IDTchecker: Rule-based Integrity Checking of Interrupt Descriptor Tables in Cloud Environments

Irfan Ahmed, Aleksandar Zoranic, Salman Javaid, Golden G. Richard III, Vassil Roussev

Department of Computer Science, University of New Orleans,
Lakefront Campus, New Orleans, LA 70148, United States
(irfan.ahmed, azoranic, sjavaid1)@uno.edu, (golden,vassil)@cs.uno.edu

Abstract

An interrupt descriptor table (IDT) is used by the processor to transfer the execution of a program to special software routines that handle interrupts, which might be raised during the normal course of operation by hardware or to signal exceptional conditions, such as a hardware failure. Attackers frequently modify the pointers in the IDT in order to execute malicious code. In this paper we present `IDTchecker`, which provides a comprehensive rule-based approach to check the integrity of the IDT and the corresponding interrupt handling code, based on a particular scenario commonly found in cloud environments. In this scenario, multiple virtual machines (VMs) run the same version of an OS kernel, which implies that IDT related code should also be identical across the pool of VMs. IDTchecker uses this scenario to compare the IDTs and the corresponding interrupt handlers across the VMs for any inconsistencies, based on a pre-defined set of rules. We thoroughly evaluate the effectiveness and runtime performance of IDTchecker and find that it can detect any change in the IDT or interrupt handling code without having any significant impact on a guest VMs' system resources. Moreover, IDTchecker itself has a very small memory footprint (i.e. 10-15MB).

Keywords: Cloud Computing, Virtualization, Malware, Interrupt Descriptor Table, Forensic Analysis

1. Introduction

Memory forensics involves extracting digital artifacts from the physical memory of a computer system. The computer's Interrupt Descriptor Table (IDT) is one such artifact, which is a well-known target for malware (especially rootkits). The IDT provides an efficient way to transfer control from a program to an interrupt handler, which is a special software routine, to handle exceptional conditions occurring within the system, processor, or currently executing program. For instance, hardware failure and division by zero are two unusual conditions that are handled through the IDT. Malware manipulates the IDT in order to change the system's control flow and run malicious code. The changes may occur either to the pointer in the IDT or in the interrupt handler itself, to redirect execution to malicious code that has been injected into the system.

A state of the art solution [1] [2] checks the integrity of IDT by keeping a valid state of the table and comparing it with the table's current state. For instance, Microsoft (MS)

introduces kernel patch protection (a.k.a., PatchGuard) [2] in 64-bit MS Windows to detect any modifications in kernel code and certain critical data structures including the IDT. PatchGuard caches the legitimate copy and the checksum of the IDT, then compares them with the current IDT in memory to check for any modifications. Similarly, another tool, CheckIDT [1], examines the integrity of the IDT by storing the entire table in a file so that it can be compared later with the current state of the table in memory. Despite these approaches being applicable to different operating systems, they still suffer at least two major limitations:

- They require an initialization phase where it is assumed that the IDT is not infected at the time the valid state of the IDT is obtained. This may not be the case especially if the interrupt handler is patched in the kernel file on disk because when the system restarts and the IDT is created, the IDT will point to the malicious interrupt handler before the valid state of the IDT can be obtained. The pointers in the IDT may change after the system boots if the kernel or kernel modules are loaded at different memory locations. Thus, every time the system restarts, such approaches need to record the valid state of the table.

- Current solutions do not specifically consider the interrupt handler's code for integrity checking. Although they do check the integrity of the entire kernel code and modules that also include interrupt handler's code, they do not ensure that the pointer in the IDT points to a valid interrupt handler. A state of the art solution for checking the integrity of kernel code (and modules) requires maintaining a dictionary of cryptographic hashes of trusted code [3] [4], which compares the hash of the current code with the one stored in the dictionary. Such an approach requires maintaining the dictionary across every kernel update for effective integrity checking.

In this paper, we propose `IDTchecker`, which provides a comprehensive, rule-based approach to check the integrity of the IDT in real time without requiring an initialization phase, a "known good" copy of the IDT, or a dictionary of hashes. IDTchecker works in a virtualized environment where a pool of virtual machines (VMs) run identical guest operating systems with the same kernel version - a typical scenario in cloud servers. Such pools of virtual machines are maintained to simplify the maintenance process so that applying patches and upgrading systems can be automated. IDTchecker works by retrieving the IDT and its corresponding interrupt handler code from the physical memory of the guest VM and comparing them across the VMs in the pool. It uses a pre-defined set of rules to perform comprehensive integrity checking. IDTchecker runs on a privileged virtual machine where it has access to a guest VMs' physical memory through virtual machine introspection (VMI). None of its components run inside the guest VMs, which makes IDTchecker more resistant to tampering by malware.

We performed extensive evaluation of IDTchecker in terms of effectiveness and efficiency. We used real-world malware and popular IDT exploitation techniques to modify the IDT and its corresponding interrupt handler code, all of which are successfully detected by IDTchecker. We analyzed the runtime performance of IDTchecker during best and worst case scenarios. Our results show that IDTchecker does not have any significant impact on a guest VMs' resources, since none of its components run inside the VMs. Its memory footprint is around

10-15MB, which is quite negligible as compared to the size of physical memory typically found in a cloud server (tens to hundreds of gigabytes).

The paper is organized as follows: Section 2 presents related work. Section 3 provides an overview of the IDT and describes IDTchecker's architecture in detail. Section 4 describes the IDTchecker's implementation followed by the evaluation in section 5 and conclusion in section 6.

2. Related work

There has been relatively little work done on IDT integrity checking; this section presents a brief summery.

Kad proposes CheckIDT [1] a Linux-based tool that is able to detect any modification in the IDT by storing the IDT descriptor values in a file and later comparing it to the current state of the IDT in memory. If any discrepancy between the state of the two tables is detected, CheckIDT can restore the table in memory by copying the IDT values from the file, assuming that the file's integrity is still intact. In order to access the in-memory table, CheckIDT uses Sd *et al.*'s technique [5] on /dev/kmem that allows CheckIDT to access the table from user space without using a Linux kernel module.

Kernel Patch Protection [2] (or PatchGuard) is an approach that checks the integrity of kernel code (including modules) and important data structures such as the IDT, Global Descriptor Table (GDT), System Service Descriptor Table (SSDT) and syscalls table. It is currently introduced by Microsoft in 64-bit Windows operating systems. PatchGuard stores a legitimate known-good copy and checksum of kernel code and data structures and compares them with the current state of the code and the data structures at random intervals of time. It is implemented as a set of routines, which are protected in the system by using anonymity techniques such as misdirection, misnamed functions and general code obfuscation.

Volatility [6] has a plugin [7] that checks the integrity of the pointers to interrupt handlers in the IDT. It walks through each entry in the IDT and checks whether the pointer in the entry is within the address range of the kernel code (including modules). It ensures that the pointers do not point to an unusual location in memory, however, it cannot detect attacks [1] [8] that do not require modifying a pointer in the table, but instead directly patch interrupt handler code.

IDTGuard [9] is an MS Windows-based tool, which checks the integrity of pointers in the IDT. It separately computes the pointer values of the IDT by finding the offset of the interrupt handler in the kernel file (such as `ntoskrnl.exe`) and adds it to the base address of the kernel in memory. The computed values are then matched with the pointers in the IDT table for integrity checking. However, this tool cannot check the integrity of pointers corresponding to kernel modules where the pointers point to `Kinterrupt` data structures, instead of the interrupt handler in the kernel code.

3. IDT Integrity Checking

With increases in computational power and physical memory, a physical machine can easily accommodate the computational needs of more than one average user at a time. Virtualization provides an opportunity for efficient resource utilization of a physical machine

by concurrently running several VMs over a virtual machine monitor (VMM) or hypervisor – an additional layer between hardware and hosted guest operating systems. The VMM also allows a privileged VM to monitor the runtime resources of other (guest) VMs, such as memory, I/O, disk etc., through virtual machine introspection - an approach that facilitates building more robust security tools for the environment. IDTchecker uses introspection while running in a privileged VM to access the physical memory of guest VMs, retrieves the IDTs and their corresponding interrupt handlers and matches them (according to the pre-defined set of rules) in order to check for any inconsistencies.

3.1. Overview of the IDT

Interrupts and exceptions are system events that indicate that a condition or an event requires the attention of a processor [10]. Interrupts can be generated in response to hardware signals such as hardware failure or by software through `INT n` instruction. Exceptions are generated when the processor detects an error during the execution of an instruction such as divide by zero. Each such condition indicated by an interrupt or exception requires special handling by the processor and thus, is represented with a unique identification number, which is referred to as an interrupt vector. (In this paper, the difference between interrupts and exceptions is not important and thus we refer to both of them as interrupts.) When an interrupt occurs, the current execution of a program is suspended and the control flow is redirected to an interrupt handler routine through IDT.

An IDT is an array of interrupt vectors where each vector provides an entry point to an interrupt handler. There can be at most 256 interrupt vectors. Each vector is represented by 8 bytes, containing the information about the index to a local/global descriptor table, request/descriptor privilege levels, offset to interrupt handler etc. There are up to three types of interrupt (vector) descriptors in an IDT: interrupt gate, trap gate and task gate. Interrupt and trap gate descriptors are similar, but they differ in the functionality and the type field in the descriptor that identifies the gate. Unlike for trap gates, the processor, while processing an interrupt gate, clears IF flag in the EFLAGS register in order to prevent other interrupts from interfering with the current interrupt handler. Task gate descriptors, on the other hand, have no offset value to the interrupt handler. This handler is reached through the segment selector field in the descriptor.

The Global Descriptor Table (the GDT)[1] is utilized when the interrupt handler needs to be accessed in protected mode (where protection ring [10] is enforced). An entry in the table is called segment descriptor. Each descriptor describes the base address and the size of a memory segment along with the segment's access rights information. Each descriptor is also associated with a segment selector that provides the information about the index to the descriptor, access rights and a flag to determine whether the index points to an entry of the global descriptor table. Each interrupt vector has a segment selector that is used to find the base address of the segment. In case of interrupt and trap gates, the base address of the interrupt is obtained by adding segment's base address to the offset in the vector.

[1]The Local Descriptor Table (LDT) is also used to access the interrupt handler. However, this usage is beyond the scope of this paper.

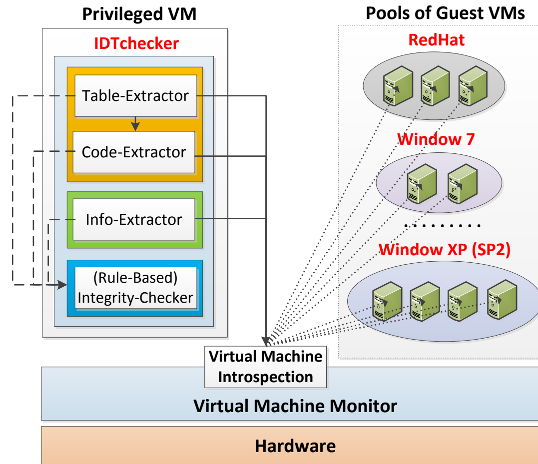

Figure 1: IDTchecker Architecture

3.2. Assumptions

We assume a fully virtualized environment where the VMM supports memory introspection of guest VMs, and that there exist different pools of VMs where each pool runs an identical guest operating system with the same kernel version. This provides an opportunity for IDTchecker to probe and compare the IDTs and interrupt handlers within each pool.

3.3. IDTchecker architecture

The basic idea behind IDTchecker is to obtain the IDTs and corresponding interrupt handler code from a pool of VMs and perform a comprehensive integrity check based on a predefined set of rules. To achieve this task, we have divided the IDTchecker into four modules that accommodate different functions of IDTchecker needed during the entire process of integrity checking. The four components are Table-Extractor, Code-Extractor, Info-Extractor and Integrity-Checker.

- **Table-Extractor & Code-Extractor:** We define separate modules (Table-Extractor and Code-Extractor) for extracting tables and interrupt related code because the structure of the tables (IDT and GDT) are dependent on the processor while the organization of the interrupt related code in the system is mostly dependent on the operating system. For instance, MS Windows uses a `Kinterrupt` structure to store the information about an interrupt handler that is provided by kernel drivers. Having extraction of code and table into two separate modules increases the portability of IDTchecker. Moreover, Code-Extractor receives interrupt vector descriptor values from Table-Extractor after the descriptor is parsed, which Code-Extractor needs to find the index of the GDT segment and offset of interrupt handler (if the descriptor type is not a task gate).

- **Info-Extractor:** The Info-Extractor module is used to fetch any additional information from memory, such as the address range of kernel modules. Such information might be required by a rule to assess some aspect of the IDT or the interrupt handling code.

- **Integrity-Checker:** The Integrity-Checker module is used to apply a pre-defined set of rules on the data obtained from the last three modules in order to comprehensively check the integrity of the IDT. Unlike the other three modules, Integrity-Checker does not need to access the memory of guest VMs, since all necessary data is already made available by other modules.

Figure 1 illustrates the overall architecture of IDTchecker in a typical setting where IDTchecker would be effective. It shows different pools of guest VMs where each pool is running the same version of a guest operating system. The VMs are running on top of a VMM, and the VMI facility is available for privileged VMs to introspect the guest VMs' system resources. To clarify several points, IDTchecker only needs to perform read-only operations on the guest VMs' physical memory and no component of the IDTchecker runs inside the guest VMs.

Discussion: Given that IDTchecker can compare IDTs and their corresponding interrupt handler code across VMs, it is able to detect any inconsistency in the IDT among VMs. We can use a majority voting algorithm to determine which VM has been infected. However, the majority vote can only be effective if the majority of VMs have uninfected IDTs. In this case, IDTchecker is more effective in detecting the first sign of infection, which can then be used to trigger a thorough forensic investigation in order to find the root cause of the infection.

It is also worth discussing whether the IDT should always be identical across VMs when identical kernel code (including modules) is running. The initial 32 interrupt vectors (i.e. 0 to 31) are pre-defined. These interrupt vectors will always remain identical across VMs. However, other interrupt vectors (from 32 to 255) are user-defined and may vary across VMs in that the same interrupt entry or descriptor can be associated with different interrupt vectors across VMs. Thus, one-to-one matching of such interrupt entries may not be feasible at all the times. A more robust approach is to find the equivalent interrupt vector entry (that is being matched) in other IDT tables across VMs before rules are applied to them. Moreover, the current version of IDTchecker does one-to-one matching and can further be improved with this approach.

3.4. Rules for integrity checking

IDTchecker currently uses four rules to perform integrity checking across VMs and within each VM:

Rule 1: All the values in each interrupt vector should be the same across VMs (excluding the interrupt handler offset field, which is checked for integrity by the subsequent set rules). This rule ensures that the fields in the IDT are original.

Rule 2: The interrupt handler code should be consistent across VMs. This rule detects any modification in the code. The rule is effective in detecting the modifications, unless the identical modification is done across all VMs in the pool.

Rule 3: The interrupt handler is located either in basic kernel code or a kernel module. This means, the base address of interrupt handler should be within the address range of either basic kernel code or any module's code. This rules ensures that the base address is not pointing to an unusual location.

Rule 4: Given that the base address of an interrupt handler is within the address range of the kernel code or any module, the offset of the base address of the interrupt handler from the starting address of its corresponding driver or basic kernel code should be the same across all virtual machines. This rule detects instances of random injections of malicious code within a driver or the basic kernel code.

4. Implementation

IDTchecker's design is simple in that all its components reside locally on a privileged VM, which can be implemented on any VMM that has introspection support (such as Xen, KVM or VMware ESX) without requiring modifications to the VMM itself. For the proof of concept, we developed IDTchecker on Xen [11], with MS Windows (Service Pack 2) XP guest operating systems and used the LibVMI introspection library [12]. We also used Opdis [13] (a disassembler library) and OpenSSL [14] (for computing cryptographic hashes).

The rest of the section describes the low-level implementation details of the components of IDTchecker.

4.1. Table-Extractor

The IDT and GDT are created each time a system starts. The processor stores their base address and size in `IDTR` and `GDTR` registers for protected mode operations. In each register the base address specifies the linear address of byte 0 of the table, while the size specifies the number of bytes in the table. Table-Extractor obtains this information from the registers in the guest VM and extracts the IDT and GDT tables from the guest VM's memory. It further interprets the raw bytes of the tables as table entries and their respective fields, then forwards them to the Code-Extractor.

4.2. Code-Extractor

After receiving the tables from the Table-Extractor, Code-Extractor retrieves the code corresponding to each IDT entry. The process of Code-Extraction varies with the interrupt vector type (i.e., interrupt gate, task gate, trap gate), thus, we discuss the process for each type separately. In the Windows XP VMs we used, we found no trap gate entries in the IDT. Thus, in this section, we only discuss interrupt gates and task gates. We also found that most vectors in Windows XP's IDTs are interrupt gates except for three task gate vectors.

4.2.1. Interrupt gate

Each interrupt gate entry in the IDT has a segment selector associated with the GDT. It also has an interrupt handler offset that can be added to the base address of the segment described in the GDT to form the base address `ptr` of the interrupt handler. The interrupt handler can be located either in the basic kernel code (i.e. `ntoskrnl.exe` for Windows XP) or in a kernel module. If the interrupt handler is in a kernel module, `ptr` then points to the `Kinterrupt` data structure, which is a kernel control object that allows device drivers to register an interrupt handler for their devices. It contains information that the kernel needs to associate the interrupt handler with a particular interrupt, such as the base address of the interrupt handler in the module, vector number of the IDT entry etc. In order to determine whether the handler code is in a kernel module, we match the vector number in `Kinterrupt`

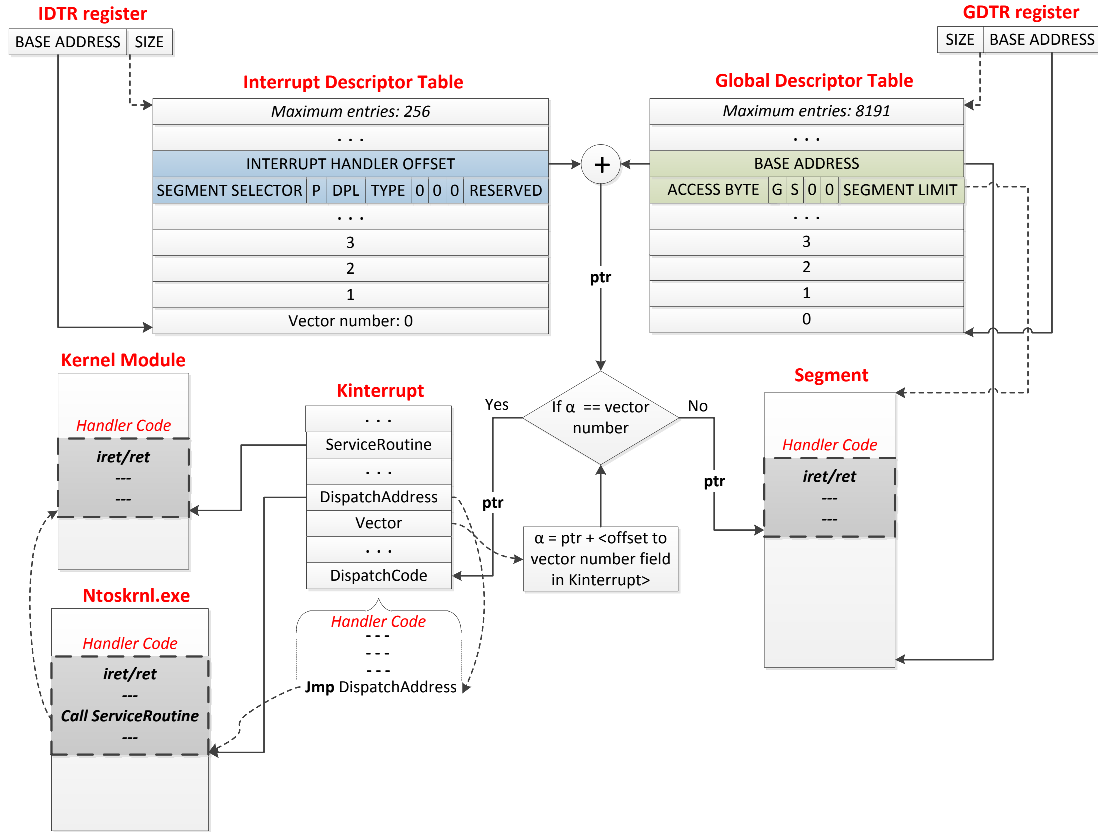

Figure 2: Code extraction of Interrupt gate. (The GDT/IDT descriptor format is adjusted for illustration purposes.)

structure with the vector number of the IDT entry. If matched, the handler code is in the kernel module; otherwise it is in the basic kernel.

At this stage, Code-Extractor needs to find the base address and size of all the interrupt handling code in order to make a clean extraction of the code. Figure 2 illustrates the entire extraction process.

- **Finding the base address:** When the code is located in the basic kernel, `ptr` contains the base address of the interrupt handler which is the only code needed for integrity checking. When the code is in a module, `ptr` points to `DispatchCode`, which executes and at some stage jumps to another code (`InterruptDispatcher`). This code further executes and at some stage calls the interrupt handler from the device driver. In this case, we have to extract three chunks of code. The base addresses of all the three pieces of code are in the `Kinterrupt` structure, which Code-Extractor processes to obtain the base addresses.

- **Finding the code size:** Code-Extractor finds the size of the code by disassembling it starting from the base address of the code, assuming that the first occurrence of a return instruction points to the end of the code. This assumption is valid through function prologue and epilogue convention, which is followed by assembly language programmers

and high language compilers. These are a few lines of code placed at the start and end of the function, which store the state of the stack and registers when the function is called and restore them at the time when function returns. Thus, a return instruction is required at the end of the function in order to ensure that the restoration code executes before function returns. We have not encountered situations where returns in interrupt handler code occur before the end of the handler. Furthermore, we performed a simple experiment to see if the Windows Driver Model (WDM) compiler strictly follows the function prologue and epilogue convention. We placed a few return instructions between the if-else statements in the interrupt handler code of a hello-world driver. When we compiled it, we found that the return instructions were replaced with jump instruction pointing to a return instruction placed at the end of the code. This shows that the WDM compiler follows the convention upon which our heuristic relies.

4.2.2. Task gate

Each (task gate) entry in the IDT has no interrupt handler offset, and therefore there is no direct pointer to any handler or code. Instead, the segment selector in the entry has the index of a GDT entry. The GDT entry is a task state segment (TSS) descriptor, which provides the information about the base address and the size (i.e., the segment limit) of a task state segment (TSS). The TSS stores the processor state information (such as segment registers and general-purpose registers etc.), which is required to execute the task. TSS contains the code segment (`CS`) that points to one of the descriptors in the GDT that defines a segment where the interrupt handler code is located. TSS contains the instruction pointer (`EIP`) value, as well. When a task is dispatched for execution, the information in TSS is loaded into the processor and the task execution begins with the instruction pointer (`EIP`) value, which provides the base address of the interrupt handler.

Once we have the base address of the interrupt handler, we use the same method that we used for interrupt gate to find the size of the code. When we disassemble the code, the first occurrence of the return instruction we considering to be the end of the interrupt handler code. Figure 3 illustrates the entire extraction process.

4.3. Info-Extractor

Info-Extractor is a generic module used to obtain any additional information associated with the IDT and its code and make it available to the Integrity-Checker. We use it to obtain the address range of the basic kernel and its associated modules that Integrity-Checker requires for Rule 3 and Rule 4. Info-extractor also takes into consideration other modules already loaded in memory.

Windows XP maintains a doubly linked list corresponding to locations of the basic kernel code and modules, where each element in the list is a `LDR_DATA_TABLE_ENTRY` data structure containing the base address `DllBase` and the size of the module `SizeOfImage`. Windows XP also stores the pointer to the first element of the list in a system variable `PsLoadedModuleList`, which Info-Extractor uses to reach the list, browse each element and store it into a local buffer. The pointer to the buffer is then forwarded to the Integrity-Checker. Figure 4 shows the doubly linked list.

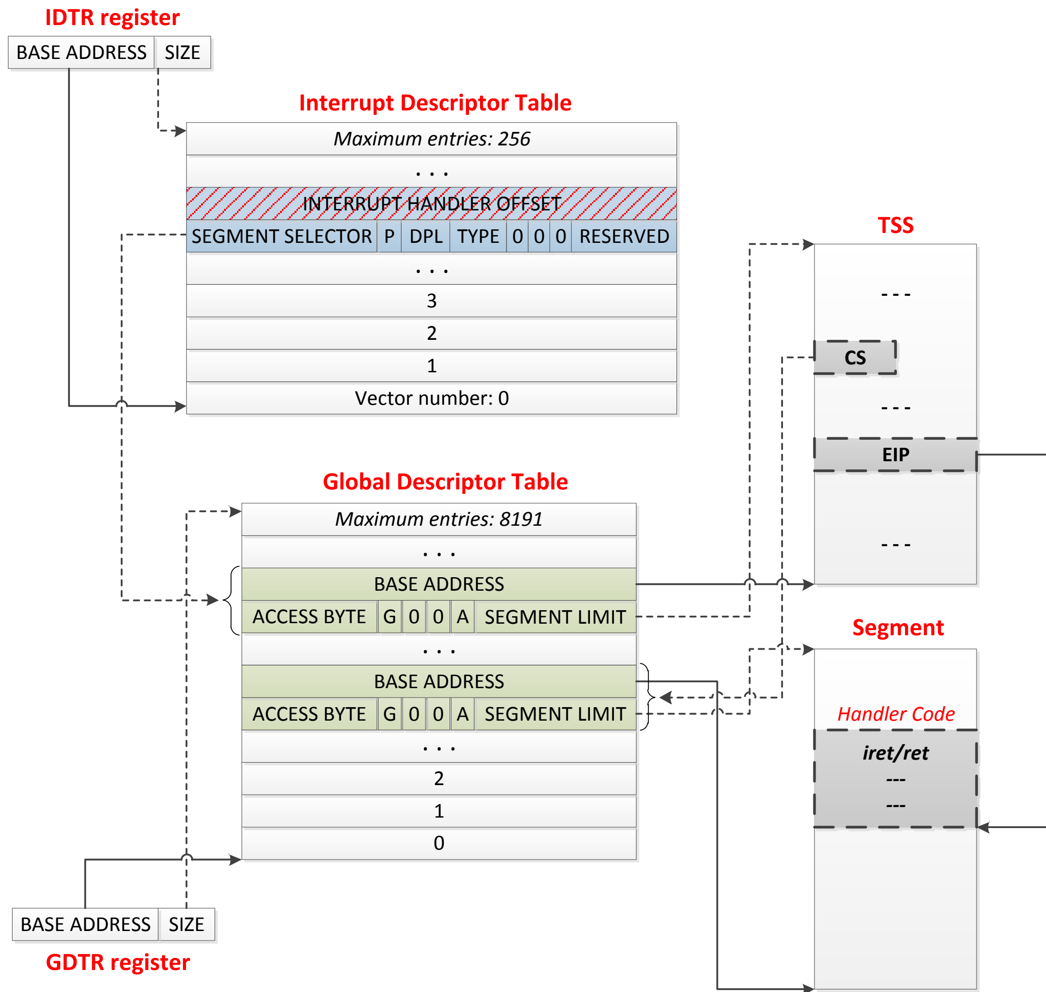

Figure 3: Code extraction of task gate. (The GDT/IDT descriptor format is adjusted for illustration purposes.)

4.4. Integrity-Checker

Integrity-Checker applies the rules to the data obtained from the last three modules. However, sometimes it needs to also manipulate the data in order to apply the rules. This section discusses both aspects of this component corresponding to each rule.

Rule 1: This rule compares IDTs across VMs. Integrity-Checker does this by comparing each value of every IDT entry across VMs. However, this does not include interrupt handler offsets.

Rule 2: This rule compares the interrupt handler code across VMs. At this stage, Integrity-Checker already has the handler code obtained by the Code-Extractor. However, since the code has been extracted from the memory of different VMs, it may not always be matched as is. This is because the code of the basic kernel and its modules in files have relative virtual addresses (RVAs) or offsets. When a module is loaded into memory, the loader replaces the RVAs with absolute addresses by adding the base address of the module

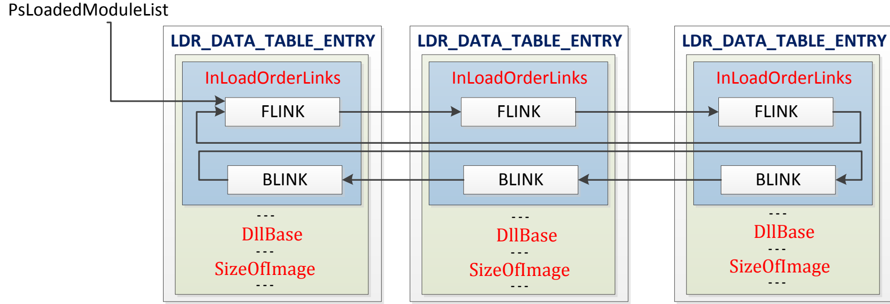

Figure 4: Doubly linked list of kernel modules

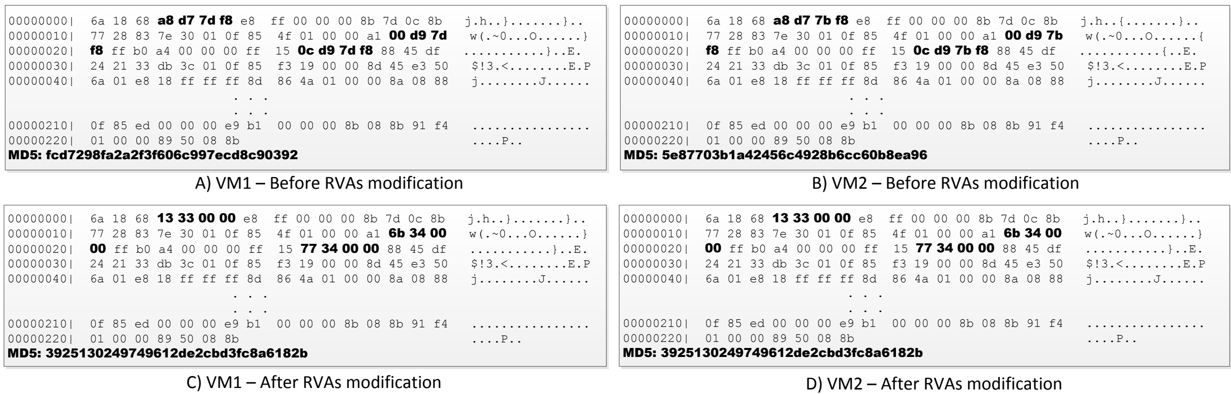

Figure 5: Relative virtual address (RVA) modification on interrupt handler (`i8042prt!I8042KeyboardInterruptService`), associated with interrupt vector `0x31`. The module's (32-bit) base address for virtual machines VM1 and VM2 are `F8 7B A4 95`' and `F8 7D A4 95`'.

(i.e. the pointer to the 0th byte of the module in memory) with RVAs. If the same module is loaded at different locations across VMs, the kernel/module's code (including the interrupt handler in the code) will have different absolute addresses and as a result is not consistent and will not be matched as is. Integrity-Checker reverses this change (using equation 1) by subtracting the absolute addresses in the code with their respective base addresses of the modules. This brings the absolute addresses back to RVAs, which represent the values in files and should be the same across VMs.

$$RVA = Absolute\ address\ (in\ the\ code\ ) -\ Base\ address\ (of\ Kermel\ or\ Module) \quad (1)$$

Figure 5 illustrates the RVAs modification. It shows the same interrupt handler code extracted from the physical memory of two virtual machines. Integrity-Checker assumes that the differences of bytes in the code represent the absolute addresses. This assumption is valid until the code in any of the VMs is modified since the base address of the module where the handler is located is different in both the VMs. There should be a difference of

4 bytes on a 32-bit machine. Depending on where the difference of bytes starts in the base address of the module in the two VMs, this may not always be the case.

Discussion: At this time, Integrity-Checker considers only the interrupt handler code for integrity checking, which is sufficient unless routines called by the handler are patched with malicious code. In this case, the integrity of the IDT is violated, even though the IDT table and its related interrupt handler code is still intact. Instead of finding such routines and checking their integrity, it is more efficient to check the integrity of the entire module where handler code is located. This may also include the routines that are not being called, however this approach still can save the parsing time to search for the calling functions. There are several existing techniques that check the integrity of entire modules [4][15] [16] [17] [18] [19] [20].

Rule 3 & Rule 4: Rules 3 and 4 check the base address of the interrupt handler code within the address range of kernel modules and check the offset of the handler's base address from the base address of its respective module. Integrity-Checker has the list of kernel modules (along with their address ranges) where the address ranges are exclusive and not overlapped. It searches the base address of interrupt handler to figure out whether it is within the address range of any module. Integrity-Checker uses binary search that requires the module's list to be sorted according to the base address. Moreover, merge sort is used for sorting. When the handler's base address is found within the address range of a module, the module is considered to be a holder of the interrupt handler. The module's base address is further used to compute the offset between the base addresses of the module and the handler, which is then matched across VMs.

5. Evaluation

We performed several experiments to evaluate the effectiveness and efficiency of the IDTchecker; the results are presented in this section.

5.1. Experimental settings

We built a small-scale cloud server for performing the experiments. The server had Xen 4.1.3 running on Intel core 2 Quad (4 * 2.83GHz cores) and 8GB RAM. We ran 7 VMs (i.e., DomUs) using Xen. Each VM had 1GB RAM, 10GB Hard Disk and was running MS Windows XP (Service Pack 2) using hardware-assisted virtualization (HVM). The privileged VM (i.e. Dom0) was running Fedora 16 (`3.4.9-2.fc16.x86_64` kernel).

5.2. Integrity checking

IDTchecker is designed to detect any integrity violation within the IDT and its corresponding interrupt handlers. This section discusses the experiments that violate the IDT integrity, and whether IDTchecker is able to detect them. These experiments also include real-world malware that manipulate the IDT to run malicious code.

5.2.1. Hooking an interrupt

As discussed earlier, each descriptor in the IDT has a (32-bit) pointer that either points to an interrupt handler or the `Kinterrupt` structure. It is formed from the two (16-bit) fields in the descriptor, which are the lower and higher 16-bits of the pointer address. There are

techniques [1] that exploit this pointer in order to redirect the control flow to a malicious code. For instance, while modifying the pointer, the technique stores the original pointer, which is then restored after the (malicious) code has executed. Moreover, a jump instruction is added to the (malicious) code that jumps to the original code after the malicious code is executed. This technique runs both the malicious and original code to support normal system operation.

In this experiment, we modified the pointer in the IDT in order to check whether IDTchecker could detect the modification. We used IDTGuard [9] for making this modification by using an implicit malfunctioning behavior of this tool. As discussed in the related work, this tool is designed to check the integrity of the IDT by separately computing the pointer values and comparing it to the one in the IDT. However, the computation is only possible when interrupt handlers are located in the kernel code (i.e. `ntoskrnl.exe`). When IDTGuard computes the pointer value that points to `Kinterrupt` structure (because the interrupt handler is in kernel module), it computes a pointer value of a random location in a kernel code. Thus, we used IDTGuard to replace the original pointer value in the IDT with this random pointer value and checked whether it was detected by IDTchecker. IDTChecker did detect this by showing that the pointer was pointing to a code different than those in other VMs.

5.2.2. Hooking an Interrupt handler

The interrupt handler can also be patched in order to run malicious code [1]. This change would not modify the pointer in the IDT but the actual code that is run to handle an interrupt. In this section, we performed a simple experiment where we used a customized driver for a simple Programmed IO device [21] that also had an interrupt handler. When the driver was loaded, the interrupt handler was registered with an interrupt vector. We used IDT entry (`0x3e`), which was originally registered with `atapi!IdePortInterrupt` handler. We then disabled the IDE Channel to free the system resources, which is then occupied by the Programmed IO device [21]. We further installed the driver for this device, which also registered its interrupt handler with vector `0x3e`. When we ran the IDTchecker and compared the IDT's across other VMs, it showed that the interrupt handler code for vector `0x3e` was different than those in other VMs.

5.2.3. IDT manipulation through malware

In this section, we describe the real-world malware and some popular IDT exploitation techniques we used to further evaluate the effectiveness of the IDTchecker. Malware modified the pointers in the IDT and the interrupt handler code in order to run their malicious code which IDT checker successfully detected.

Subverting the Windows Kernel: As pointed out by Skape [8], rootkits that directly replace IDT entries make a lot of noise and are, therefore, not stealthy. The proof of concept rootkit solutions that are largely undetectable by common rootkit scanners, however, rely on overwriting an interrupt handler such as the `KiInterruptTemplate` routine pointed by the Interrupt vector. Mxatone *et al.* [8] demonstrated a multipurpose proof of concept of attaching keylogging or packet sniffing code via IDT hooking. This process modifies the pointer in `KiInterruptTemplate` by searching for the following static code: "`mov edi, <&Kinterrupt>; jmp edi;`", to point to the maliciously crafted `Kinterrupt` structure which contains calls to the kernel routines that can gather keyboard strokes or

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT driver,IN PUNICODE_STRING path)
{
//save the original addresses of INT 1 and INT 3 handlers
    _asm
    {
        /////save INT1
        ……………
        /////save INT3
        ……………
    }

//hook INT 1 and INT 3 handlers

    a=KeNumberProcessors[0];
    while(1)
    {
        PsCreateSystemThread(&threadhandle,
                (ACCESS_MASK) 0L,0,0,0,
                (PKSTART_ROUTINE)HookIDT,0);
        KeWaitForSingleObject(&event,
           Executive,KernelMode,0,0);
        if(IdtsHooked==a)
            break;
    }
// proceed to overwriting memory
    _asm
    {
        //remove protection before overwriting
        mov eax,cr0
        push eax
        and eax,0xfffeffff
        mov cr0,eax

        //insert breakpoint (0xCC opcode)

        mov ebx,a
        mov al,0xcc

        mov byte ptr[ebx],al

        //restore protection
        pop eax
        mov cr0,eax
    }
    return 0;
}
```

```
void HookIDT()
{
    ULONG handler1,handler2,idtbase,tempidt,a;
    UCHAR idtr[8];
    //get the addresses that we have write to IDT
    handler1=(ULONG)&replacementbuff[0];
    handler2=(ULONG)&replacementbuff[32];
    tempidt=(ULONG)ExAllocatePool(NonPagedPool,2048);
      ……………
    for(a=0;a<IdtsHooked;a++)
    { …………//If IDT already hooked, re-enable interrupts and return
}
    _asm
    {  // load the copy of IDT into IDTR register
        ………………
        // modify IDT
        mov ecx,idtbase

        //hook INT 1
        add ecx,8
        mov ebx,handler1

        mov word ptr[ecx],bx
        shr ebx,16
        mov word ptr[ecx+6],bx

        ///hook INT 3
        add ecx,16
        mov ebx,handler2

        mov word ptr[ecx],bx
        shr ebx,16
        mov word ptr[ecx+6],bx

        //reload the original idt
        lea ebx,idtr
        mov eax,idtbase
        mov dword ptr[ebx+2],eax
        lidt idtr
        sti
    }//add the address of newly hooked IDT the list of hooked IDTs
    idtbases[IdtsHooked]=idtbase;
    IdtsHooked++;
    ExFreePool((void*)tempidt);
    KeSetEvent(&event,0,0);
    PsTerminateSystemThread(0);
}
```

Figure 6: Hooking the Kernel Directly [22]

network packets. The original interrupt handler will still execute after the malicious one since the malicious code will return to the legitimate `Kinterrupt` structure. After the modifications to the `Kinterrupt` structure, IDTChecker was able to detect the code injection by KinterruptTemplate pointer modification.

Direct Kernel Hooking: Another proof of concept [22] registers a dummy driver that hooks interrupt vectors `0x01` and `0x03` to functions that will represent a USB storage device as a regular disk drive. This proof of concept malware achieves this goal by capturing calls to `IoCreateDevice()` that takes a pointer into the `DRIVER_OBJECT` of the newly added device and replaces the MajorFunction (`IRP_MJ_DEVICE_CONTROL`) with the malicious function sitting within the dummy driver. As a result, every system call to the another USB device driver `USBSTOR` can be intercepted and monitored. In order to do so, the malware hooks `IoCreateDevice()` by inserting instructions into the executable code. Figure 6 is a breakdown of an IDT hooking function which is called directly from and contained within the dummy driver. The `hookIDT()` calling function preservers the old interrupts (`0x01` and `0x03`) that are about to be hooked.

As seen in Figure 6, after the original IDT has been preserved, it is then safe to unleash the hooking mechanism which hooks debugging interrupts `0x01` and `0x03`. Afterwords, the original IDT is restored and addresses of newly created hooks are added to the list of hooked IDT entries.

IDTchecker ran a comparative analysis of two virtual machines, where one of the VMs (VM1) had registered this malicious driver. The output generated by the IDTchecker illus-

```
lkd> !idt -a
Dumping IDT
 [.........]
2d:    8053d790 nt!KiDebugService
2e:    8053c651 nt!KiSystemService
2f:    8053f950 nt!KiTrap0F
30:    806d647c
31:    822b7044 f881a495 (KINTERRUPT 822b7008)
 [.........]
lkd> u 8053c651 l 10
nt!KiSystemService:
8053c651 6a00            push    0
8053c653 55              push    ebp
8053c654 53              push    ebx
8053c655 56              push    esi
8053c656 57              push    edi
8053c657 0fa0            push    fs
8053c659 bb30000000      mov     ebx,30h
8053c65e 668ee3          mov     fs,bx
8053c661 ff3500f0dfff    push    dword ptr ds:[0FFDFF000h]
8053c667 c70500f0dfffffffffff mov dword ptr ds:[0FFDFF000h],0FFFFFFFFh
8053c671 8b3524f1dfff    mov     esi,dword ptr ds:[0FFDFF124h]
8053c677 ffb640010000    push    dword ptr [esi+140h]
8053c67d 83ec48          sub     esp,48h
8053c680 8b5c246c        mov     ebx,dword ptr [esp+6Ch]
8053c684 83e301          and     ebx,1
8053c687 889e40010000    mov     byte ptr [esi+140h],bl
```

A) Uninfected IDT with Interrupt Handler Code

```
lkd> !idt -a
Dumping IDT
 [.........]
2d:    8053d790 nt!KiDebugService
2e:    f8bae2a0
2f:    8053f950 nt!KiTrap0F
30:    806d647c
31:    822c14d4 f87da495 (KINTERRUPT 822c1498)
32:    8053bd24 nt!KiUnexpectedInterrupt2
 [.........]
lkd> u f8bae2a0 l 10
f8bae2a0 60              pushad
f8bae2a1 9c              pushfd
f8bae2a2 0fa0            push    fs
f8bae2a4 66bb3000        mov     bx,30h
f8bae2a8 668ee3          mov     fs,bx
f8bae2ab 1e              push    ds
f8bae2ac 06              push    es
f8bae2ad 8bd8            mov     ebx,eax
f8bae2af e870030000      call    f8bae624
f8bae2b4 3b05c8ecbaf8    cmp     eax,dword ptr ds:[0F8BAECC8h]
f8bae2ba 7514            jne     f8bae2d0
f8bae2bc 3b1dccecbaf8    cmp     ebx,dword ptr ds:[0F8BAECCCh]
f8bae2c2 7d0c            jge     f8bae2d0
f8bae2c4 8915e0ecbaf8    mov     dword ptr ds:[0F8BAECE0h],edx
f8bae2ca 53              push    ebx
f8bae2cb e8b7000000      call    f8bae387
```

B) Infected IDT with Interrupt Handler Code

Figure 7: Comparison of IDT and Interrupt Handler dumps before and after `STrace Fuzen` malware infection (in WinDbg)

trates detection of the changes made to the interrupt `0x01` and `0x03` handler code by identifying that the dispatcher code size was mismatched. Furthermore, IDTChecker detected that the offset of the start address for the handler from driver's base address in infected VM1 had a value of `F7C477C0`. Keeping in mind that the maximum address range for the kernel functions is `F7C3A000`, the address we detected is obviously outside the address range of the kernel code and furthermore, does not match the value in VM2 `8053d4E4`, which is inside the address range of the kernel code. Further examination of the assembly dump provided by the IDTchecker, points to the expected assembly instructions overwritten by performing the interrupt hooking and is shown in the figure above.

STrace Fuzen Interrupt Hooking: In this experiment, we used `STrace Fuzen` [23] malware, which hooks the IDT on interrupt vector `0x2E`. This vector represents the System Service Descriptor Table (SSDT). When an application needs the assistance of the operating system via SSDT, `NTDLL.DLL` issues interrupt `0x2E` to transfer from user land to kernel land. This malware saves the address of the original interrupt handler and changes it to the address of its own code. When an application makes a request via the SSDT, this hook is called before the kernel function in the SSDT.

We used two identical VMs that had Windows XP (SP2) running. We ran the malware on one machine and noticed the changes through WinDbg (an MS Windows debugger) and then, compared them with the other uninfected machine. Figure 7 shows the results of the comparison where we noticed that the pointer for the `0x2E` vector was modified, along with the interrupt handler code in the infected machine. IDTchecker detected the modifications successfully.

5.3. Runtime Performance

This section discusses the runtime performance of IDTchecker when the guest VMs are idle or exhaustively using their resources. It also discusses the impact of IDTchecker on a guest VM's system resources and the memory overhead of IDTchecker in the privileged VM.

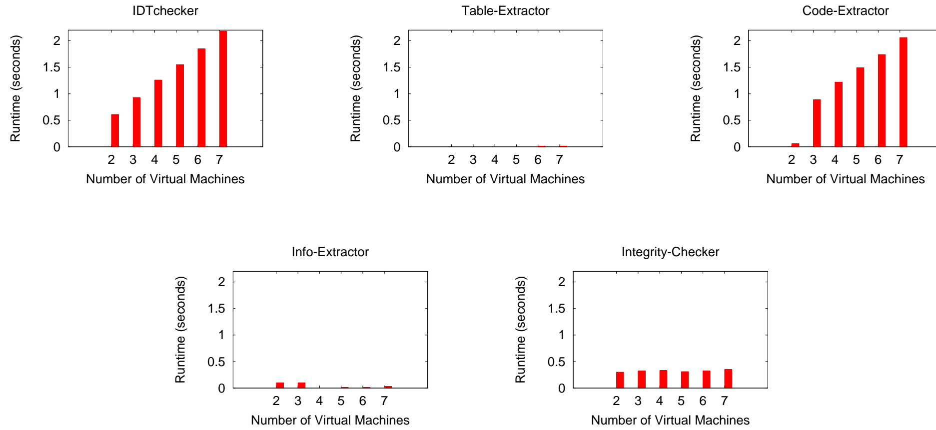

Figure 8: Execution time of IDTchecker (and its components) on different number of VMs when they are mostly idle.

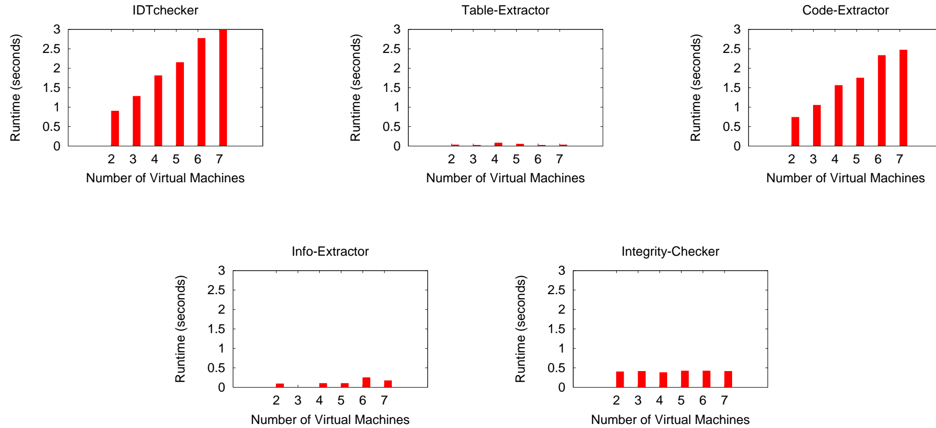

Figure 9: Execution time of IDTchecker (and its components) on different number of VMs when they are exhaustively using their resources.

5.3.1. Best and worst case scenarios of IDTchecker

We compare an IDT across a number of VMs and compute the best and worst running times. In the best case scenario, we kept the guest VMs idle so that IDTchecker would have all the available system resources. In the worst case scenario, we ran resource intensive processes that were intended to consume most of the system resources (such as RAM, IO and CPU) of guest VMs. Given the fact that all the VMs were running on the same hardware, this case would give IDTchecker fewer physical resources for execution.

Figures (8 & 9) show the runtime performance of IDTchecker and its components. Comparing both figures, we find that IDTchecker's components show similar runtime patterns. However, in the worst case scenario IDTchecker takes more time to execute. We also find that Code-Extractor consumed most of the resources. This is because, unlike the Table-Extractor and the Info-Extractor, the Code-Extractor has to access Guest VMs memory several times in order to retrieve different chunks of memory corresponding to interrupt vectors in the

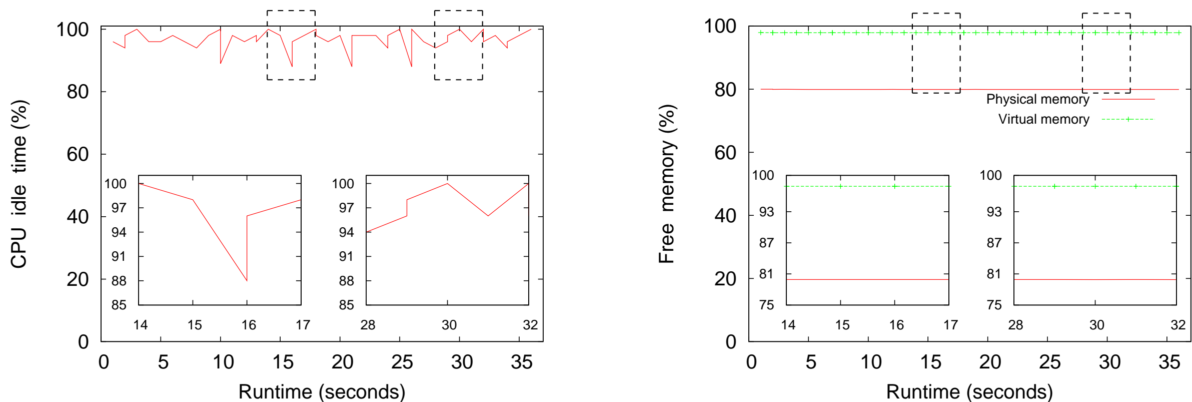

Figure 10: Inside virtual machine – CPU and memory impact of IDTchecker. Box represents the time span when IDTchecker was accessing the guest virtual machine's memory. (The boxes are zoomed-in in the small graphs).

IDT. For instance, if there are 100 interrupt gates in the IDT, Code-Extractor has to access memory 300 times. This is because there are three associated memory accesses per interrupt gate. On the other hand, the Table-Extractor has to access memory only twice; once to access the IDT and the second time to access the GDT. We also find a linear growth in the execution time of IDTchecker when we increase the number of VMs. This is because IDTchecker accesses the VMs sequentially, reading the memory of one VM at a time. This is the reason that Code-Extractor shows the same behavior as IDTchecker. Integrity-Checker showed consistent time when the number of VMs was increased. This is because Integrity-Checker does not access the guest VM's memory; instead, it only needs to apply simple rules to the VMs' data.

5.3.2. Inside virtual machine – IDTchecker's impact on system resources:

Recall that no component of IDTchecker runs inside the VM. It is for this reason we assume that there should not be any significant impact of IDTchecker on the guest VM's system resources. Figure 10 shows the processor and memory usage of an almost idle guest VM. The slight sign of disturbance is caused by a lightweight tool running to monitor the system resource usage from within the VM. The boxes point out the time frame when IDTchecker was running on the guest VMs extracting the tables and different memory chunks from VM's physical memory. After looking at the graphs, we find that there is no significant perturbation induced by IDTchecker on the guest VM's processor and memory resources. Thus, we can conclude that IDTchecker does not have significant impact on the guest VM's resources.

5.3.3. Memory overhead of IDTchecker

Figure 11 shows the memory overhead of IDTchecker on privileged VM running Fedora 16. There were around 500MB RAM available for IDTchecker since other seven guest VMs occupied 7GB of memory and around 500MB was occupied by the Fedora. In order to observe the memory usage of IDTchecker, we kept the machine idle. However, the tool that we used in the last section for acquiring the stats of the VM's system resources was running

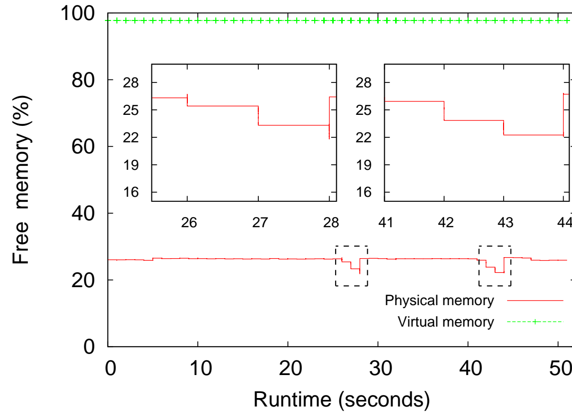

Figure 11: Memory overhead of IDTchecker on privileged VM. Box represents the time span when IDTchecker was running in the VM. (The boxes are zoomed-in in the small graphs).

on privileged VM. When we ran IDTchecker, we noticed 10-15MB perturbation, which is around 2-3 percent of the total available memory. We did not notice any usage of virtual memory, which is also shown in Figure 11. The perturbation is shown in the boxes and zoomed-in in the small graphs in Figure 11.

6. Conclusion

In this paper, we presented a comprehensive, rule-based approach, IDTchecker, to check the integrity of IDTs. Our approach examines the pointers in the IDT tables and the corresponding interrupt handler code that the pointers are intended to point to. We evaluated the effectiveness of IDTchecker by changing pointer values and the interrupt handler code in the IDT. To achieve this, we ran real-world malware that explicitly modified the pointers in the table and hooked the code with an interrupt vector using a customized kernel driver. The results showed that IDTchecker detected all the changes effectively.

We also evaluated IDTchecker's runtime performance during the best and worst case scenarios where all VMs were idle or were exhaustively using system resources. In both cases, we observed a linear growth in the execution time of IDTchecker when we increased the number of VMs. This is because IDTchecker accesses the VMs' memory sequentially. However, in the worst case scenario, IDTchecker took more time to execute because all the VMs were exhausting system resources, which left less physical resources for IDTchecker to run efficiently at its full capacity.

We also measured the impact of IDTchecker on guest VM's system resources (i.e., processing and memory usage) by keeping the VM idle and letting IDTchecker access the VM's memory through introspection. Since no component of IDTchecker runs inside the VMs, we did not notice any significant perturbation on guest VM's system resources during this span of time. Moreover, we also measured the memory overhead of IDTchecker on privileged VM and found that IDTchecker required only 10-15MB of memory to run smoothly.

IDTchecker is robust in that it runs outside the guest VMs and in case of any compromise of the guest VMs, an attacker cannot hinder the IDTchecker service unless it can attack the VMM directly and impact the privileged VM. IDTchecker is able to detect the change in IDTs even when most of the VMs are compromised. However, it cannot exactly identify which VMs are compromised. Thus, IDTchecker can be best utilized to detect the first signs of compromise, which then trigger a resource intensive forensic investigation to find the root cause of the problem.

Acknowledgment

This work was supported by the NSF grant CNS #1016807.

7. References

[1] Kad, Handling interrupt descriptor table for fun and profit, http://www.phrack.org/issues.html?issue=59&id=4&mode=txt.

[2] S. Skape, Bypassing patchguard on windows x64, http://uninformed.org/?v=3&a=3&t=sumry.

[3] Hacking with linux kernel module, http://newdata.box.sk/raven/lkm.html.

[4] Digital signatures for kernel modules, http://msdn.microsoft.com/enus/library/bb530195.aspx.

[5] Sd, Devik, Linux on-the-fly kernel patching without lkm, http://www.phrack.org/issues.html?id=7&issue=58.

[6] Volatility, http://code.google.com/p/volatility/.

[7] Volatility plugin, http://code.google.com/p/volatility/source/browse/trunk/volatility/plugins/linux/check_idt.py?spec=svn2273&r=2273.

[8] Mxatone, Ivanlef0u, Stealth hooking : Another way to subvert the windows kernel, http://www.phrack.org/issues.html?issue=65&id=4.

[9] IDTGuard, http://www.msuiche.net/2006/12/10/idtguard-v01-december-2005-build/.

[10] Intel 64 and IA-32 architectures software developers manual, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[11] Xen, http://www.xen.org/.

[12] LibVMI, http://code.google.com/p/vmitools/.

[13] Opdis, http://mkfs.github.com/content/opdis/.

[14] OpenSSL, http://www.openssl.org/.

[15] J. Rutkowska, System virginity verifier defining the roadmap for malware detection on windows system, in: Hack in the Box, 2005.

[16] T. Garnkel, M. Rosenblum, A virtual machine introspection based architecture for intrusion detection, in: Symposium on Network and Distributed System Security (NDSS), 2003.

[17] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, C. D. McDonell, Linux kernel integrity measurement using contextual inspection, in: ACM workshop on Scalable trusted computing, STC 07, 2007, p. 2129.

[18] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, P. Khosla, Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems, in: Twentieth ACM Symposium on Operating Systems Principles, 2005, pp. 1–16.

[19] G. Kroah-Hartman, Signed kernel modules, Linux Journal 117 (2004) 48–53.

[20] I. Ahmed, A. Zoranic, S. Javaid, G. G. Richard III, Modchecker: Kernel module integrity checking in the cloud environment, in: ACM symposium on Applied computing, 4th International Workshop on Security in Cloud Computing (CloudSec '12), 2012.

[21] W. Oney, Programming the Microsoft Windows Driver, second edition Edition, Microsoft Press, New York, 2002.

[22] Hooking the kernel directly, `http://www.codeproject.com/Articles/13677/Hooking-the-kernel-directly`.

[23] J. Butler, G. Hoglund, Rootkits: Subverting the Windows Kernel, Addison-Wesley, Boston, 2005.