ModChecker: Kernel Module Integrity Checking in the Cloud Environment

Irfan Ahmed, Aleksandar Zoranic, Salman Javaid, Golden G. Richard III
Dept. of Computer Science, University of New Orleans,
Lakefront Campus, New Orleans, LA 70148, United States
Email: {iahmed4, azoranic, sjavaid, golden}@uno.edu

***Abstract*—Kernel modules are an integral part of most operating systems (OS) as they provide flexible ways of adding new functionalities (such as file system or hardware support) to the kernel without the need to recompile or reload the entire kernel. Aside from providing an interface between the user and the hardware, these modules maintain system security and reliability. Malicious kernel level exploits (e.g. code injections) provide a gateway to a system's privileged level where the attacker has access to an entire system. Such attacks may be detected by performing code integrity checks. Several commodity operating systems (such as Linux variants and MS Windows) maintain signatures of different pieces of kernel code in a database for code integrity checking purposes. However, it quickly becomes cumbersome and time consuming to maintain a database of legitimate dynamic changes in the code, such as regular module updates. In this paper we present `ModChecker`, which checks in-memory kernel modules' code integrity in real time without maintaining a database of hashes. Our solution applies to virtual environments that have multiple virtual machines (VMs) running the same version of the operating system, an environment commonly found in large cloud servers. ModChecker compares kernel module among a pool of VMs within a cloud. We thoroughly evaluate the effectiveness and runtime performance of ModChecker and conclude that ModChecker is able to detect any change in a kernel module's headers and executable content with minimal or no impact on the guest operating systems' performance.**

***Keywords*-Xen; cloud computing; kernel module; malware; code integrity; virtual machine;**

I. Introduction

Dynamic kernel modules are used to extend the functionality of the static kernel within an operating system, such as adding support for new hardware. These modules, commonly known as drivers, can be dynamically attached or detached from the kernel without restarting the system or recompiling the kernel. By retaining only the primary functionality that is required to run an operating system, the static kernel can remain small [1]. Most modern operating systems such as Linux variants (FreeBSD or Solaris) and Microsoft (MS) Windows support dynamic loadable kernel modules. This provides more flexibility than the traditional method of recompiling the kernel to add additional functionality while retaining the same capabilities as the code compiled into the kernel [1]. Nonetheless, dynamic kernel modules can be exploited to add malicious functions to the kernel or subvert the entire operating system. For instance, a rootkit [2] acts like a kernel module and can hide directories, files, processes and network connections. Therefore, in order to prevent the kernel from being compromised, a kernel module must be trusted and its integrity must not be compromised.

A state-of-the art solution maintains a dictionary of cryptographic hashes of trusted kernel modules [3], [4] and verifies the validity of a kernel module by matching its hash with the stored value. For example, MS Windows registers and maintains digital signatures for kernel modules that MS Windows uses to verify the module's integrity upon loading it into memory [3]. MS Windows does not, however, check modules for malicious contents and does not use the signatures to perform any integrity checking after the module is loaded in memory. Furthermore, it is cumbersome to maintain the dictionary for kernel updates [5], third party drivers, and valid customized modules.

In this paper, we present `ModChecker`, which examines the integrity of kernel modules in real time without maintaining a dictionary of hashes. ModChecker works in virtualized environments where multiple VMs have the same operating system running. By comparing a module's hash value among the VMs, ModChecker verifies that module's integrity. Only modules actually loaded in memory are checked. The module file contains relative virtual addresses that the module loader replaces with corresponding absolute addresses when it is loaded into memory. The absolute address is computed by adding the relative virtual address to module's base address (i.e., the address of the first byte of the memory loaded module). Since the module is often relocated once loaded in memory, its absolute addresses vary accordingly. This makes computing the hash of a complete in-memory module unsuitable. Thus, ModChecker extracts all the headers and read-only executable contents of a module and computes their hashes separately. ModChecker was evaluated in a cloud computing environment [6]. This cloud provided us with a realistic environment where multiple identical VMs are running at the same time [7]. To benchmark ModChecker, several kernel exploitation techniques were used to exploit several system modules under MS Windows XP SP2. The results show that ModChecker can detect various types of kernel exploits with minimal impact on a guest operating system's performance.

The rest of the paper is organized as follows: Section II summarizes the related work. Section III presents ModChecker and details its architectural assumptions. Section IV outlines the implementation and section V is the evaluation. Section VI concludes the paper.

II. Related work

There has been a large volume of work published on kernel integrity monitoring, specifically focusing on control flow integrity, as well as data and code integrity. This section covers the existing approaches and tools that are most related to ModChecker.

Rutkowska proposes System-Virginity-Verifier (SVV) [8] that verifies the integrity of the in-memory code section of system drivers and Dynamic Link Libraries (DLL). It compares the in-memory code to the corresponding (reference) Portable Executable (PE) file on disk. SVV and ModChecker are similar in that they are based on a cross-view approach and share the same goal of verifying the integrity of the in-memory modules. Unlike ModChecker, however, SVV must be deployed within the system. Moreover, most malware infects files on disk first, and then loads the infected file into memory [9]. Therefore, SVV cannot pinpoint the infection when both memory and the file contain the same infected code. ModChecker, on the other hand, is more robust since it uses the cross-view from a separate, isolated system. Unless all the VMs in a cloud are compromised, ModChecker can effectively detect code integrity violations.

Commodity operating systems, such as MS Windows and several Linux variants, enable digitally signed kernel modules [3], [4]. This means that they compute and maintain a database of cryptographic hash values for kernel modules. The operating systems use this hash value to verify the integrity of the module before it is loaded into memory. However, this mechanism neither checks the module for malicious content when the module is initially signed nor does it guarantee integrity of the module after it is loaded into memory.

Garfinkel *et al.* propose LIVEWIRE [10], which is a Virtual Machine Monitor (VMM) based intrusion detection system that has the capability of checking the integrity of well known user programs such as sshd, ietd and syslogd. Livewire keeps a hash of a known good program and periodically compares it with the hash of in-memory code sections to detect code integrity violations. The Livewire approach is also applicable to well known kernel modules and dynamically linked libraries. However, it cannot be generalized due to the fundamental requirement of having a good hash of the in-memory code.

Loscocco *et al.* present Linux Kernel Integrity Monitor (LKIM) [11], which verifies the integrity of static kernel code with cryptographic hashes and examines the dynamic data structures to verify the integrity of function pointers. LKIM does not maintain a list of module hashes to check kernel module integrity, since the modules are relocated at runtime and the addresses of key data structures cannot be known until relocation. To solve this problem, LKIM modifies the Linux kernel that supplies it with modules' loading information (i.e., its name and the addresses of each section). In order to evaluate a module's integrity, LKIM uses this information along with the untainted copy of the code to simulate the loading process.

Arvind *et al.* propose Pioneer [12], which verifies code integrity and ensures that code remains intact during execution inside an untrusted environment. Pioneer is based on a challenge-response protocol between a dispatcher (an external trusted entity) and the untrusted computing environment. The dispatcher sends a challenge in order to invoke a self-checking function that computes a checksum of the code and sends it back to the dispatcher. The dispatcher has a copy of the code that it uses to verify the checksum is correct and is received in an expected amount of time. This guarantees the dispatcher that a "dynamic root of trust" exists within the untrusted environment.

Neugschwandtner *et al.* propose *d*Anubis [13], which dynamically analyzes malicious MS Windows device drivers. It includes integrity checking of drivers by placing them under supervision. In case of integrity violation, it matches the kernel function addresses with the ones obtained from Windows debugging symbols. The difference in addresses helps in identifying the function that has been patched.

III. Kernel-Module Integrity Checker

In a fully virtualized environment such as a Cloud [14], a VMM is the heart of that environment [7]. A VMM allows multiple operating systems to run on VMs that are inherently running on the same computer hardware concurrently. This type of infrastructure allows one VM to monitor various runtime resources (memory, disk etc.) of other VMs through virtual machine introspection (VMI) [15]. While running under a main, privileged virtual machine, ModChecker acquires memory access of other virtual machines through VMI and cross-matches the contents of a module across a pool of VMs in order to verify the integrity of a module. The efficacy of ModChecker is based on the following realistic assumptions.

A. Assumptions

We assume a typical cloud computing environment consisting of hardware, VMM (or hypervisor such as Xen [4]), guest VMs and a privileged VM (where ModChecker accesses the memory contents of the guest VMs through introspection).

B. ModChecker architecture

ModChecker is designed to be simple, effective and easily deployable. It runs on a privileged VM and through introspection it performs read-only operations of the memory of guest VMs.

Figure 1 shows the architecture of ModChecker that has three components: Module-Searcher, Module-Parser and Integrity-Checker.

1) Module-Searcher: Module-Searcher is the only component of ModChecker that accesses the memory of guest VMs. It finds the list of active modules and then looks for the module that is being checked for integrity violation. If the module is in memory, ModChecker extracts the entire module from the guest VM's memory and passes it to the Module-Parser that extracts the headers and executable contents.

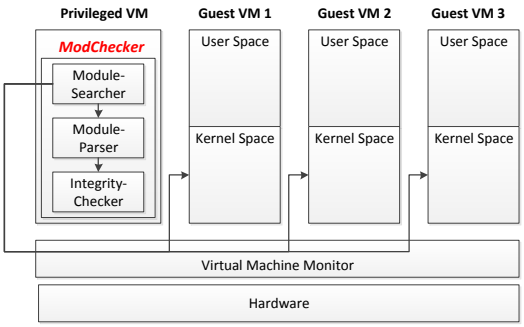


Fig. 1. ModChecker Architecture

2) Module-Parser: Module-Parser processes the module after receiving it from Module-Searcher. The module in general maintains a specific format when it is loaded into memory, that specifies its header and content information. The format depends on the operating system platform. For instance, in MS Windows, modules have portable executable (PE) format. The Module-Parser extracts headers and executable contents and passes them to Integrity-Checker that evaluates their integrity.

3) Integrity-Checker: Integrity-Checker has two primary functions. Firstly, it computes the hashes of the headers and the contents of the module and compares them to the same module loaded into other VMs. Secondly, it adjusts the relative virtual addresses (RVAs) in the executable content. RVA is the offset from the base address (the first byte of the module loaded into the memory). Once the kernel module loader loads a module, it adds the base address to the RVAs to get their absolute virtual addresses. The loader replaces the RVAs with their absolute addresses and that makes the same executable content inconsistent among multiple VMs. Integrity-Checker reverses these changes by computing the RVAs and replacing the absolute addresses with their corresponding RVAs. This functionality is essential in order to match the hashes of the same executable content that are fetched from other VMs.

Discussion: Integrity-Checker compares a module from a VM with modules from other $t-1$ VMs (where t is the total number of machines in a cloud that are running the same version of a guest operating system) to figure out whether the module has been altered or infected. It keeps track of the number of times the hashes of all the headers and content have been matched successfully. If the number of successes n are in majority from the total number of comparisons (i.e. $n > (t-1)/2$), Integrity-Checker concludes that the module has not been altered. This approach is only effective if majority of the VMs are running the original (or uninfected) modules. However, there are cases when malware such as `SQL Slammer` [1] [16] can rapidly infect most of the machines in a network and this would possibly make the above approach raise false alarms. However, in either of the above cases,

[1]It is worth noting that SQL Slammer is a buffer overflow exploit that does not modify kernel code and thus, cannot be detected by ModChecker.

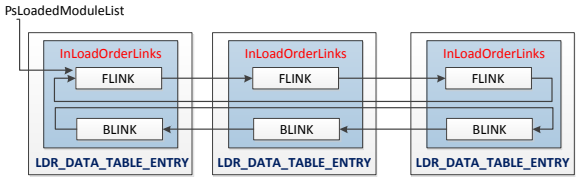


Fig. 2. Doubly linked list of the in-memory kernel modules

ModChecker is capable of detecting discrepancies among VMs that can trigger a more comprehensive, deeper analysis tools for further investigation and clean up. Furthermore, in a virtualizated environments, it is possible to keep clean snapshots of VMs and upon detecting any discrepancy, the machine(s) can be reverted back to their clean state to flush infections.

IV. Implementation

The ModChecker design is portable to any VMM (e.g. Xen, KVM or VMware ESX) that has support for VM introspection. For the proof of concept, we developed a prototype of ModChecker on Xen [4] that had MS Windows XP (Service Pack 2) VMs running. We used the libVMI library [15] to introspect the memory of Windows XP virtual machines. Moreover, we used the OpenSSL [17] library to get the support for computing MD5 message digests.

This section further describes the low-level implementation details of the components of the ModChecker.

A. Module-Searcher

Module-Searcher looks for the module in memory that is being checked for integrity. The list of the active modules (that are currently loaded into memory) is maintained as a doubly linked list (refer to Figure 2). Each node in the list is represented by the structure `LDR_DATA_TABLE_ENTRY` that contains the module name `BaseDllName`, and the base address `DllBase` (i.e. starting byte of the module in memory). It also contains the pointers to the next `FLINK` and the previous `BLINK` modules that are used to traverse the list in forward and backward directions. Module-Searcher obtains the pointer of the first element in the list using a system global variable `PsLoadedModuleList` and traverses the list to look for the module that is being checked for integrity violation by name. Once the module is found in the list, Module-Searcher obtains the base address of that module and copies the whole module from the virtual machine's memory to a local buffer and then passes the buffer pointer to Module-Parser.

B. Module-Parser

The Module-Parser receives the module from Module-Searcher and begins extracting the headers and contents. The format of the module in MS Windows is a portable executable. Figure 3 shows the relationships among portable executable headers. Module-Parser starts with

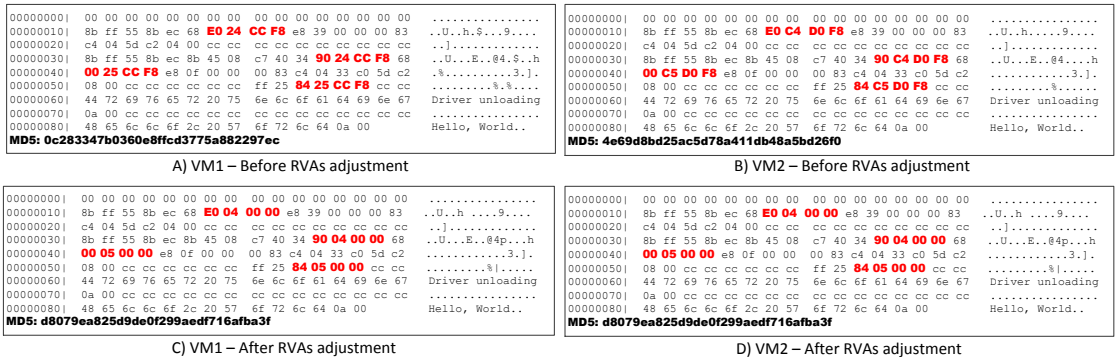


Fig. 4. RVAs adjustment illustration for `.text` section-data of `Hello World` kernel module. The Modules (32-bit) base addresses for virtual machines VM1 and VM2 are '`00 20 CC F8`' and '`00 C0 D0 F8`'.

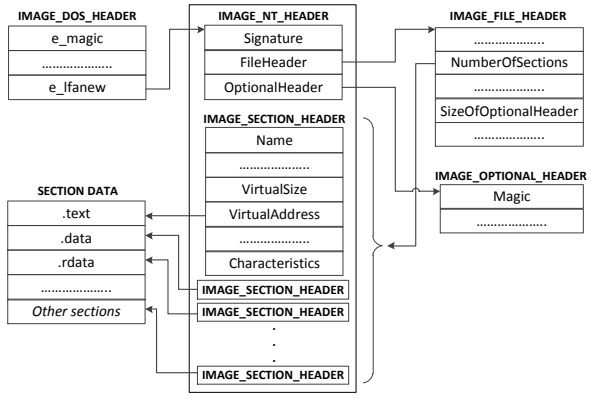


Fig. 3. Correlations among Portable-executable headers

`IMAGE_DOS_HEADER`. The first two bytes of the module are the `e_magic` magic numbers (specifically, "`MZ`") in the DOS header. The header has a field `e_lfanew` that points to `IMAGE_NT_HEADER`. The NT header has a four-byte signature (whose first two bytes are "`PE`") and the pointers (`FileHeader` and `OptionalHeader`) to `IMAGE_FILE_HEADER` and `IMAGE_OPTIONAL_HEADER`. The FILE header contains the size of the OPTIONAL header `SizeOfOptionalHeader` and the number of sections `NoOfSections` that come after NT header. A section has a section header `IMAGE_SECTION_HEADER` and section data. The section headers have the same size and are located in a sequence. Each section header has a unique name `Name`, a pointer `VirtualAddress` to its section data and the size `VirtualSize` of the section data. It also contains a `Characteristics` field that describes the features (e.g. read-only or executable code) of section data. Module-Parser uses this information to identify the section data that contains executable code.

After verifying the magic number in the DOS header, Module-Parser obtains the pointer to `IMAGE_NT_HEADER`. It accesses the NT header that is followed by the FILE and OPTIONAL headers. The section header starts just after the NT header. Module-Parser extracts `NoOfSections` section headers and further processes them to obtain their corresponding section data. Module-Parser then passes the headers and section data to the Integrity-Checker. Algorithm 1 represents Module-Parsing code for extracting headers and section data.

Algorithm 1 Extracting headers and section data from kernel module

```
IMAGE_DOS_HEADER dos
IMAGE_NT_HEADER nt
IMAGE_FILE_HEADER file
IMAGE_OPTIONAL_HEADER opt
IMAGE_SECTION_HEADER sec
Copy dos ← buffer[1,sizeof(IMAGE_DOS_HEADER)]
Obtain NTOffset ← NT header offset
Copy nt ← buffer[NToffset, sizeof(IMAGE_NT_HEADER)]
Copy file ← buffer[NToffset+sizeof(SignatureNTheader),
    sizeof(IMAGE_FILE_HEADER)]
Copy opt ← buffer[NToffset+sizeof(SignatureNTheader)+
    sizeof(IMAGE_FILE_HEADER),
    sizeof(IMAGE_OPTIONAL_HEADER)]
for i = 1 to file.NumberOfSections do
    sec ← Copy buffer[NToffset+sizeof(SignatureNTheader)+
    sizeof(IMAGE_FILE_HEADER)+
    sizeof(IMAGE_OPTIONAL_HEADER)+
    (sizeof(IMAGE_SECTION_HEADER)×i),
    sizeof(IMAGE_SECTION_HEADER)]
    Copy SectionData[i] ← buffer[sec.VirtualAddress, sec.VirtualSize]
end for
```

C. Integrity-Checker

Integrity-Checker computes the MD5 hash for each header and the section data (that contains the executable code) and compares their hash values with the ones obtained from the same modules from other virtual machines. At this stage, it is highly unlikely that the hash values of the executable will match among the modules. This is due to the fact that RVAs in the executable code are replaced with their absolute addresses and these are different among virtual machines, as the modules are loaded at different base addresses (refer to Figure 4A & 4B). At this stage, the Integrity-Checker scans the executable code, looks for the absolute addresses,

computes their corresponding RVAs (which are the same across virtual machines) and replaces the absolute addresses with their RVAs. This makes the executable code consistent and their MD5 hashes are matchable (refer to Figure 4C & 4D). Algorithm 2 represents Integrity-Checker code for adjusting RVAs. Integrity-Checker computes the RVAs using the following equation.

$$RVA = Absolute\,address\ -\ Base\,address \quad (1)$$

Finding the absolute addresses: Integrity-Checker assumes that while comparing the same executable code from two virtual machines, the difference in bytes within the code represents the absolute addresses. (The assumption is valid until the code is altered by an adversary). On a 32-bit machine, the address consists of 4 bytes and ideally there should be a difference of four consecutive bytes within the code. However, it is possible that difference could be less than four bytes depending on how many identical initial bytes the module's base addresses have. For instance, if the base addresses are '`00 CC 20 F8`' and '`00 CC 90 70`', the first two bytes of the base address are the same. Thus, the absolute address starts two bytes ahead from where the difference has been detected.

Algorithm 2 Adjusting Relative Virtual Addresses (RVAs)

```
offset ← 0
IsDifferenceExist ← 0
for i = 1 to 4 do
    offset++;
    if BaseAddress1[i] ≠ BaseAddress2[i] then
        IsDifferenceExist ← 1
        break
    end if
end for
if IsDifferenceExist == 1 then
    for j = 0 to SectionHeader.VirtualSize do
        if SectionData1 [j] ≠ SectionData2 [j] then
            Copy(4-bytes) AbsoluteAddress1 ←
            SectionData1[j − offset + 1]
            Copy(4-bytes) AbsoluteAddress2 ←
            SectionData2 [j − offset + 1]
            RVA1 ← AbsoluteAddress1 - BaseAddress1
            RVA2 ← AbsoluteAddress2 - BaseAddress2
            if RVA1 == RVA2 then
                Replace AbsoluteAddress1 with RVA1
                Replace AbsoluteAddress2 with RVA2
            end if
        end if
        j ← j − offset + 1 − 4
    end for
end if
```

V. Evaluation

A. Experimental Settings

For experimentation, we built a cloud environment. This test bed featured a Quad Core i7 (2.67 GHz * 8) server with HyperThreading enabled and 18 GB of RAM. This server had a 64-bit privileged virtual machine (`DomO`) running Fedora 16 (kernel 3.3.2-6) along with Xen 4.1.2 [4]. We instantiated 15 VM clones (`DomU: Dom1-Dom15`) in Xen from a single 32 bit Window XP (SP2) installation to make sure that all VMs are identical. Moreover, the automatic updates were disabled to ensure no accidental updates alter any of the modules. We used the introspection library for VMI (libvmi-0.6) [15] in all the experiments.

B. Integrity checking

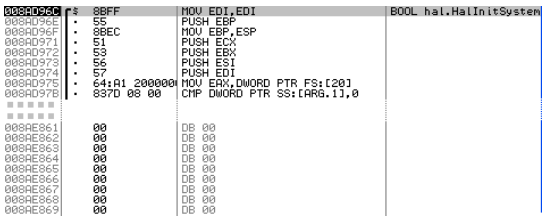
ModChecker is designed to detect the integrity violations (that are often introduced by malware infection) in headers and executable content of kernel modules running on multiple VMs. We evaluated the effectiveness of ModChecker by manually infecting common kernel modules in a way that imitates the infection techniques often used by common rootkits to exploit the kernel modules. Such techniques include, but are not limited to, single opcode replacement, inline hooking of DLLs and .sys files and PE file header infections [18], [19].

The following section describes detailed experiments that affirm ModChecker's ability to detect any modification in headers and executable content within the kernel modules. This detection is accomplished in real time provided that at least one virtual machine runs the original (uninfected) module.

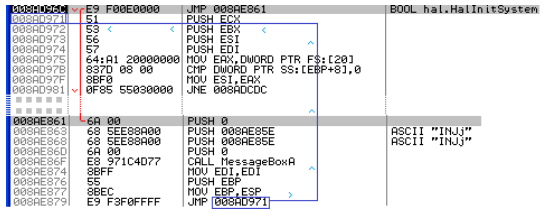
1) Single opcode replacement: Initial experiments consist of manually modifying the hardware abstraction layer kernel module, `C:\WINDOWS\System32\hal.dll`, within a single VM. Using Ollydbg [20] to open `hal.dll` library statically, we have made minimal changes to the `.text` code section. We modified a counter register decrement instruction `DEC ECX`, HEX opcode `49`, to its alternate instruction `SUB ECX, 1`, opcode `83E901`. Upon system restart, the newly modified `hal.dll` file was loaded into memory containing above mentioned modifications, despite the fact that this one to three byte modification shifted the jmp offsets. ModChecker analyzed the kernel modules from all the VMs, then reported a different hash value of the `.text` section of the `hal.dll` module within the VM containing the simple change. All other sections and fields of the module continued to be consistent with the kernel module sections of other concurrently running machines. An example of this minimal code change is malware's insertion of a specially crafted jump instruction or modification of the pointer that references a legitimate function. This is done to divert module's control flow to an address designated by the malware. This part of the experiment shows that a change to a single opcode will result in ModChecker raising a flag.

2) Inline Hooking: Inline hooking is a commonly used approach to divert control flow from legitimate code to malicious code that has been injected [9], [18], [19]. A typical method for accomplishing this is to insert `jmp` functions within the body of a function. An example of such malware is the `TCPIRPHOOK` rootkit. As shown in [19], this rootkit inserts hooks into the kernel module to intercept function calls and modify the result data obtained by network connection queries. Similarly, `Win32.Chatter` virus [9] infects `.sys` files by hooking kernel level functions to perform user level payload injection.

We use the `hal.dll` kernel module to create hooks for the purposes of our experiment. As shown in Figure 5 the inline hooking mechanism relies on finding appropriate non-executable code segments, known as opcode caves, such as `00` instructions, to place its payload. The first three lines of assembly instructions are replaced with a jmp hook which



(a) Before Inline Hooking



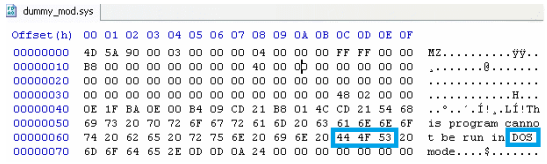
(b) After Inline Hooking

Fig. 5. Change made by Inline Hooking to `hal.dll`

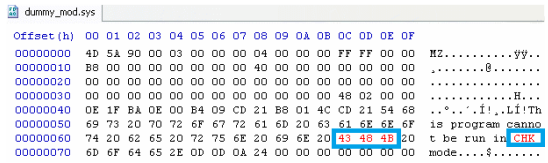
redirects execution to the newly injected payload. Furthermore, the injecting malware performs sanitation of overwritten bytes before returning to the original entry function `hal.HalInitSystem`. Windows OS did not catch this change in control flow at runtime and loaded the infected `hal.dll` into memory. Modchecker noted that the `hal.dll` module loaded into memory contains code different from that in the other VMs.

3) Trivial modification in stub program: The next experiment involved replacing only three characters of the stub program within the 'Hello World' dummy driver without changing the code alignment as shown in Figure 6. The word `DOS` in "`This program cannot be run in DOS mode`" at the beginning of the header now becomes `CHK`. The driver was loaded into the kernel using the OSR Driver Loader [21] program. ModChecker's analysis of the DOS header distinguished the modified driver's hash from the hashes of the non-modified drivers in parallel VMs. As other sections of the '`Hello World`' driver were left intact, ModChecker found no differences in hash values.

4) PE header modification via DLL hooking: DLL hooking within PE headers is a method of attaching extra, specially crafted DLLs to a module in kernel or user space. In fact, this mechanism can be used to hook functions not just to `.dll` modules but also `.sys` modules as they are structurally the same. Using the previously described inline hooking technique, kernel module procedures may be redirected to any dynamically linked procedure within the maliciously referenced DLL. This is particularly useful when an inline hooking method fails to find enough opcode caves to inject its malicious payload. Using `CFF Explorer` [22], a sample `inject.dll` module was inserted into a sample driver, `dummy.sys`, and a sample message popup function was made available to the driver by exporting a `callMessageBox()` procedure. To successfully inject a dynamically linked library



(a) Before modification



(b) After modification

Fig. 6. Modification in the dummy driver

into a kernel module, PE headers have to be adjusted to reference the attached functionality. Since additional code is made available to the main executable process, the size of the code visible to the module will change, thus increasing the `VirtualSize` value in the section header [23]. Furthermore, injecting extra code into the kernel module shifts the locations of subsequent section headers and also modifies the `.text` section data. Finally, the pointers that reference these new header locations will be adjusted appropriately. All these changes in the PE headers and the code section contribute to differentiating this kernel module's PE hashes, as well as the hash of the `.text` section. A rootkit such as `Rustok.B` creates hooks inside kernel's `Ntfs.sys` module that reference external functions [19], imitating the behavior of our example above.

ModChecker performed integrity analysis of the loaded dummy driver module and compared it to the dummy module that had `inject.dll` attached to it. Using ModChecker to detect executable kernel modules that have additional libraries hooked to it is straight forward at this point due to the mismatch of hashes in a number of module sections. Hash mismatches were detected in `IMAGE_NT_HEADER`, `IMAGE_OPTIONAL_HEADER`, all `SECTION_HEADER's` and `.text` field.

C. Runtime Performance

This section describes the runtime performance of ModChecker. It discusses the best and the worst runtime case scenarios and analyzes the impact of ModChecker on guest VM's system resources.

1) Best and Worst Case Scenarios of ModChecker: This section details the best and worst runtime cases of ModChecker. We used the module `http.sys` and compared it across a number of VMs. We computed the best and the worst running time of ModChecker by keeping the virtual machines mostly idle, as well as increasing resource consumption by running resource exhausting processes. For the latter case, we used `HeavyLoad` (a stress testing software) [24] that is capable of stressing all the resources (such as CPU, RAM and

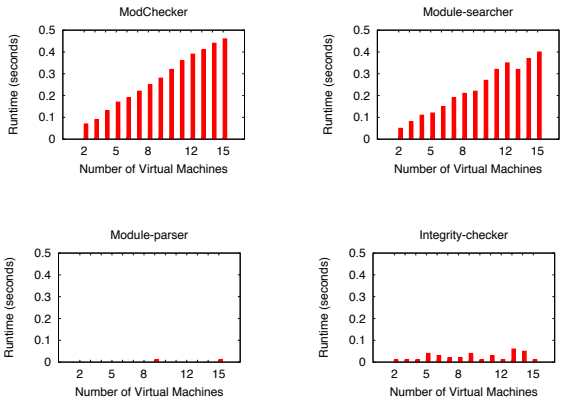


Fig. 7. Runtime performance of ModChecker (and its components) on different number of VMs when they are mostly idle

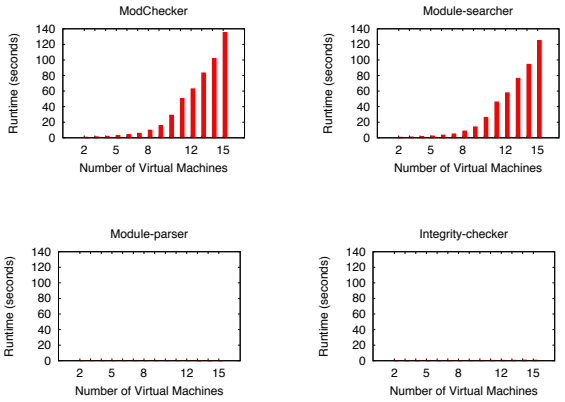


Fig. 8. Runtime performance of ModChecker (and its components) on different number of VMs when they are exhaustively using their resources

disk) of an MS Windows machine.

Figures (7 & 8) illustrate the runtime performance of ModChecker and its components under the best and the worst case scenarios. In the best case, we noticed a linear increment in the runtime as we increase the number of VM for comparison. We also noticed that, unlike Module-Parser and Integrity-Checker, the linear increment is also shown by Module-Searcher that significantly effects the overall runtime performance of ModChecker. The linear increment is comprehensible given that the current version of ModChecker accesses the virtual machines' memory in a sequence. The modular design of ModChecker can support parallel access of virtual machines' memory which would considerably enhance the runtime performance of ModChecker. Moreover, Module-Searcher takes more time than other modules since Module-Searcher has to access the memory by pages; an action that requires an iterative access of the memory until the whole module is copied to a local buffer.

The virtual machines share the same hardware resources (CPU, RAM, disk) with the privileged VM where ModChecker is being run. In the worst case, when the VMs exhaustively use their resources, the privileged virtual machine gets less hardware resources for ModChecker that effects the ModChecker's runtime performance. Moreover, we noticed a sudden nonlinear growth in the ModChecker's runtime when the number of heavily loaded VMs exceeded the number of available virtual cores, which we considered was the cause of this growth.

2) Inside virtual machine - ModChecker's impact on system resources: This section measures ModChecker's impact on the system resources inside the virtual machine while it accesses the machine's memory. We expect ModChecker to have insignificant impact on the machine's internal resource consumption since none of its components run inside the guest VMs. We kept the virtual machine idle in order to detect any significant change in the system's resource utilization when ModChecker accesses the memory.

We wrote a light-weight tool in `Python` to continuously record the current state of a VM's system resources. The tool helped us baseline the normal usage of the resources and identify any significant perturbation when the memory is accessed by ModChecker. Our tool records the CPU state (such as idle time, privileged time and user time), memory state (such as percentage of free physical and virtual memory and number of page faults), disk state (such as queue length and disk read/write per second rate) and network state (such as number of packets sent/received). The tool was run inside the VM, upon which it recorded its state and sent it to an external remote network storage device. The readings were simple ASCII characters that did not put any significant impact on the network. Moreover, this information was not stored on the local file system since local disk is an important part of virtual memory analysis.

Figure 9 shows the state of the CPU and memory of the guest VM and also explicitly zooms in on the time spans when the memory was being accessed by ModChecker. The graphs depict no significant perturbation during the time span when memory was accessed by ModChecker. Thus, we conclude that ModChecker does not place any considerable burden on the system resources of the virtual machine.

VI. Conclusion

The detection of sophisticated modern malware infections inside kernel space poses challenges in terms of effectiveness and efficiency. In this work, we presented a modular approach (for a cloud environment) to detect any integrity violation (often introduced by malware infections) in kernel modules.

We evaluated ModChecker with techniques often used by common rootkits to alter kernel module's headers and executable contents by injecting DLLs and explicitly modifying header values. The results showed that ModChecker detected any such infection in kernel modules that pertained to the alteration of headers and executable code.

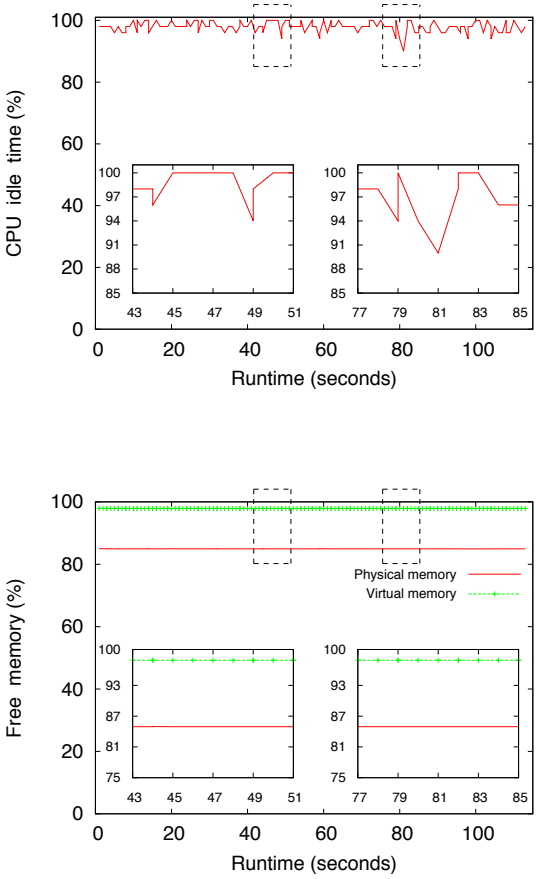


Fig. 9. Inside virtual machine - CPU and memory impact of ModChecker. Box represents the time span when the virtual machine's memory was being accessed by ModChecker. (The boxes are zoomed-in in the small graphs.)

We also evaluated the best and worst runtime cases of ModChecker when VMs were idle and when they were heavily loaded. In the best case, ModChecker showed steady linear growth of runtime. This is because the current version of ModChecker sequentially accesses virtual machines. However, in the worst case, the growth was nonlinear after the heavily loaded VMs exceeded the number of virtual cores.

We measured the impact of ModChecker on system resources inside the virtual machines. The machines were kept idle to ensure that only ModChecker induced perturbation was observed. We found no considerable impact of ModChecker on the system resources.

ModChecker runs outside the virtual machines using VMI that makes it difficult for malware residing inside the virtual machine to compromise. In the worst case, when malware infection spreads to the majority of the virtual machines in the cloud, ModChecker is still be able to detect discrepancies among kernel modules across virtual machines. Therefore, we conclude our approach is most effective in conducting initial light-weight consistency checks. Upon detecting module's inconsistent state a more comprehensive, but also more resource intensive, analysis tool can be used to trace the malware infection that has been flagged by the ModChecker.

ACKNOWLEDGMENT

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

REFERENCES

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. New Jersey, US: John Wiley and Sons, 2009.
[2] "Hacking with linux kernel module," http://newdata.box.sk/raven/lkm.html.
[3] "Digital signatures for kernel modules," http://msdn.microsoft.com/en-us/library/bb530195.aspx.
[4] "Xen," http://www.xen.org/.
[5] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," in *Proceedings of the* 2^{nd} *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*, Lisbon, Portugal, March 2007, pp. 327–340.
[6] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Information Technology Laboratory, Special Publication SP-800-145, Tech. Rep., 2007.
[7] R. Buyyaand, J. Broberg, and A. Goscinski, *Cloud Computing: Principles and Paradigms*, 1st ed. New Jersey, US: John Wiley and Sons, 2011.
[8] J. Rutkowska, "System virginity verifier defining the roadmap for malware detection on windows system," in *proceedings of the* 5^{th} *Hack In The Box Security Conference*, Kuala Lumpur, Malaysia, 2005.
[9] K. Kasslin, "Kernel malware: The attack from within," in *proceedings of the* 9^{th} *Annual Association of anti-Virus Asia Researchers Conference (AVAR)*, Auckland, New Zealand, December 2006.
[10] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *proceedings of the* 10^{th} *Annual Network and Distributed System Security Symposium*, San Diego, California, US, Febraury 2003, pp. 191–206.
[11] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell, "Linux kernel integrity measurement using contextual inspection," in *proceedings of the ACM workshop on Scalable trusted computing (STC'07)*, New York, NY, USA, 2007, pp. 21–29.
[12] A. Seshadr, M. Luk, E. Shi, A. Perrig, L. V. Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *proceedings of the* 12^{th} *ACM Symposium on Operating Systems Principles*, New York, NY, US, October 2005, pp. 1–16.
[13] M. Neugschwandtner, C. Platzer, P. M. Comparetti, and U. Bayer, "*d*Anubis - dynamic device driver analysis based on virtual machine introspection," in *proceedings of the* 7^{th} *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'10)*, Bonn, Germany, July 2010, pp. 41–60.
[14] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2009.
[15] "LibVMI," http://code.google.com/p/vmitools/.
[16] "SQL slammer," http://www.sans.org/security-resources/malwarefaq/ms-sql-exploit.php.
[17] "OpenSSL," http://www.openssl.org/.
[18] "Inline hooking in windows," www.exploit-db.com/download_pdf/17802/.
[19] C. Xuan, J. Copeland, and R. Beyah, "Toward revealing kernel malware behavior in virtual execution environments," in *proceedings of the* 12^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, Saint-Malo, France, 2009, pp. 304–325.
[20] "OllyDbg," http://www.ollydbg.de/.
[21] "OSR driver loader," http://www.osronline.com/article.cfm?article=157.
[22] "CFF explorer," http://www.ntcore.com/exsuite.php.
[23] "Peering inside the PE: A tour of the win32 portable executable file format," http://msdn.microsoft.com/en-us/library/ms809762.aspx.
[24] "Heavyload," http://www.jam-software.com/heavyload/.