attack OS and VMM and presents the target with less physical memory than is available on the system.

5. Defending against virtual-machine based rootkits

In this section, we explore techniques that can be used to detect the presence of a VMBR. VMBRs are fundamentally more difficult to detect than traditional malware because they virtualize the state seen by the target system and because an ideal VMBR modifies no state inside the target system. Nonetheless, a VMBR does leave signs of its presence that a determined intrusion detection system can observe. We classify the techniques that be used to detect a VMBR by whether the detection system is running below the VMBR, or whether the detection system is running above the VMBR (i.e., within the target system).

5.1. Security software below the VMBR

The best way to detect a VMBR (indeed, any malware) is to run at a layer that is not controlled by the VMBR. Detectors that run below the VMBR can see the state of the VMBR because their view of the system does not go through the VMBR's virtualization layer. Such detection software can read physical memory or disk and look for signatures or anomalies that indicate the presence of a VMBR, such as a modified boot sequence. Other low-level techniques such as secure boot [9] can ensure the integrity of the boot sequence and prevent a VMBR from gaining control before the target OS.

There are various ways to gain control below the VMBR. One way to gain control below the VMBR is to use secure hardware. Intel's LaGrande [25], AMD's platform for trustworthy computing [2], and Copilot [36] all propose hardware that can be used to develop and deploy low-layer security software that would run beneath a VMBR.

Another way to gain control below the VMBR is to boot from a safe medium such as a CD-ROM, USB drive or network boot server. This boot code can run on the system before the VMBR loads and can view the VMBR's quiescent disk state. Strider GhostBuster is an example of security software that uses a bootable CD-ROM to gain control before the OS boots [48]. As we point out in Section 3.4, VMBRs can avoid booting from safe medium by emulating system shutdowns and reboots, thus we recommend physically unplugging the machine before attempting to boot from a safe medium.

A third way to gain control below the VMBR is to use a secure VMM [17]. Like alternative bootable media, secure VMMs gain control of the system before the operating system boots. Running a secure VMM does not by itself stop a VMBR, as a VMBR can still insert itself between the VMM and the operating system. However, a secure VMM does retain control over the system as it runs and could easily add a check to stop a VMBR from modifying the boot sequence above the secure VMM.

Using a secure VMM, we implemented an enhanced version of secure boot which can prevent VMBR installations. The goal of our secure boot system is to provide attestation for existing boot components, such as the disk's master boot record, the file system's boot sector, and the OS's boot loader and also to allow legitimate updates of these components. All attempted updates of these components are verified (by checking the cryptographic signature) before they are allowed to complete. The verification code resides in a separate virtual machine, so it is protected from malicious code running within the guest. We implement this secure boot system using a Virtual PC VMM and a Windows XP guest operating system.

[2] *Zero pages* are pages within the OS that are completely zeroed out.

5.2. Security software above the VMBR

While running detection software below a VMBR is the best way to detect the VMBR, it is generally inconvenient to run software at such a low level. For example, booting from a safe medium can detect a VMBR, but such a reboot may only occur infrequently. In this section, we explore the question of whether software running above the VMBR (i.e., in the target system) can detect the presence of the VMBR.

There are several reasons why traditional techniques (e.g., having the target system scan memory or disk) may not be able to detect a VMBR. First, the target system is confined to its virtual machine. Since the VMBR stores its state outside that virtual machine, the target system will see no suspicious changes in its own state. Second, even if the target system did see something amiss, the VMBR could tamper with the execution of the detector and force it to report incorrect results.

With these limitations in mind, we analyze the possible perturbations that a detector could notice while running inside the virtual machine.

A fundamental perturbation of the VMBR is that it uses machine resources such as CPU time, memory and disk space, and possibly network bandwidth. A VMBR adds CPU overhead to trap and emulate privileged instructions, as well as to run any malicious services. These timing differences can be noticed by software running in the virtual machine by comparing the running time of benchmarks against wall-clock time [42, 28]. A VMBR can make the detector's task more difficult by slowing down the time returned by the system clock, but the detector can overcome this by using a clock that can be read without interference from the VMBR (e.g., the user's wristwatch).

In addition to CPU overhead, a VMBR uses memory and disk space. A detector can detect this overhead by running a program that requires the entire machine's memory or disk space. A VMBR can hide its space overhead in several ways. It can hide its memory overhead by mapping target pages with the same contents to the same physical page [47] or by paging some data to disk. While this extra paging activity may increase the running time of the program, the memory overhead of VMware was too small to measure in our tests. A VMBR can hide its disk overhead by compressing a portion of the disk, sharing disk blocks with the same contents, or pretending that the disk space it uses have bad sectors. We implemented the bad sector trick for Windows XP target systems; if the target system attempts to use these sectors, we emulate bad sectors by squashing writes and returning random data on reads.

Another type of perturbation is caused by the variety of I/O devices on the computer. Recall that a VMM typically virtualizes all I/O devices. However, virtualizing the I/O device without changing the target's view of the device requires one to understand the interface and semantics of each device. This is a formidable task, given the wide variety of I/O devices present on today's computing platforms. Instead, current VMMs emulate a small number of virtual devices (often with customized interfaces to improve performance [1, 34]). The target OS must then use the drivers for the emulated virtual devices. For example, when targeting Windows XP systems, we had to update `ntoskrn.exe` and `hal.sys` drivers to match the virtual hardware. Intrusion detectors can observe the change in devices that occurs when the VMBR is installed. A VMBR can simplify the task of emulating all devices by choosing to virtualize only those devices that are required to protect its state (e.g., the boot disk) or to carry out malicious activity (e.g., network card). The VMBR can allow the target OS to access remaining devices directly [32]. This solves the problem of virtualizing all devices, but on today's x86 platform it potentially exposes VMBR state because target OS drivers with direct access to devices can access all physical memory through DMA.

A final source of perturbations are caused by the imperfect virtualization of today's x86 processors. Sensitive, non-privileged instructions like `sidt` leak information about the VMM yet do not trap to the VMM [31, 37]. Commercial VMMs use binary translation to emulate these instructions, but only when they are executed at the processor's highest privilege level (ring 0). An intrusion detector can execute these instructions at a lower processor privilege level (rings 1 - 3) to detect a VMBR. In Section 3.3, we describe how we defend against a common virtual-machine detection technique (*redpill*) that uses such an instruction. However, defending against all invocations of this instruction would require binary translation for all instructions, not just ring 0 instructions, which may slow performance.

We expect future enhancements to the x86 platform to reduce these perturbations. Upcoming virtualization support from Intel [45] and AMD [7] will enable more efficient virtualization. These enhancements eliminate sensitive, non-privileged instructions so they cannot be used from the CPU's user-mode to detect the presence of a VMM. These enhancements may also accelerate transitions to and from the VMM, and this may reduce the need to run specialized guest drivers. In addition to virtualization support, I/O MMUs [6] provide the ability to protect physical memory pages from device DMA. This protection will enable a VMBR to emulate a subset of devices and allow the target OS to drive all other hardware directly. As a result, the devices visible to the target can be similar or identical to the underlying physical hardware.

6. Trends toward virtualization

In this section we discuss how the trend of hardware virtualization support and the trend of widespread VMM adoption might affect VMBRs.

First, future enhancements to the x86 hardware platform may make VMBRs more practical. As we point out in Section 5, limitations to the current x86 hardware platform force VMM designers to introduce perturbations into the system which make it straightforward to detect the presence of VMMs; future enhancements to the x86 platform should help reduce these perturbations. This trend toward hardware virtualization support might also make VMBRs more practical by reducing the amount of state needed to support VMBRs, reducing the amount of time needed to boot VMBRs, and allowing hardware devices to perform at full capacity. For example, if a VMBR gives the target OS video driver direct access to the underlying video card, the VMM could omit all video drivers and graphics subsystems (e.g., X), thus reducing the size of the VMBR payload. Also, the VMM would not have to initialize the video card since the target OS video driver will take the appropriate initialization steps, resulting in faster boot time. Finally, all features of the hardware video card would be available to the target system, so high-end video applications, like 3D games, would not suffer any performance degradation like they might if the VMBR used a virtualized video card.

Second, the trend towards widespread VMM use might help defenders detect and prevent VMBRs, but simply running a VMM will not avert the threat. It might be possible for VMBRs to reside in between the VMM and the target OS, thus still asserting full control over the target system. In fact, since the target OS is already expecting to run above a VMM, not physical hardware, inserting a VMBR in between a VMM and a target OS might be easier than inserting one between physical hardware and an operating system. This task might be easier because the interface to the virtual hardware will likely be much simpler than the interface to physical hardware. Fortunately, having a secure VMM in place gives defenders the advantage since it runs beneath VMBRs and has complete control over software running above. Also, a secure VMM is flexible enough to support the development and deployment new security services designed to thwart the VMBR threat.

7. Related work

Our work on VMBRs is related to four areas of prior work: layer-below attacks, using virtual machines to enhance security, detecting the presence of VMMs, and inserting new software layers into existing systems.

Layer-below attacks are a well-known technique for compromising the security of a system [21]. By accessing or controlling a layer of software that is below a defense mechanism, an attacker can avoid and disable that mechanism. For example, rootkits implemented in the operating system kernel are a common layer-below attack [22, 41]. While kernel-level rootkits can hide easily from user-level intrusion detection systems, kernel-level detectors like VICE [11] and Klister [38] can detect kernel-level rootkits because both run at the same privilege level and in the same memory space. Shadow Walker hides some of the memory footprint of a standard kernel-level rootkit by manipulating the page table, TLB data, and page fault handler [43]. While this resembles some of the techniques used in virtual machines, Shadow Walker only partially virtualizes the OS; for example, the OS still runs in kernel mode and can see and restore the modified page fault handler and page tables. In contrast, VMBRs operate below any kernel-level detector and thus can hide all their state and events from these detectors.

A second area of research related to VMBRs are projects that use virtual machines to enhance security. Researchers have used virtual machines to detect intrusions [18, 8, 27], isolate services [33], encrypt network traffic [33], analyze intrusions [14], and implement honeypots [26, 46]. These services leverage the advantages of virtual machines, in particular isolation and interface compatibility, to enhance the security of systems without requiring the cooperation or correctness of higher levels of software. VMBRs exploit these same features to protect and hide malicious software from operating system and application-level security services. Besides different goals, VMBRs differ from past VM-based services in how they are installed. VMBRs must install themselves beneath a system while preserving the persistent state of that system. In contrast, prior VM-based services have assumed that a system would be installed into a virtual machine rather than being migrated from a non-virtual machine.

A third area of related research are projects that detect the presence of VMMs by observing timing perturbations. Pioneer [42] attempts to detect the existence of a VMM by measuring the amount of time it takes to execute a specialized checksum over the code of its verification function. This checksum is designed to be optimal for the particular processor it runs on and includes sensitive, non-privileged instructions in its computation. Because sensitive, non-privileged instructions must be emulated by the VMM, this computation will be sub-optimal when run within a virtual machine. Pioneer uses a remote *dispatcher* machine to measure timing, so the timing does not rely on the local clock which is under the control of the VMBR.

Finally, VMBRs apply the general idea of inserting a new layer into an existing system. Other applications of this idea include virtual machines [15], stackable file systems [23], and preserve compatibility with existing systems by not modifying the network firewalls. A key feature of all these applications is that they preserve compatibility with existing systems by not modifying interfaces of the existing layers. For example, a virtual-machine monitor is inserted between the hardware and the operating system, and an operating system running on an ideal VMM sees the same hardware interface as an operating system running directly on the hardware.

8. Conclusions

Traditional malicious software is limited because it has no clear advantage over intrusion detection systems running within a target system's OS. In this paper, we demonstrated how attackers can gain a clear advantage over intrusion detection systems running in a target OS. We explored the design and implementation of VMBRs, which use VMMs to provide attackers with qualitatively more control over compromised systems. We showed how attackers can leverage this advantage to implement malicious services that are completely hidden from the target system and to enable easy development of general-purpose malicious services. We evaluated this new malware threat by implementing two proof-of-concept VMBRs. We used our proof-of-concept VMBRs to subvert Windows XP and Linux target systems and implemented four example malicious services.

In addition to evaluating the VMBR threat, we also explored techniques for detecting a VMBR. The best way to detect a VMBR is to control a layer beneath the VMBR, such as through bootable CD-ROMs, secure VMMs, or secure hardware. It might also be possible to detect a VMBR from software running above the VMM, but the high level of control VMBRs have over software running above turns this style of detection into an arms race where the VMBR has the fundamental advantage.

However, VMBRs have a number of disadvantages compared to traditional forms of malware. When compared to traditional forms of malware, VMBRs tend to have more state, be more difficult to install, require a reboot before they can run, and have more of an impact on the overall system. Although VMBRs do offer greater control over the compromised system, the cost of this higher level of control may not be justified for all malicious applications.

Despite these shortcomings, we believe that VMBRs are a viable and likely threat. Virtual-machine monitors are available from both the open-source community and commercial vendors. We built VMBRs based on two available virtual-machine monitors, including one for which source code was unavailable. On today's x86 systems, VMBRs are capable of running a target OS with few visual differences or performance effects that would alert the user to the presence of a VMBR. In fact, one of the authors accidentally used a machine which had been infected by our proof-of-concept VMBR without realizing that he was using a compromised system!

9. Acknowledgments

We thank the anonymous reviewers and our shepherd, Steve Gribble, for suggestions that helped us improve this paper. We thank Peter Biddle, Brandon Baker, and Eric Traut for providing valuable insight and discussions about this topic. We would also like to thank Rick Rashid, Ted Wobber, Butler Lampson, and Paul Barham for providing feedback on early versions of this paper. This research was supported in part by National Science Foundation grants CCR-0098229 and CCR-0219085, by ARDA grant NBCHC030104, by Intel Corporation, and by Microsoft.

References

[1] VMware Virtual Machine Technology. Technical report, VMware, Inc., September 2000.

[2] AMD platform for trustworthy computing. In *WinHEC*, September 2003.

[3] Advanced Configuration and Power Interface Specification, September 2004. http://www.acpi.info/DOWNLOADS/ACPIspec30.pdf.

[4] Installing VMware Tools, Novemeber 2005. http://www.vmware.com/support/ws5/doc/new_guest_tools_ws.html.

[5] ttylinux, 2006. http://www.minimalinux.org/ttylinux/showpage.php?pid=1.

[6] Advanced Micro Devices Inc. BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors, April 2004.

[7] Advanced Micro Devices Inc. AMD Pacifica Virtualization Technology, March 2005. http://enterprise.amd.com/downloadables/Pacifica.ppt.

[8] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the 2004 USENIX Security Symposium*, August 2005.

[9] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of 1997 IEEE Symposium on Computer Security and Privacy*, pages 65–71, 1997.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[11] J. Butler and G. Hoglund. VICE—Catch the Hookers! *Black Hat USA*, July 2004. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf.

[12] J. Butler, J. L. Undercoffer, and J. Pinkston. Hidden Processes: The Implication for Intrusion Detection. In *Proceedings of the 2003 Workshop on Information Assurance*, pages 116–12, June 2003.

[13] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.

[15] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996.

[16] Fuzen_Op. The FU rootkit. http://www.rootkit.com/project.php?id=12.

[17] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[18] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)*, February 2003.

[19] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, February 1973.

[20] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.

[21] D. Gollmann. *Computer Security, 2nd edition*. John Wiley and Sons, Inc., January 2006.

[22] Halflife. Abuse of the Linux Kernel for Fun and Profit. *Phrack*, 7(50), April 1997.

[23] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[24] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

[25] Intel Corp. LaGrande Technology Architectural Overview, 2003.

[26] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In *Proceedings of the 2004 USENIX Security Symposium*, August 2004.

[27] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP)*, pages 91–104, October 2005.

[28] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[29] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: a file system integrity checker. In *Proceedings of 1994 ACM Conference on Computer and Communications Security (CCS)*, pages 18–29, November 1994.

[30] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, pages 1–15, April 2005.

[31] K. Lawton. Running multiple operating systems concurrently on an IA32 PC using virtualization techniques, 1999. http://plex86.org/research/paper.txt.

[32] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[33] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*, 9(4), September 2000.

[34] Microsoft Corp. Microsoft Virtual PC 2004 Technical Overview, 2005. http://www.microsoft.com/windows/virtualpc/evaluation/techoverview.mspx.

[35] Microsoft Corp. Windows Preinstallation Environment Overview, 2005. http://www.microsoft.com/whdc/system/winpreinst/WindowsPE_over.mspx.

[36] J. Nick L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot–a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 2004 USENIX Security Symposium*, August 2004.

[37] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[38] J. Rutkowska. Detecting Windows Server Compromises. In *HivenCon Security Conference*, November 2003. http://www.invisiblethings.org/papers/hivercon03_joanna.ppt.

[39] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction, 2005. http://invisiblethings.org/papers/redpill.html.

[40] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, December 2002.

[41] sd and devik. Linux on-the-fly kernel patching without LKM. *Phrack*, 11(58), December 2001.

[42] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, 2005.

[43] S. Sparks and J. Butler. Shadow Walker: Raising The Bar For Windows Rootkit Detection. *Phrack*, 11(63), August 2005.

[44] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June 2001.

[45] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, May 2005.

[46] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the 2005 Symposium on Operating Systems Principles*, October 2005.

[47] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[48] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, June 2005.

[49] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.