https://doi.org/10.2352/ISSN.2470-1173.2021.3.MOBMU-034

Characteristic Features of the Kernel-level Rootkit for Learning-based Detection Model Training

Mohammad Nadim; Department of Electrical & Computer Engineering, University of Texas at San Antonio; San Antonio, Texas, USA
Wonjun Lee; The Katz School of Science and Health, Yeshiva University; New York City, New York, USA
David Akopian; Department of Electrical & Computer Engineering, University of Texas at San Antonio; San Antonio, Texas, USA

Abstract

The core part of the operating system is the kernel, and it plays an important role in managing critical data structure resources for correct operations. The kernel-level rootkits are the most elusive type of malware that can modify the running OS kernel in order to hide its presence and perform many malicious activities such as process hiding, module hiding, network communication hiding, and many more. In the past years, many approaches have been proposed to detect kernel-level rootkit. Still, it is challenging to detect new attacks and properly categorize the kernel-level rootkits. Memory forensic approaches showed efficient results with the limitation against transient attacks. Cross-view-based and integrity monitoring-based approaches have their own weaknesses. A learning-based detection approach is an excellent way to solve these problems. In this paper, we give an insight into the kernel-level rootkit characteristic features and how the features can be represented to train learning-based models in order to detect known and unknown attacks. Our feature set combined the memory forensic, cross-view, and integrity features to train learning-based detection models. We also suggest useful tools that can be used to collect the characteristics features of the kernel-level rootkit.

Keywords: Cyber-security, Digital forensic, Kernel-level rootkit, Machine Learning.

Introduction

A Kernel-level rootkit is one of the most elusive types of malware in recent years. It can exploit the vulnerabilities existing in the operating system (OS) kernel to hide its presence and malicious activities. It is difficult for user-level applications to detect kernel-level rootkit as it operates in the kernel with the highest privileges. The stealthy nature of the kernel-level rootkit makes it the most lethal and sophisticated attacking tool for cyber offenders. ZeroAccess malware used rootkit techniques to hide in an infected machine and was used to download other malware from a botnet [1]. It infected millions of Microsoft Windows OS machines. Zacinlo malware leverages rootkit technique to propagate adware in Windows 10 OS [2]. Most of the traditional security systems are focused on user-level threats and failed to detect the kernel-level rootkit.

According to Hoglund and Butler, a rootkit is a set of programs that remain undetected on a computer and have a permanent effect [3]. Rootkits can be categorized into five different classes: user-level, kernel-level, hypervisor-level, boot-kits, and firmware rootkits. In this paper, we only focus on the kernel-level rootkit. Many approaches have been proposed to detect the kernel-level rootkit. Signature-based approaches may check the kernel module static signatures [4] or data access signatures of the kernel dynamic objects [5]. Behavior-based detection approaches may check the kernel memory access behavior [6], analyze the execution path [7], abnormal behavior within a herd [8], or data structure invariants behavior [9]. Another approach for detecting the kernel-level rootkit is cross-view-based detection. The basic idea of cross-view-based detection is to compare two different views of the system [10, 11]. Volatile memory traces are a great source to construct an unmodified view for the kernel-level rootkit detection [12]. The kernel-level rootkits can tamper with the integrity of both the static region and the dynamic region of the OS. While some research focuses on only static region integrity, recent research focuses on dynamic region integrity as modern kernel-level rootkits mostly alter the dynamic data structures. The integrity-based detection approaches focusing on static regions either check the write attempt to read-only memory section [13, 14] or periodically check the hash of the known memory region [15, 16]. By verifying the function pointers or kernel data-layout partitioning [17] dynamic region integrity can be checked. External hardware can also be used to detect the kernel-level rootkit [18, 19, 20, 21].

With the increase of cybercrime in recent years, the automatic detection of known and unknown attacks now become important in modern security systems. A learning-based detection is an excellent approach to automatically detect known and unknown attacks with high accuracy. The purpose of this paper is to have an insight into the kernel-level rootkit characteristic features and how the features can be represented to train learning-based models in order to detect known and unknown kernel-level rootkit attacks. Also, to get familiar with the tools that can be used to collect the features.

The rest of the paper is composed as follows: prior research on learning-based kernel-level rootkit detection is introduced in Section II. A brief discussion about the kernel-level rootkit is discussed in Section III. The characteristic features of the kernel-level rootkit are elaborately described in Section IV, followed by the useful tools to collect the features in Section V. Finally, we conclude this paper with future research direction in Section VI.

II. Related Works

There has been a race between kernel-level rootkit evolution and detection approaches. The researchers are now focusing on learning-based detection techniques to detect kernel-level rootkit because machine learning and deep learning technology have proved high accuracy to automatically detect known and unknown malware. Researchers have trained learning models with a variety of features to detect kernel-level rootkit such as hardware events counts using hardware performance counter (HPC) [22], virtual memory access pattern of an application [23], or system call execution times [24]. These features have some limitations in detecting the DKOM attack. A learning algorithm is applied to a set of kernel driver run-time features derived from the execution behavior using an emulator [25]. The additional delay in driver loading time is a weakness of this approach to detect kernel-level rootkit. The obfuscation technique employed in kernel-level rootkit binaries makes the static analysis difficult. Still, the kernel-level rootkit can be detected through static analysis by disassembling the kernel driver and extract features like general behavior, communications, suspicious behaviors, etc. [26]. As the detector work inside the host in this detection system, the detector is vulnerable to the advanced kernel-level rootkit. Tian et al. [27] experimented with behavior features of the kernel module to train multiple machine learning algorithms. The features included important kernel API invocation, executing code in the kernel data region, write operation to a kernel memory area, write operation to important hardware registers, etc. The authors isolate the kernel module memory which may introduce significant overhead. Hardware events like data dependencies between registers, OS privilege transition, and branches in program execution flow can be involved to interpret the program data/control transfer flow features. Zhou and Makris [28] introduced a hardware-assisted machine learning-based rootkit detection mechanism that first identifies the process class using machine learning algorithms and then employs Kernel Density Estimation (KDE) to indicate a compromise in process behavior caused by a kernel-level rootkit.

Wang et al. [29] proposed a machine learning-based trusted kernel rootkit detection method. They combine the memory forensic analysis with bio-inspired machine learning technology. The training features are extracted from volatile memory dumps using the Volatility framework [30]. The extracted features include hidden kernel modules, device tree, the SSDT function, callbacks, timers, orphan threads, and driver objects. Seven different machine learning classifiers are used to train the model and detect kernel rootkits. Lee and Nadim suggested some key features of the kernel-level rootkit and showed some possible attack scenarios in the container-based cloud computing system [31].

Our kernel-level rootkit features are closely related to the features used by Wang et al. [29]. The advantage of using volatile memory traces is that the detection system can be implemented separately. The drawback of this approach is that the attack can happen between snapping two volatile memory traces (transient attack). We explain the characteristic features more elaborately. We also represent all possible states of the features and how they can be labeled as normal or malicious to train learning-based models. Importantly, we tried to cover some features that will not be affected by the transient attack. Some cross-view features and integrity features are also included in our feature set. In the feature set '1' indicates the presence of a feature and '0' indicates the absence.

III. Kernel-level Rootkit

The rootkits of the first generation are mainly user-level rootkits that conceal themselves as disk-resident system programs by mimicking the system process files. Those rootkits are easy to detect and remove by using file integrity tools. So, the modern rootkits have moved to memory-residency to evade the detection by file integrity tools. The rootkits of the second generation modify the control flow to execute malicious code by using the hooking technique. By executing the malicious code, the return value or functionality requested from the OS can be altered. User-mode hooking is easy to detect compared to kernel-mode hooking, as it is implemented in the user-space. Kernel-mode hooking injects malicious code into the kernel-space via device driver which makes it difficult to detect by a user-mode intrusion detection system (IDS). System Service Descriptor Table (SSDT), Interrupt Descriptor Table (IDT) and I/O Request Packet (IRP) function tables are the most common target for implementing kernel hooks. The execution of malicious code by the second-generation rootkit leaves a memory footprint in both user-space and kernel-space that can be detected and analyzed. The rootkits of the third generation are mostly kernel-level rootkits. Though they have limited applications, they are difficult to detect as they modify the dynamic kernel data structures. Direct Kernel Object Manipulation (DKOM) attack, implemented by the third-generation rootkits, targets the dynamic data structures in kernel whose values change during runtime. We can summarize the action of the kernel-level rootkit into the following categories: System Service Hijacking (system call table hooking, replacing system call table), Dynamic Kernel Object Hooking (virtual file system hooking), and Direct Kernel Object Manipulation (DKOM).

System Service Hijacking

A system call is implemented in such way that it works as an interface between user-level processes and an OS. Through this interface, user-level programs access the system resources. All the actual system call routine memory addresses are stored in a table named System Service Descriptor Table (SSDT) or System Call Table. The kernel-level rootkits can attack the system call table in different ways. For example, attackers can modify the system call routine address in the system call table to replace the legitimate system call with their own malicious system call. By modifying the code in the target address, attackers can also change the control flow of a system call. The control flow is passed to the malicious code usually by injecting jump instructions. Additionally, attackers can overwrite the memory that stores the system call table address to replace the whole system call table with their own version of the system call table [32]. Another important hooking target is the Interrupt Descriptor Table (IDT). The processor uses the IDT to determine the correct response to interrupts and exceptions. As interrupts have no return values, interrupt requests can only be denied by hooking the IDT. In a multiprocessing system, an attacker needs to hook all IDTs as each CPU has its own IDT.

Dynamic Kernel Object Hooking

The OS kernel uses Virtual File System (VFS) to handle the file system operations across different types of file systems such as EXT2, EXT3, and NTFS. Thus, VFS is a layer between the actual file systems and the user-level programs that make the file handling system calls to access the files. Different data structures are used by VFS to achieve a common file model such as the file object, inode object, and dentry object. The kernel-level rootkit can modify the file object data structure field that contains a pointer to the file_operation structure (f_op) to hide without modifying the system call table. Function pointers to inode operation functions such as lookup function are stored in the inode data structure. The kernel-level rootkit can hide a process by modifying the function pointer of the lookup function for the process directory's (/proc) inode data structure [33].

Direct Kernel Object Manipulation

By using the DKOM technique, the kernel-level rootkits can also modify the kernel data structures. As the DKOM technique aims to modify dynamic kernel data structures, it is harder to detect than kernel hooking because the dynamic object changes during normal runtime operations. Malicious process hiding is a perfect example of the DKOM technique. EPROCESS data structure is the OS kernel's representation of a process object. To hide a malicious process, kernel-level rootkits unlink the malicious process's EPROCESS data structure that is maintained in a doubly linked list. Unlinking an element from the process list implemented in a doubly linked list makes the process invisible to both user and kernel-mode programs. Other than process, Kernel device drivers, active ports can also be hidden by using this technique. Implementation of DKOM is extremely difficult because an incorrect change in OS kernel data structure may result in system crashes.

IV. Characteristic Features of the Kernel-level Rootkit

In this section, we will describe the important characteristic features of the Kernel-level rootkit and show how the features can be represented to train learning-based models.

Modules

Kernel-level rootkits are often loaded into the kernel as an LKM. When a kernel module is loaded into the kernel, LDR_DATA_TABLE_ENTRY, a metadata structure is generated to create a doubly linked list pointed to by PsLoadedModuleList. In Windows OS, Get-Module -ListAvailable command looks into C:\Program Files\WindowsPowerShell\Modules and C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules directories to list all modules loaded on the system [34]. In the Linux OS, the 'lsmod' command searches the /proc/modules directory for listing all loaded modules. If kernel-level rootkit hides the module from those directories using the file hiding technique, then user-level applications and utility tools will not find the malicious module. In that case, we can check the memory for the doubly linked list associated with the modules to find the hidden module. Unfortunately, the kernel-level rootkit can also modify the module's doubly linked list by unlinking the corresponding entry using the DKOM technique to hide its presence (figure 1). In that case, we need to scan the memory to find out the unlinked module.

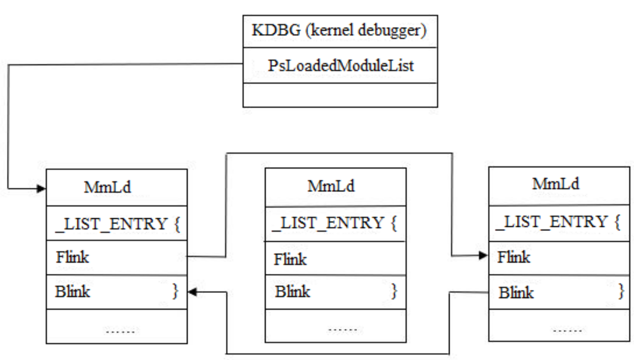

Figure 1. Hiding module from the doubly linked list.

The data structure entry for the malicious module will still be in the memory though it is unlinked from the doubly linked list. All unlinked and unloaded modules can be detected from volatile memory by using a pool tag scanning approach that looks for the pool tag (MmLd) associated with the kernel module in the physical address space [35]. If the module is not in the list of unloaded modules, that indicates an unlinked malicious module hidden by the kernel-level rootkit. In Windows OS, the Windows Debugger can be used to extract the list of unloaded modules. If the pool tag 'MmLd' of the metadata structure LDR_DATA_TABLE_ENTRY is corrupted or destroyed by the kernel-level rootkit, a pool tag scanning in the physical memory address for DRIVER_OBJECT data structure reveals the list of kernel modules.

Processes

In Windows OS, an EPROCESS data structure is associated with each process and all active processes' EPROCESS data structure creates a doubly linked list pointed to by PsActiveProcessHead (Figure 2). A kernel-level rootkit can hide processes from system utilities by hooking NtQuerySystemInformation. However, by checking the doubly linked list in the memory, a hidden process from the system utility can be detected when hooking is conducted. A kernel-level rootkit can also use the DKOM to unlink the process's EPROCESS data structure from the doubly linked list for hiding the process information. The ActiveProcessLinks field in the EPROCESS data structure contains two members: Flink (forward link) points to the next EPROCESS data structure and Blink (backward link) points to the previous EPROCESS data structure. A kernel-level rootkit can modify this ActiveProcessLinks to unlink the malicious process from the doubly linked list. Each EPROCESS data structure contains a pool tag 'Proc' that is searchable in a pool tag scanning approach resulting in the detection of the unlinked process [35]. Inactive or terminated processes can also be detected if they reside in memory. When distinguishing the unlinked and terminated processes, the exit time of processes is useful.

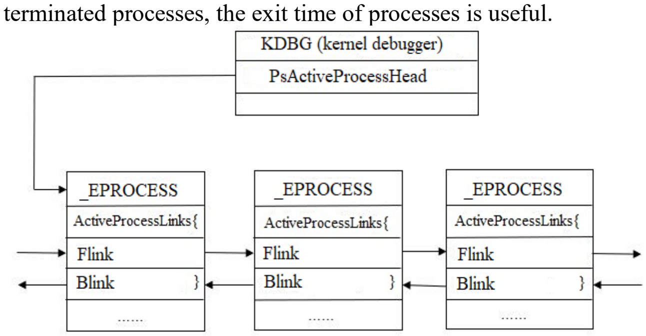

Figure 2. The doubly linked list of EPROCESS data structure.

Threads

A Thread is a flow of instruction execution within a process with an ETHREAD data structure. A thread that does not belong to any active module can be named as an orphan thread. The starting address of a thread (thread.StartAddress) points to the owning driver of that thread. If the start address of a thread does not match with any kernel module in the PsLoadedModuleList, this may indicate an orphan thread left by the kernel-level rootkit. A process can own multiple threads to perform parallel execution of instructions. Through the list-traversal in volatile memory, all the threads hidden from a utility debugger can be identified [36]. A pool tag 'Thre' scanning in physical memory can also detect all the hidden threads. As the ETHREAD data structure contains information about its parent process, it is possible to identify any hidden processes by carefully examining the thread information. Table 1 shows possible states for modules, processes, and threads information for the feature set.

Kernel Hooks

The SSDT is a critical target for kernel-level rootkit as it contains the system service routines pointers in kernel space. A kernel-level rootkit can overwrite the SSDT function pointers for pointing to malicious modules.

Table 1. Possible states for Modules, Processes, and Threads as a feature set.

Feature / Label	Diff_1	Diff_2	Diff_3	Diff_4	Diff_5	Diff_6	Orphan Thread
Normal	0	0	0	0	0	0	0
Malicious	0	0	0	0	0	0	1
Malicious	0	0	0	0	0	1	1/0
Malicious	0	0	0	0	1	1/0	1/0
Malicious	0	0	0	1	1/0	1/0	1/0
Malicious	0	0	1	1/0	1/0	1/0	1/0
Malicious	0	1	1/0	1/0	1/0	1/0	1/0
Malicious	1	1/0	1/0	1/0	1/0	1/0	1/0

Diff_1 = difference between system utility output and memory scanning of the doubly linked list for loaded modules.
Diff_2 = difference between memory scanning of the doubly linked list and 'MmLd' pool tag scanning for loaded modules.
Diff_3 = difference between memory scanning of the doubly linked list and pool tag scanning of DRIVER_OBJECT data structure for loaded modules.
Diff_4 = difference between system utility output and memory scanning of the doubly linked list for active processes.
Diff_5 = difference between memory scanning of the doubly linked list and 'Proc' pool tag scanning for active processes.
Diff_6 = difference between utility debugger output and memory scanning for active threads

In Windows OS, the SSDT table stores the pointers to core kernel API functions of NT modules, and the SSDT shadow table stores the pointer to GUI related functions of win32k.sys module. Scanning all the ETHREAD objects in the memory and checking ETHREAD.Tcb.ServiceTable pointers make it easy to detect any modification of SSDT. The Interrupt Descriptor Table (IDT) that stores the function pointer of interrupt service routines or interrupt handlers is another critical target and the kernel-level rootkits can modify IDT entries to redirect the control flow to the malicious code for execution. By checking the memory of IDT, we can find the hooked IDT entry that resides outside the known clean memory region. It is possible to find the hooked address containing malicious code by checking the volatile memory. The kernel-level rootkit is not only limited to system table hooking. It can also target a function in the kernel and forcing it to jump to a memory address containing malicious code.

Callbacks and Timers

To monitor the occurrence of a particular event in the Windows OS, drivers need to be registered for a callback routine. The callback function allows the kernel-level rootkit driver to monitor system activities and take different malicious actions accordingly [37]. For example, a kernel-level rootkit can install a callback routine for monitoring process execution and termination on the system. Similar to callbacks, a kernel-level rootkit can create a timer to get notification of a specific time elapsed. A kernel-level rootkit can schedule periodic operations by using this functionality. We can check the volatile memory for callback objects and timer objects to find any malicious or unknown modules indicating a kernel-level rootkit.

Special Machine Registers

This feature section is similar to the features suggested by Lee and Nadim [31]. The kernel-level rootkits can tamper with the machine register values to alter the kernel control flow and that makes the machine registers an important feature to detect the kernel-level rootkit. Machine registers were focused on prior research to monitor the integrity of the kernel and any violation of integrity indicates a kernel-level rootkit detection [9, 38, 39]. Since some machine registers hold the memory location of important kernel tables, by altering the value of the register, kernel-level rootkits can redirect the control flow to the memory address where malicious executables reside. After system boot, the value stored in some machine registers become fixed. That means any alteration to those machine registers will indicate suspicious activity.

Interrupt Descriptor Table register

The IDT is a data structure that stores the list of interrupt descriptors to determine the correct response of interrupts and exceptions. Interrupt descriptor table register (IDTR) stores both the physical base address and length of the IDT. Using the Load instruction of IDT (LIDT) the kernel-level rootkit can change the base address of the IDT and redirect all interrupt requests to the malicious address. The process of changing the value stored in the IDTR register using the load instruction (LIDT) is well described by Kad [40]. A traditional security scanner may check the integrity of the old IDT and the kernel-level rootkit will remain undetected. So, the write operation to the IDTR can be incorporated into the feature set of the learning model.

Global and Local Descriptor Table registers

The characteristics of various memory areas used during the program execution are defined in the global descriptor table (GDT) and the local descriptor table (LDT) data structures. GDT contains the global segment (memory area), while LDT contains a program-specific private memory segment. Global descriptor table register (GDTR), and local descriptor table register (LDTR) stores the value that points to GDT and LDT, respectively. Kernel-level rootkits can modify these register values to point to the memory address where a malicious executable exists [27].

Cr0 Control register

The general behavior of the CPU and other devices can be controlled or changed by the control registers [41]. cr0 is a 32-bit control register with various flags that can modify the basic operation of the processor. For the write protection, the 16th bit of the cr0 register is used. If it is set, then the CPU will not be able to write to the read-only memory section. 16th bit of cr0 register can be modified to bypass the write protection and the kernel-level rootkit can then write malicious executable in the read-only memory or hook SSDT [27]. The technique of SSDT hooking by modifying the cr0 register has been described by Dejan [42]. Some legitimate kernel drivers like Anti-virus or firewall products may need to modify cr0 register. It is still an important feature to incorporate into the learning model.

Table 2 shows possible states for the kernel hooks, callbacks and timers, and special machine registers as feature set.

Table 2. Possible states for Kernel hooks, Callbacks and Timers, and Special Machine Registers as feature set.

Feature / Label	SSDT hook	IDT hook	Inline hook	Abnormal callbacks	Abnormal timers	IDTR value changed	GDTR value changed	cr0 value changed
Normal	0	0	0	0	0	0	0	1/0
Malicious	0	0	0	0	0	0	1	1/0
Malicious	0	0	0	0	0	1	1/0	1/0
Malicious	0	0	0	0	1	1/0	1/0	1/0
Malicious	0	0	0	1	1/0	1/0	1/0	1/0
Malicious	0	0	1	1/0	1/0	1/0	1/0	1/0
Malicious	0	1	1/0	1/0	1/0	1/0	1/0	1/0
Malicious	1	1/0	1/0	1/0	1/0	1/0	1/0	1/0

The write protection bit of the cr0 register in Linux can be disabled as follows:

```
write_cr0(read_cr0( ) & (~ 0x10000))
```

After the malicious write operation, the write protection bit of the cr0 register needs to be reset, otherwise, the system will crash. It can be reset as follows:

```
write_cr0(read_cr0( ) | 0x10000)
```

V. Useful Tools for Feature Collection

Memory forensic is widely used in prior research to detect the malicious behavior of the computer system. Kernel-level rootkit behaviors such as malicious code injection, hooking, process hiding, module hiding, etc. can be easily detected by memory forensic techniques. Prior works have used volatile memory to detect the kernel-level rootkit [29, 43 - 46]. The most commonly used memory forensic framework is Volatility [30]. It can extract digital artifacts from volatile memory without interrupting the system being investigated. This open-source tool supports all three major OS (Windows, Linux, and macOS). Other memory forensic tools such as BlackLight[47], SANS SIFT[48] can also be used to analyze the volatile memory. The volatility framework does not provide memory acquisition capability, but it is flexible to support the different file formats of volatile memory.

The detection system can be isolated from the target OS by running the target OS in a virtualized environment. VirtualBox [49] is one of the most popular open-source hypervisors to create virtual environments. Command-line tools can be used to read the machine register value of a target OS running inside a virtual machine. For example, the ‘VBoxManage debugvm vm_name getregisters idtr’ command will return the value stored in the IDTR register (base address and length of IDT). ‘VBoxManage debugvm vm_name getregisters gdtr’ command, ‘VBoxManage debugvm vm_name getregisters cr0’ command will return the value stored in the GDTR register and cr0 register, respectively. Here, vm_name in the command will be the name of the virtual machine. These values can be stored for a clean system and checked later for detecting any modification. Different system utility tools can be used to construct lists of modules and processes inside the host. Windows (tasklist, driverquery), Linux (lsmod, ps aux) have their own implementation of system utility tools. Then socket connection can be used to send the lists to the detection system outside the host for a view comparison.

VI. Conclusion and Future Work

In this paper, we elaborately describe and suggest some characteristic features of the kernel-level rootkit and how they can be represented to train learning-based models. We also suggest some useful tools that can be used to collect the features. Volatile memory traces used in prior research have flaws to detect the transient attacks. We include some characteristic features of the kernel-level rootkit that will come from continuous monitoring so that the transient attacks can be detected. Our future work includes creating an open-source dataset and then train learning-based models to detect the kernel-level rootkit.

References

[1] The ZeroAccess Rootkit. J. Wyke and S. Labs. 2020. Retrieved from: https://nakedsecurity.sophos.com/zeroaccess/.

[2] Zacinlo malware ad fraud. 2020. Retrieved from: https://labs.bitdefender.com/2018/06/six-years-and-counting-inside-the-complex-zacinlo-ad-fraud-operation/

[3] Hoglund, G. and Butler, J., 2006. Rootkits: subverting the Windows kernel. Addison-Wesley Professional.

[4] Kruegel, C., Robertson, W. and Vigna, G., 2004. Detecting kernel-level rootkits through binary analysis. In 20th Annual Computer Security Applications Conference (pp. 91-100). IEEE.

[5] Rhee, J., Lin, Z. and Xu, D., 2011. Characterizing kernel malware behavior with kernel data access patterns. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (pp. 207-216).

[6] Rhee, J., Riley, R., Xu, D. and Jiang, X., 2009, March. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In 2009 international conference on availability, reliability and security (pp. 74-81). IEEE.

[7] Wang, X. and Karri, R., 2013, May. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC) (pp. 1-7). IEEE.

[8] Bianchi, A., Shoshitaishvili, Y., Kruegel, C. and Vigna, G., 2012, October. Blacksheep: detecting compromised hosts in homogeneous crowds. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 341-352).

[9] Hofmann, O.S., Dunn, A.M., Kim, S., Roy, I. and Witchel, E., 2011. Ensuring operating system kernel integrity with OSck. ACM SIGARCH Computer Architecture News, 39(1), pp.279-290.

[10] Wang, Y.M., Beck, D., Vo, B., Roussev, R. and Verbowski, C., 2005, June. Detecting stealth software with strider ghostbuster. In 2005 International Conference on Dependable Systems and Networks (DSN'05) (pp. 368-377). IEEE.

[11] Rhee, J., Riley, R., Xu, D. and Jiang, X., 2010, September. Kernel malware analysis with un-tampered and temporal views of dynamic

kernel memory. In International Workshop on Recent Advances in Intrusion Detection (pp. 178-197). Springer, Berlin, Heidelberg.
[12] Hua, Q. and Zhang, Y., 2015, October. Detecting malware and rootkit via memory forensics. In 2015 International Conference on Computer Science and Mechanical Automation (CSMA) (pp. 92-96). IEEE.
[13] Garfinkel, T. and Rosenblum, M., 2003, February. A virtual machine introspection-based architecture for intrusion detection. In Ndss (Vol. 3, No. 2003, pp. 191-206).
[14] Baliga, A., Chen, X. and Iftode, L., 2006. Paladin: Automated detection and containment of rootkit attacks. Department of Computer Science, Rutgers University.
[15] Quynh, N.A. and Takefuji, Y., 2007, March. Towards a tamper-resistant kernel rootkit detector. In Proceedings of the 2007 ACM symposium on Applied computing (pp. 276-283).
[16] Petroni Jr, N.L. and Hicks, M., 2007, October. Automated detection of persistent kernel control-flow attacks. In Proceedings of the 14th ACM conference on Computer and communications security (pp. 103-115).
[17] Srivastava, A. and Giffin, J., 2012, December. Efficient protection of kernel data structures via object partitioning. In Proceedings of the 28th annual computer security applications conference (pp. 429-438).
[18] Petroni Jr, N.L., Fraser, T., Molina, J. and Arbaugh, W.A., 2004, August. Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor. In USENIX security symposium (pp. 179-194).
[19] Baliga, A., Ganapathy, V. and Iftode, L., 2010. Detecting kernel-level rootkits using data structure invariants. IEEE Transactions on Dependable and Secure Computing, 8(5), pp.670-684.
[20] Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y. and Kang, B.B., 2012, October. Vigilare: toward snoop-based kernel integrity monitor. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 28-37).
[21] Lee, H., Moon, H., Jang, D., Kim, K., Lee, J., Paek, Y. and Kang, B.B., 2013. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In 22nd {USENIX} Security Symposium ({USENIX} Security 13) (pp. 511-526).
[22] Singh, B., Evtyushkin, D., Elwell, J., Riley, R. and Cervesato, I., 2017, April. On the detection of kernel-level rootkits using hardware performance counters. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (pp. 483-493).
[23] Xu, Z., Ray, S., Subramanyan, P. and Malik, S., 2017, March. Malware detection using machine learning based analysis of virtual memory access patterns. In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017 (pp. 169-174). IEEE.
[24] Luckett, P., McDonald, J.T. and Dawson, J., 2016, April. Neural network analysis of system call timing for rootkit detection. In 2016 Cybersecurity Symposium (CYBERSEC) (pp. 1-6). IEEE.
[25] Wilhelm, J. and Chiueh, T.C., 2007, September. A forced sampled execution approach to kernel rootkit identification. In International Workshop on Recent Advances in Intrusion Detection (pp. 219-235). Springer, Berlin, Heidelberg.
[26] Musavi, S.A. and Kharrazi, M., 2014. Back to static analysis for kernel-level rootkit detection. IEEE Transactions on Information Forensics and Security, 9(9), pp.1465-1476.
[27] Tian, D., Ma, R., Jia, X. and Hu, C., 2019. A Kernel Rootkit Detection Approach Based on Virtualization and Machine Learning. IEEE Access, 7, pp.91657-91666.
[28] Zhou, L. and Makris, Y., 2018, March. Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution. In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1580-1585). IEEE.
[29] Wang, X., Zhang, J., Zhang, A. and Ren, J., 2019. TKRD: Trusted kernel rootkit detection for cybersecurity of VMs based on machine learning and memory forensic analysis. Mathematical Biosciences and Engineering, 16(4), pp.2650-2667.
[30] Volatility framework. 2020. Retrieved from: https://www.volatilityfoundation.org/
[31] Lee, W. and Nadim, M., 2020, August. Kernel-Level Rootkits Features to Train Learning Models Against Namespace Attacks on Containers. In *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)* (pp. 50-55). IEEE.
[32] Levine, J., Grizzard, J. and Owen, H., 2004. A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table. In Second IEEE International Information Assurance Workshop, 2004. Proceedings. (pp. 107-125). IEEE.
[33] Jakobsson, M. and R. Zulfikar, Eds. 2008. Crimeware: Understanding New Attacks and Defenses, Addison-Wesley Professional
[34] Microsoft Powershell, 2020, Retrieved from: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/get-module?view=powershell-7
[35] Sylve, J.T., Marziale, V. and Richard III, G.G., 2016. Pool tag quick scanning for windows memory analysis. Digital Investigation, 16, pp.S25-S32.
[36] Investigating Windows Threads with Volatility, 2020, Retrieved from: http://mnin.blogspot.com/2011/04/investigating-windows-threads-with.html
[37] The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory.
[38] Wang, Z., Jiang, X., Cui, W. and Ning, P., 2009, November. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security* (pp. 545-554).
[39] Zhang, F., Wang, J., Sun, K. and Stavrou, A., 2013. Hypercheck: A hardware-assistedintegrity monitor. *IEEE Transactions on Dependable and Secure Computing*, *11*(4), pp.332-344.
[40] Handling Interrupt Descriptor Table for fun and profit - kad. Phrack Vol. 0x0b, Issue 0x3b
[41] Control register, 2020. Retrieved from: https://en.wikipedia.org/wiki/Control_register
[42] Hooking the System Service Dispatch Table (SSDT), 2020. Retrieved from: https://resources.infosecinstitute.com/hooking-system-service-dispatchtable-ssdt/#gref
[43] Hay, B. and Nance, K., 2008. Forensics examination of volatile system data using virtual introspection. *ACM SIGOPS Operating Systems Review*, *42*(3), pp.74-82.
[44] Guri, M., Poliak, Y., Shapira, B. and Elovici, Y., 2015, August. JoKER: Trusted detection of kernel rootkits in android devices via JTAG interface. In *2015 IEEE Trustcom/BigDataSE/ISPA* (Vol. 1, pp. 65-73). IEEE.
[45] I. Korkin, and I. Nesterov, "Applying Memory Forensics to Rootkit Detection" (2014). *Annual ADFSL Conf. on Digital Forensics, Security and Law*. 1.
[46] Ring, S. and Cole, E., 2004, October. Volatile memory computer forensics to detect kernel level compromise. In International Conference on Information and Communications Security (pp. 158-170). Springer, Berlin, Heidelberg.
[47] SANS SIFT. 2020. Retrieved from: https://digital-forensics.sans.org/community/downloads
[48] BlackLight. 2020. Retrieved from: https://www.blackbagtech.com/products/blacklight/
[49] Oracle VM VirtualBox. 2020. Retrieved from: https://www.virtualbox.org/