JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 445-461 (2017)
DOI:10.1688/JISE.2017.33.2.10

An Online Approach for Kernel-Level Keylogger Detection and Defense*

DONGHAI TIAN[1,2], XIAOQI JIA[2,3], JUNHUA CHEN[4,+] AND CHANGZHEN HU[1]
[1]*Beijing Key Laboratory of Software Security Engineering Technique*
Beijing Institute of Technology
Beijing, 100081 China
[2]*Key Laboratory of Network Assessment Technology, Institute of Information Engineering*
Chinese Academy of Sciences
Beijing, 100093 China
[3]*University of Chinese Academy of Sciences*
Beijing, 100049 China
[4]*Key Laboratory of IOT Application Technology of Universities in Yunnan Province*
Yunnan Minzu University
Kunming, 650500 China
E-mail: chenjunhuabj@163.com

Keyloggers have been studied for many years, but they still pose a severe threat to information security. Keyloggers can record highly sensitive information, and then transfer it to remote attackers. Previous solutions suffer from limitations in that: (1) Most methods focus on user-level keylogger detection; (2) Some methods need to modify OS kernels; (3) Most methods can be bypassed when the OS kernel is compromised. In this paper, we present LAKEED, an online defense against the kernel-level keylogger by utilizing the hardware assisted virtualization technology. Our system is compatible with the commodity operating system, and it can protect the running OS transparently. The basic idea of our approach is to isolate the target kernel extension that may contain the keylogger from keyboard drivers' execution environment and then monitor their potential interactions. By comparing the runtime information with the execution baseline that is obtained by the offline analysis, the keylogger can be identified. The evaluation shows that LAKEED can defeat kernel-level keyloggers effectively with low performance overhead.

Keywords: keylogger detection, virtualization, OS kernel, on-the-fly, driver

1. INTRODUCTION

As more and more users' privacy information gets stolen, keyloggers pose a serious threat to information security. Once a keylogger is installed into the end host, the keyboard activity can be maliciously captured. As a result, the attacker can easily obtain user-sensitive information (*e.g.*, user names and passwords). Although keyloggers have been well studied for many years, they are still widely used for stealing personal information in the wild. Typically, keyloggers can be implemented as software or hardware

Received December 29, 2015; revised March 15, 2016; accepted April 8, 2016.
Communicated by Meng Chang Chen.
* This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61602035, 61100228 and 61202479, the National Key Research and Development Program of China under Grant No. 2016YFB0800700, the National High-tech R&D Program of China under Grant No. 2012AA-013101, and Open Found of Key Laboratory of Network Assessment Technology, Institute of Information Engineering, Chinese Academy of Sciences.
+ The corresponding author.

devices [1].

Software keyloggers can be divided into two categories: user-level keyloggers and kernel-level keyloggers. The first ones stay in the user-level, and they utilize high-level APIs provided by the OS to intercept keystrokes. Compared with the first ones, kernel-level keyloggers are more dangerous in that they operate in higher privilege mode, which can completely control the OS kernel code and data. Moreover, the kernel-level keylogger is difficult to be identified for its small footprint in terms of memory and processor utilization.

To defeat keyloggers, many defense approaches have been proposed. Most of these methods can only detect user-level keyloggers, and they cannot deal with kernel-level keyloggers. Some methods rely on the emulation-based techniques so they will introduce considerable performance overhead. Some methods require some modifications to OS kernels. As a result, they may not be applied widely.

To address the above problems, in this paper, we present the design and implementation of LAKEED, a novel kernel-level keylogger detection system based on the hardware assisted virtualization technology. Compared with the previous methods, our system can be easily deployed in the real production environment to defeat kernel-level keyloggers on-the-fly.

Our approach is motivated by the key observation: most kernel-level keyloggers will change the working procedure of keyboard drivers. For example, the keylogger may utilize the inline hook technique to subvert the execution control flow of keyboard drivers. Based on this observation, we leverage the hardware assisted virtualization technology to monitor the potential interactions between keyboard drivers and a target kernel extension. For ease of presentation, we use the terms driver, kernel extension and kernel module interchangeably.

We have implemented a prototype of LAKEED based on a tiny hypervisor. Most of the functionalities are built into the hypervisor layer. Thanks to the late launch feature provided by the recent hardware, the hypervisor can be loaded on-demand so that it can protect the target OS transparently.

In summary, our method makes the following contributions:

- We propose a lightweight approach for keylogger detection. This method can detect a keylogger in the kernel space, and prevent it from sniffing user keystrokes.
- We leverage the hardware assisted virtualization technology to achieve transparent kernel-level keylogger detection.
- We design and implement a prototype of LAKEED based on Windows systems. The evaluations show that our system can detect kernel mode keyloggers effectively.

2. SECURITY ASSUMPTION AND THREAT MODEL

Our defense method against kernel-level keyloggers is based on three security assumptions. First, we assume that the target operating system runs on the hardware that supports hardware assisted virtualization. Moreover, we assume the keyboard is connected to the host computer via PS/2 port. Our second assumption is that our hypervisor is trusted thanks to its small TCB. Third, we assume the keylogger resides in one of the

existing kernel modules. For example, a device driver developed by the third party may contain the keylogger functionality to monitor keyboard activity.

Our threat model allows the keylogger to access the data and code regions of all kernel components with full privilege. To get the keyboard input data, the keylogger can either hijack the keyboard drivers' execution, or access the keyboard buffer directly. Under this model, the attacker is powerful enough to capture user keystrokes in the kernel space.

3. BACKGROUND

Before introducing our method, it is necessary to present the general working flow of a keyboard in a Windows system. In general, the working procedure of the keyboard in Windows can be summarized as follows:

(1) When a user presses or releases a key on the keyboard, it generates a hardware interrupt.
(2) The CPU calls the Interrupt Service Routine (ISR) (*i.e.*, I8042KeyboardInterruptService) inside the i8042prt driver to handle the interrupt.
(3) The i8042prt driver reads the input data located in the keyboard controller.
(4) After the driver finishes its urgent task, it puts the other non-urgent task, referring to Deferred Procedure Call (DPC), into a kernel callback queue.
(5) The OS kernel executes the DPC so that the function I8042KeyboardIsrDpc inside the i8042prt driver gets invoked.
(6) The i8042prt driver executes the callback function KeyboardClassServiceCallback, which is registered by the kbdclass driver.
(7) The raw input thread in the user space send an IRP (I/O Request Package) read request to the keyboard driver.
(8) The kbdclass driver extracts the read request from the raw input thread, and then invokes the function KeyboardClassRead to obtain the pressed key information.
(9) The kbdclass driver returns the keyboard data to the raw input thread.

In addition, we need to introduce some background of Intel VT (Virtualization Technology) that our system relies on. Intel VT defines two new processor modes, called VMX root mode and VMX non-root mode. The hypervisor runs in VMX root mode, while the guest OS runs in VMX non-root mode. Intel VT supports a special feature, called late launching of VMX modes. This feature allows us to launch a hypervisor on the live system on-the-fly. When the hypervisor is launched, certain events (*e.g.*, privilege instructions) cause the processor mode to transfer from VMX root mode to VMX non-root mode, which is called VMExit.

4. OVERVIEW OF OUR APPROACH

The goal of LAKEED is to build a system that can detect a kernel-level keylogger and prevent it from stealing keyboard data. Different from previous detection systems, our approach exploits the hardware assisted virtualization technology to defeat the keylogger. The basic idea of our approach is to leverage the hypervisor's higher privilege to monitor the execution of a kernel extension (that may contain the keylogger functionality)

along with the keyboard drivers' execution. By comparing the runtime information with the normal execution profile, our system could judge whether the target extension contains the functionality to collect keyboard data.

As shown in the Fig. 1, LAKEED achieves the keylogger detection in two key stages: offline analysis and online detection. In the first stage, LAKEED leverages the hardware assisted paging to transparently isolate keyboard drivers from the OS kernel in a clean execution environment. By doing so, we can ensure that the driver's code cannot be executed during the OS kernel's execution; while the OS kernel's code cannot be accessed during the driver's execution. As a result, our system could capture the execution transfers between the keyboard drivers and OS kernel. By recording the invocation entry points of keyboard drivers as well as the associated call stacks, we obtain the normal execution profile for the keyboard.

Fig. 1. Overview of LAKEED functionalities.

Next, LAKEED enters the online detection stage. Similar to the first stage, the target kernel extension is isolated from keyboard drivers. Based on the hardware assisted paging, the hypervisor creates three protection domains for the keyboard drivers and the target extension. These protection domains have the same memory mapping, but different access permissions. In this way, the hypervisor can monitor the invocations of the kernel extension (or driver) and the associated call stacks. If this runtime information does not match with the normal execution profile, it indicates that the target extension may contain the keylogger functionality.

To prevent the keylogger from stealing the keyboard data, we propose a defensive technique to protect the user input. Specifically, we exploit the hypervisor's higher privilege to intercept the keystroke first, and then transfer the input data to the protected user process directly.

5. SYSTEM DESIGN AND IMPLEMENTATION

We have developed LAKEED, a prototype based on the Hyperdbg [2] (a tiny hypervisor) to demonstrate our approach. As Fig. 2 shows, most components reside in the hypervisor level, including Policy, Enforcer, and Monitor. The Policy component stores the normal execution profile of keyboard drivers. The role of the Enforcer is to isolate the keyboard drivers and kernel extension in different protection domains. The monitor is used to trap and analyze the execution of the kernel extension and keyboard drivers. There is only one component, called Controller, locating in the user level. This component is applied to specify the target kernel extension to be monitored. Moreover, the Controller is responsible for transferring the execution profile to the hypervisor. To protect the Controller from being tampered by kernel-level attacks, the latest hypervisor-based methods [3, 4] can provide the required protection.

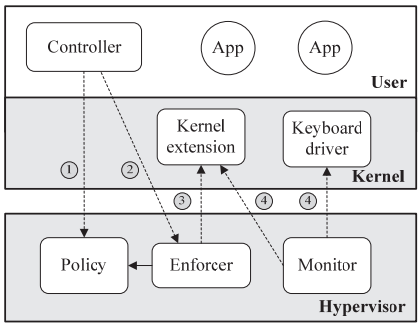

Fig. 2. The LAKEED architecture.

In general, the workflow of LAKEED can be summarized as follows: First, the Controller transfers the execution profile to the Policy. Then, the Enforcer is notified to isolate the target kernel extension from the keyboard driver. After that, the Monitor traps and analyzes execution transfer between the kernel extension and keyboard drivers. If the kernel extension's runtime behavior affects the keyboard driver's execution, the Monitor will generate a keylogger alert.

5.1 Extension Identification

To identify the target kernel extension, a traditional method is to utilize the kernel module's information inside the OS kernel. For example, this method first finds the global head of the module list whose virtual address can be pre-determined. Next, it walks the module list to locate the module descriptor. By reading the descriptor, it can gather the module's name, the base address as well as the module's size. Unfortunately, recent study [5] shows that advanced attackers may manipulate the module's descriptor to mislead the security software. Consequently, we cannot rely on the descriptor information to identify the memory region of the target kernel extension in some cases.

To address these problems, we propose a signature-based approach to identify the kernel module from the kernel memory. Specifically, we first parse the PE file of the kernel module to locate and analyze the text section. Then, we compute the checksum of this section as the signature. To make the signature robust and small, we only choose a very small part of the text section as the computing target. With the computed checksum, we scan the kernel memory to locate the kernel extension.

To access the kernel memory, the hypervisor needs to traverse the page tables and then map the physical pages into its own address space. Before calculating the checksum in the memory, we should first locate the appropriate address whose memory content should contain the function prologue instructions (*i.e.*, push ebp, mov ebp, esp). Next, we compare the checksum with the memory content. If it matches, the memory region of the kernel extension can be easily located. Particularly, the locating method is based on our three observations:

(1) The virtual address of a kernel extension is always contiguous and page-aligned.
(2) The code section of this kernel extension resides in the first memory page.

(3) The kernel extension usually does not contain self-modifying code.

Relying on the first page that includes the target checksum and the PE file size of the kernel extension, we can identify the specific memory pages where the kernel extension is located.

On the other hand, if the checksum does not match, the hypervisor will scan the next memory page for the extension identification. The basic workflow of extension identification is illustrated in Fig. 3.

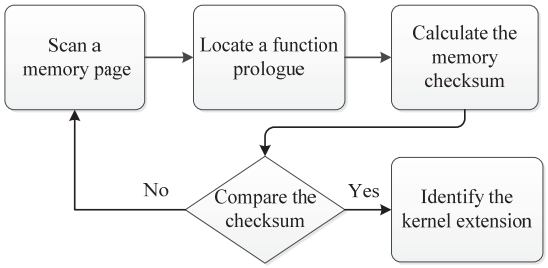

Fig. 3. The workflow of extension identification.

5.2 Extension Isolation

To isolate the target kernel extension1 from kernel space, we utilize Intel's Extended Page Tables (EPT) technology. When this feature is enabled, the hardware MMU will maintain additional page tables, called EPTs. These tables are used to carry out translation of Guest Physical Address (GPA) used in a VM to Host Physical Addresses (HPA) of the real hardware. Specifically, the EPT has 4 paging structures, including namely PML4 (Page Map Level 4 Table), PDPT (Page Directory Pointer Table), PD (Page Directory), and PT (Page Table). To allocate memory for these paging structures, we leverage Windows non-paged pool allocation. By manipulating the access rights of the EPTs, the hypervisor can run the target extension and keyboard drivers in different execution environments.

Since Windows systems mainly utilize two kernel modules to drive a keyboard, we need to maintain 3 separate EPTs for the two keyboard drivers and the target kernel extension. All these EPTs have the same memory mapping, but with different access permissions.

Before setting the memory permissions, we first apply the method (mentioned in Section 5.1) to locate the memory area in which the kernel extension and keyboard drivers reside. To set the memory access rights in EPT, we need to traverse the paging structures to modify the permission bits in the associated page table entries. Once the different memory permissions are set, we can divide the kernel address space into three separate address spaces, which is shown in Fig. 4.

In the target kernel extension space, the target extension can only access its own memory region, and cannot access the code and data region of the keyboard drivers. In the keyboard driver spaces, both of the two drivers can access each other's data region, in addition to its own code and data regions. However, the drivers' code permissions are

mutually exclusive, and they are not allowed to access the code and data regions of the target kernel extension.

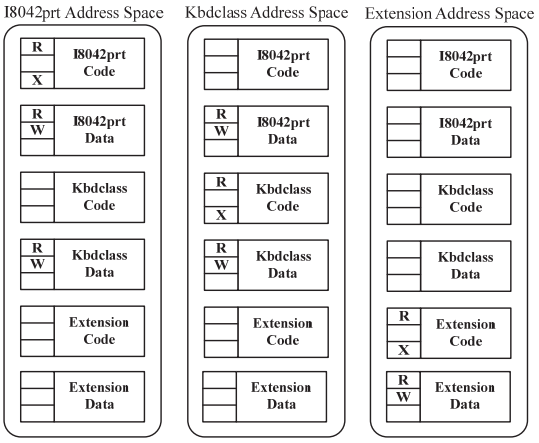

Fig. 4. Memory protections in the target extension and keyboard driver address space.

Initially, all of the three kernel extensions are marked as non-executable in the EPTs. In this way, when one of these kernel extensions is invoked by the OS kernel, it will trigger an EPT violation due to the memory protection. Then, we need to remark the associated memory area as executable in the original EPT to continue the kernel execution. Next, when an EPT violation is triggered again, we do not remark the memory permissions, but just change the value of the EPT base pointer to another EPT root that defines another set of access rights. Since switching the address space will flush the TLB, the system performance should be affected due to the TLB misses. To address this problem, we make use of Intel's Virtual Processor Identifiers (VPID) technology [6] to avoid the TLB flush. For each invocation of one kernel extension, only one EPT is active.

It is worth noting that we make some reverse engineering efforts to locate the keyboard buffers and set their memory permissions. Specifically, we first utilize the signature-based method [7] to locate KiInitialPCR, the Kernel Processor Control Region (KPCR) for the processor 0. Traversing from KiInitialPCR, we can find the Object Manager Namespace Directory (OMND), which provide a path to the driver object i8042prt. Based on the driver object, we get a pointer to a device object. Furthermore, the object contains a pointer to the Device Extension structure. By analyzing this structure, we obtain the address of the keyboard buffer (*i.e.*, PKEYBOARD INPUT DATA). Similarly, we make use of the KPCR to identify the keyboard buffer inside the kbdclass driver. Then, we set the associated access rights in the EPT to isolate the keyboard buffer from the target kernel extension.

5.3 Keylogger Detection

After the extension isolation is performed, we leverage the underlying hypervisor to trap and analyze the invocation of the kernel extension. Before detecting the anomaly

invocation, we need to generate an execution baseline for the working procedure of keyboard drivers. To this end, we apply a dynamic analysis method to record the invocation information of keyboard drivers in a clean Windows system. Similar to the extension isolation, we first manipulate two exclusive EPTs for two keyboard drivers in the hypervisor. Next, we utilize the hypervisor's higher privilege to capture the normal invocation of one kernel module that causes an EPT violation. Fig. 5 shows the execution baseline of keyboard drivers.

Specifically, when a user presses or releases a key, the ISR function I8042KeyboardInterruptService inside the i8042prt driver is invoked first (step a). Then, this driver queues the DPC function I8042KeyboardIsrDpc. Next, the system calls the DPC, which in turn invokes the callback function KeyboardClassServiceCallback inside the kbdclass driver (step b). After the callback function finishes its operation, it returns the kernel execution to the i8042prt driver (step c). After that, the kbdclass driver is invoked to handle the IRP read request (step d). Compared with the entire working procedure of keyboard drivers, the execution baseline exhibits the intrinsic features when the keyboard drivers work in different protection domains. In addition to record the invocation entry points, we also log the call stack information during dynamic analysis. This information can reflect the execution transfers inside the OS kernel. With the invocation entry point and the call stack information as a baseline, the hypervisor can detect the abnormal invocation of a kernel extension during the working procedure of keyboard drivers.

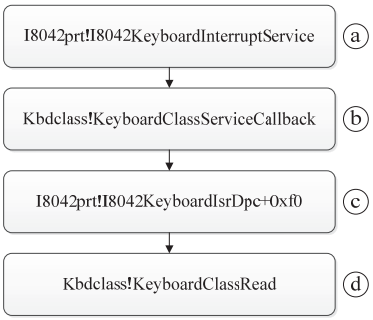

Fig. 5. Keylogger detection baseline.

To our knowledge, there are two types of kernel-level keylogger: the codebased [8] and the data-based [9]. The first one hooks specific system calls or driver functions to capture user keystrokes. The second one does not make any kernel code modification, but use a heuristic method to access the keyboard buffer directly.

For the code-based keylogger, it hijacks the normal execution of keyboard drivers. As a result, the keylogger's execution will be inlined into the keyboard driver invocations. To judge whether the target extension's execution is inlined into the drivers', we compare the runtime execution information (*e.g.*, the driver invocation trace) with the detection baseline. If it matches (when the target extension's invocation information is removed), we think the target extension's function does not interleave with the keyboard drivers'. Otherwise, the extension may contain the keylogger functionality.

For the data-based keylogger, it may not be invoked during the working procedure

of keyboard drivers. Thanks to our extension isolation, the target kernel extension cannot access the keyboard buffer when it is executed. If the keyboard buffer is accessed during the kernel extension's execution, we believe that this extension would probably sniffer user keystrokes.

5.4 Keylogger Defense

To prevent the kernel-level keylogger from sniffing keystrokes, we propose an anti-keylogger method based on the virtualization technology. The general defense architecture is shown in the Fig. 6.

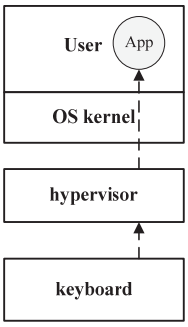

Fig. 6. Keylogger defense architecture.

Different from the keylogger detection, the hypervisor should first intercept each keystroke before it is sent to the OS. For this purpose, the hypervisor should be configured to trap external interrupts. In specific, the external-interrupt exiting field of VM-Execution Control in VMCS should be set to 1. To determine whether the interrupt is generated by the keyboard, the hypervisor should look up the IOAPIC redirection table to get the interrupt number. After that, the hypervisor will further judge whether the target process is protected according to the pre-defined process ID, which can be obtained from the process descriptor. To locate the process descriptor, we need to first find the head of the descriptor list, and then traverse the list according to the cr3 register value. If the target process is not protected, the hypervisor will inject the interrupt event into the OS. After that, the OS will turn to handle this keyboard interrupt. If the target process is protected, the hypervisor will not transfer the keystroke event to the guest OS but store it temporarily in the hypervisor space.

To access the keystroke, the protected process needs to invoke a hypercall, which enables a user program to communicate with a hypervisor directly. Particularly, the hypercall should contain 3 parameters, including the service number, the virtual address of a buffer (inside the protected process), and the buffer size. The first parameter is a unique identifier to allow the hypervisor knows that the protected process will begin to receive user input from the keyboard. Since adversaries may figure out the service number to abuse the hypercall, we should keep the service number random. To deal with this problem, we can leverage the online patching provided by hypervisor to change the user

code and the corresponding hypercall handler dynamically. The second and third parameters are used for recording keystrokes into a user buffer.

To facilitate our implementation, users are required to press the F9 key twice to notify the beginning and end of user input. To retrieve keyboard scan code in PS/2 keyboards, the hypervisor can read the 0x60 I/O port directly. After the keyboard data is read, the OS kernel cannot fetch it any more thanks to the hardware feature. To prevent the kernel-level keyloggers from accessing the user buffer that contains the keystrokes, we leverage the hardware assisted paging to set the associated memory permission non-readable. By doing so, the user buffer cannot be accessed by any kernel component when the protected process is executed.

6. EVALUATION

In this section, we evaluate both the detection effectiveness and the performance of LAKEED. All the experiments are carried out on a Dell Power Edge T410 work station with one 2.13G Intel Xeon E5606 CPU and 4 GB memory.

6.1 Effectiveness

We evaluate the effectiveness of LAKEED for kernel-level keylogger detection with six real-world instances and three synthetic examples, which is shown in Table 1. In the first two examples, both of them are attached to the keyboard drivers to catch keyboard read requests. The third one modifies the IOAPIC redirection table to capture keystrokes. The fourth one is a commercial keylogger, which is mainly implemented by a driver (*i.e.*, nwlnk2k.sys). The fifth one is a filter driver that is attached to the kbdclass driver. The sixth one is a driver that hooks the IDT (Interrupt Descriptor Table) for keystroke sniffing.

Table 1. Effectiveness of LAKEED.

Kernel-level keylogger	Detection
Klog [10]	Yes
Ctrl2cap [11]	Yes
Ps2intcap	Yes
Elite keylogger [12]	Yes
KDL 1.0.3	Yes
bhwin keysniff	Yes
Hook-kbdclass (synthetic)	Yes
Hook-i8042prt (synthetic)	Yes
Keylogger-1 (synthetic)	Yes

To implement the keylogger Hook-kbdclass, we utilize the inline hook technique to hijack the function KeyboardClassServiceCallback inside the kbdclass driver. Similarly, we exploit a function pointer located in the i8042prt driver to change its execution control flow

for recording the keyboard data. Keylogger-1 is a data-based keylogger developed by our lab. To sniffer user keystrokes, this keylogger does not need to change the kernel code and kernel control data, but access the keyboard buffer directly. The experiments show that all these kernel-level keyloggers are successfully detected by our system.

To analyze our detection procedure, Fig. 7 shows two case studies of kernel-level keyloggers which violate the execution baseline. The grey sections illustrate the abnormal invocation of the kernel module. In Fig. 7 (a), since the keylogger Hook-kbdclass hijacks the callback function KeyboardClassServiceCallback registered by the kbdclass driver, this kernel extension will be invoked when the i8042prt driver executes the callback function. After the kernel extension finishes its operations, it returns the execution back to the kbdclass driver. Thanks to our memory isolation, both of the two execution transfers will trigger two EPT violations so that the hypervisor can capture and identify the abnormal events for keylogger detection.

In Fig. 7 (b), the keylogger ctrl2cap.sys hijacks the IRP function inside the kbdclass driver to dispatch the IRP MJ READ request. As a result, this kernel extension will be executed when the IRP dispatch function is invoked. In particular, since this keylogger register a callback function in the IRP dispatch routine, this registered function will be invoked when the IRP request is fully served. After that, the keylogger invokes the kernel function IoCallDriver to transfer the execution control back to the kdbclass driver. After the driver finishes its task, the execution is transferred back to the keylogger again. Similarly, these operations trigger the EPT violations due to our memory isolation. Then, by comparing the runtime execution profile with the detection baseline, the keylogger is identified successfully.

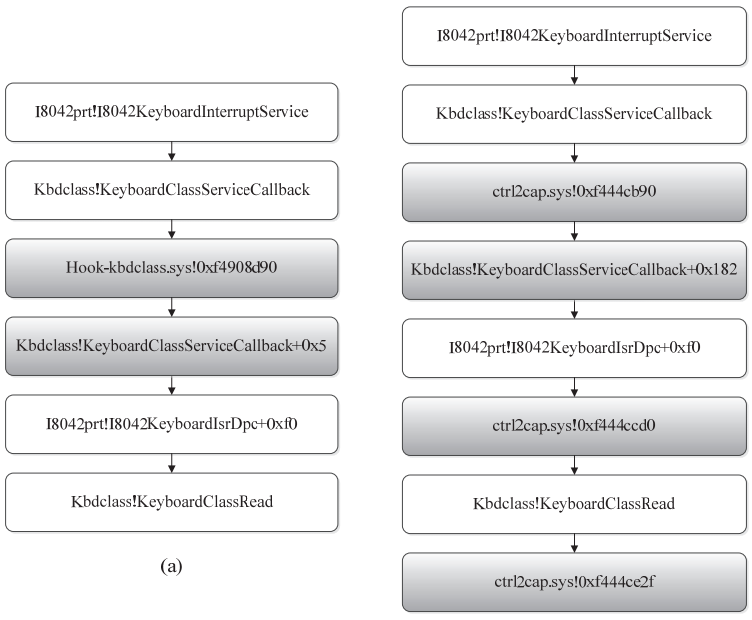

Fig. 7. Keylogger detection examples.

To evaluate the effectiveness of our approach for the keylogger defense, we develop a custom application to invoke hypercalls for retrieving keystroke information from the hypervisor. Meanwhile, we install a kernel-level keylogger for sniffing keystrokes. The experiments show that the hypervisor can capture all the keystrokes that are then transferred to the target application correctly. However, the keylogger does not get any keystroke information for the target application.

6.2 Performance Overhead

To measure the application level overhead, we test several application benchmarks in the protected system where the MinGW and MSYS facilities are installed. For each benchmark, we isolate one relevant kernel extension from the protection domains of keyboard drivers. In the first two benchmarks, our target kernel extension is the ntfs file system module (*i.e.*, ntfs.sys). To test the first benchmark, we decompress the standard Linux kernel source package (*i.e.*, linux-2.6.24.tar.gz) using the tar program from the MSYS. For the second application test, we use the MinGW to compile the latest Hyperdbg [2] with the default configuration. In the third benchmark, we make use of the cp program from the MSYS to copy a 57695KB file into a USB disk. Before conducting this test, the USB driver (*i.e.*, usbstor.sys) is isolated from the keyboard drivers. In the next two benchmarks, our target kernel module is the tcpip driver (*i.e.*, tcpip.sys). To carry out these tests, we first install an Apache web server in the protected system. Then, we utilize the ApacheBench program to measure the average response time and the transfer rate of this Server. More precisely, the ApacheBench is configured to set up 6 concurrent clients with each generating 30 requests to the Apache server that serves a 2568KB html webpage. In the final benchmark, our target kernel extension is a custom driver (*i.e.*, crypt file.sys) for file encryption. To conduct the test, we measure the execution time for this driver to encrypt a 97.2 MB file. To figure out the add-on overhead, we also carry out these experiments in the native system.

Table 2. Running overhead of LAKEED.

Benchmark	Target kernel extension	Native performance	LAKEED performance	Add-on overhead
Kernel decompression	Ntfs.sys	329110ms	345296ms	4.92%
Hyperdbg build	Ntfs.sys	13843ms	14328ms	3.50%
Copy file	Usbstor.sys	3180Kb/s	3035Kb/s	4.78%
ApacheBench transfer rate	Tcpip.sys	1845Kb/s	1776Kb/s	3.89%
ApacheBench response time	Tcpip.sys	1391ms	1428ms	2.66%
File encryption time	Crypt file.sys	673235ms	707682ms	5.12%

Table 2 shows the results of our user-level benchmarks. The second column shows the target isolated kernel extension. The third column shows the performance of the native sys-

tem without virtualization, while the fourth column shows the performance of the protected system with our detection enabled. The last column presents the add-on performance overhead. From Table 2, we can see that the overhead introduced by LAKEED is minimal. In general, the performance overhead relies on the virtualization cost. In addition to intercepting the sensitive events that cause VMExit unconditionally, the hypervisor should trap external interrupts so that it can capture the keystroke before the OS kernel.

Besides application level benchmarks, we also carry out a micro-benchmark to evaluate the performance overhead of invoking keyboard drivers. For this purpose, we implement a simple device driver that filters the kbdclass driver to intercept the IRP MJ READ request. In particular, this driver registers a callback function that will be invoked when the IRP is severed. The driver figures out the tick count for keyboard drivers to handle the IRP MJ READ request. When a user presses and releases a key, the keyboard drivers spend 12 tick counts to process the IRP in the native system; while it takes 16 tick counts for the keyboard drivers to handle the IRP in the protected system. The add-on performance cost is 4 CPU ticks.

In addition, we implement a custom driver to hook the IRP dispatch function in the kbdclass driver. Then, we calculate the execution time of the dispatch function. In the native system, it takes 8 CPU ticks to complete the operation. In our protected system, the function spends 13 CPU ticks to finish its execution. The add-on overhead is 5 CPU ticks.

The performance experiments indicate that our protection mechanism introduces very small performance cost if the target kernel extension does not need to interact with the keyboard drivers. However, the performance overhead will be higher when the target extension interacts with the keyboard drivers frequently. The main reason for this overhead is that since the keyboard drivers are isolated from the target extension address space, the underlying hypervisor needs to change the current active EPT for the interactions between the target extension and keyboard drivers.

7. DISCUSSION AND LIMITATIONS

In this section, we discuss several issues related to our system. First, since the target kernel extension should be specified by administrators, it may take some human efforts for keylogger detection. To address this issue, the whitelisting approach based on trust computing [13] could be used to limit the number of potential target extension. Moreover, the whitelisting approach can be applied to filter the false positive.

Second, LAKEED is limited to detecting the keylogger based on Return-Oriented Programming (ROP), which does not introduce new code but leverages control of the call stack to execute the existing code. However, the ROP-based keylogger is not common in the real world.

Finally, our current implementation mainly focuses on detecting the keyloggers based on the PS/2 keyboard. Nevertheless, by isolating the USB related kernel extensions and building an associated execution profile, our method can also be applied to defeat the keyloggers based on the USB keyboard. Moreover, our solution is applicable to other operating systems. (*e.g.*, Linux).

8. RELATED WORK

Stefano *et al*. [14] introduce a keylogger detection technique based on fine-grained profiling of memory write patterns. Although this technique can be transparently deployed online, it cannot be applied to defeat kernel-level keyloggers. Peter and Glenn [15] propose a hypervisor based mitigation technique for keylogger spyware attacks. This technique requires deploying two virtual machines, which include the trusted VM and the untrusted VM. Most of tasks are performed in the untrusted VM, while any keyboard activity can only happen in the trusted VM. As a result, this method may not be applied to protect the pre-installed OS. HookScout [16] introduces a binary-centric method to detect the hooks inside the kernel memory. By monitoring these hooks, HookScout can detect the keyloggers that exploit the function pointers inside keyboard drivers. However, this method is limited to detect the data-based keyloggers. Duy Le *et al*. [17] propose a novel method based on the dynamic taint analysis technique [18] to detect kernel-level keyloggers. Since this method relies on an emulator to track the data flow of a keyboard driver, it imposes considerable performance overhead. Chaoting Xuan *et al*. [19] propose an on-demand emulation-based method for shepherding loadable kernel modules. By specifying a group of security policies for confidentiality-violation rootkits, this approach can identify kernel-level keyloggers effectively. Nevertheless, it may introduce significant performance cost for the emulation. Jesus Navarro *et al*. [20] propose a Virtualization-Aware method to mitigate kernel-level keyloggers. Since this method requires modifying the existing OS kernel, it cannot be applied to protect the commodity OS. Recently, Fengwei Zhang *et al*. [21] present a framework to secure password-based logins on commodity OS by leveraging CPU System Management Mode. Unfortunately, this method cannot defend against the keyloggers that read the keyboard buffer directly to obtain the keystrokes using DMA. Yueqiang Cheng *et al*. [22] present a fine-grained I/O protection framework based on cryptographic and virtualization techniques. Compared with this method, our approach has three advantages: (1) lightweight; (2) on-the-fly protection; and (3) no modification to the target OS. Gabor Pek *et al*. [23] propose a virtualization-based protection system to detect unknown malware on live systems. Different from our work, this system focuses on user-level protection, and it cannot handle kernel-level malware.

9. CONCLUSION

In this paper, we present LAKEED, a lightweight kernel mode keylogger detection system based on the virtualization technology. We exploit the late launch feature provided by recent hardware so that our detection mechanism can be added on-demand. Moreover, we utilize the hardware assisted paging to isolate the target kernel extension and keyboard drivers in different protection domains. By monitoring the potential execution transfers between the protection domains, the keylogger behaviour can be identified. Our evaluations show that LAKEED can detect the keylogger functionality in the kernel extension and prevent it from recording user keystrokes.

REFERENCES

1. KeyGrabber, "Hardware keylogger-wifi usb hardware keyloggers," http://www. keelog.com, 2013.
2. A. Fattori, R. Paleari, L. Martignoni, and M. Monga, "Dynamic and transparent analysis of commodity production systems," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 417-426.
3. Y. Cheng, X. Ding, and R. H. Deng, "Efficient virtualization-based application protection against untrusted operating system," in *Proceedings of the 10th ACM Symposium on Information*, *Computer and Communications Security*, 2015, pp. 345-356.
4. O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 265-278.
5. H. Y. Aravind Prakash, E. Venkataramani, and Z. Lin, "Manipulating semantic values in kernel data structures: Attack assessments and implications," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013, pp. 1-12.
6. Intel Corporation, "Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide," http://www.intel.com/Assets/PDF/manual/253669.pdf, 2012.
7. P. Stewin and I. Bystrov, "Understanding DMA malware," in *Proceedings of the 9th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2012, pp. 21-41.
8. G. Hoglund and J. Butler, *Rootkits*: *Subverting the Windows Kernel*, Addison-Wesley Professional, USA, 2005.
9. E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis and S. Ioannidis, "You can type, but you can't hide: A stealthy GPU-based key-logger," in *Proceedings of the European Workshop on System Security*, 2013, pp. 1-6.
10. Clandestiny, "Klog – a filter driver example using a kernel key logger," http://www.rootkit.com/newsread.php?newsid=187, 2009.
11. M. Russinovich, "Ctrl2cap," http://technet.microsoft.com/en-us/sysinternals/bb8975-78.aspx, 2006.
12. Elite Keylogger Software, "Elite keylogger for windows," http://www.elite-keylogger.net/, 2015.
13. R. Sailer, X. Zhang, T. Jaeger, and L. V. Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of USENIX Security Symposium*, 2004, pp. 223-238.
14. S. Ortolani, C. Giuffrida, and B. Crispo, "Klimax: Profiling memory write patterns to detect keystroke-harvesting malware," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, 2011, pp. 81-100.
15. P. C. S. Kwan, and G. Durfee, "Practical uses of virtual machines for protection of sensitive user data," in *Proceedings of the 3rd International Conference on Information Security Practice and Experience*, 2007, pp. 145-161.
16. H. Yin, P. Poosankam, S. Hanna, and D. X. Song, "Hookscout: Proactive binary-

centric hook detection," in *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2010, pp. 1-20.
17. D. Le, C. Yue, T. Smart, and H. Wang, "Detecting Kernel level keyloggers through dynamic Taint Analysis," Technical Report, WM-CS-2008-05, Department of Computer Science, College of William Mary, 2008.
18. J. Newsome, and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005, pp. 1-17
19. C. Xuan, J. A. Copeland, and R. A. Beyah, "Shepherding loadable kernel modules through on-demand emulation," in *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2009, pp. 48-67.
20. J. Navarro, E. Naudon, and D. Oliveira, "Bridging the semantic gap to mitigate kernel-level keyloggers," in *Proceedings of the Security and Privacy Workshops*, 2012, pp. 97-103.
21. F. Zhang, K. Leach, H. Wang, and A. Stavrou, "Trustlogin: Securing password-login on commodity operating systems," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 333-344.
22. Y. Cheng, X. Ding, and R. H. Deng, "DriverGuard: virtualization-based fine-grained protection on I/O flows," *ACM Transactions on Information and System Security*, 2013, pp. 1-30.
23. G. Pek and L. Buttyan, "Towards the automated detection of unknown malware on live systems," in *Proceedings of IEEE International Conference on Communications*, 2014, pp. 847-852.

Donghai Tian (田东海) received his Ph.D. degree in Computer Science from Beijing Institute of Technology in 2012. He is currently an Assistant Professor in School of Software, Beijing Institute of Technology, China. His research interests include system security and virtualization technologies.

Xiaoqi Jia (贾晓启) received his Ph.D. degree in Computer Science from the Graduate University of Chinese Academy of Sciences in 2010. He is currently a Professor in the Institute of Information Engineering, Chinese Academy of Sciences, Beijing. His research interests include operating system security, software security, cloud security, and virtual machine technologies.

Junhua Chen (陈君华) received his Ph.D. degree in Computer Science from Beijing Institute of Technology in 2010. He is currently an Associate Professor in Key Laboratory of IOT Application Technology of Universities in Yunnan Province, Yunnan Minzu University, China. His research interests include information security and Internet of things.

Changzhen Hu (胡昌振) received his Ph.D. degree in Mechatronics Engineering from Beijing Institute of Technology in 1996. He is currently a Professor at Beijing Institute of Technology. His research interests include information security, network, and pattern recognition.