Software Virtualization Based Rootkits

ABSTRACT

The most popular virtual execution technologies include pure emulator, API emulator and virtual machine, the typical representations of all these three technologies are Bochs, Wine and VMware respectively. The implementation of virtual machine could be divided into two categories by the virtualization extent employed, those are full virtualization and para-virtualization, the latter has come up and been used in most recent years; the “para” means making some assumptions about or doing some modifications on the Target OS. In addition, according to the difference of virtual machine monitor(VMM)’s structure, there are type I and type II VMM, and the type II could be referred to as the hosted VMM as well. As the virtualization technology continues developing and becomes more prevalent, the two biggest processor manufacturers in the world both have developed their new CPU technologies which support virtualization, the Intel VT-x and AMD Pacifica. However, all of the following discussion will still aim at the traditional x86 platform as that is by far the most widely used. The virtualizable processor architecture requires that all the sensitive instructions could be trapped when running with the degraded privilege

level. Unfortunately there exists a few non-virtualizable instructions that make x86 a non-strictly virtualizable architecture. In order to achieve the full virtualization of x86, some extremely complicated software techniques should be used to overcome these architectural limitations.

Being an excellent virtual machine software product, VMware has implemented the full virtualization of x86. We take the VMware Workstation version as an example to give a brief introduction about the architecture, basic working principle and some core technologies used of the type II VMM. As a hosted structure, there are two OS contexts, the host and the guest, so VMware must use the total context switch method to ensure necessary execution environment isolation. VMware uses privilege compression and dual execution modes (direct execution and binary translation) to virtualize the instruction execution part of x86 CPU. Direct execution with ring degradation would let a majority of common instructions run natively, and trap most of the sensitive ones at the same time. However, the binary translator would be used to deal with the non-virtualizable instructions. The execution mode selection will be made by the appropriate decision-making module which depends on the CPU's current mode, privilege level and segment state. It is worthy to point out that binary translation is a very complicated but useful software technique which is also referred to as dynamic recompilation. Besides the fact that the code generation is highly difficult and complex, it will also handle the

translation cache's synchronization and coherency. However, another technique, used by Plex86, that solves the same problem seems much simpler, which should be a good choice for implementing a light weight VMM. In addition to virtualizing the instruction execution part, VMware also applies some other techniques to virtualize the CPU's segmentation, paging and interrupt/exception system components, such as deferred shadow segmentation, shadow paging and interrupt/exception forwarding. As to device virtualization, VMware chooses the method of full emulation which provides a complete set of virtual devices that are totally different from the true hardware devices. Since the device emulation depends completely on how each given hardware device works, the implementation method varies.

At present, the virtualization technology has been widely used in many computer related fields, but the research and application of virtualization on the security area is still in an elementary state. Virtualization technology could be applied to developing a non-intrusive debugger, a honey pot that traps malicious programs, and VM Based Rootkits. The Microsoft research team working together with Michigan University has developed a VM Based Rootkit prototype named SubVirt. SubVirt could gain system control after the Target OS was infected and rebooted. SubVirt then made the Target OS run within the context of a VM, while SubVirt itself would run directly on the true hardware.

SubVirt could control the behavior of the target system via a VMM and provide some malicious services externally. However some inherent and obvious defects of SubVirt greatly reduced its practicality. Finally, as the most key part of the presentation, we will discuss the complete technical scheme of a novel VM Based Rootkit. The VMBR itself is sort of light weight VMM. After the VMBR is loaded, the VMBR will ensure the target system is still running by placing it into a rootkit created virtual execution environment. It then becomes very difficult for the victim to perceive the rootkits' presence or to find any virtualization footprint. Although this novel VMBR is just a proof of concept, it has at least achieved the coexisting transparently and perfectly with the target system.

KEY WORDS virtual machine, virtual machine monitor, virtual machine based rootkit

Contents

Chapter 1 Introduction

1.1 Popular "Virtual Execution" Techniques

1.1.1 **Pure Emulator**

Pure Emulator simulates all the necessary components which constitute a complete computer system by using a pure software method, including the processor and all hardware equipments, and its processor part is very similar to the way in which the compilation and execution environments of some interpreted high-level languages (Visual Basic, Java) work, which cycle for fetching instructions, decoding and executing them. Pure Emulator has the advantage of good portability, the target (simulated) platform and actual execution platform can be heterogeneous, its disadvantage is poorer performance. The typical representative is Bochs (http://www.bochs.com).

1.1.2 **API Emulator**

The function of API Emulator is to allow binary code of applications compiled for an operating system to run directly on another operating system, its working principle is to intercept the API invokations issued by applications and use the API of current operating system to simulate them. The merits of API Emulator are the application binary compatibility between operating systems, and better performance, but the drawbacks here are that this method relies on the internal implementations of specific operating systems, and poor generality and portability. The typical representative is Wine (http://www.winehq.com).

1.1.3 **Virtual Machine**

The initial goal for designing a Virtual Machine is to run several operating systems concurrently, its working principle is little bit similar to the above Pure

Emulator: it uses the pure emulation method for hardware devices, and direct execution plus emulation (binary translation) for processor part, that's namely one kind of Quasi-Emulation technology, which lets most of instructions run naturally on the hardware processor, but captures and emulates a small part of incompatible (sensitive) instructions by a Virtual Machine Monitor. The merit of Virtual Machine is the better performance, however the shortcoming is the poor portability, also it requires that the target (virtualized) platform to be the same as the actual execution platform. The typical representative is VMware (http://www.vmware.com), Plex86 (http://www.plex86.org, actually the new Plex86 belongs in Para-Virtualization, and can only supports Linux as its Guest OS).

1.2 Full Virtualization vs. Para-Virtualization

Full Virtualization means virtualizing all features of the processor and hardware devices. The typical representatives that use Full Virtualization include IBM VM/370, VMware etc.

Due to some restrictions of processor architecture, for example x86 includes some non-virtualizable instructions, it is impossible to achieve Full Virtualization without using binary translation system or hardware virtualization support. Under such circumstances, the so-called Para-Virtualization appears, Para-Virtualization means to do some appropriate modifications on the target operating system, such as getting rid of those non-virtualizable instructions and using customized I/O communication protocols, which then can reduce the complexity of VMM implementation and also enhance the performance. But Para-Virtualization technique also exists some shortages, for example it can not be used in a non-open-source operating system (such as Windows) , also modifications might be intrusive to target operating system and so on. Owing to this VMware company puts forward a kind of standard called Virtual Machine Interface (VMI) API, every operating system that follows this interface can not only run naturally, but can also run in a Para-VM, and without the need of kernel recompilation, this standard is in fact a kind of transparent Para-Virtualization. The typical representatives which use Para-Virtualization include Xen, Denali.

1.3 Virtual Machine Monitor (VMM)

1.3.1 **Type I VMM**

Type I VMM refers to that the VMM runs directly on hardware (Bare Machine), and actually it can be seen as an operating system with virtualization mechanism, which is responsible for its Virtual Machines (VMs) scheduling and resource allocation. The typical representative of Type I VMM is VMware ESX Server. The diagram of Type I VMM is shown as follows:

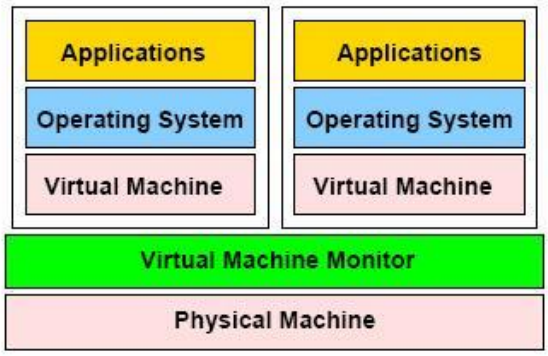

Diagram 1－1 Type I VMM

1.3.2 **Type II VMM**

Type II VMM is also referred to as Hosted VMM, the VMM itself runs as an application program in host operating system (Host OS), which manipulates hardware devices directly and is responsible for creating virtual execution environment and allocating resources for the guest operating system (Guest OS). By using Total Context Switch technique, Host OS and Guest OS each lives in their own isolated world, however the emulation of Guest OS device I/O operations is accomplished by some virtual machine software runs in Host world by making use of its facilities (API and Device Driver etc). The typical representative of Type II VMM is VMware Workstation. The diagram of Type II VMM is shown as follows:

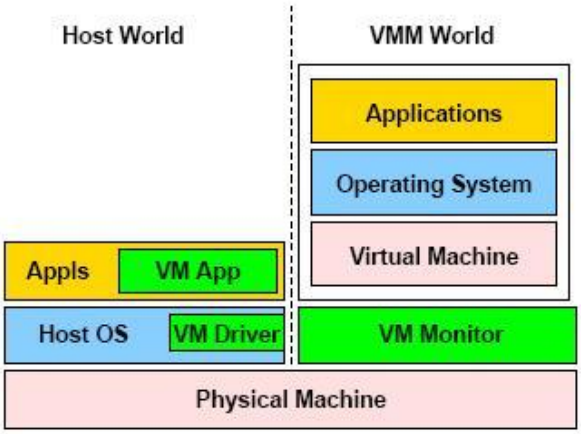

Diagram 1－2 Type II VMM

1.4 Hardware Based Virtualization Techniques

The two new processor techniques, Intel VT-x (Vanderpool & Silverdale) and AMD Pacifica, are enhancements to traditional processors to support software virtualization. Initially, they are mainly concentrated in handling those non-virtualizable instructions, and do not provide virtualization schemes for all processor components. The main feature is the introduction of a new CPU privileged root mode for the use of VMM itself, and non-root mode for Guest OS. In addition, they also provide some special VM instructions for switching and controlling VM (such as vmlaunch/vmrun, vmresume, vmexit/vmmcall, vmread, vmwrite etc.), and data structures used to save/ restore the state of Guest OS (VMCS and VMCB) as well. AMD Pacifica seems to go further on the road of hardware based virtualization, which can support SMM (System Management Mode) and SMI (System Management Interrupt), and use the NPT (Nested Page Table) and DEV (Device Exclusion Vector) technologies to achieve memory virtualization and DMA protection.

If readers are interested in these rising technologies, please find related documents for more technical information by yourself.

1.5 Virtualize x86

1.5.1 **The Standards of Virtualizable Processor Architecture**

R. Goldberg put forward 4 requirements of the third generation hardware architecture suitable for virtual machine in his doctoral dissertation << Architectural Principles for Virtual Computer Systems >> in 1972:

1. At least two processor privilege modes.
2. A method used by non-privileged program to invoke privileged system routines.
3. Memory relocation or protection mechanism, such as segmentation or paging system.
4. Asynchronous interrupt mechanism, which allows the communication between I/O system and CPU.

x86 meets R. Goldberg's requirements well:

1. 4 privilege modes (rings 0~3).
2. Interrupt Gate, Call Gate etc.
3. Segmentation and Paging.
4. Interrupts and Exceptions.

John Scott Robin and others gave further standards about virtualizable processor architecture in the paper << Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor >> :

1. The execution manner of non-privileged instructions are almost identical in two modes, the user mode and the privileged mode, no distinguishing instruction words or extra bits in instruction address part must be used in privileged mode by CPU.
2. Having protection mechanism or address translation system to isolate and protect the real machine from the virtual one (or between several virtual machines).
3. When the Virtual Machine tries to execute sensitive instructions, the Virtual Machine Monitor have to be notified automatically, and it must be able to emulate the results of those instructions.

The sensitive instructions include:

a. Instructions that attempt to modify or reference the VM operation mode or machine state.

b. Instructions that read or change the sensitive registers and/or specific memory regions, such as clock and interrupt register.

c. Instructions that access storage protection system, memory system or address relocation system, which will allow VM to access any address that beyond its own virtual memory.

d. All I/O instructions

x86 almost meets the requirements mentioned above:

1. No differences in execution manner of non-privileged instruction exist between privilege modes, however the execution results might be slight different, such as popf.

2. Segment and page based protection system.

3. By lowering the running privilege level of VM Operating System (VMOS), the exception handler of VMM can trap and emulate most of the sensitive instructions, which means most sensitive instructions are privileged. Unfortunately, however, there are some sensitive but non-privileged ones (violate 3b, 3c), which makes x86 a non-strictly virtualizable architecture.

1.5.2 **The Challenges On x86 Virtualization**

1.5.2.1 **Limitations From Hardware and Processor**

Hardware: usually be designed to be controlled exclusively by only one device driver, otherwise it may result in hardware state confusion and inconsistency.

x86 processor: its system features part are designed to be configured and used by only one operating system. In addition, there also exist some other problems as follow:

- Tight-coupled between some non-strictly related mechanisms: there exists a very close coupling between the privilege level and segment mechanisms, the RPL and DPL are contained respectively in segment selector and segment descriptor which are both manipulated directly by CPU. Thus the VMM has no chance to interpose timely and perform emulation and therefore may expose potentially to the VMOS the fact that its privilege being lowered.
- Hidden part of segment registers: All 6 segment registers contain hidden information, such as segment base address, length, and privilege level, and these concealed information will be flushed only upon segment reloading.

When segment descriptors in memory being modified but without a timely segment reloading, then appears the so-called segment inconsistent or irreversible problem. At this time an interrupt/exception occurred which traps into VMM will result in the total loss of the hidden information, then the VMM will not be able to emulate segment related instructions correctly, and also VMOS may not be restored running.

- Non-trapped sensitive instructions: such as popf, pushf, sgdt, sldt, sidt.

1.5.2.2 **Sensitive x86 Instructions List**

The following is the incomplete list of the sensitive instructions in x86 architecture, and most of them are privileged instructions:

- clts
- hlt
- in
- ins
- out
- outs
- lgdt
- lidt
- lldt
- lmsw
- ltr
- mov r32, CRx
- mov CRx, r32
- mov r32, DRx
- mov DRx, r32
- popf/popfd
- pushf/pushfd
- cli
- sti
- sgdt
- sidt
- sldt
- smsw

- str
- arpl
- verr
- verw
- lar
- lsl
- lds/les/lfs/lgs/lss
- mov r/m, Sreg
- mov Sreg, r/m
- push Sreg
- pop Sreg
- sysenter
- sysexit
- invlpg
- invd
- wbinvd
- rdmsr
- wrmsr
- rdpmc
- rdtsc

Because of the restriction of paper length, for more details readers have to make reference to Intel instruction manual by themselves.

1.5.2.3 **Non-trapped Sensitive x86 Instructions List**

Most of the non-trapped sensitive instructions are segment or flags bits related instructions:

- lar/lsl/verr/verw: the results of execution will depend on CPL and RPL value of target segment selector, and lar/lsl might reveal the changes of segment access right and limit.
- sgdt/sidt/sldt/str: reveal the changes of GDT/IDT base, length and LDT/TSS selector value.
- smsw: expose some critical flag bits in CR0, such as PE bit.
- popf/popfd: can modify some critical flag bits in EFLAGS, also the execution results will depend on current processor mode and privilege level

(CPL and IOPL), and can not be trapped in protected mode.

- pushf/pushfd: expose some critical flag bits in EFLAGS, such as IF, IOPL bit etc.
- mov r/m, Sreg: reveal the changes of RPL value in segment selectors.
- mov Sreg, r/m
- push Sreg: reveal the changes of RPL value in segment selectors.
- pop Sreg

In addition, I/O and control transfer instructions are also closely related to virtualization topic, so deserve a thorough discussion.

- in/ins/out/outs
- sysenter/sysexit
- call/jmp/int n/ret/iret

An in-depth analysis of these non-virtualizable instructions will be given in the following sections which deal with the complete technical scheme of VM Based Rootkit.

1.6 Related Concepts and Terminologies

- Host OS: the Host Operating System of Type II VMM, which takes charge of creating virtual execution environment and resource allocation for Guest OS.
- Guest OS: the Guest Operating System of Type II VMM (in narrow sense), which runs within the VM and is under the control of VMM.
- VM: Virtual Machine, the virtual execution environment that hosts the Guest OS.
- VMM: Virtual Machine Monitor, the software component used to achieve virtual machine hardware abstract.
- Hypervisor: the software that runs directly on the underlying hardware, with the responsibilities for lodging and managing the VMs, it usually refer in particular to Type I VMM.
- Non-virtualizable Instruction: those instructions existed in some processor architectures, which are either sensitive or behave differently under different privilege level, but both don’t incur traps.
- Segment Inconsistent Problem: also known as Segment Irreversible Problem,

which means that the current processor operation mode is different from the mode when segment being loaded, or the hidden information in segment register is not the same with the current values of segment descriptors in memory. Because the VMM must handle the interrupt/exception transparently, however the occurrence of interrupt/exception will result in the segments switch and then the flush of concealed contents in segment registers, thus if those segments are in the irreversible states to the moment, VMM will be unable to restore them.

1.7 VMware's Working Principle

The following picture depicts the whole structure of VMware Workstation:

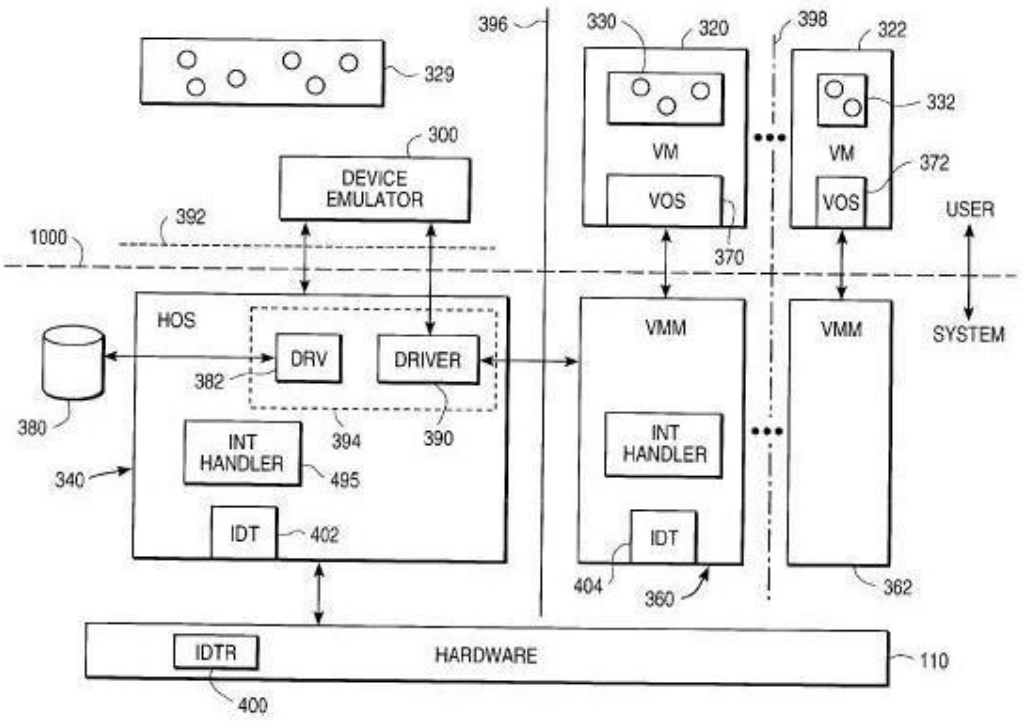

Diagram 1－3 The Whole Structure of VMware Workstation

Here I will not plan to spend a mass of words to describe the detailed working principle of VMware, which should be your own work. However there are some important points as follows you must be clear, which also help to understand how a VM Based Rootkit works:

1. Total Context Switch: Total Context Switch differs with ordinary Process Context Switch in that it will save and restore all context information of a given processor, which include address space state, general purpose registers set, floating registers, privileged registers (control registers, segment descriptors table register), interrupt/exception vectors etc. After switch, Host OS and Guest OS/VMM (in order to control and manage the execution of Guest OS and make the switching back possible, VMware maps its VMM module into Guest context, which is invisible and non-influential to Guest OS) each lives in an isolated and independent world, and there is no easy way to let the processor go back to another context's instruction path, in order to do this, VMware uses a "span page" in memory, which keeps the same and starts at the same linear address in both contexts. "Span page" contains a set of instruction sequence specific to a given processor architecture, which achieves the address space switching and allows the processor to continue executing its next instruction. Guest OS/VMM just occupies the VMware application's time quantum to run for a while in Guest world, and VMM must yield the control voluntarily and switch back to Host world according to the clock frequency programmed by Host OS, so that other applications in Host OS may get chances to be scheduled.

2. x86 Processor Virtualization: To virtualize x86 instruction execution, VMware basically uses two critical techniques: the privilege (ring) compression and the dual execution modes (direct execution and binary translation). After the running privilege being lowered, most privileged instructions issued by Guest OS will incur exceptions (#GP), then VMware will have chance to handle them with its own IDT, exception handler and privileged instruction emulation module. Because there exists a few number of non-virtualizable instructions in x86 architecture, the only using of direct execution seems not adequate, as a necessary complement of direct execution, the binary translator will be applied in some cases. The execution mode decision process will depend on current processor mode, privilege level and segment state comprehensively. VMware doesn't use binary translation under Guest user mode for performance consideration, which has become a leak used to detect VM presence by some malicious codes. As to other components and mechanisms of x86 processor, such as segmentation, paging, task, interrupt/exception etc, VMware also involves some sub-modules in its VMM to virtualize them, for example the MMU emulation module 516 is for virtualizing the x86 paging system by using a very complex but interesting shadow paging technique.

3. Hardware Devices Virtualization: The device emulation could be divided into two kinds: those without the need to interact with real physical devices will be emulated directly by VMM in Guest world, such as the access to state or latched data ports, on the other hand all device operations which need to access the real devices must be forwarded to device emulator software in Host world via a context switch, and emulated by using the Host OS's facilities. In addition, all hardware interrupts occurred in Guest/VMM context do not belong to Guest OS, under such case the VMM must make a switch back immediately and fake this interrupt in Host world as if it happens here just now.

Chapter 2 Implementation of a VM Based Rootkit

2.1 Previous Works

As far as I known, there do exist VM Based Rootkits by using AMD and Intel's processor virtualization enhancements (AMD Pacifica & Intel VT-x), which have been demonstrated in Black Hat USA this year, but Hardware Virtualization Based Rootkit is definitely not my concern here.

The Microsoft research team working together with Michigan University has developed a Software VM Based Rootkit prototype named SubVirt. SubVirt could gain system control after the Target OS was infected and rebooted. SubVirt then made the Target OS run within the context of a VM, while SubVirt itself would run directly on the true hardware. SubVirt could control the behavior of the target system via a VMM and provide some malicious services externally.

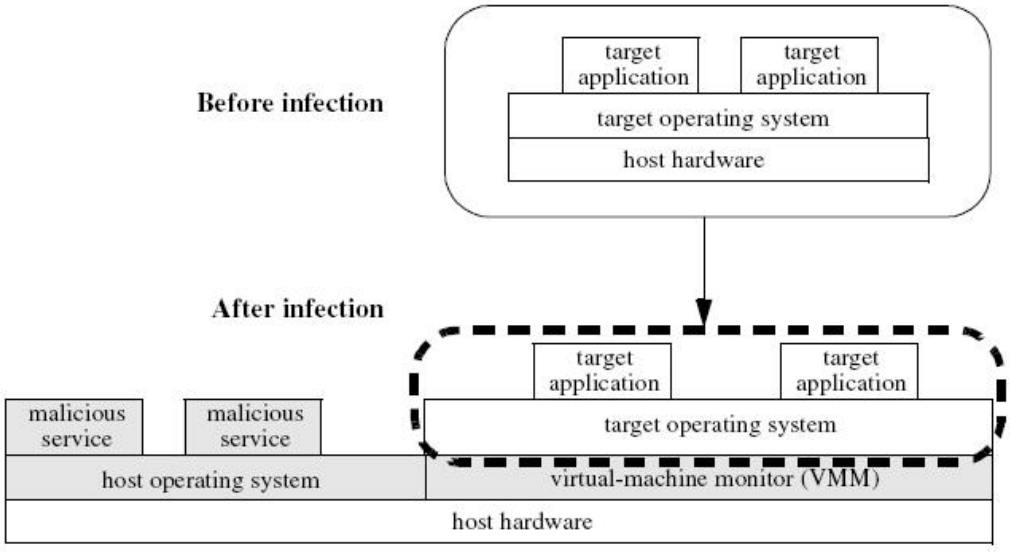

Diagram 2－1 The Working Principle of SubVirt

As a prototype of Software VM Based Rootkit, SubVirt has brought forward a

kind of very novel and challenging concept, however some inherent and obvious defects of it greatly reduce its practicality:

- Firstly, it depends on large commercial VM software (VMware with source code not opened or VPC) and a supporting Host OS (Linux), which make itself too big and not easy for spreading out.
- Secondly, it requires to modify hard disk MBR and insert itself to system Boot sequence, but the consistency of system Boot code could be ensured easily by using some TPM or secure Boot software.
- Finally, VM software will provide a totally different set of virtual devices, which requires re-installation of affected device drivers and thus allows the Target OS being infected to be aware of the device changes.

As the most important part, we are going to discuss the complete technical scheme of a novel VM Based Rootkit (VMBR) next, which can overcome all shortcomings of SubVirt listed above. Actually this VMBR itself is sort of a light weight VMM, that means no binary translation or dynamic scanning methods used, and no immoderate assumptions, requirements and modifications made on Target OS. After being loaded, the VMBR will ensure the target system is still running by placing it into a rootkit created virtual execution environment. It then becomes very difficult for the victim to perceive the rootkits' presence or to find any virtualization footprint. In addition, the VMBR may exercise various interferences and controls on Target OS at its will.

2.2 The Whole Structure

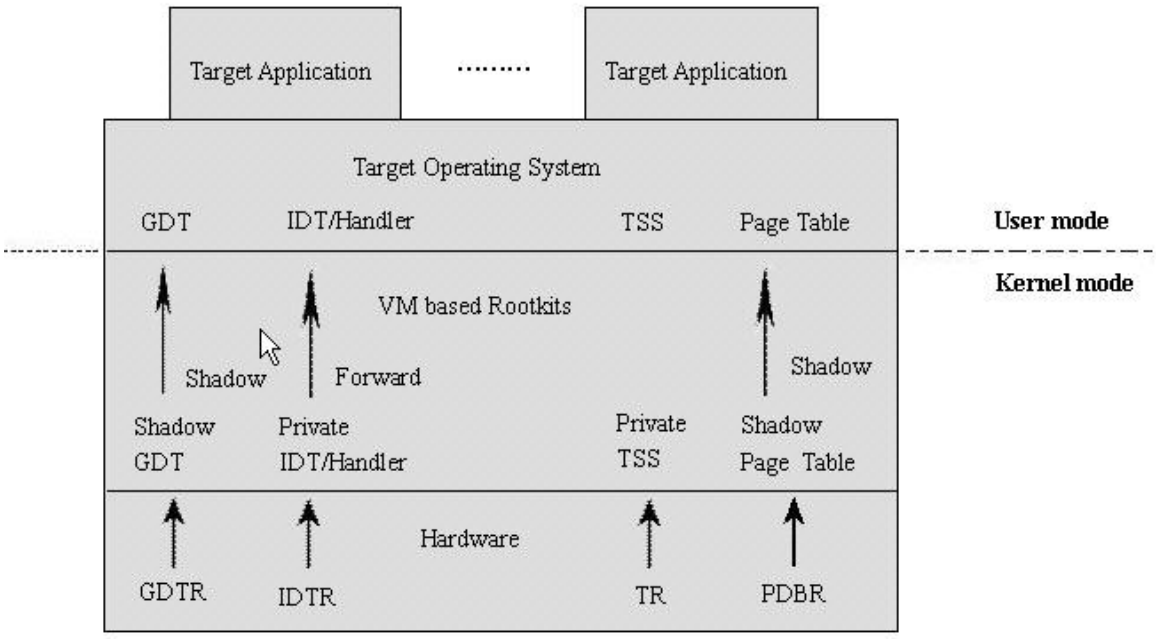

Diagram 2－2 The Whole Structure of VM Based Rootkit

There is no concepts of Host OS or Guest OS in this new VMBR scheme, thus no need for a Total Context Switch that is absolutely necessary in Type II VMM, the OS being controlled by VMBR is referred to as Target OS.

The VMBR can be loaded via a kernel mode driver under Target OS, after initialization process finished, the driver then becomes useless and can be unloaded as normal. The physical memory pages that occupied by VMBR can be 1) allocated by driver module by invoking the related kernel functions exported by Target OS and freed when driver being unload, 2) found in free page list by searching the PFN database maintained by Target OS, 3) reserved expediently by using a Boot option (/burnmemory) in system Boot configuration file (boot.ini). The linear address region where VMBR resides will be topmost 4M of the entire 4G space, an example of detailed address space layout is shown as follows:

```
    Windows 2000 build 2195 sp4 with no 4M page support patch in ntoskrnl.exe
    Total physical memory range 512M with /burnmemory=200M
    0x00000000 ~ 0x0009ffff      (0xa0000 640k)
    0x00100000 ~ 0x1ffeffff      (0x1feeffff 523,200k)
    Physical memory range occupied by VMBR
    0x18000000 ~ 0x18400000

    Linear memory range occupied by Target OS in topmost 4M
    0xffd00000 ~ 0xffd13fff (0x14000 80K)   ACPI
    0xffdf0000 ~ 0xffdfffff (0x10000 64K)   System (PCR &
KI_USER_SHARED_DATA)
    0xfffe0000 ~ 0xfffe0fff (0x1000 4K)     HAL (maybe Timer Counter)

    Linear memory range occupied by VMBR
    0xffe00000 ~ 0xffe18fff (0x19000 100K)  VMBR.sys
    0xffe19000 ~ 0xffe19fff (0x1000 4k)     Nexus
    0xffe1a000 ~ 0xffe1afff (0x1000 4k)     VM structure
    0xffe1b000 ~ 0xffe1bfff (0x1000 4k)     IDT
    0xffe1c000 ~ 0xffe1cfff (0x1000 4k)     GDT
    0xffe1d000 ~ 0xffe1dfff (0x1000 4k)     LDT
```

0xffe1e000 ~ 0xffe1efff (0x1000 4k)	TSS
0xffe1f000 ~ 0xffe1ffff (0x1000 4k)	IDT stubs
0xffe20000 ~ 0xffe20fff (0x1000 4k)	Log buffer
0xffe21000 ~ 0xffe21fff (0x1000 4k)	Monitor page directory
0xffe22000 ~ 0xffe22fff (0x1000 4k)	Nexus page table
0xffe23000 ~ 0xffe72fff (0x50000 320k)	Monitor/Target page table
0xffe73000 ~ 0xffe73fff (0x1000 4k)	Target CPU state
0xffe74000 ~ 0xffe74fff (0x1000 4k)	Code physical page
0xffe75000 ~ 0xffe75fff (0x1000 4k)	Temporary physical page0
0xffe76000 ~ 0xffe76fff (0x1000 4k)	Temporary physical page1
0xffe77000 ~ 0xffe77fff (0x1000 4k)	Transition page table
0xffe78000 ~ 0xffe78fff (0x1000 4k)	Page table linear address map

After being initialized successfully, the VMBR would have replaced the real information in hardware processor loaded by Target OS originally with its own ones to fully virtualize the continue running of Target OS, which include Shadow GDT, Shadow Page Table, Private TSS and IDT. Thereafter, a transparent intermediate layer (VMM) has been created and functions properly between the Target OS and real hardware, usually the processor will spend most of time on running codes of Target OS, while Rootkit can only get controls of execution for a while during the occurrence of hardware interrupts or exceptions, and must resume the running of Target OS quickly after various necessary processing.

2.3 Virtualize 4 Modes of x86

Because the Rootkit in this scheme will be loaded only after the Target OS has been booted successfully, we only discuss the virtualization of Protected Mode of x86 here, although Target OS may switch to Virtual 8086 Mode (a ntvdm launched) or System Management Mode (SMM) occasionally (a SMI asserted). To keep a light-weight implementation, we use completely direct execution to virtualize Protected Mode.

2.4 Virtualize x86 Instructions Execution

We use complete direct execution mode combined with privilege level (ring)

compression to virtualize the x86 instruction execution, that means all Target codes (unmodified) will run natively on hardware processor, only with the privilege level of Target kernel mode being lowered, therefore the execution of some privileged instructions would consequentially incur exceptions due to privilege level nonmatched, which then gives Rootkit good chances to emulate them. As to privilege level compression, there are two feasible schemes (because most modern operating system only use two rings, here we assume that the Target OS also uses just ring0 and ring3):

Scheme 1: Compress Target OS kernel mode (original ring0) to ring3. As could be seen from the following chapters, this scheme is not good for distinguishing the segment and page accessibility between Target OS user mode and kernel mode, and will make virtualization implementation become extremely complex.

Scheme 2: Compress Target OS kernel mode (original ring0) to ring1. This is a recommended scheme.

Just as we have talked before, x86 belongs to a non-strictly virtualizable architecture, so it is almost impossible to achieve a perfect virtualization by only using the direct execution mode, but we would disclose various potential deficiencies intensively in the next few chapters, and discuss how to try our best to reduce or remedy them.

2.5 Virtualize Processor State Information

Rootkit reserves a small region in its memory space to maintain Target OS's virtual processor state information, which include general-purpose register set, flag register, segment register set, control registers, debug registers, GDTR/LDTR/IDTR/TR, some MSRs and PIC/APIC. Some state information which are not used or modified by Rootkit can be left in hardware registers without any extra saving operation. The general-purpose registers, arithmetic flag bits in EFLAGS and segment registers should be saved each time when Target OS traps into Rootkit due to interrupts or exceptions, because the succedent running codes of Rootkit might change the contents in these registers. While the sensitive registers (include privileged flag bits in EFLAGS) should usually be saved when Target OS attempts to modify them, which inevitably traps into Rootkit for emulation, such as the issuing of a lgdt instruction by Target OS to reload GDTR. Rootkit must follow the real processor's processing logics strictly when it performs the emulation, meanwhile which also

involves frequent references to the virtual processor state information, for example when forwarding the hardware interrupts to Target OS, Rootkit should make its decisions according to the current states of Target OS's virtual interrupt flag (IF), virtual interrupt controller and virtual IDT.

2.5.1 **The Changes in Processor State**

- smsw is a non-privileged instruction, which stores the lowest 16 bits of CR0 register to specified general-purpose register or memory location. A Real Mode Target OS may use this instruction to find that it is actually running at Protected Mode.

  0：PE (Protection Enable)

  1：MP

  2：EM

  3：TS

  4：ET

  5：NE

- pushf/pushfd can not be trapped under Protected Mode (can be trapped under Virtual 8086 when IOPL != 3), thus may expose potentially some critical flag bits in EFLAGS, such as IF and IOPL, while VM and RF bits are always cleared in stack images. In order to prevent Target OS from enabling/disabling hardware interrupts by using sti/cli instruction pair, Rootkit would always set hardware IOPL flag bit to 0, at the same time the hardware IF flag bit to 0 to ensure itself always has chance to obtain controls. When running at Target OS user mode, IF and IOPL are generally set to 1 and 0 respectively (prevent user application from disabling interrupt), this is just the same as what Rootkit sets, so Target OS would see no change. However, when being at Target OS kernel mode, if Target OS disables interrupt (cli, may trapped. popf/popfd or iret, failed silently) or sets IOPL not equal to 0 (popf/popfd or iret, failed silently, Target OS will usually not do such a stupid thing) and follows a immediate check, it will surprisedly find the hardware IF and IOPL are still 1 and 0 in fact.

  8：TF Trap

  9：IF Interrupt Enable

  10：DF

12-13：IOPL I/O Privilege Level

14：NT

16：RF

17：VM V86 mode

18：AC

19：VIF

20：VIP

21：ID

- popf/popfd can not be trapped under Protected Mode (can be trapped under Virtual 8086 when IOPL != 3), and its execution results would heavily depend on processor current mode and privilege level (CPL and IOPL), under Protected Mode, IOPL can only be modified in ring0, while IF can be modified only when CPL <= IOPL. Usually Target OS user mode would not be affected, but its kernel mode may suffer from being not able to modify some flag bits (IOPL and IF) due to privilege compression, these instructions fail silently with no exceptions occurred, thus no emulation chance for Rootkit. In most cases Target OS will not use popf/popfd to modify IOPL and IF flags, but it may choose to use the iret (the required protection checking for IOPL and IF modification is similar to popf/popfd, but it can be used to change VM flag) or sti/cli instead.

2.6 Virtualize Segmentation

The privilege compression is the core of the whole virtualization scheme, moreover the privilege level of running code under x86 are decided by CPL (in invisible parts of current code and stack segment registers) or segment selector RPL (when interrupt/exception occurs, the RPL saved in stack represents the code privilege before control transfer), therefore the implementation of privilege compression actually requires to play tricks with segmentation of Target OS. Rootkit will not allow the Target OS's segment descriptor table to be used directly by hardware, instead it provides a shadow descriptor table (Shadow DT) of its own for real processor, the "Shadow" here means that Rootkit's DT is very closely associated with Target OS's DT, any descriptor change happened in Target OS's DT will be eventually reflected to Shadow DT with appropriate adjustment on descriptor's DPL when performing the synchronization action.

The Following discussion will not deal with the LDT and Conforming Code Segment, since few operating system uses them.

2.6.1 **The Structure of Shadow Descriptor Table and Shadowing Algorithm**

The Shadow DT contains some shadow descriptor entries (no more than 8192) from the beginning, which virtualize the descriptors in Target OS's DT, then six cached descriptor entries that correspond to six segment registers and are used to emulate the x86 segment caching mechanism, finally a number of entries reserved for the use by Rootkit to describe itself (Rootkit descriptors). The segment present bit (SPB) of shadow descriptors are initialized to be 0 (unshadowed state), while the SPB and DPL of cached descriptors or Rootkit descriptors are 1 and 0 respectively. In this way, any attempt to access the unshadowed (#NP), cached or Rootkit descriptor (#GP) by Target OS would inevitably incur exception, which gives Rootkit chance to emulate.

The Shadowing algorithm used when synchronizing descriptor pairs is shown as follows: For a Data/Code segment descriptor of Target OS, its DPL must be changed from 0 to 1 (or 3), and limit be truncated if overlapped with Rootkit space. For a Gate descriptor (such as call gate), its target code segment selector value must be modified to 0.

2.6.2 **Target Redefine DT**

lgdt instruction can be trapped, so Rootkit may desynchronize all shadow descriptors at this time, which could be done quickly by unmapping the pages where descriptors reside. After a new DT being defined, all previous traces installed on old DT will be invalidated, yet the new DT needs not to be traced immediately. We only install physical and linear traces on the Target OS pages which contain at least one synchronized descriptor. Before desynchronization, we must cache the six active shadow descriptors in advance, which means copying six shadow descriptors which have been loaded into six segment registers currently to their corresponding cached slots.

The introduction of Physical and Linear Trace will be arranged at the following chapter (Virtualize Paging).

2.6.3 **Target Modify Descriptors**

When Rootkit receives a notification that Target OS attempts to modify synchronized descriptors via physical trace, it may desynchronize all affected synchronized descriptors in Shadow DT, which could be done by setting SPB of these descriptors to 0 or simply unmapping the whole page if the page only contains desynchronized descriptors. If a page in Target OS's DT doesn't contain any synchronized descriptor longer, all physical and linear traces would totally be cancelled then. Again, we must firstly perform the caching before desynchronization as long as the modification involves to some active descriptors (currently used by one of six segment register). We can see that Rootkit doesn't synchronize affected descriptors at this point, but instead it defers the new descriptors synchronization to the moment when they are loaded subsequently, this deferred and asynchronous shadowing scheme is advantageous for the emulation of updating segment cache and synchronizing segment accessed bit.

2.6.4 **Target Remap/Unmap DT**

Rootkit may receive a notification when the page mapping of some regions in Target OS DT which contain synchronized descriptors has changed via linear trace, and the disposition method it uses is basically identical to descriptors modification above. Rootkit makes no difference between remap and unmap here, the distinction between these two actions would present only when Target OS loads descriptors from that regions subsequently.

2.6.5 **Target Load Segments**

The action that Target OS loads synchronized shadowed descriptors may go naturally, while loading unshadowed or desynchronized shadowed descriptors would lead to #PF or #NP, upon that Rootkit will carry out the descriptor pairs shadowing, segment register cache update and segment access bit synchronization. Here we assure that the first loading of a segment descriptor shouldn't happen naturally, and must be trapped and emulated by Rootkit.

2.6.6 **Inter-Segments Control Transfer**

Inter-segment control transfer involves code segment, stack segment (when privilege level changed) switching and a series of protection checking, and most importantly it must still go successfully under the condition of our privilege compression, which becomes a crucial and relatively complex problem of the whole segment virtualization scheme. We are going to discuss inter-segment control transfer topic from the following five transfer fashions.

2.6.6.1 **Task Switching**

Because most modern operating systems didn't use the hardware task mechanism, we also skip it for the moment.

2.6.6.2 **Call Gate**

The DPL (0) or target code segment selector value (0) field has been adjusted by Rootkit when performing the Target OS descriptors synchronization, thereby Rootkit may get a chance for interposition when Target OS tries to transfer through a call gate, the emulation logic is very similar to transferring with an interrupt/trap gate which will be talked later.

2.6.6.3 **Direct jmp and call/ret Between Segments**

Direct jmp and call/ret between segments don't involve privilege level switching, so privilege compression scheme would not affect this kind of transfer, and Rootkit may let them go naturally. However, what should be noticed is that if we adopt the first compression scheme that pushes Target OS kernel mode to ring3, then those direct transfers from Target user mode to its kernel mode (or in reverse) which are impossible at all under the normal condition would successfully happen by error.

2.6.6.4 **Interrupt/Exception**

Please refer to the chapter "virtualize interrupt/exception" for detailed analysis of inter-segment transfer with interrupt/trap gate.

2.6.6.5 **sysenter/sysexit**

In order to trap the actions that transfer with sysenter/sysexit instruction pairs by #GP exceptions, Rootkit may simple set the hardware model specified register (MSR) SYSENTER_CS_MSR (which defines the target segments of transfer directly or indirectly) to 0, and the emulation process is also similar to interrupt/trap gate.

2.6.7 **Synchronize Segment Accessed Bit**

When Target OS attempts to load a segment descriptor for the first time, Rootkit starts to synchronize this descriptor pair and update the accessed bit of it as well.

2.6.8 **The Changes in Segmentation System**

The Change of GDTR: sgdt instruction can not be trapped at any privilege level, so by which Target OS may find the change of GDT base address and limit. The solution to this problem is using dynamic scanning technique or mapping Shadow GDT at the Target expected location when running in Target user mode. Fortunately GDT in most operation systems would usually be created at a fixed memory address known in compilation phase, so not necessary to be obtained by sgdt instruction in runtime.

The Change of GDT Descriptors: When Target OS is running at ring3, Rootkit may enforce page protection (supervisor bit) on the pages of its Shadow GDT. While if the Target kernel mode has been compressed to ring1, Rootkit may truncate the overlapped portions of Target OS's code/data segment descriptors with the reduction of their limits to protect itself, thus any access (for instance, to Shadow GDT) beyond segment limit will incur #GP. Generally the DPL field of GDT descriptor in most operating systems is static, and never checked.

The Change of Segment Selector: The change of RPL field in Target OS current code segment selector, which is pushed on the stack when an interrupt/exception occurs, would make the Target OS's interrupt/exception handler impossible to distinguish the format of trap frame correctly, and expose the fact that the running privilege level has been lowered as well, fortunately this problem could be solved by Rootkit emulating (capture, and then forward) the interrupt/exception for Target OS. In a word, Rootkit should try its best to keep the Index and RPL fields in segment register selectors as Target OS expected while segment descriptors' DPL being lowered, but unfortunately the RPL of CS and SS segment register selectors must be

strictly identical to CPL.

Now we are about to discuss several sensitive (segment related) but non-trapped instructions.

- When Target kernel mode has been compressed to ring3, then Target user mode may have the accessibility to those segments that belong only to kernel mode originally by using lar/lsl/verr/verw or segment loading instructions.
- lar/lsl: Expose the change in segment limit and DPL of Target OS kernel mode segments.
- mov r/m, Sreg/push Sreg: Expose the change of RPL in visible part (segment selector) of Target OS kernel mode segment registers. Therefore, for some segment registers, we should try to keep the RPL unchanged when segments loading, but with their corresponding descriptors DPL lowered.
- Segment loading instructions (lds/les/lfs/lgs/lss /pop Sreg /mov Sreg, r/m): All instructions above may not be used to load CS, otherwise it leads to an #UD exception, so we put the discussion of CS loading at inter-segment control transfer. SS loading requires RPL=CPL=DPL, otherwise an #GP is generated, so it's impossible to keep RPL unchanged here. However, the loading of DS, ES, FS, GS would only need Max(RPL,CPL)<=DPL as long as the target segment is a data segment or non-conforming code segment, so in this case RPL can be kept unchanged. For example, the selector value of DS is 0x0020, but the descriptor DPL it referenced is 3.

2.6.9 **Segment Irreversibility**

At first, we try to solve the segment inconsistency (or irreversibility) problem by saving the active descriptors to corresponding cached descriptors (those cached ones are used to emulate the hidden part of six segment registers) before they are desynchronized, and updating the cached descriptors upon segment loading. However, the deferred segment shadowing scheme has guaranteed that the cached descriptors can be updated duly at the first loading of segment descriptors, so the caching operation before desynchronization seems redundant then. Furthermore, because usually this problem only happens during system Boot phase when switching from Real Mode to Protected Mode (haven't reloaded PM segments), we don't talk it much more.

2.7 Virtualize Paging

Because Rootkit “steals” some portions in both linear address space and physical memory of Target OS, and it also needs to use the access and protection bits in hardware page table to play all kinds of virtualization tricks, then Rootkit will not allow the Target OS’s page directory table/page table to be used directly by hardware, instead it provides a set of shadow table (Shadow Page Table) of its own for real processor, its working principle is almost identical to Shadow Descriptor Table mentioned previously, any PDE/PTE change happened in Target OS table will be eventually reflected to Shadow table with appropriate adjustments on entry’s access, protection attributes and physical address mapped when performing the shadowing.

2.7.1 **The Structure of Shadow Page Table and Shadowing Algorithm**

Rootkit spend a number of physical memory to emulate the page directory table/page table of Target OS, and this shadow table is initialized to be empty except for those entries that map itself, which can be done by marking most entries as invalid (valid bit cleared).

- Target kernel mode being compressed to ring1: The lowest 12 access bits are kept unchanged during shadowing, while the adjustments of physical address part should be decided according to different cases. If the physical address mapped has been in used by Rootkit itself, it will then allocate a new page to substitute the requested one, at the same time it also records the association relationship between these two physical addresses in case that Target OS requests the same physical page again. Another solution is that Rootkit abandons the physical page used and looks for another free page for itself, this solution can effectively protect against Target OS’s DMA operations, but it requires a lot of content movement (copy) at the same time, therefore now my Rootkit prototype only supports linear address relocation, but not physical relocation.
- Target all modes being compressed to ring3: The shadowing algorithm is similar as the above one, but there may exist an extra problem of how to distinguish the page accessibility between different modes of Target OS. A conceivable solution is that we split those tables that contain at least one system entry into two shadow tables, which correspond to Target OS user

mode (ring3) and system mode (ring0~2) respectively: For system mode, all entries in Target table will be shadowed as user in shadow table (for system mode use) except for those ones that assist virtualization. For user mode, we keep the original settings of Target table, or simply don't include those system entries in its shadow table (valid bits cleared). However, a better solution would be that Rootkit flushes totally the shadow table and restarts with a empty one upon every privilege switching, and creates a proper set of page table according to current privilege (CPL).

2.7.2 **The Procedure of Shadow Paging**

In initial state, all entries in shadow table are marked as invalid except for those entries that map Rootkit itself, to be more exactly, only a nearly empty shadow page directory table exists except for its last PDE, while dynamic shadow page tables are created subsequently. When #PF occurs during Target OS running (the P flag in error code pushed on stack is 0), Rootkit emulates the action of hardware MMU by traversing Target OS's page directory table/page table. If no valid mapping is found, then it ends up with forwarding and letting #PF to be handled by Target OS. When a valid mapping is found and also it is not caused by physical traces installed by Rootkit (such as the read traces when Target OS is running at ring1), then Rootkit begins shadowing, first shadows the corresponding PDE (if it is not be shadowed), next creates a new shadow page table (if not be created) and shadows the corresponding PTE with all other PTEs marked as invalid, then write protects the appropriate portion of Target OS table which contains shadowed entries, finally re-executes the faulting instruction (may be a processor instruction fetching). When #PF address (CR2) is overlapped with the Rootkit space, Rootkit may need to relocate itself to somewhere else (linear address space conflict and linear address relocation), but this would usually incur a #GP first due to truncated segment limit.

2.7.3 **Physical Trace**

Physical Tracing is a mechanism that Rootkit has the ability to install read, write, read/write traces on Target OS's physical memory pages, and be notified when accesses (read and/or write) to these pages occur. Because the trace is based on physical memory page, it requires Rootkit to maintain a set of reverse mapping, that is

mapping from physical address to linear address. Physical tracing is implemented by degrading the access bits of all entries that map to a specified physical page in Rootkit's shadow page table, for read/write trace mappings are degraded to invalid (not present), while for write trace mappings are degraded to read-only (CR0.WP bit must be also set when ring1 compression scheme used). Each time when a new mapping from linear address to physical address being inserted to shadow table, its access bits may also need to be degraded according to the physical traces installed on that physical page. Rootkit would record those physical page address which have been installed with traces, and the recorded addresses are Target OS requested addresses, but not the adjusted addresses in shadow table, choosing Target requested address is because that Target OS may generally use consecutive physical pages, which would simplify the codes of Rootkit that manage protected regions.

2.7.4 **Linear Trace**

Linear tracing is mechanism used by Rootkit to detect the mapping change (unmap or remap) of a given linear region range of Target OS. Rootkit achieves the linear trace by detecting the changes of Target OS' page table and trapping the MMU instruction that loads PDBR, while the changes of Target OS' page table can be monitored by installing physical traces (write trace) on relevant physical pages, where the entries in Target OS page table that map the given linear region reside.

2.7.5 **Target Store PDBR**

All instructions that store PDBR (CR3) can be trapped, Rootkit then emulates them so that Target OS may not perceive the change of physical address of page directory table. Usually, operating system would map the page directory table (and each page table) at a fixed linear address for a convenient memory management, for example in Windows 2000, the page directory table of any process is arranged to start at 0xc0300000.

2.7.6 **Target Load PDBR**

Here we don't discuss the context switch via hardware task mechanism at the moment. All instructions that load PDBR (CR3) can be trapped, Rootkit then may

update the entire shadow table at this time. What should be noticed is that all hardware TLB entries must be flushed except for those used to map Rootkit itself after table update.

- Synchronous Scheme (not recommended): Converts and inserts each entry in Target OS newly loaded page directory table/page table to shadow table, and then installs physical traces (write trace) on those physical pages where Target OS table reside after the shadowing finished.
- Asynchronous Scheme: Empties entirely the shadow table and starts anew, then cancels all the traces aimed at Target OS page directory table/page table as well. Although all physical traces installed previously are not in existence and don't function well anymore at present, they can be restored gradually along with mappings being inserted dynamically.

2.7.7 **Target Modify PDE/PTE**

Because of the write traces installed by Rootkit, any attempt to modify the shadowed entries by Target OS would incur a #PF, then Rootkit handles it accordingly, and finally it flushes the corresponding TLB entries.

- Synchronous Scheme (not recommended): The write trace is base on the granularity of a page size, therefore #PF handler should decide all affected entries according to the target address of write operation. Those entries affected must be shadowed again.
- Asynchronous Scheme: All affected entries would be unshadowed (set to a non-present state as default), while the actual shadow actions of these entries would be deferred until Target OS accesses the addresses mapped by them later.

2.7.8 **Target Flush TLB**

The TLB flush operations of Target OS may include:

- Use invlpg instruction to flush a TLB entry correspond to a given linear page that is specified with instruction operand.
- Reload CR3 register to flush all non-global TLB entries.
- Modify any paging flag (PE, PG in CR0, and PGE, PSE, PAE in CR4) to

flush all TLB entries.

We don't discuss the third case here, and the second case has also been analyzed. As to the first case, the processing of TLB flush with invlpg instruction is actually very similar to the modification of PDE/PTE discussed above: first Rootkit unshadows the shadow entry that maps the desired page specified by instruction operand, and then flushes the corresponding hardware TLB entry.

2.7.9 **Maintain The Accessed/Dirty Bits**

Processor will update the Accessed (PDE) and Dirty bits automatically so that operating system may get feedbacks of its page usage, therefore Rootkit should correctly emulate the Accessed/Dirty bits in Target OS page directory table/page table.

- Scheme 1: When Target OS accesses its own page table or Rootkit empties a set of previous virtual page table, Rootkit may update the A&D bits to Target table timely. In order to interpose when Target OS accesses its own table, Rootkit may impose read traces on the physical pages of Target table, which would not incurs #PF too frequently because Target table is not used directly by hardware processor.
- Scheme 2: For asynchronous shadowing scheme, because each entry will be shadowed when it is first accessed, then Rootkit may set it to Accessed (in both shadow table and Target table) and temporary read-only (in shadow table) at this time. When relevant pages mapped being written for the first time, shadow entry (in shadow table) will be set to Dirty automatically by hardware processor, then Rootkit only needs to set shadowed entry (in Target table) to Dirty also and restore the access bits of shadow entry according to that of the current shadowed entry. Because here we set shadow entry to be temporary read-only factitiously, then the #PF handler must depend on the real state of shadowed entry to decide whether it is a genuine #PF or an extra one we brought in.

2.7.10 **Handle the Page Fault**

When #PF occurs during Target OS running (the error code pushed on stack call tell us the reason for fault), Rootkit emulates the action of hardware MMU by traversing Target OS's page directory table/page table. If no valid mapping is found,

then it ends up with forwarding and letting #PF to be handled by Target OS. When a valid mapping is found and also it is caused by physical traces installed by Rootkit (known from the fault address in CR2), then Rootkit disables the trace installed on this mapping temporarily, and finally restores it to degradation state after single stepping the faulting instruction. What should be noticed is that Rootkit should use invlpg instruction to flush TLB entry for this mapping after restoration, otherwise the trace might not function well any longer.

2.7.11 **Linear Address Space Confliction**

When Target OS accesses (via #PF before shadowing) or maps (via linear tracing facility) the linear range that has been occupied by Rootkit, Rootkit must relocate itself to some other free region by adjusting the base address of its code segment descriptor, which requires no content movement. In addition, after this relocation hardware GDTR, IDTR must be reloaded with descriptors of Rootkit itself inside these tables being adjusted accordingly, and MSR registers (describes the syscall entry points) are also updated to point to the new entry address. As an optimization, Rootkit may load initially into a special region known as not used by given Target OS.

2.7.12 **The Changes in Paging System**

- The Change of CR3: All instructions that access CR3 can be trapped, so Rootkit may simply emulate them.
- The Change of Page Directory Table/Page Table: Because Target OS can not perceive the change of CR3, also generally operating systems would map the page directory table (and each page table) at a fixed linear address, Target OS shouldn't be able to access the region of shadow table unless it does by accident, event so Rootkit can still protect its shadow page table in the same way like what it does to protect its shadow GDT that we have talked before.

2.8 Virtualize TSS

Virtualizing TSS is essential to interrupt/exception emulation (ring 0~2 stack pointer) and I/O instructions capture (I/O permission bitmap).

2.8.1 **The Structure of Private TSS**

Most modern operation systems are not inclined to use the hardware task mechanism, but the software task method instead to perform task switching manually. For example, Windows XP just uses one TSS for all processes, whose descriptor selector value is 0x0028 with DPL equal to 0, most fields in TSS are insignificant and usually it doesn't contain a valid I/O permission map (its offset beyond TSS limit), and XP only updates PDBR, ring0 stack and I/O map offset fields in TSS upon task switching (SwapContext).

Rootkit will provide a private (virtual) TSS and a TSS descriptor of its own for use directly by real processor: the private TSS resides in Rootkit's linear region, which is pointed by Base field in TSS descriptor, while the private TSS descriptor belongs to the Rootkit Descriptors portion in Shadow GDT with a relative higher selector value (the selector value of VMware's private TSS descriptor is 0x4000). The ring0 stack field in TSS points to the private ring0 stack of Rootkit, which is represented by a ring0 stack segment selector: stack pointer pair and must has sufficient space, while the I/O permission map field may be configured according to the current privilege level.

2.8.2 **Target Store TR**

str instruction can not be trapped at any privilege level, so by which Target OS may find the value change of TSS selector, but it is still impossible for it to access the TSS descriptor. Fortunately operation system would usually map the TSS to a fixed linear address and keep the address in PCR structure (be the same as GDT and IDT), the TSS descriptor selector value is also invariable (constant) and not necessary to be obtained by str instruction at runtime.

2.8.3 **Target Load TR**

ltr instruction can be trapped, then Rootkit only needs to save the new TSS selector value set with no traces installed on new TSS, and Rootkit will access Target OS's TSS only when it is necessary (forward interrupt/exception or emulate I/O). Because most modern operating systems only use one TSS shared by all tasks (not one TSS per task as before), Rootkit may hardly come across this instruction.

2.9 Virtualize Interrupt/Exception

Virtualizing interrupt/exception is a prerequisite. Firstly because the kernel mode privilege level has been lowered, it is impossible for Target OS itself to handle the interrupt/exception directly. The interrupt/exception emulation helps to hide the fact of virtualization, and avoids exposing the changes of segment selectors and some flags bit in trap frame. In addition, interrupt/exception is the only approach for Rootkit to gain execution control, and also handling various exceptions is the foundation for it to play all kinds of virtualization tricks.

2.9.1 **The Structure of Private IDT**

Firstly let's see a example of IDT under Windows XP (sp2):

```
lkd> !idt –a

Dumping IDT:

0:   804e0350 nt!KiTrap00          #DE
1:   804e04cb nt!KiTrap01          #DB
2:   Task Selector = 0x0058        NMI
3:   804e089d nt!KiTrap03          #BP (int3)
4:   804e0a20 nt!KiTrap04          #OF( into)
5:   804e0b81 nt!KiTrap05          #BR
6:   804e0d02 nt!KiTrap06          #UD
7:   804e136a nt!KiTrap07          #NM
8:   Task Selector = 0x0050        #DF
9:   804e178f nt!KiTrap09          #Coprocessor segment overrun
a:   804e18ac nt!KiTrap0A          #TS
b:   804e19e9 nt!KiTrap0B          #NP
c:   804e1c42 nt!KiTrap0C          #SS
d:   804e1f38 nt!KiTrap0D          #GP
e:   804e264f nt!KiTrap0E          #PF
f:   804e297c nt!KiTrap0F          reserved
10: 804e2a99 nt!KiTrap10           #MF
11: 804e2bce nt!KiTrap11           #AC
12: Task Selector = 0x00A0         #MC
```

```
13: 804e2d34 nt!KiTrap13                #XF
14: 804e297c nt!KiTrap0F                reserved
15: 804e297c nt!KiTrap0F                reserved
16: 804e297c nt!KiTrap0F                reserved
17: 804e297c nt!KiTrap0F                reserved
18: 804e297c nt!KiTrap0F                reserved
19: 804e297c nt!KiTrap0F                reserved
1a: 804e297c nt!KiTrap0F                reserved
1b: 804e297c nt!KiTrap0F                reserved
1c: 804e297c nt!KiTrap0F                reserved
1d: 804e297c nt!KiTrap0F                reserved
1e: 804e297c nt!KiTrap0F                reserved
1f:  804e297c nt!KiTrap0F               reserved
20: 00000000 00000000 (KINTERRUPT ffffffc4)
.
.
.
29: 00000000 00000000 (KINTERRUPT ffffffc4)
2a: 804dfb92 nt!KiGetTickCount
2b: 804dfc95 nt!KiCallbackReturn
2c: 804dfe34 nt!KiSetLowWaitHighThread
2d: 804e077c nt!KiDebugService
2e: 804df631 nt!KiSystemService
2f:  804e297c nt!KiTrap0F
30: 806f2d50 hal!HalpClockInterrupt                IRQ0
31: 821bcdd4
32: 804ded04 nt!KiUnexpectedInterrupt2
33: 81c40dd4
34: 822b543c
35: 804ded22 nt!KiUnexpectedInterrupt5
36: 804ded2c nt!KiUnexpectedInterrupt6
37: 804ded36 nt!KiUnexpectedInterrupt7
38: 806ecef0 hal!HalpProfileInterrupt
39: 823dc174
```

```
3a: 804ded54 nt!KiUnexpectedInterrupt10
3b: 823e5984                                    IRQB(PCI IRQ)
3c: 82065cdc
3d: 804ded72 nt!KiUnexpectedInterrupt13
3e: 8234c044
3f: 8234bb3c                                    IRQ0F
40: 804ded90 nt!KiUnexpectedInterrupt16
.
.
.
ed: 804df452 nt!KiUnexpectedInterrupt189
ee: 804df459 00e96800 (KINTERRUPT 804df41d)
        00000000 (KINTERRUPT 000000e4)
ef: 804df460 nt!KiUnexpectedInterrupt191
f0: 804df467 nt!KiUnexpectedInterrupt192
f1: 804df46e 0000eb68 (KINTERRUPT 804df432)
        00000000 (KINTERRUPT e8fffffc)
f2: 804df475 nt!KiUnexpectedInterrupt194
f3: 804df47c 0085e900 (KINTERRUPT 804df440)
        00000000 (KINTERRUPT 0000008b)
f4: 804df483 nt!KiUnexpectedInterrupt196
f5: 804df48a nt!KiUnexpectedInterrupt197
f6: 804df491 000000ef (KINTERRUPT 804df455)
        00000000 (KINTERRUPT 0000ee64)
f7: 804df498 000000f0 (KINTERRUPT 804df45c)
        00000000 (KINTERRUPT 0000ef64)
f8: 804df49f 000000f1 (KINTERRUPT 804df463)
        00000000 (KINTERRUPT 0000f064)
f9: 804df4a6 000000f2 (KINTERRUPT 804df46a)
        00000000 (KINTERRUPT 0000f164)
fa: 804df4ad 000000f3 (KINTERRUPT 804df471)
        00000000 (KINTERRUPT 0000f264)
fb: 804df4b4 000000f4 (KINTERRUPT 804df478)
        00000000 (KINTERRUPT 0000f364)
```

```
fc: 804df4bb 000000f5 (KINTERRUPT 804df47f)
        00000000 (KINTERRUPT 0000f464)
fd: 804df4c2 000000f6 (KINTERRUPT 804df486)
        00000000 (KINTERRUPT 0000f564)
```

In the IDT shown above: the first 32 entries (0x0~0x1f) are exceptions reserved by Intel, among these entries, NMI, #DF, #MC are handled particularly by task gate mechanism, #BP and #OF are interrupt gates with DPL = 3, and all the rests are interrupt gates with DPL = 0. The following 0x2a~0x2f are software interrupts portion, which are interrupt gates with DPL = 3 except for 0x2f (DPL = 0), and 0x2d, 0x2e serve as debug service and system call service respectively. Finally comes the hardware interrupts portion, which are 0x30~0x3f and correspond to IRQ0~IRQf.

Rootkit provides a private IDT (resides in Rootkit space) that is pointed by hardware IDTR, in which the basic layout of descriptors is almost a reprint of Target OS' IDT, and also a separate handler (stub) for each vector, which can be indexed by each corresponding interrupt gate descriptor in private IDT. Because the generation of hardware interrupts and exceptions don't need to check (ignore) the DPL of gate descriptors, all gate descriptors of Target IDT may keep their DPL unchanged in the private IDT except for those software interrupts with DPL = 0, whose DPL must be modified to 1 (or 3). Recurring to private IDT and TSS, Rootkit would be able to get execution control at the first time when interrupt/exception occurs.

2.9.2 **Handle Interrupt/Exception**

When interrupt/exception occurs, the corresponding handler of Rootkit gets invoked, and processor has created in advance a interrupt/exception frame in ring0 stack specified in Rootkit's private TSS as follows:

```
On rootkit's own stack (inter-privilege)
Low address:
    Error Code (if needed)
    EIP
    CS
    EFLAGS
    ESP
    SS
```

High address:

The stack images pushed of SS, CS and EFLAGS would be the real values in hardware registers during the running of Target OS, which may not be what Target OS expects and must be fixed accordingly when forwarding. Before further processing, Rootkit must save the Target OS' context firstly, which involves primarily the general-purpose registers and data segment registers. In order to access the whole 4G address space, Rootkit may load DS with its own data segment in and switch it back before returning to Target OS, otherwise iret instruction would invalidate DS register due to privilege level unmatched (however, in practice Rootkit also may not use flat mode segments of its own for the sake of easier code relocation). Rootkit may get to know the running privilege level of Target OS when interrupt/exception happens according to the SS and CS pushed in trap frame, moreover it can judge whether it is an exception resulted from the virtualization of itself or not by considering all facts comprehensively, which include the exception vector, error code, faulting address (EIP) and #PF address (in CR2). For the exceptions as results of virtualization, Rootkit may perform necessary processing and then return directly, on the other hand for those hardware interrupts, software interrupts and exceptions incurred by Target OS, Rootkit has to forward them to Target OS correctly.

The exceptions introduced by Rootkit virtualization include #PF, #GP and #NP etc. Usually, #PF is due to the physical traces installed on some pages of Target OS by Rootkit (or unmapping the pages where desynchronized segment descriptors reside), the reason of #GP is in that all kinds of privilege related checkings may fail when Target OS is running with degraded privilege level, while #NP is the result of Target loading desynchronized segments. After various appropriate processing (omitted) according to various situations, Rootkit may return to the location of faulting instruction or its next instruction directly, moreover it may return immediately to the location of transfer target if the faulting instruction is a certain control transfer instruction, the detailed return steps include restoring segment and general-purpose registers at first, then following a stack pop of error code, and a final iret instruction.

In contrast with the exceptions processing above, the forwarding of those hardware interrupts, software interrupts and exceptions by Target OS are not as easy as the emulation of those deferred interrupts by Windows HAL, which only needs to simply execute a int xx (HalpHardwareInterruptxx), here in fact the word "forward" means that Rootkit uses software method to accurately emulate the generation process

of interrupt/exception by hardware processor, the involved steps are listed as follows:

- If it is a hardware interrupt, then decides whether it is allowed to generate or not according to the related bits in IMR of virtual PIC and IF flag of EFLAGS.
- Locates Target OS' IDT and indexes the corresponding gate descriptor by interrupt/exception vector and virtualized IDTR of Target OS.
- Decides whether a privilege switch should happen or not by comparing Target OS's current privilege level with the privilege level of target code segment specified in gate descriptor.
- If a privilege switch should happen, then locates the corresponding internal-ring stack pointer pair according to the privilege level of target code segment and the virtualized TSS of Target OS (TR and GDT as well).
- Fakes a correct interrupt/exception frame on appropriate stack (the current stack or stack of internal-ring) .
- Clears some flag bits in virtualized EFLAGS of Target OS according to the type of gate descriptor.
- Transfers control to proper handler of Target OS, and the forwarding ends.

Actually we have omitted a series of protection checkings performed by real processor in above-mentioned steps for simplicity, which may include the checking of privilege level, type and limit etc. Anyone of these checkings fails, then the resulting exception would be forwarded to Target OS in the first place before the current interrupt/exception to be forwarded.

Before actual transferring to interrupt/exception handler of Target OS, Rootkit would fake a correct interrupt/exception frame as follows on the appropriate stack of Target OS:

```
On target's current stack (intra-privilege)
Low address:
    Error Code (if needed)
    EIP
    CS
    EFLAGS
High address:
```

On target's inner stack (inter-privilege)
Low address:
Error Code (if needed)
EIP
CS
EFLAGS
ESP
SS
High address:

The stack images pushed of SS, CS and EFLAGS would all be the values when Target OS runs natively (not be virtualized), which should absolutely be what Target OS expects. For the reason that interrupt/exception handlers would usually distinguish the format of trap frame by judging the RPL value of CS selector image on stack, for example Linux decides whether or not it is a inter-privilege level transfer by comparing RPL of CS selector image with 3. Therefore creating correct interrupt/exception frames on appropriate stacks of Target OS may ensure the proper running of Target OS, and also avoid exposing the fact of virtualization.

The actual transfer from Rootkit to Target OS's interrupt/exception handler is completed by a iret instruction, before that Rootkit should fake a interrupt/exception frame on the stack of its own, which would eventually emulate a inter-privilege level return to Target OS. The interrupt/exception frame created as follows:

On rootkit's own stack (inter-privilege)
Low address:
EIP
CS
EFLAGS
ESP
SS
High address:

The CS:EIP pair is the address of Target OS's interrupt/exception handler that is specified in gate descriptor, and the RPL of CS must be fixed according to the segment compression scheme used. The SS:ESP pair is either the current stack of Target OS or internal-ring stack taken from its current TSS, also the RPL of SS (internal-ring stack) must be adjusted. As to the EFLAGS, we may clear some flag

bits (TF, NT, VM, RF and IF when an interrupt gate used) on the base of the value pushed on the stack when Target OS traps to Rootkit, however the hardware IF must always set, so we only clear IF flag in the virtualized EFLAGS of Target OS in stead of the real one.

Here we assume that all interrupt/exceptions are generally handled by Target OS kernel mode, although there may surely exist some exceptions, such as the XOK operating system which handles some non-critical exceptions and software interrupts in user mode routines. When Target kernel mode being compressed to ring1, because the CS selector RPL (0) in trap frame < CPL (1), Target OS's inner-privilege level return (return to the same privilege level) would definitely incur a #GP, which then needs Rootkit to help to emulate the iret instruction, while its inter-privilege level return may go natively. When Target kernel mode being compressed to ring3, the inner-privilege level return would also need to be emulated by Rootkit because it is impossible to return to a code segment with higher privilege level in x86, while the inter-privilege level return may happen by error since it is considered as an inner return (CS RPL and CPL are both 3), which then leads Target OS to a wrong routing. It is advantageous for perfect virtualization of the flags bits modification by iret instruction that Rootkit interposes and emulates the interrupt/exception return of Target OS, a feasible approach is that Rootkit puts a invalid EIP that points to its own linear space when it fakes the trap frame on Target stack, in this way it not only doesn't expose the virtualization footprints, but also ensures that every iret of Target OS would incur a #GP because the target EIP beyond the limit of target code segment, and then give Rootkit a opportunity to perform emulation.

2.9.3 **Target Store IDTR**

sidt instruction can not be trapped at any privilege level, so by which Target OS may find the change of IDT base (usually limit is unchanged 0x7ff), but it is still impossible for it to access the descriptors inside it. Fortunately operation system would usually map the IDT at a fixed linear address and keep the address in PCR structure (be the same as GDT and TSS), therefore no need to obtain it via sidt instruction at runtime.

2.9.4 **Target Load IDTR**

lidt instruction can be trapped, then Rootkit only needs to save the new IDT's base and limit set with no traces installed on it, and Rootkit will access Target OS's IDT only when it is necessary (forward interrupt/exception). Because most operating systems only set IDT in system initialization phase and don't modify it anymore, Rootkit may hardly come across this instruction.

2.10 Virtualize Devices

Because VM Based Rootkit is not a full-fledged machine virtualization project, it is impossible and unnecessary to provide a complete set of virtual devices for Target OS, what Target OS saw are still those real physical hardware, however this doesn't mean Rootkit has no ability to interpose and control the device related operations of Target OS. Because of the opportunity when Rootkit starts, it has no chance at all to do anything to the device enumeration and configuration process (also known as PCI probing), but it can undoubtedly control the succedent device operations after that. Next we are going to discuss briefly how to virtualize four general device operations.

2.10.1 **Port Mapped I/O**

Rootkit can utilize the IOPL flag bit combined with I/O permission map to capture all in/out instructions that issued by Target OS to access I/O address space. It goes like this: Whichever privilege level Target OS is running at, the hardware IOPL will be always set to 0 to prevent Target OS from enabling/disabling interrupt by using sti/cli instruction pair. When virtualized CPL <= virtualized IOPL , that means Target OS is running with I/O access right, all other relevant bits in private TSS I/O permission map will be cleared except for those ports for which Rootkit cares. While virtualized CPL > virtualized IOPL, all cleared bits above mentioned will be set to the current values of Target OS's real TSS. Once the I/O port accesses being captured, Rootkit may discard, modify (perform real I/O access for the Target OS, but do some modifications on incoming/outgoing data) or emulate them according to the requirements.

2.10.2 **Memory Mapped I/O**

Some devices have registers or storage regions which are shown as the form of

mapped memory, that means these stuffs are mapped directly into the processor's physical memory address space and processor may use ordinary memory-reference instructions to access them, for instance the registers of Local APIC and video buffer (VRAM). Rootkit may utilize the paging mechanism provided by processor to capture these memory mapped I/O: simply unmapping its physical pages or enforcing some page protections on its corresponding linear pages both allow Rootkit to get chances to virtualize the access to memory mapped devices.

2.10.3 **Hardware IRQ**

We have discussed fully the capture and forwarding of hardware interrupt in previous chapters, what still deserves a few words here is the virtualization of interrupt controller. Target OS may frequently configure the interrupt mask register when running, while for Rootkit, masking some IRQs (clock interrupt IRQ0) may result in the its loss of control, therefore it must maintain a virtual interrupt controller by itself. The structure and working principle of APIC used in SMP system are relatively complex, while virtualizing a PIC seems much easier, which is some similar to the virtualization of IRQL in Windows kernel. Rootkit creates and maintains a virtual PIC of its own, which further includes the interrupt mask register, interrupt request register, interrupt in-service register (IMR, IRR, ISR) etc, all captured operations to relevant ports of hardware PIC would eventually be reflected to this virtual PIC, and then Rootkit may use the state information in such a virtual PIC to make decision when forwarding hardware interrupts.

2.10.4 **DMA**

The DMA operation of device is a big headache to virtualization, especially under the circumstance when the current PCI bus allows the PCI devices (interfaces in fact) to become the master devices of bus, and to perform so-called Bus Mastering DMA. Whichever DMA method being used, the third-party DMA through system DMA controller (two cascaded 8237) or the Bus Mastering DMA, it is extremely disadvantageous for virtualization in that Target OS may exploit device DMA to access any desired real physical memory page, which may bypass all the memory protections imposed by Rootkit's segment/paging virtualization strategies, and potentially destroy the memory regions occupied by Rootkit itself. Presently the best

solution is to capture the address setting actions to system DMA controller or related DMA registers in PCI interfaces (such as DMA region table and BMIDTPX register in IDE interface), and to fully emulate them, however this would undoubtedly reduce the generality heavily, and increase implementation complexity at same time. Besides there is another workaround, which marks the physical pages occupied by Rootkit as bad (such as parity error) in PFN database maintained by Target OS to prevent it from using those pages thereafter, but its obvious disadvantage is the introducing of new virtualization footprints. AMD Pacifica uses DEV (Device Exclusion Vector) technique to enforce DMA protection.

2.11 The Payload of VMBR

After VM Based Rootkit being loaded as an intermediate layer between Target OS and the real hardware, what it can see directly are just those hardware related events, such as instruction executions, interrupt requests and I/O operations, however Rootkit may also control the internal running logics and states of Target OS by using VMI (Virtual Machine Introspection) technique.

In fact, what a VM Based Rootkit can really do only depends on your imagination, here I will demonstrate how to hide specified process in icesword (a famous rootkit revealer by Chinese) in my VMBR prototype. For the purpose of anti-debugging, icesword periodically restores the int 1 and int 3 vectors in IDT to their original values which should point to the debug and breakpoint exception handlers provided by Target OS, and also it may clear the debug registers (dr0 ~ dr3) to prohibit hardware breakpoints, however all of these anti-debugging methods are useless in a virtualized environment because all sensitive operations are actually emulated by Rootkit. Having broken through the obstacle of anti-debugging, the left works seem much easier, we set a hardware breakpoint at the function used by icesword to enumerate processes (ExEnumHandleTable, undocumented), then another breakpoint at the location of the callback function which is passed as an argument. When callback function called, we simply filter out the desired process by comparing the process identifier. A companion application is used combined with VMBR, which communicates to VMBR stealthily by a software breakpoint instruction (int 3) with eax equal to a magic number and ebx equal to the desired process identifier. By comparing the two following pictures, we can see that the command line process (CMD.exe, pid 808) has been hidden from the process list.

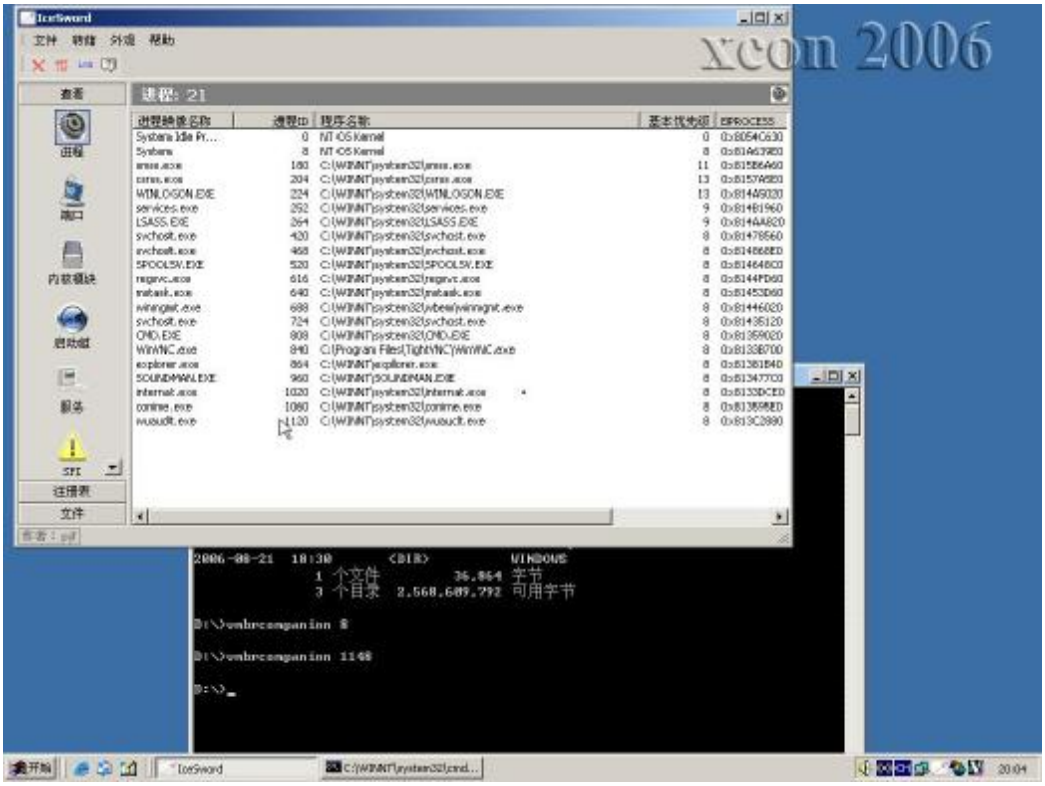

Diagram 2－3 the demonstration of hiding specified process (before)

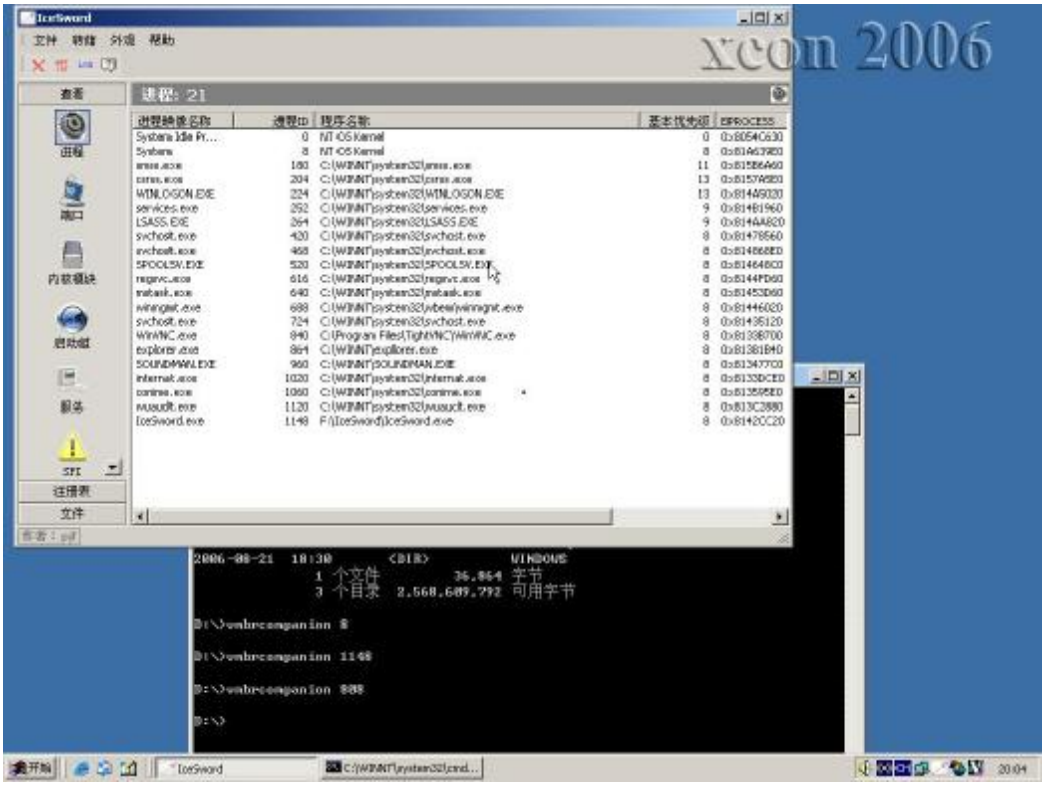

Diagram 2－4 the demonstration of hiding specified process (after)

Chapter 3 Conclusion

Because x86 is not a strictly virtualizable processor architecture, at present my VMBR prototype has not implemented a perfect virtualization with the lack of binary translation or dynamic scanning techniques, however this proof of concept has at least proved that it can coexist with Target OS and work properly, which further validates the correctness of the whole virtualization scheme that we discussed.

My future works include 1) trying to support more Target OSes (only Windows 2000 is eligible for serving as Target OS now), 2) adding more supports for advanced processor features, such as SMP and SMM etc, 3) introducing simple dynamic scanning module to deal with those non-virtualizable instructions of x86.

We can see very clearly from the following diagram, when running at a virtualized environment, the executions of all privileged instructions of Target OS would inevitably incur exceptions and be emulated by VMBR.

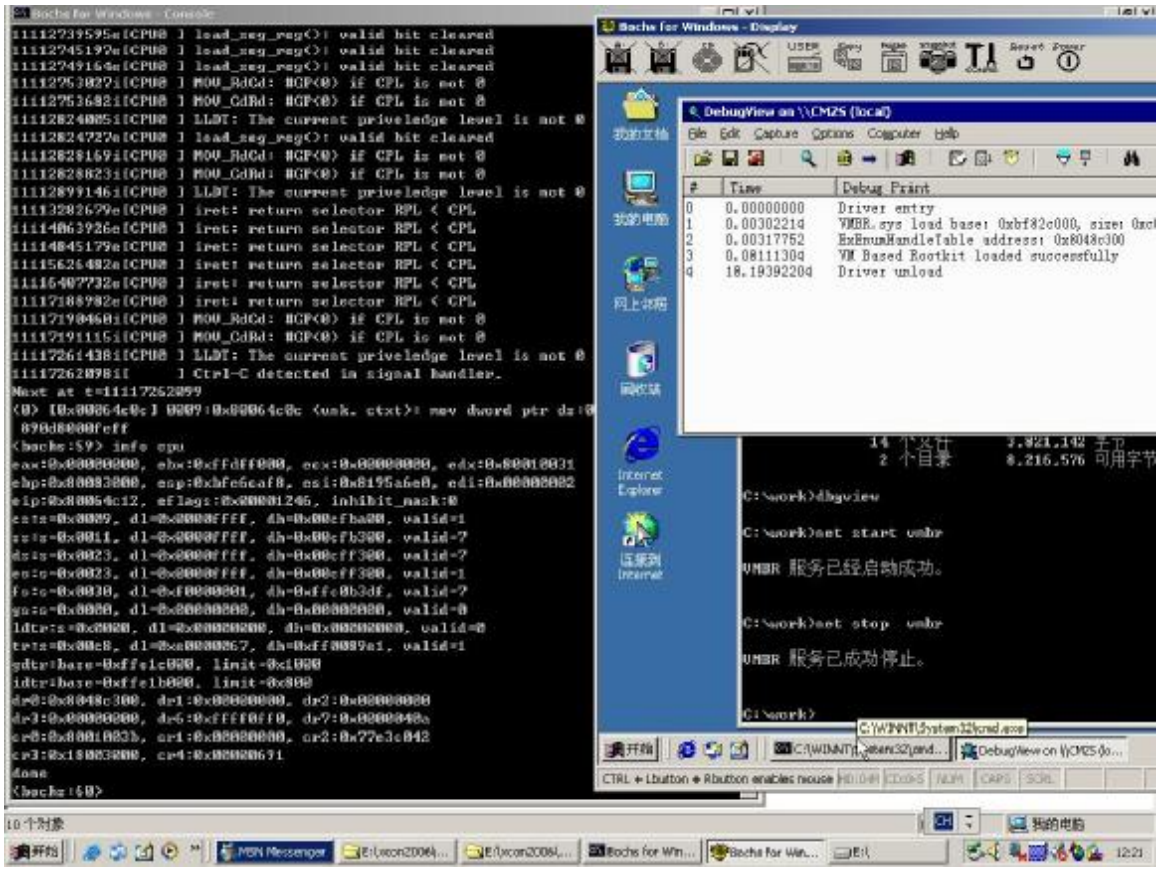

Diagram 3－1 VMBR prototype running within a debug version of Bochs

Acknowledgement

First, I have to say that actually many contents in this paper are the abstractions and summaries of viewpoints or concepts of many forthgoers in virtualization field, without their great works, it was impossible for me to complete this paper so smoothly.

Secondly, I would like to thank my colleague Borun Song. The discussions between us have provided me many inspirations on technical aspect.

In addition, I must say thanks to my colleagues Jiayu Zang and my friend Chuandong Zheng. Miss Zhang teaches me how to use MS Word for typesetting, and as the final reviser Mr. Zheng ensures the text quality of this paper.

Finally, once again, let me express my most sincere thanks to all people who make contributions to this paper!

References

【1】 Intel Corporation 《Intel Architecture Software Developer's Manual Volume 2》 1997

【2】 Intel Corporation 《Intel Architecture Software Developer's Manual Volume 3》 2003

【3】 VMware Inc.《System And Method For Virtualizing Computer Systems》 Dec. 2002

【4】 VMware Inc. 《System And Method For Facilitating Context-Switching In a Multi-Context Computer System》 Sep. 2005

【5】 VMware Inc. 《Deferred Shadowing of Segment Descriptors In a Virtual Machine Monitor For a Segmented Computer Architecture》 Aug. 2004

【6】 VMware Inc. 《Dynamic Binary Translator With A System And Method For Updating And Maintaining Coherency Of A Translation Cache》 Mar. 2004

【7】 VMware Inc. 《Method And System For Implementing Subroutine Calls And Returns In Binary Translation SubSystem Of Computers》 Mar. 2004

【8】 John Scott Robin, Cynthia E. Irvine 《Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor》 Aug. 2000

【9】 K. Lawton 《Running Multiple Operating Systems Concurrently On an IA32 PC Using Virtualization Techniques》 1999

【10】Samuel T. King, Peter M. Chen, Yi-Min Wang 《SubVirt Implementing Malware With Virtual Machines》 2006

【11】Jeremy Sugerman, Ganesh Venkitachalam, Beng-Hong Lim 《Virtualizing IO Devices on VMware Workstation's Hosted Virtual Machine Monitor》 Jun. 2001

【12】Charles L. Coffing 《An x86 Protected Mode Virtual Machine Monitor for the MIT Exokernel》 May. 1999

【13】Prashanth P. Bungale, Swaroop Sridhar, Jonathan S. Shapiro 《Low-Complexity Dynamic Translation in VDebug》 Mar. 2004

【14】Prashanth P. Bungale, Swaroop Sridhar, Jonathan S. Shapiro 《Supervisor-Mode Virtualization for x86 in VDebug》 Mar. 2004

【15】Jack Lo 《VMware and CPU Virtualization Technology》 2005
【16】Dave Probert 《Windows Kernel Internals II Virtual Machine Architecture》Jul. 2004
【17】Steve McDowell, Geoffrey Strongin 《Virtualization Technology For AMD Architecture》
【18】Narendar B.Sahgal, Dion Rodgers 《Understanding Intel® Virtualization Technology (VT) 》