Rule-Based Integrity Checking of Interrupt Descriptor Tables in Cloud Environments

Irfan Ahmed, Aleksandar Zoranic, Salman Javaid, Golden Richard Iii, Vassil Roussev

Chapter 21

RULE-BASED INTEGRITY CHECKING OF INTERRUPT DESCRIPTOR TABLES IN CLOUD ENVIRONMENTS

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

Abstract An interrupt descriptor table (IDT) is used by a processor to transfer the execution of a program to software routines that handle interrupts raised during the normal course of operation or to signal an exceptional condition such as a hardware failure. Attackers frequently modify IDT pointers to execute malicious code. This paper describes the IDTchecker tool, which uses a rule-based approach to check the integrity of the IDT and the corresponding interrupt handling code based on a common scenario encountered in cloud environments. In this scenario, multiple virtual machines (VMs) run the same version of an operating system kernel, which implies that IDT-related code should also be identical across the pool of VMs. IDTchecker leverages this scenario to compare the IDTs and the corresponding interrupt handlers across the VMs for inconsistencies based on a pre-defined set of rules. Experimental results related to the effectiveness and runtime performance of IDTchecker are presented. The results demonstrate that IDTchecker can detect IDT and interrupt handling code modifications without much impact on guest VM resources.

Keywords: Cloud forensics, interrupt descriptor table, integrity checking

1. Introduction

Memory forensics involves the extraction of digital artifacts from the physical memory of a computer system. The interrupt descriptor table (IDT) is a valuable artifact, more so because it 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, a special software routine that handles exceptional conditions occurring within the system, processor or executing program. (A hardware failure and division by zero are examples of unusual conditions that are handled by an IDT.) Malware often manipulates an IDT to change the system control flow and run malicious code. The change may occur either to a pointer in the IDT or in the interrupt handler itself, which then redirects execution to malicious code that has been injected into the system.

PatchGuard [6, 18] checks IDT integrity by keeping a valid state of the table and comparing it with the current table state. Microsoft includes kernel patch protection (PatchGuard) [18] in 64-bit Windows systems to detect modifications to kernel code and critical data structures such as the IDT. PatchGuard caches the legitimate copy and the checksum of the IDT, and compares them with the current IDT in memory to check for modifications. Another tool, CheckIDT [6], examines IDT integrity by storing the entire table in a file so that it can be compared later with the current state of the table in memory.

While these tools are applicable to different operating systems, they suffer from at least two major limitations. Both tools require an initialization phase, where it is assumed that the IDT is not infected at the time that the valid state of the IDT is obtained. This may not be the case if the interrupt handler is patched in the kernel file on disk because, when the system restarts and the IDT is created, the IDT points 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, the tools need to record the valid state of the table.

Current solutions do not specifically consider the interrupt handler code for integrity checking. Although they do check the integrity of the kernel code and modules that include interrupt handler 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 the maintenance of a dictionary of cryptographic hashes of trusted code [9, 14] in order to compare the hash of the current code with the hash stored in the dictionary. Such an approach requires maintaining the dictionary across every kernel update to implement effective integrity checking.

This paper describes the IDTchecker tool, which provides a comprehensive, rule-based approach to check IDT integrity 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. The pools of virtual machines simplify the maintenance process to facilitate the automation of patch applications and system upgrades. IDTchecker works by retrieving the IDT and its corresponding interrupt handler code from the physical memory of a guest VM and comparing it across the VMs in the pool. The tool uses a pre-defined set of rules to perform comprehensive integrity checking. It runs on a privileged virtual machine where it has access to guest VM physical memory through virtual machine introspection (VMI). None of the components of IDTchecker run inside the guest VMs, which makes IDTchecker more resistant to tampering by malware.

IDTchecker was evaluated extensively in order to assess its effectiveness and efficiency. Effectiveness testing used real-world malware and popular IDT exploitation techniques to modify the IDT and interrupt handler code. Efficiency testing analyzed the runtime performance of IDTchecker under the best case and worst case scenarios. The results demonstrate that IDTchecker does not have any significant impact on guest VM resources because none of its components run inside the VMs. Its memory footprint is 10 to 15 MB, which is quite negligible compared with the amount of physical memory typically found in a cloud server (tens to hundreds of gigabytes).

2. Related Work

This section discusses research related to IDT integrity checking.

CheckIDT [6] is a Linux-based tool that detects IDT modifications by storing the IDT descriptor values in a file and later comparing them with the current values of the IDT in memory. If a discrepancy between the two tables is detected, CheckIDT restores the table in memory by copying the IDT values from the saved file, assuming that the integrity of the file has been maintained. CheckIDT uses the technique described in [16] to access the table from user space without using a Linux kernel module.

Kernel Patch Protection [18] (or PatchGuard) checks the integrity of kernel code (including modules) and important data structures such as the IDT, global descriptor table (GDT) and system service descriptor table (SSDT). It is currently implemented in 64-bit Windows operating systems. PatchGuard stores legitimate known-good copies and checksums of kernel code and data structures, and compares them with the current state of the code and the data structures at random times. PatchGuard is implemented as a set of routines that are protected by using anonymization techniques such as misdirection, misnamed functions and general code obfuscation.

Volatility [21] has a plugin [22] that checks the integrity of IDT pointers to interrupt handlers. It walks through each IDT entry and checks if the pointer in the entry is within the address range of the kernel code (including modules). Volatility ensures that the pointers do not point to unusual locations in memory. However, it cannot detect attacks [6, 10] that directly patch interrupt handler code and do not modify pointers in the table.

IDTGuard [19] is a Windows-based tool that checks the integrity of IDT pointers. The tool separately computes an IDT pointer value by finding the offset of the interrupt handler in the kernel file (such as `ntoskrnl.exe`) and adding it to the base address of the kernel in memory. The computed IDT pointer values are then matched with the pointers in the IDT table to verify their integrity. However, IDTGuard cannot check the integrity of pointers corresponding to kernel modules where the pointers point to `Kinterrupt` data structures instead of interrupt handlers in the kernel code.

3. IDT Integrity Checking

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 the hardware and the hosted guest operating systems. The VMM also allows a privileged VM to monitor the runtime resources of other (guest) VMs (e.g., memory and I/O) through virtual machine introspection. IDTchecker uses introspection while running on a privileged VM to access the physical memory of guest VMs. It retrieves the IDTs and their corresponding interrupt handlers and matches them according to pre-defined rules in order to check for inconsistencies.

3.1 Overview

Interrupts and exceptions are system events that indicate that a condition or an event requires the attention of the processor [5]. Interrupts can be generated in response to hardware signals such as hardware failure or by software through the `INT n` instruction. Exceptions are generated when the processor detects an error during the execution of an instruction such as divide by zero. Each condition indicated by an interrupt or exception requires special handling by the processor and, thus, is represented using a unique identification number referred to as an interrupt vector. In this paper, the difference between interrupts and exceptions is not important and, thus, both are referred to as interrupts. When an interrupt occurs, the execution of a program is suspended and the control flow is redirected to an interrupt handler routine through an IDT.

An IDT is an array of interrupt vectors, each vector providing an entry point to an interrupt handler. There can be at most 256 interrupt vectors. Each vector is eight bytes long and contains 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 functionality and in the type field in the descriptor that identifies the gate. Unlike the situation for trap gates, when handling an interrupt gate, the processor clears the IF flag in the EFLAGS register to prevent other interrupts from interfering with the current interrupt handler. Task gate descriptors, on the other hand, have no offset values to the interrupt handler. Instead, the interrupt handler is reached through the segment selector field in the descriptor.

The global descriptor table (GDT) is utilized when the interrupt handler has to be accessed in protected mode (where a protection ring [5] is enforced). An entry in the table is called a segment descriptor. Each descriptor describes the base address and the size of a memory segment along with information related to the access rights of the segment. Each descriptor is also associated with a segment selector that provides information about the index to the descriptor, access rights and a flag that determines if 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 a case of interrupt and trap gates, the base address of the interrupt is obtained by adding the base address of the segment to the offset in the vector.

3.2 Assumptions

We assume the presence of a fully-virtualized environment where the VMM supports memory introspection of guest VMs. Also, we assume that different pools of VMs are present, 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

IDTchecker is designed to obtain the IDTs and corresponding interrupt handler code from a pool of VMs and perform a comprehensive integrity check based on a pre-defined set of rules. To achieve this task, IDTchecker employs four modules to implement the various functions needed to perform integrity checking. The four components are: (i) Table-Extractor; (ii) Code-Extractor; (iii) Info-Extractor; and (iv) Integrity-Checker.

- **Table-Extractor and Code-Extractor:** IDTchecker incorporates separate modules (Table-Extractor and Code-Extractor) for extracting tables and interrupt related code because the IDT and GDT structures are dependent on the processor, while the organization of the interrupt-related code is mostly dependent on the operating system. For instance, Microsoft Windows uses a `Kinterrupt` structure to store the information about an interrupt handler that is provided by kernel drivers. Separating the extraction of code and tables into two modules increases the portability of IDTchecker. Moreover, Code-Extractor receives the interrupt vector descriptor values from Table-Extractor after the descriptors are parsed. This data is used by Code-Extractor to locate the index of a GDT segment and the offset of an interrupt handler (if the descriptor type is not a task gate).

- **Info-Extractor:** The Info-Extractor module fetches any additional information required by a rule from memory, such as the address range of kernel modules.

- **Integrity-Checker:** The Integrity-Checker module applies a predefined set of rules to the data obtained from the Table-Extractor, Code-Extractor and Info-Extractor modules in order to comprehensively check IDT integrity. Unlike the other three modules, Integrity-Checker does not need to access the memory of guest VMs. This is because all the needed data is made available by the other modules.

Figure 1 presents the overall architecture of IDTchecker. The figure shows multiple pools of guest VMs, each pool running the same version of a guest operating system. The VMs run on top of a VMM and the VMI facility is available for a privileged VM to introspect guest VM resources. Note that IDTchecker only needs to perform read-only operations on guest VM physical memory and no IDTchecker component runs inside a guest VM.

By comparing IDTs and their corresponding interrupt handler code across VMs, IDTchecker is able to detect IDT inconsistencies between the VMs. A majority vote algorithm is used to identify an infected VM. Of course, the majority vote strategy is effective only if the majority of VMs have uninfected IDTs. In this case, IDTchecker is more effective

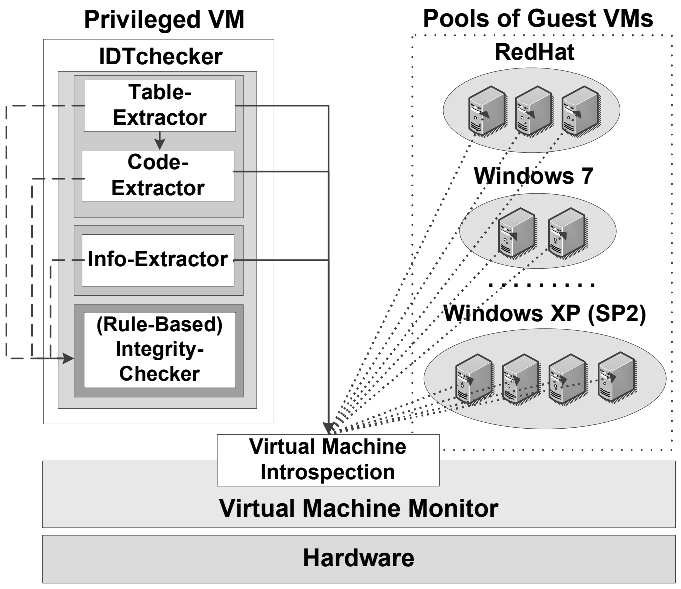


Figure 1. IDTchecker architecture.

at detecting the first sign of infection, which can then be used to trigger a thorough forensic investigation to find the root cause of the infection.

It is also worth discussing if the IDT should always be identical across VMs when identical kernel code (including modules) is executing. The initial 32 interrupt vectors (0 to 31) are pre-defined. These interrupt vectors always remain identical across VMs. However, other interrupt vectors (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 interrupt entries may not be feasible at all times. A more robust approach is to find the equivalent interrupt vector entry that is being matched in other IDT tables across VMs before the rules are applied to them. The current version of IDTchecker performs one-to-one matching and can be enhanced using this approach.

3.4 Integrity Checking Rules

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 rules). This rule ensures that all the fields in the IDT are original.

- **Rule 2:** The interrupt handler code should be consistent across VMs. This rule detects modifications to the code. The rule is effective unless identical modifications are made to the code in all the VMs.

- **Rule 3:** The interrupt handler is located in basic kernel code or in a kernel module. This means that the base address of the interrupt handler should be within the address range of the basic kernel code or within the address range of the code of a module. This rule ensures that the base address does not point 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 of a 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 VMs. This rule detects instances of random injections of malicious code within the basic kernel code or within a driver.

4. IDTchecker Implementation

The IDTchecker design is simple in that all its components reside locally on a privileged VM, which can be implemented on any VMM that has memory introspection support (e.g., Xen, KVM or VMware ESX) without requiring modifications to the VMM itself. For the proof of concept, we developed IDTchecker on Xen [23] with the Microsoft Windows (Service Pack 2) XP guest operating system. We used the LibVMI introspection library [20] and the Opdis disassembler library [12]; cryptographic hashes were computed using OpenSSL [13].

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

4.1 Table-Extractor

The IDT and GDT are created each time a system starts. The processor stores their base addresses and sizes 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 memory. It further interprets the raw bytes of the tables as table entries and their respective fields, and forwards them to Code-Extractor.

4.2 Code-Extractor

Code-Extractor receives the tables from Table-Extractor and retrieves the code corresponding to each IDT entry. Code-Extractor handles each interrupt vector type (interrupt gate, task gate and trap gate) differently. Since no trap gate entries are found in Windows XP VMs, only the interrupt and task gate extraction are discussed below.

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 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, then `ptr` points to the `Kinterrupt` data structure, which is a kernel control object that allows device drivers to register an interrupt handler for their devices. The data structure contains information that the kernel needs to associate the interrupt handler with a particular interrupt (e.g., the base address of the interrupt handler in the module and the vector number of the IDT entry). In order to determine if the handler code is in a kernel module, the vector number in the `Kinterrupt` structure is matched with the vector number of the IDT entry. If the two values match, 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 addresses and sizes of all the interrupt handling code segments in order to make a clean extraction of the code. Figure 2 shows the extraction process. Note that the IDT and GDT descriptor formats in the figure are adjusted for illustrative purposes.

- **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 point jumps to other code (`InterruptDispatcher`). This code executes and at some point calls the interrupt handler from the device driver. In this case, three chunks of code have to be extracted. The base addresses of the three pieces of code are in the `Kinterrupt` structure, which Code-Extractor processes to obtain the addresses.

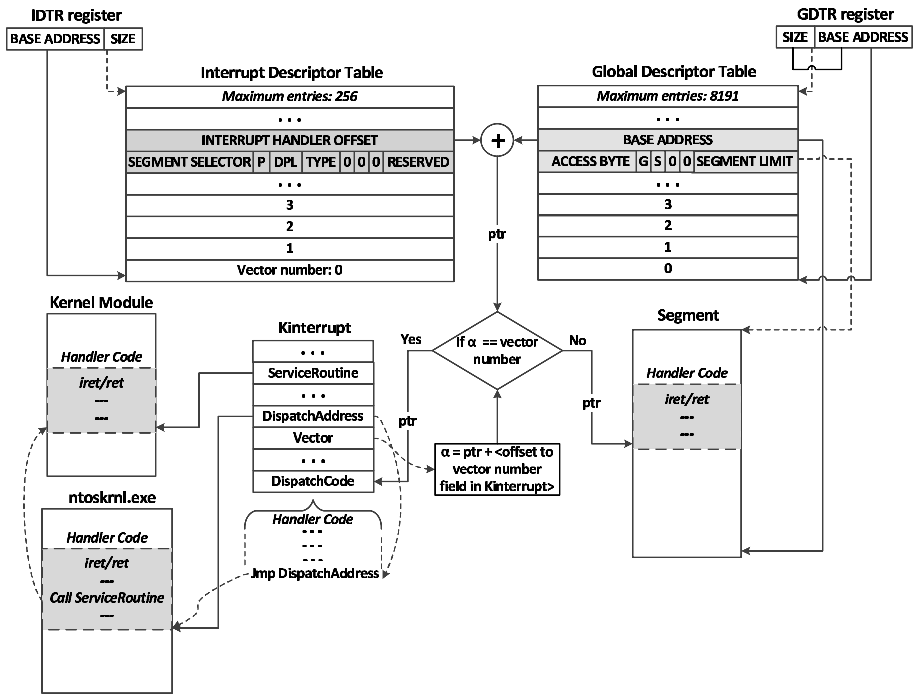


Figure 2. Code extraction of an interrupt gate.

- **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 based on the function prologue and epilogue convention, which is followed by assembly language programmers and high-level language compilers. The function prologue and epilogue are small amounts of code placed at the start and end of a function, respectively; the prologue stores the state of the stack and registers when the function is called and the epilogue restores them when the function returns. Thus, a return instruction is required at the end of a function in order to ensure that the restoration code executes before function returns.

  We have not encountered a situation where a return in interrupt handler code occurs before the end of the handler. Also, we performed an experiment to see if the Windows Driver Model (WDM) compiler 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. After compiling the code, we discovered that the return instructions were replaced with jump instructions pointing to return instructions 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 a handler or code. Instead, the segment selector in the entry is the index of a GDT entry. The GDT entry is a task state segment (TSS) descriptor that provides information about the base address and size (i.e., segment limit) of a TSS. The TSS stores the processor state information (e.g., segment registers and general purpose registers) that is required to execute the task. The TSS also 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. Additionally, the TSS contains the instruction pointer (`EIP`) value. When a task is dispatched for execution, the information in the TSS is loaded into the processor and task execution begins with the instruction pointer (`EIP`) value, which provides the base address of the interrupt handler.

After locating the base address of the interrupt handler, the process for determining the code size is the same as that used for interrupt gates. Figure 3 shows the extraction process. Note that the IDT and GDT descriptor formats in the figure are adjusted for illustrative purposes.

4.3 Info-Extractor

The Info-Extractor module is used to obtain additional information associated with the IDT and its code and make it available to Integrity-Checker. In addition, the module obtains the address range of the basic kernel and its associated modules that Integrity-Checker requires for Rules 3 and 4. Info-Extractor also takes into consideration the other modules that are already loaded in memory.

Windows XP maintains a doubly-linked list (Figure 4) corresponding to the locations of the basic kernel code and modules, where each element in the list is a `LDR_DATA_TABLE_ENTRY` data structure that contains 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 in a local buffer. The pointer to the buffer is then forwarded to Integrity-Checker.

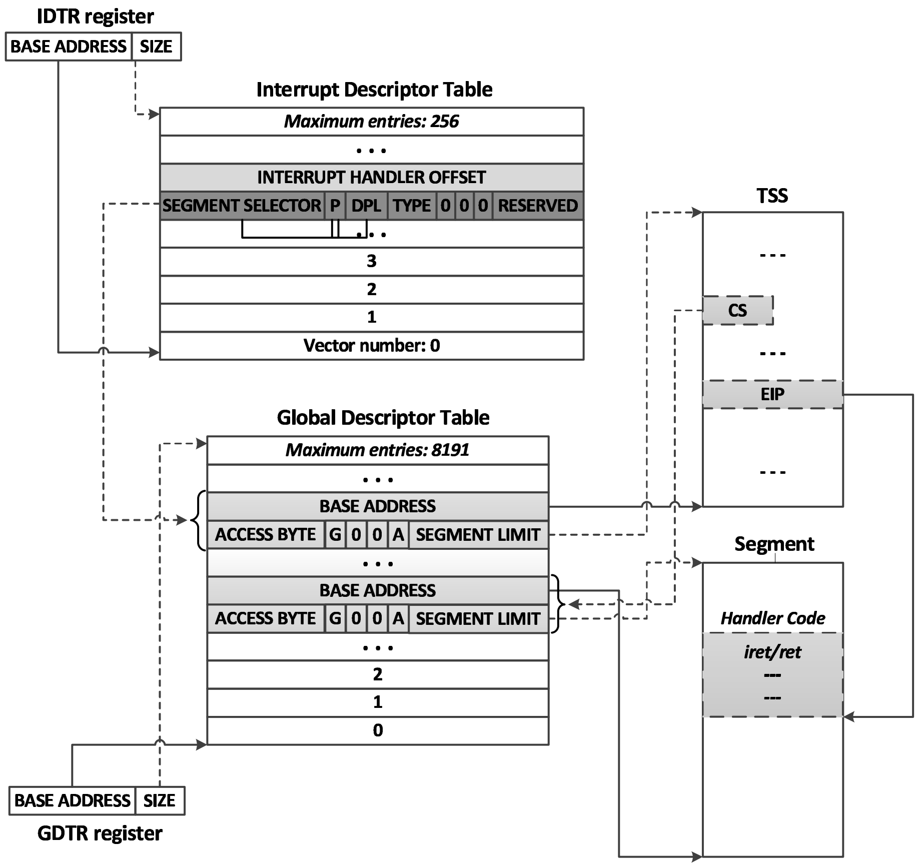


Figure 3. Code extraction of a task gate.

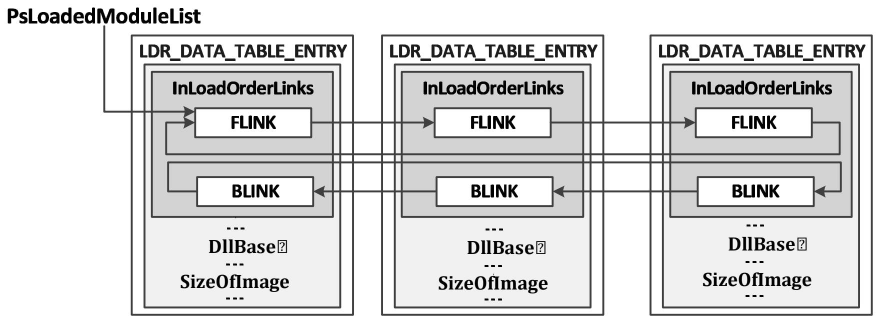


Figure 4. Doubly-linked list of kernel modules

4.4 Integrity-Checker

Integrity-Checker applies the rules to the data obtained from the other three modules. However, Integrity-Checker sometimes needs to also manipulate the data in order to apply the rules. We discusses both these aspects of Integrity-Checker with regard 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. Integrity-Checker uses the interrupt handler code obtained by Code-Extractor. However, because the code has been extracted from the memory of different VMs, it may not always match. The reason is that the code of the basic kernel and its modules in the 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 (i.e., the pointer to the zeroth byte of the module in memory) to the RVAs. If the same module is loaded at different locations across VMs, the kernel/module code (including the interrupt handler in the code) will have different absolute addresses and, as a result, will not be consistent and will not match. Integrity-Checker reverses this change by subtracting the base addresses of the modules from the absolute addresses in the code. This brings the absolute addresses back to RVAs, which represent the values in files and should be the same across VMs.

  Figure 5 illustrates the RVA modification of the interrupt handler `i8042prt!I8042KeyboardInterruptService` associated with interrupt vector `0x31`. The 32-bit base addresses of the module for virtual machines VM1 and VM2 are `F8 7B A4 95` and `F8 7D A4 95`, respectively. The figure shows the same interrupt handler code extracted from the physical memory of the two VMs. Integrity-Checker assumes that the differences of bytes in the code represent the absolute addresses. This assumption is valid until the code in one of the VMs is modified because the base address of the module containing the handler is different for the two VMs. Ideally, there should be a difference of four bytes for a 32-bit machine. However, 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.

```
00000000|  6a 18 68 a8 d7 7d f8 e8  ff 00 00 00 8b 7d 0c 8b   j.h..}.......}..
00000010|  77 28 83 7e 30 01 0f 85  4f 01 00 00 a1 00 d9 7d   w(.~0...O......}
00000020|  f8 ff b0 a4 00 00 00 ff  15 0c d9 7d f8 88 45 df  ...........}..E.
00000030|  24 21 33 db 3c 01 0f 85  f3 19 00 00 8d 45 e3 50   $!3.<........E.P
00000040|  6a 01 e8 18 ff ff ff 8d  86 4a 01 00 00 8a 08 88   j........J......
                            . . .
                            . . .
00000210|  0f 85 ed 00 00 00 e9 b1  00 00 00 8b 08 8b 91 f4   ................
00000220|  01 00 00 89 50 08 8b                               ....P..
MD5: fcd7298fa2a2f3f606c997ecd8c90392
```

(a) VM1 before RVA modification.

```
00000000|  6a 18 68 a8 d7 7b f8 e8  ff 00 00 00 8b 7d 0c 8b   j.h..{.......}..
00000010|  77 28 83 7e 30 01 0f 85  4f 01 00 00 a1 00 d9 7b   w(.~0...O......{
00000020|  f8 ff b0 a4 00 00 00 ff  15 0c d9 7b f8 88 45 df  ...........{..E.
00000030|  24 21 33 db 3c 01 0f 85  f3 19 00 00 8d 45 e3 50   $!3.<........E.P
00000040|  6a 01 e8 18 ff ff ff 8d  86 4a 01 00 00 8a 08 88   j........J......
                            . . .
                            . . .
00000210|  0f 85 ed 00 00 00 e9 b1  00 00 00 8b 08 8b 91 f4   ................
00000220|  01 00 00 89 50 08 8b                               ....P..
MD5: 5e87703b1a42456c4928b6cc60b8ea96
```

(b) VM2 before RVA modification.

```
00000000|  6a 18 68 13 33 00 00 e8  ff 00 00 00 8b 7d 0c 8b   j.h..}.......}..
00000010|  77 28 83 7e 30 01 0f 85  4f 01 00 00 a1 6b 34 00   w(.~0...O......}
00000020|  00 ff b0 a4 00 00 00 ff  15 77 34 00 00 88 45 df   ...........}..E.
00000030|  24 21 33 db 3c 01 0f 85  f3 19 00 00 8d 45 e3 50   $!3.<........E.P
00000040|  6a 01 e8 18 ff ff ff 8d  86 4a 01 00 00 8a 08 88   j........J......
                            . . .
                            . . .
00000210|  0f 85 ed 00 00 00 e9 b1  00 00 00 8b 08 8b 91 f4   ................
00000220|  01 00 00 89 50 08 8b                               ....P..
MD5: 3925130249749612de2cbd3fc8a6182b
```

(c) VM1 after RVA modification.

```
00000000|  6a 18 68 13 33 00 00 e8  ff 00 00 00 8b 7d 0c 8b   j.h..}.......}..
00000010|  77 28 83 7e 30 01 0f 85  4f 01 00 00 a1 6b 34 00   w(.~0...O......}
00000020|  00 ff b0 a4 00 00 00 ff  15 77 34 00 00 88 45 df   ...........}..E.
00000030|  24 21 33 db 3c 01 0f 85  f3 19 00 00 8d 45 e3 50   $!3.<........E.P
00000040|  6a 01 e8 18 ff ff ff 8d  86 4a 01 00 00 8a 08 88   j........J......
                            . . .
                            . . .
00000210|  0f 85 ed 00 00 00 e9 b1  00 00 00 8b 08 8b 91 f4   ................
00000220|  01 00 00 89 50 08 8b                               ....P..
MD5: 3925130249749612de2cbd3fc8a6182b
```

(d) VM2 after RVA modification.

Figure 5. VM1 and VM2 before and after RVA modification.

Currently, Integrity-Checker considers only the interrupt handler code for integrity checking, which is sufficient unless the routines called by the handler are patched with malicious code. In this case, IDT integrity is violated although the IDT table and its related interrupt handler code are still intact. Instead of finding such routines and checking their integrity, it is more efficient to check the integrity of the entire module where the handler code is located.

This may also include the routines that are not being called, but this approach can reduce the time required to search for the calling functions. Several techniques have been proposed for checking the integrity of entire modules (see, e.g., [1, 4, 7–9, 15, 17]).

- **Rules 3 and 4:** Rules 3 and 4 check the base address of the interrupt handler code in the address range of the kernel modules and check the offset of the handler base address from the base address of its respective module. Integrity-Checker has the list of kernel modules and their address ranges (where the address ranges are exclusive and do not overlap). Integrity-Checker searches the base address of interrupt handler to check if it is within the address range of a module. It uses a binary search that requires the list associated with a module to be sorted according to the base address. When the handler base address is found in the address range of a module, the module is considered to be a holder of the interrupt handler. The module base address is also used to compute the offset between the base addresses of the module and the handler, which is then matched across VMs.

5. Evaluation

This section presents the results of several experiments that evaluated the effectiveness and efficiency of IDTchecker.

5.1 Experimental Setup

We built a small-scale cloud server for the experiments. The server ran Xen 4.1.3 on an Intel Core 2 Quad (4 × 2.83 GHz cores) with 8 GB RAM. We created seven VMs (i.e., DomUs) using Xen. Each VM had 1 GB RAM, a 10 GB hard disk and ran Windows XP (Service Pack 2) using hardware-assisted virtualization. The privileged VM (i.e., Dom0) ran Fedora 16 with the `3.4.9-2.fc16.x86_64` kernel.

5.2 Integrity Checking

IDTchecker is designed to detect integrity violations in the IDT and its corresponding interrupt handlers. This section presents the results of experiments that violated IDT integrity in various ways. The experiments employed real-world malware to manipulate the IDT and execute malicious code.

5.2.1 Hooking an Interrupt. Each IDT descriptor has a 32-bit pointer that points to an interrupt handler or the `Kinterrupt` structure. Each pointer is formed from the two 16-bit fields in the descriptor, which are the lower and higher 16 bits of the pointer address. Techniques are available to exploit this pointer to redirect control flow to malicious code [6].

This experiment modified the IDT pointer and tested if IDTchecker could detect the modification. An implicit malfunctioning behavior of the IDTGuard tool [19] was used to effect the modification. As discussed in Section 2, this tool is designed to check IDT integrity by separately computing the pointer values and comparing them with the values in the IDT. However, the computation is only possible when the interrupt handlers are located in the kernel code (i.e., `ntoskrnl.exe`). When IDTGuard computes the value of the pointer to `Kinterrupt` (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 the random pointer value. IDTchecker was able to detect the modification by showing that the code pointed to by the pointer was different from the code pointed to by the pointers in the other VMs.

5.2.2 Hooking an Interrupt Handler .

An interrupt handler can also be patched in order to run malicious code [6]. This change would not modify the pointer in the IDT, but the actual code that is executed to handle an interrupt. This experiment used a customized driver for a programmed I/O device [11] with an interrupt handler. When the driver was loaded, the interrupt handler was registered with an interrupt vector. We used an IDT entry (`0x3e`) that was originally registered with the `atapi!IdePortInterrupt` handler. Next, we disabled the IDE channel to free system resources to hold the programmed I/O device [11]. We then installed the driver for this device, which also registered the interrupt handler with vector `0x3e`. IDTchecker detected the modification by comparing the IDTs across the other VMs and showing that the interrupt handler code for vector `0x3e` was different from the corresponding code in the other VMs.

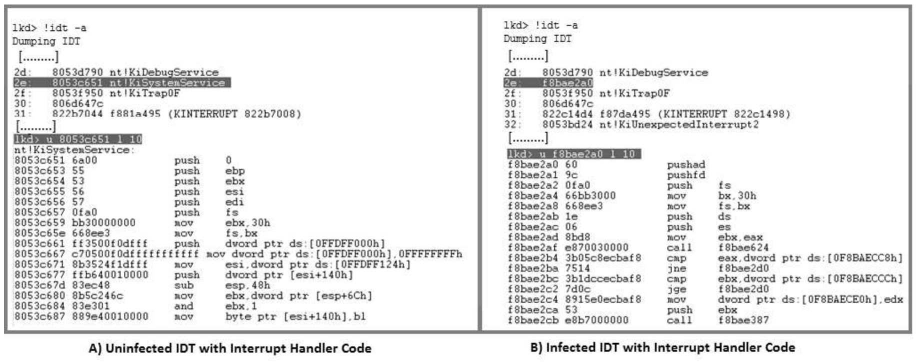
5.2.3 IDT Manipulation via Malware.

Real-world malware and IDT exploitation techniques can modify IDT pointers and interrupt handler code in order to run malicious code. Three experiments were conducted to test the performance of IDTchecker in the face of IDT manipulation via malware.

- **Subverting the Windows Kernel:** As discussed by Skape [18], rootkits that directly replace IDT entries leave many traces and are, therefore, not stealthy. Rootkits that are largely undetectable by common scanners rely on overwriting an interrupt handler such as the `KiInterruptTemplate` routine pointed to by the interrupt vector. mxatone and ivanlef0u [10] have demonstrated how to attach keylogging or packet sniffing code via IDT hooking. Their technique searches for the code "`mov edi, <&Kinterrupt>; jmp edi;`" and modifies the pointer in `KiInterruptTemplate` to point to the maliciously crafted `Kinterrupt` structure that contains calls to the kernel routines that can gather keyboard strokes or network packets. The original interrupt handler will still execute after the malicious interrupt handler because the malicious code returns to the legitimate `Kinterrupt` structure. After modifications were made to the `Kinterrupt` structure, IDTchecker was able to detect the code injection by `KinterruptTemplate` pointer modification.

- **Direct Kernel Hooking:** A proof-of-concept malware [2] registers a dummy driver that hooks interrupt vectors `0x01` and `0x03` to functions that represent a USB storage device as a regular disk drive. This is done by capturing calls to `IoCreateDevice()` that take a pointer to the `DRIVER_OBJECT` of the newly-added device and replaces the MajorFunction (`IRP_MJ_DEVICE_ CONTROL`) with the malicious function sitting in the dummy driver. As a result, every system call to the 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. An IDT hooking function contained within the dummy driver is called directly from the dummy driver. The `hookIDT()` calling function preserves the old interrupts (`0x01` and `0x03`) that are to be hooked. After the original IDT has been preserved, the hooking mechanism can be unleashed, which hooks the debugging interrupts `0x01` and `0x03`. After this is done, the original IDT is restored and the addresses of newly-created hooks are added to the list of hooked IDT entries.

  IDTchecker ran a comparative analysis of two VMs, where one of the VMs (VM1) had registered the malicious driver. IDTchecker detected the changes made to the interrupt handler code for `0x01` and `0x03` by identifying that the dispatcher code size was mismatched. Furthermore, IDTchecker detected that the offset of the start address of the handler from the driver base address in the infected VM1 had a value of `F7C477C0`. Since the maximum address range for the kernel functions was `F7C3A000`, the address detected was outside the address range of the kernel code and, furthermore, did not match the value `8053d4E4` in VM2, which was



```
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 6. IDT and interrupt handler dumps.

inside the address range of the kernel code. Further examination of the assembly dump provided by the IDTchecker revealed that the expected assembly instructions were overwritten by the interrupt hooking.

- **STrace Fuzen Interrupt Hooking:** This experiment used the `STrace Fuzen` [3] malware, which hooks the IDT on the system service descriptor table (SSDT) interrupt vector `0x2E`. When an application needs the assistance of the operating system via SSDT, `NTDLL.DLL` issues interrupt `0x2E` to transfer from user space to kernel space. The 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, the hook is called before the kernel function in the SSDT.

  We used two identical VMs that ran Windows XP (SP2). The malware was executed on one machine and the changes were observed using the WinDbg Windows debugger; the changes were then compared with those in the uninfected machine. Figure 6 shows the IDT and interrupt handler dumps in WinDbg before and after the `STrace Fuzen` malware infection. Note that the pointer for the `0x2E` vector and the interrupt handler code in the infected machine were modified. IDTchecker successfully detected both the modifications.

5.3 Runtime Performance

This section discusses the runtime performance of IDTchecker for guest VMs that were idle and for VMs that were exhaustively using their resources. It also discusses the impact of IDTchecker on guest

VM resources along with the memory overhead of IDTchecker in the privileged VM.

5.3.1 Best Case and Worst Case Scenarios. The best and worst running times for IDTchecker were identified by tests using idle and fully-loaded VMs. For the best case scenario, the guest VMs remained idle so that IDTchecker would have all the available system resources. In the worst case scenario, the guest VMs executed resource-intensive processes that consumed most of the system resources (CPU, RAM and I/O), leaving IDTchecker very limited physical resources for execution.

Figures 7 and 8 show the execution times of IDTchecker and its components for different numbers of idle VMs and fully-loaded VMs, respectively. The figures show similar runtime patterns for IDTchecker components, with Code-Extractor consuming most of the resources. This is because, unlike Table-Extractor and Info-Extractor, Code-Extractor has to access guest VM memory several times in order to retrieve different chunks of memory corresponding to interrupt vectors in the IDT. For instance, if there are 100 interrupt gates in the IDT, then Code-Extractor has to access memory 300 times, once for each of the three associated memory elements per interrupt gate. On the other hand, Table-Extractor has to access memory only twice: once to access the IDT and the second time to access the GDT.

Linear growth is observed in the execution time of IDTchecker as the number of VMs is increased. This is because IDTchecker accesses the VMs sequentially, reading the memory of one VM at a time. This is also the reason why Code-Extractor shows the same behavior as IDTchecker. On the other hand, the execution time of Integrity-Checker remains constant as the number of VMs is increased. This is because Integrity-Checker does not access the guest VM memory, and only needs to apply the four rules to the processed VM data.

5.3.2 Impact on Guest VM Resources. Because the components of IDTchecker execute outside a guest VM, there should be a minimal performance impact on the guest VM resources. Figures 9 and 10 show the processor and memory usage for an almost idle guest VM. The boxes in the figures show zoomed-in portions from the original graphs when IDTchecker was accessing guest VM memory. The slight sign of disturbance is caused by the monitoring of system resource usage from within the VM. The boxes correspond to the time frames when IDTchecker was running on the guest VM and extracting the tables and memory chunks from the physical memory of the VM. The graphs show

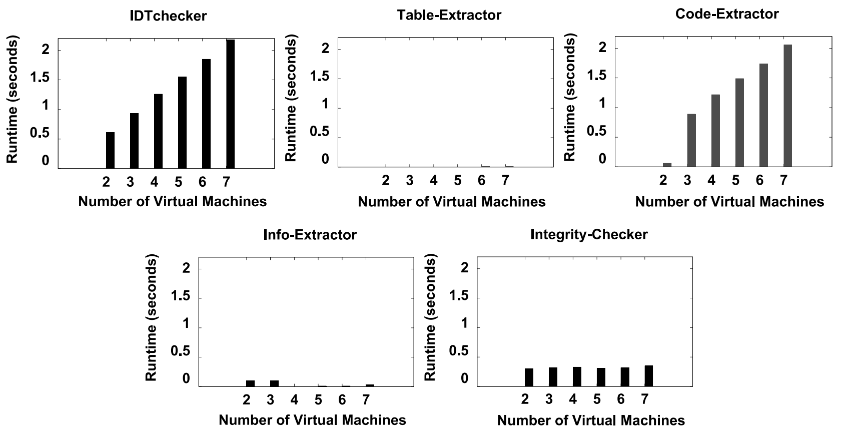


Figure 7. Execution times of IDTchecker and its components for an idle VM.

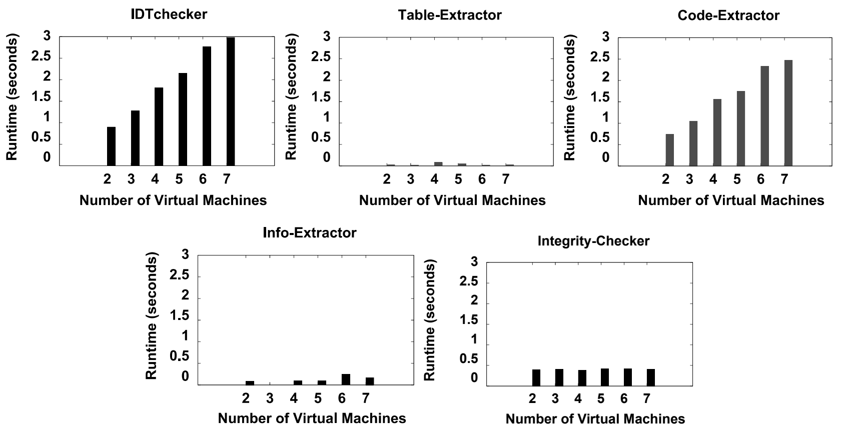


Figure 8. Execution times of IDTchecker and its components for a fully-loaded VM.

that no significant perturbations are induced by IDTchecker on the processor and memory resources of the guest VM. Thus, we can conclude that IDTchecker does not have a significant impact on guest VM resources.

5.3.3 Memory Overhead. Figure 11 shows the memory overhead of IDTchecker on a privileged VM running Fedora 16. The boxes correspond to the time frames when IDTchecker was running on the VM, and they show zoomed-in portions from the original graphs. Approximately 500 MB RAM was available to IDTchecker because the other seven guest VMs occupied 7 GB of memory and the Fedora operating system occupied 500 MB. During the tests, the machine was idle with

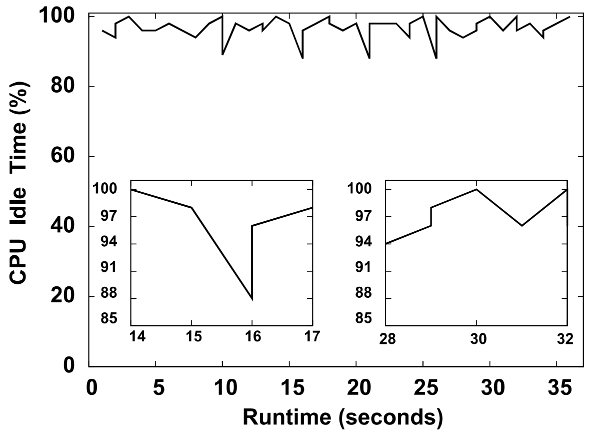


Figure 9. IDTchecker CPU usage.

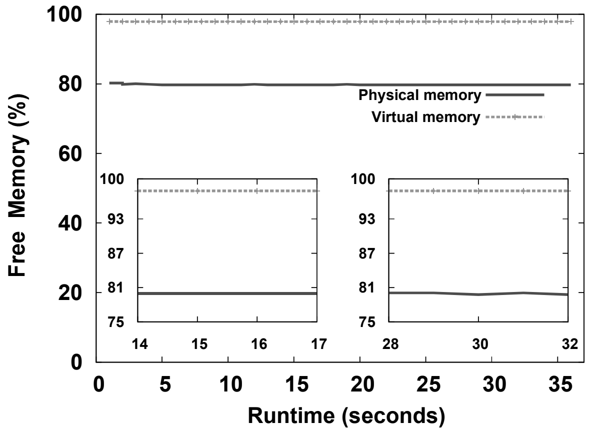


Figure 10. IDTchecker memory usage.

the exception of the usage monitoring process. A 10 to 15 MB perturbation in memory usage was caused by IDTchecker – this is just 2 to 3% of the total available memory. No usage of virtual memory was observed. The boxes in Figure 11 show the zoomed-in portions of the perturbations related to physical memory usage.

6. Conclusions

The IDTchecker tool is designed to check the integrity of IDTs and the corresponding interrupt handling code in guest VMs running in cloud environments. IDTchecker provides alerts when a VM is compromised using a majority vote based on the outputs of a set of pre-defined rules. However, IDTchecker cannot identify exactly which VM is compromised.

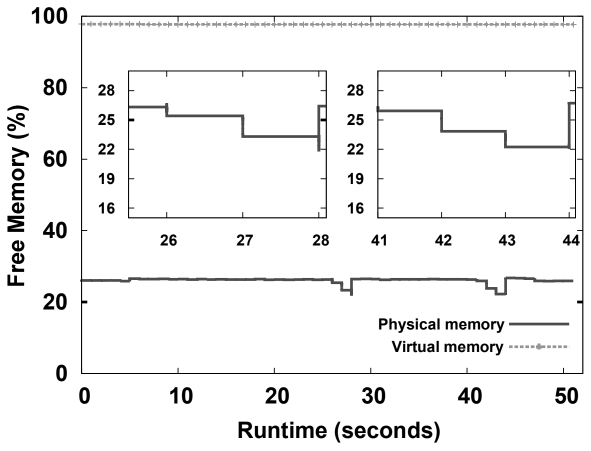


Figure 11. IDTchecker memory overhead for a privileged VM.

Thus, the tool is best used to detect the first signs of compromise, which can then trigger a resource-intensive forensic investigation to find the root cause of the problem.

Experiments demonstrate that IDTchecker is effective at detecting modifications to pointer values and interrupt handler code. Runtime performance testing of IDTchecker shows linear growth in execution time as the number of VMs is increased. Also, IDTchecker has a minimal impact on guest VM resources such as processor and memory.

Acknowledgement

This research was supported by NSF Grant No. CNS 1016807.

References

[1] I. Ahmed, A. Zoranic, S. Javaid and G. Richard III, ModChecker: Kernel module integrity checking in the cloud environment, *Proceedings of the Forty-First International Conference on Parallel Processing Workshops*, pp. 306–313, 2012.

[2] A. Bassov, Hooking the kernel directly (`www.codeproject.com/Articles/13677/Hooking-the-kernel-directly`), 2006.

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

[4] T. Garfinkel and M. Rosenblum, A virtual machine introspection based architecture for intrusion detection, *Proceedings of the Network and Distributed System Security Symposium*, pp. 191–206, 2003.

[5] Intel, Intel 64 and IA-32 Architectures Software Developer's Manuals, Santa Clara, California (`www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`), 2013.

[6] Kad, Handling the interrupt descriptor table for fun and profit, *Phrack*, vol. 0x0b(0x3b), 2002.

[7] G. Kroah-Hartman, Signed kernel modules, *Linux Journal*, vol. 2004(117), article no. 4, 2004.

[8] P. Loscocco, P. Wilson, J. Pendergrass and C. McDonell, Linux kernel integrity measurement using contextual inspection, *Proceedings of the Second ACM Workshop on Scalable Trusted Computing*, pp. 21–29, 2007.

[9] Microsoft, Digital Signatures for Kernel Modules on Windows, Redmond, Washington (`msdn.microsoft.com/en-us/library/windows/hardware/gg487332.aspx`), 2007.

[10] mxatone and ivanlef0u, Stealth hooking: Another way to subvert the Windows kernel, *Phrack*, vol. 0x0c(0x41), 2008.

[11] W. Oney, *Programming the Microsoft Windows Driver Model*, Microsoft Press, Redmond, Washington, 2002.

[12] Opdis Project, Opdis (`mkfs.github.com/content/opdis`).

[13] OpenSSL Core and Development Team, OpenSSL Cryptography and SSL/TLS Toolkit (`www.openssl.org`), 2009.

[14] pragmatic, (Nearly) complete Linux loadable kernel modules: The definitive guide for hackers, virus coders and system administrators (`newdata.box.sk/raven/lkm.html`), 1999.

[15] J. Rutkowska, System virginity verifier: Defining the roadmap for malware detection in Windows systems, presented at the *Hack in the Box Conference*, 2005.

[16] sd and devik, Linux on-the-fly kernel patching without LKM, *Phrack*, vol. 0x0b(0x3a), 2001.

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

[18] S. Skape, Bypassing PatchGuard on Windows x64 (`uninformed.org/?v=3&a=3&t=sumry`), 2005.

[19] M. Suiche, IDTGuard v0.1 December 2005 Build (`www.msuiche.net/2006/12/10/idtguard-v01-december-2005-build`), 2005.

[20] VMI Tools Project, LibVMI (`code.google.com/p/vmitools`).

[21] Volatility Project, The Volatility Framework (code.google.com/p/volatility).

[22] Volatility Project, Volatility Plugin (code.google.com/p/volatility/source/browse/trunk/volatility/plugins/linux/check_idt.py?spec=svn2273&r=2273).

[23] Xen Project, Xen, Cambridge, United Kingdom (www.xenproject.org).