Shepherding Loadable Kernel Modules through On-demand Emulation

Chaoting Xuan[1], John Copeland[1], and Raheem Beyah[1,2]

School of Electrical and Computer Engineering, Georgia Institute of Technology
Department of Computer Science, Georgia State University

Abstract. Despite many advances in system security, rootkits remain a threat to major operating systems. First, this paper discusses why kernel integrity verification is not sufficient to counter all types of kernel rootkits and a confidentiality-violation rootkit is demonstrated to evade all integrity verifiers. Then, the paper presents, DARK, a rootkit prevention system that tracks a suspicious loadable kernel module at a granite level by using on-demand emulation, a technique that dynamically switches a running system between virtualized and emulated execution. Combining the strengths of emulation and virtualization, DARK is able to thoroughly capture the activities of the target module in a guest OS, while maintaining reasonable run-time performance. To address integrity-violation and confidentiality-violation rootkits, we create a group of security policies that can detect all avialiable Linux rootkits. Finally, it is shown that normal guest OS performance is unaffected. The performance is only decreased when rootkits attempt to run, while most rootkits are detected at installation.

Keywords: Rootkit Prevention, Virtual Machine Monitor, Emulator, On-demand Emulation.

1 Introduction and Background

The security issue of the operating system extensions has been studied for years. Unfortunately, the fact is that many commodity operating systems (e.g., Windows XP and numerous Linux distributions) don not provide the defense against those malicious kernel extensions. In recent years, academics propose a "Out-of-the-Box" approach [2][3][4][5][6][32][35][36][37] to protect detection software by placing it outside the target (guest) OS, e.g. hypervisors (virtual machine monitor), external co-processor. This approach creates strong isolation between detection software and malware such that the former is "invisible" to the latter, (most likely) surviving its attacks accordingly.

Rutkowska [7] proposed a taxonomy that classifies rootkits according to how they interact with operating systems. Type I rootkits refers to those that tamper with the static part of an operating system, e.g. kernel text, system call table and IDT; Type II rootkits refers to those that modify the dynamic part of an operating system, e.g., the data section. Since contemporary OSs are not designed

U. Flegel and D. Bruschi (Eds.): DIMVA 2009, LNCS 5587, pp. 48–67, 2009.

to be verifiable, a large amount of dynamic kernel objects that can potentially be exploited by type II rootkits present challenges to security communities [8]. Recent progresses made in rootkit researches [9][10][11][13][14] reveal that hackers may take advantage of some hardware features to construct stealthy rootkits to beat the existing rootkit detectors.

Kernel run-time protection mechanisms can be categorized as detection and prevention. Inherent limitations of rootkit detection mechanisms are discussed in Section 2. Previous run-time rootkit prevention approaches [31][32] focus on protecting the benign kernel code and thwarting malicious kernel code. One key issue here is how to determine the goodness and trustworthiness of any piece of kernel code. Unfortunately, previous approaches did not give in-depth analysis of this problem and just simply assume it is a priori knowledge to end users or protection systems, which is not true in practice. To date, there is no such commodity operating system that strictly control the kernel code loading based on both goodness and trustworthiness of kernel code. Even Microsoft's driver code signing [34] is just employed for the identification of driver code authors, but not for assuring the goodness of signed drivers [33]. The effectiveness and robustness of this mechanism are still being questioned [1][33]. In the end, people have to make decision on whether to install a useful but potentially unsecure driver, which is a challenge that is not addressed by previous approaches.

In this paper, we propose a rootkit prevention approach that tackles the challenge above, while enhancing the existing prevention approaches. The basic idea is to sandbox a suspicious loadable kernel module in an emulator and to assure its goodness by enforcing a group of well-selected security policies. Based on open source software Qemu and Kqemu [12], we designed and implemented a software system, namely DARK that uses on-demand emulation to provide powerful defense against kernel malware. In DARK, when a rootkit tampers with a kernel object or hardware object, its illegal behavior is captured and blocked. In the meanwhile, VM emulation takes place only at the time that a suspicious module is executed. Further, most operations of the VM are performed in virtualization mode. Thus, the substantial execution overhead caused by emulation is avoided. Our contribution in this work includes:

1. Identification of non-integrity-violation rootkits that can escape kernel integrity verifiers.
2. Implementation of a novel rootkit prevention system based on on-demand emulation to sandbox a suspicious kernel module.
3. Creation of a group of security policies to detect and block all rootkits we collected.

The rest of paper is structured as follows. First, we explain the limitation of current rootkit detection mechanisms in Section 2. Section 3 presents the design and implementation of on-demand emulation. Section 4 and 5 describe the details of creating and enforcing security policy. We present the security and performance evaluation results in Section 6 and introduce related work in Section 7, while Section 8 concludes the paper.

2 Limitations of Rootkit Detection Techniques

Run-time rootkit detection methods proposed by researchers can be divided into two categories: specific rootkit detection and generic rootkit detection. Methods in the first category focus on capturing specific type of rootkits. For example, Cross-view diff-based method [6][25] just targets rootkits that conceal disk objects (files and registries); Lycosid [35] is intended to discover hidden process only. On the contrary, methods in the second category are designed to counter broad types of rootkits. To the best of our knowledge, the most generic rootkit detectors known to the public are kernel integrity verifiers [2][3][4][5][21][24][37]. Kernel integrity verifiers concentrate on examining the states of some kernel objects to ensure that illegal tampering of these objects don not occur. They are effective at defeating integrity-violation rootkits. Unfortunately, theses kernel integrity verifiers suffer two fundamental weaknesses: incompleteness of assuring the integrity of dynamic kernel objects; inability of detecting non-integrity-violation rootkits, like confidentiality-violation rootkits and hardware-exploiting rootkits. These two weaknesses are discussed in detail below.

2.1 Dynamic Kernel Objects

Most kernel rootkits are implemented in the form of kernel modules (drivers). Hence, they share the same virtual memory environments as operating system. No matter whether a kernel object (structure, list, text and so on) is exported or not by the OS, a rootkit can always directly access and tamper with it after being loaded to the kernel. In fact, direct kernel object manipulation (DKOM) is one common technique employed by rootkit writers [28]. A kernel object could reside on either permanent memory area (text, dss) or transient memory area (stack and heap); its content could be constant or changeable. A kernel object is static if its memory address is permanent. Otherwise, this object is dynamic. Defending a static kernel object is straightforward, as its location and content are relatively easy to identify. On the other hand, protecting a dynamic object could become challenging due to the following four reasons. First, in comparison with static objects, the population of dynamic objects is much larger, and enumerating all dynamic kernel objects at any time could be impractical. Second, since integrity verifiers have to wake up to work periodically, they miss catching lots of short-lived dynamic objects, e.g., local variables in stacks. Third, a detector's recognition of dynamic objects can be attacked by rootkits so that those objects are invisible to the detector. For examples, rootkits can alter the page table to hide kernel objects from detectors, or remove an element from a link list to make it untraceable. Last, the content of a kernel object can be unpredictable and detectors are unable to differentiated good and bad values. One such example is the entropy pool of Linux kernel, which can be manipulated by rootkits to compromise Linux Pseudo-Random Number Generator (PRNG) [8]. In summary, kernel integrity verifiers cannot assure the integrities of all dynamic objects in a kernel.

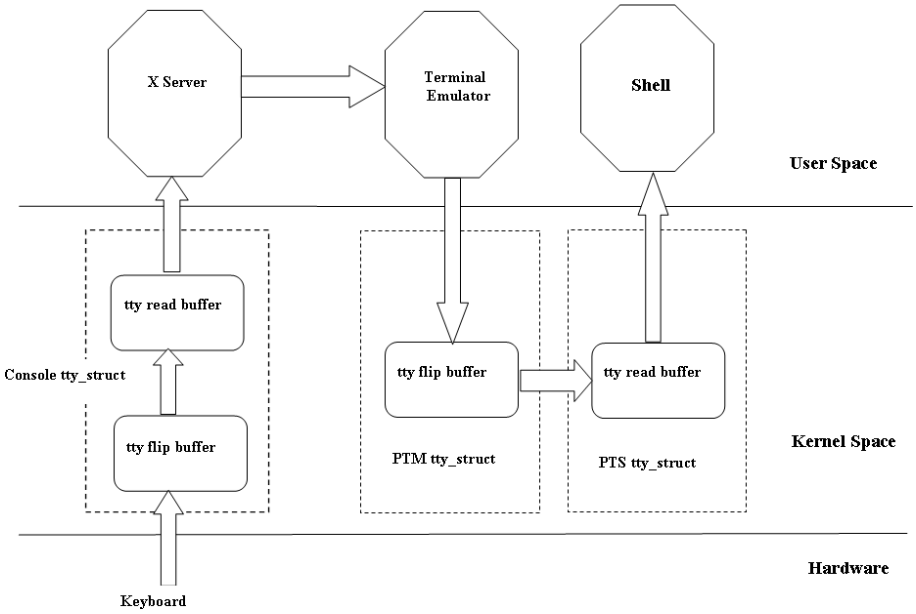

Fig. 1. Key data flow in Linux desktop

2.2 Non-integrity-violation Rootkits

Non-integrity-violation rootkits are rootkits that launch attacks while not manipulating any kernel objects, so kernel integrity verifiers can not catch them. One type of non-integrity-violation rootkits is hardware-exploiting the rootkit, which misuses hardware feature or configuration to achieve their goals. Another type of non-integrity-violation rootkits: confidentiality-violation rootkits. They break the kernel data confidentiality while preserving the data integrity. One class of candidates for the confidentiality-violation rootkits is data theft rootkits, e.g., keyloggers and network sniffers. Next, we demonstrate one confidentiality-violation rootkit: a Linux keylogger (called darklogger) that can sniff keystrokes without illegally changing any kernel object.

Today, common Linux desktop environments like Gnome and KDE use X window systems to manage terminal service: interacting with the keyboard and mouse, drawing and moving windows on the screen. The key data flow in a typical X window system is shown in figure 1. On the X server, the key reading path from keyboard to user space consists of at least two threads working in tandem: a top thread originating from a user process that issues read requests, and a bottom thread originating from the interrupt service routine that reads the key data from the keyboard. Two kernel buffers, *tty flip buffer* (`tty_struct.flip.char_buf`) and *tty read buffer* (`tty_struct.read_buf`), store the key data (interpreted by keyboard driver) and provide the synchronization between the top thread and the bottom thread. When the top thread asks for data and the tty flip buffer is empty, the thread goes to sleep; when the bottom thread fills new key data to

the *tty flip buffer*, it awakens the top thread who copies the new data from the *tty flip buffer* to *tty read buffer* and then to user space. In figure 2, when a key is generated by keyboard and travels to the shell, it may be kept in four kernel buffers. By adding hooks or patching code, traditional keyloggers hijack the control flow of kernel's processing key data. Darklogger takes a passive approach based on the observation that *tty read buffer* is a large-size circular buffer and a char data, representing a key, in the buffer is not wiped off until the head pointer of the buffer moves back to its location, where a new char data is written. Since human's keystroke speed is relatively slow (less than 30 characters/second) and the size of tty read buffer is large (4k), it takes more than 2 minutes to fill up the entire buffer. Darklogger is a kernel thread that wakes up every 10 seconds to read the *tty read buffer* and acquire all key data. Based on the positions of the head and tail pointers in the buffer, Darklogger is able to extract the key data of the last period. Because Darklogger just uses the legal kernel APIs and does not maliciously hook any function or modify any kernel data object, it can evade all kernel integrity verifiers.

Following the spirit of sandboxing program [29], DARK captures the interactions between a rootkit and the rest of a kernel. The kernel objects visited (memory read, write and function call) by a rootkit are recorded and analyzed regardless of their locations, lifespan and contents. To DARK, the rootkit defense is an access control problem and its success depends on the effectiveness of the security policies. Last, it should be pointed out that DARK is not designed to withstandrookits that access the kernel in abnormal ways, e.g., directly writing kernel memory or injecting malicious code to kernel by exploiting the vulnerabilities of benign kernel code. These attacks have been well addressed by previous rootkit prevention systems [31][32].

3 On-demand Emulation

Virtual Machine Monitors (VMM) and emulators are two types of hypervisors that support and manage multiple virtual machines (VM). A VMM seeks to achieve high performance by directly executing most instructions of a VM on the host (physical) CPU. In contrast, an emulator translates each VM's instruction to host instructions so to provide different types of virtual CPUs to its VMs, paying the cost of poor performance. Due to their deep inspection capabilities, some researchers use emulators to perform various security related tasks, e.g. malware detection and analysis.

DARK is a hybrid system that combines the strengths of VMMs and emulators to offer better system security and performance. It contains three components: a VMM, an emulator and a virtual machine (VM) where a guest OS is installed. In virtualization mode, the virtual machine runs on top of the VMM to gain nearly native speed. When a suspicious module is to be executed in the VM, the VMM is informed to take control of the VM. Then, the VMM collects the virtual CPU state and MMU status data, and sends them to the emulator. Thus, DARK is switched to emulation mode. Once receiving the VMM's virtual CPU state,

the emulator restores the VM's operation and start monitoring the module's activities and enforcing the security policies accordingly. When the execution of the module's code is completed, the emulator suspends the VM and passes its control back to VMM with the current virtual CPU state and MMU status data. The VMM restores the VM and DARK is switched to virtualization mode. The emulation is required only when the target module is executed, and most of VM codes still run on VMM.

3.1 Design

The primary task of the on-demand emulation is to trap the module execution in a VM. However, a module may have many non-privilege instructions and their executions in a VM cannot trigger exception or interrupt, which is the only way of transferring the control from VM to VMM in a virtual machine system. DARK addresses this problem by exploring the paging mechanism of operating systems. A *present bit* in the page table entry indicates whether a virtual page has been assigned a physical page frame. When the CPU accesses a virtual page whose *present bit* is 0, memory management unit (MMU) generates a page fault. Then, an interrupt routine is invoked to allocate a physical page frame and copy the page data from the swap area or disk file (demand paging) to this physical page frame. As Linux never swaps kernel codes to disk, the *present bits* of kernel code pages are always set to 1. DARK can trap a module by clearing the *present bits* of its code pages in the virtualization mode. Later, when the module is to be executed, VM issues a page fault. Thus, the VMM of DARK intercepts the exception and passes the control to emulator, who sets those *present bits* back to 1 and starts executing and monitoring the module in the emulation mode. To maintain the integrity of the existing page fault mechanisms, the page fault handler of guest OS should be modified to properly deal with these manipulated page faults.

Before loading a module to guest OS, the DARK user decides whether to monitor the module or not. If yes, the emulator is notified of the module name. To change the *present bits* of the module before its execution, the guest OS issues a software interrupt through instruction "int 0x90". The VMM catches the interrupt, and hand it over to the emulator. Then, the emulator fetches the module name from the VM image and compares it with the one defined by DARK user to decide if the current module is right target. If two names are different, DARK gives up monitoring and switches back to virtualization. Otherwise, DARK kicks off the monitoring with the following steps. First, the emulator queries the text (code) range of the target module from the module list of the guest OS, and sends it to the VMM. Then, it clears the module *present bits* and transfers the control to VMM, forcing the system into the virtualization mode. Later, when the module is to be executed, the VM generates a page fault, which is trapped to the VMM. The VMM uses the text range of the module to identify that the faulty instruction comes from the target module, and transfers the VM control to emulator. After setting the module *present bits* to 1, emulator restores the VM sessions and starts the monitoring process

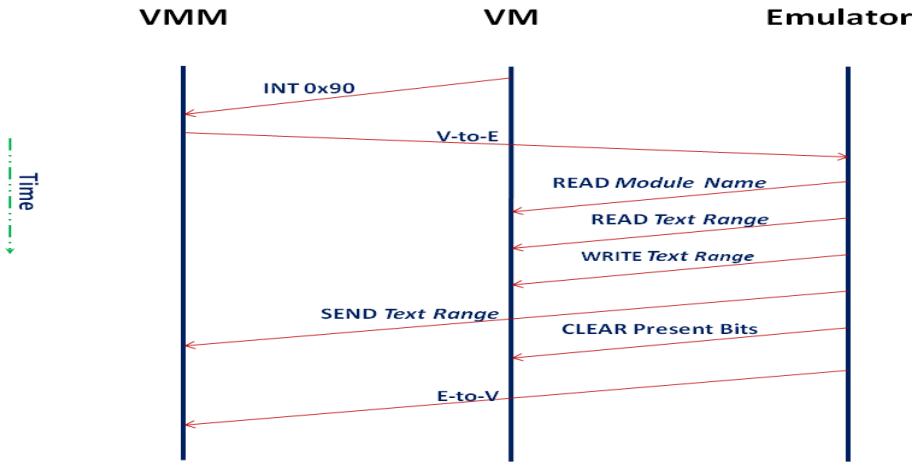

Fig. 2. Partial on-demand emulation process

again. In this way, DARK moves the VM control between VMM and emulator back and forth depending on if the VM executes module code. Figure 2 depicts this on-demand emulation process. When the module is unloaded, DARK turns off on-demand emulation by cleaning up their monitoring records and set the corresponding *present bits* in the VM to 1.

3.2 Implemenation

DARK is built on Qemu and Kqemu [12], who run on any X86 CPU regardless of the hardware virtualization support. Both the guest OS and the host OS are Redhat Linux. Qemu is a hardware emulator that uses binary translation to simulate processor and peripherals, while maintaining a reasonable speed. Kqemu is a kernel module that works with Qemu to provide virtual machine monitor function. In the full virtualization mode of a Qemu/Kqemu system, all user-mode instructions and some kernel-mode instructions of a VM can be directly executed on the host CPU. For security reason, the kernel-mode instructions for memory accesses in the VM have to be intercepted and interpreted by Kqemu. This is done by clearing the global descriptor table (GDT) and local descriptor table (LDT) when VM runs in kernel mode. Thus, any kernel-mode memory access in the VM will cause a general protection fault. Kqemu captures these faults and interprets the instructions in the kernel. Because Kqemu needs Qemu to handle some corner cases such as interpreting HLT instruction, some components of the on-demand emulation framework are already available in orginal Qemu and Kqemu software. To enable the module tracking, DARK modifies the switch control code of the existing on-demand emulation framework. In particular, DARK adds the following business logics to the interrupt handler and V-to-E (virtualization to emulation) control code (in common/module.c and common/kernel.c) of Kqemu:

1. *If an interrupt vector number is 0x90 or 0x91, perfom emulation switching*
2. *For a page fault, if the faulty instruction address is within the text range of target module, perfom emulation switching.*

Moreover, we add one boolean variable to Qemu's E-to-V (emulation to virtualization) control code to ensure that virtualization switch is disabled when the current instruction is from the target module and vice versa.

In addition, we instrument the guest OS kernel `(Linux version 2.4.18):` adding two assembly instructions to `sys_init_module and sys_delete_module` functions in `kernel/module.c`. The first instruction issues a software interrupt 0x90 before loading a module; the second one issues the interrupt 0x91 after unloading a module. DARK obtains a module name by reading the module descriptor from the kernel module list. Further, we modify the Linux module loader (insmod.c) to put the module text range in the runsize and `kernel_data` fields of the module descriptor, which allows DARK to read the text range later. In Linux, all processes share one kernel page table that can be accessed from the kernel master page global directory `swapper_pg_dir`. We use this global variable to locate the page table entries of the target module and rewrites the module *present bits* as described in Section 3.1. Last, we alter the page fault handler of the guest OS such that it ignores the page faults caused by the target module execution.

4 Security Policy

DARK does not aim to build perfect security policies to catch all rootkits. In fact, modern operating systems are not designed to be traceable and verifiable, so the creation of such "perfect" policies may be impossible. Rather, similar to SELinux [30], DARK provides a policy framework that gives security administrators the flexibility to write their own security policies. To demonstrate the effectiveness of DARK, we compose a group of security policies that are good enough to detect most existing Linux rootkits and raise the bar for future kernel exploits.

4.1 Policy Framework

DARK treats the rootkit detection as an access control problem: a malicious module needs to illegally access the other part of kernel to perform the attack. DARK's security policy is composed of a group of access control rules whose format is given in table 1.

In table1, *subject* is a kernel module that is to be monitored. A module's home space contains: object (code and global data) section, stack and heap. Any instruction issued from a module space is regarded as a representative of this module, and should be monitored. Note DARK can apply various policies to different modules, which is discussed later. *Operation* indicates the way that a module interacts with the rest of kernel. DARK tracks three types of operations performed by a module: read, write and call. First two are memory access

Table 1. Rule format of Dark

Subject	**Operation**	**Object**	**Action**
{*module a, b, c ...*}	{*read, write, call*}	{*hardware objects, kernel objects*}	{*reject, alarm*}

operations; call is an action where a module invokes functions exported by OS and other modules. Although a module may influence the kernel objects in other means, e.g., return of an external call, creating a system exception, these three operations are sufficient for DARK to detect the rootkits we know.

Object refers to those system resources and services accessed by a module. Two types of system objects are included in DARK: hardware objects and kernel objects. The former contains dedicated registers, IO ports and IO mapped memory. Many of these hardware objects are crucial to system security. For example, the register IDTR holds the linear address of interrupt descriptor table which is used by CPU to transfer an interrupt to the corresponding Interrupt handler. Hijacking this register allows hacker to amount various attacks, `e.g.`, installing a `virtual-machine-monitor` based rootkits [13][14]. Kernel object is a software concept, and one kernel object is a group of kernel data or code that is semantically meaningful to software developer such as a pointer and a function.

In DARK, a policy rule that contains a hardware object is called system rule; a rule whose object field is a kernel object is called kernel rule. A hardware object that has only one representation in DARK, and it may be a register name, or IO port number or memory address. One kernel object has two representations: one is software-level representation such as variable names and function names, which is used by DARK users to make policies; the other is hardware-level representation and it is a memory address of the corresponding software object. Since DARK enforce policy at hardware level, fore a kernel rule, it's necessary to translate its software-level representation to the hardwarelevel representation, which is called policy translation.

DARK's kernel rules may contain both static kernel objects and dynamic kernel objects. A static kernel object's memory address is determined when the kernel is build, so this object's location is fixed all the time, e.g., system call table. Conversely, a dynamic kernel object's location can only be decided at run time, e.g., a process' page table. A kernel rule containing a static object is called static kernel rule; a kernel rule containing a dynamic object is called dynamic kernel rule. Unlike static kernel rules whose policy translation can be performed before a VM is powered on, policy translation of the dynamic kernel rules has to be postponed to run time.

DARK takes two actions on a policy violation: reject and alarm. Reject denotes that DARK immediately stops executing the target module and prevents any further damages. In Linux, removing a module is more complex and risky than deleting a process from the system, and the former can corrupt the OS's operation integrity and reliability. Current implementation of reject action terminates the VM, and writes a warning message to a log file on the host OS.

Granular failure remediation is of the future work. DARK's alarm action only requires generating the logging messages instead of turning off the whole system. Determination of a reject or alarm action for a rule is based on the consideration of multiple factors: severities to system security, reliability and stability. For those attacks that not only compromise the security but also greatly degrade the system reliability and satiability, reject should be the choice, e.g., runtime patching of the kernel text; For other attacks, terminating the current system operation is not necessary, and alarming is probably sufficient such as sniffing network traffics.

4.2 Established Rules

DARK's policies are constructed based on common knowledge of the OS security and observation of attack patterns of the existing rootkits. Total 19 kernel rules were created and shown in table 2. Among them, four read rules and one call rule are used to address the data theft rootkits as discussed in 2.2. The remaining 14 write rules deal with kernel integrity. Eleven dynamic rules employ seven global variables as the starting points of policy translation. Among them, six global variables are single/double linked lists and the other one (`proc_root`) is associated with binary tree data structure. Note these global variables should be write-protection as well. Otherwise, rootkits may modify the variables to hinder the policy translation. We found that early-stage rootkits tend to manipulate the static kernel objects such as system call table and kernel text. These objects are critical to the system reliability and stability, any illegal modification of them should be rejected at once. Kernel objects contained in Rule 5 and 17 are such examples. On the other hand, some kernel rules are devised to counter the threats in the future, while not being hit by any existing Linux rootkit. For example, it has been reported that some Windows rootkits tamper with the kernel memory management system to hide some kernel objects. It can be foreseen that hackers may apply the same technique to Linux rootkits down to the road. Rule 6 and 10 are designed to achieve such purpose. Rule 9 and 16 in table 3 are optional, because many normal networking drivers may violate them and enforcing these rules possibly generates false alarms. The usages of optional rules depend on user's knowledge to the target modules. In addition to kernel rules, we created 11 system rules, and most of them are applied to special system instructions that handle critical system-level functions, e.g., SGDT and WRMSR.

5 Enforcement

DARK stores the security rules to a local file called policy.dat. This file contains the system rules, static kernel rules and software-level dynamic rules. When a VM is started, DARK forks a thread that performs three tasks: 1. loading the policy.dat to the RAM; 2. periodically translating dynamic kernel rules to the hardware-level representation; 3. transforming all memory-access rules to the hash-table based rules as discussed in Section 5.1. This thread stores all the rules in several global variables, which are used to enforce the policy at run time.

Table 2. Kenrel Rules of Dark

ID	Name	Operation	Kernel Object	Data Type	Action	Dynamic	Optional
1	Console TTY Buffer	Read	console_table	tty_struct	Alam	No	No
2	Exception Table	Write	__start_ext__table	Exception_table_entry	Alarm	No	No
3	GDT Table	Write	gdt_table	Array	Reject	No	No
4	IDT Table	Write	idt_table	Array	Reject	No	No
5	Kernel Text	Write	_text	N/A	Reject	No	No
6	MM List	Write	init_task	mm_struct	Alarm	Yes	No
7	Module List	Write	module_list	Module	Alarm	Yes	No
8	Module Text	Write	module_list	N/A	Reject	Yes	No
9	Netfilter Hooks	Call	nf_rejister_hook	N/A	Alarm	No	Yes
10	Page Table	Write	init_task	N/A	Reject	Yes	No
11	Proc Dir Entry List	Write	proc_root	proc_dir_entry	Alarm	Yes	No
12	Proc Inod Ops List	Write	proc_root	proc_inode_operation	Alarm	Yes	No
13	Proc file Ops List	Write	proc_root	Proc_file_operation	Alarm	Yes	No
14	PTM TTY Buffer	Read	ptm_table	tty_struct	Alarm	Yes	No
15	PTS TTY Buffer	Read	ptm_table	tty_struct	Alarm	Yes	No
16	Socket Buffer List	Read	skbuff_head_cache	sk_buff	Alarm	yes	Yes
17	Syscall Table	Write	syscall_table	Array	Reject	No	No
18	Task List	Write	init_task	task_struct	Alarm	Yes	No
19	Task State Segment	Write	init_tss	Array	Reject	No	No

When a suspicious module is to be executed, the emulator takes control of the VM and begins policy enforcement. Concretely, DARK intercepts all memory access instructions and some system instructions at the binary translation of Qemu. Note an alternative method is to change the Qemu's simulated MMU to capture the memory accesses. However this method cannot enjoy the benefit of code caching and suffers more performance penalty. For each of the monitored instructions, DARK checks the corresponding rules. If an instruction hit a rule, DARK takes the action defined in the rule. For alarm, DARK writes one warning messages to the system log on the host machine. The message includes the module name, the instruction's address and the rule id. For reject, DARK generates an alarm and then power off the VM by terminating the current Qemu process.

5.1 Hash Table

The data structures that hold memory access rules should be selected prudentially, as inappropriate data structure might hurt system performance. DARK's memory access rules are initially defined as a series of memory intervals. One memory interval, like (0xC03254fa, 0xC03256a0), is called one memory bucket. Some dynamic rules, like socket buffer descriptors, comprise a large amount of memory buckets. If they are stored in linked lists, DARK needs traverse thousands of memory buckets (with various sizes) to inspect one instruction in linear time of n. We present a data transformation method that converts a link list of memory buckets to two hash tables. Since hash table lookups have the complexity of O(1), it can significantly reduce the enforcement overhead.

Similar to the OS concept of a 32-bit page frame, DARK uses 10-bit and 5-bit page frames in the transformation. The memory interval of a bucket is broken into multiple 10-bit or 5-bit page frames and each page frame has one entry in a hash table. Two hash tables stores 10-bit and 5-bit page frame rules respectively. Figure 3 lists the C implementation of the converting routine. The selection of 10 and 5 bit page frames is based on the observation that most memory buckets created by DARK are either large (at the page level) or small (less than 200 bytes). This division ensures that each hash table is not overwhelmed due to the hash conflictions. Given a target memory address, DARK first computes its 5-bit page frame address by removing last 5 bit of the memory address, and searches the frame address from the 5-bit hash table; if not found, it then does the same check for 10-bit hash. Thus, only two bit operations and two hash table lookups are needed at most.

5.2 Code Cache

To reduce the emulation overhead, DARK takes advantage of the performance optimization in Qemu. The key technique is to cache the translated code sequences so that they can be directly executed in the future. Each sequence of instructions ending with a single control transfer instruction is called a block. Qemu translates a block in each main control loop and places the translated block to a code cache. All the translated blocks are organized as a hash table and a cached block can be found fast. A block can be linked to another one if

```
VOID convert_bucket (ULONG start_address, ULONG end_address, int bit) {
        ULONG   mask, page_frame, key, current_address;
        struct value_struct *val;

        mask = (1 << bit) – 1;
        page_frame = 1 << bit;
        current_address = start_address;
        while (1) {
                key = current_address & (~mask);
                val = (value_struct*)malloc(sizeof(value_struct));
                val.start_offset = current_address & mask;
                if (current_addess + page_frame < end_address) {
                        val.end_offset = 0;
                        insert_hash_entry(bit, key, val);
                        current_address += page_frame;
                }
                else { //current_address moves to the last page frame
                        val.end_offset = page_frame – (end_address & mask);
                        insert_hash_entry(bit, key, val);
                        break;
                }
        }
}
```

Fig. 3. Source code of the memory bucket transformation routine

it does not contain the indirect branches, avoiding the extra loop cost. DARK only performs the security check at binary translation, so once a block of code is put into the cache, DARK does not examine it any more. Finally, when the code cache is full, Qemu simply purges all blocks in the cache and refills the cache with new blocks. Since DARK's emulator only caches small-size module code, the chance of overflowing the cache is small.

5.3 Security Log

DARK provides the logging capability that keeps record of the interactions between a module and the rest of the kernel. The log includes: memory write and read, function call and IO operations. For memory read and write, DARK prints out the instruction address, and target memory address and content. For function invocation, DARK records the function address, calling instruction address, the first two parameters and return value of the function. However, parameter semantics of a function are unknown, so DARK logs the first 16 bytes in the stack parameter area of the function. Note that DARK only logs the external memory accesses and function invocation. In addition, we create a tool that interprets log records, identifies all heaps that are assigned to the module, and removes them from the log. Combining this logging capability with Qemu's snapshot can provide the abundant data sources for forensic analysis.

6 Evaluation

This section presents the empirical results of the DARK system. The evaluation is composed of two subsections. In the first subsection, the functional effectiveness

of DARK is investigated: whether the security policies are made properly in terms of false positive and false negative detection rates. Then, we conduct the performance evaluation and study the performance impact of on-demand emulation on the VM. DARK is built based on the QEUM 0.8.2 and KQEMU 1.3.0prell. All the experiments are performed on a Dell machine with Intel P4 CPU (2.8 GHz) and 1 GB RAM. The host OS is Fedora Core 5.0 and the guest VM was assigned 256M RAM and 6G hard drive; Guest OS is Red Hat Linux 8.0 with 2.4.18-14 kernel.

6.1 Security

Beside the classification of rootkits given by [7], Petroni [5] classified the rootkits according to their intentions: user-space object hiding (HID), privilege escalation (PE), reentry/backdoor (REE), reconnaissance (REC), and defense neutralization (NEU). In this experiment, we collect 18 rootkits that cover a wide range of attacks. Among them, there are 10 type I rootkits, 8 type II rootkits, 8 HID rootkits, 7 PE rootkits, 3 REE rootkits, 5 REC rootkits and 3 NEU rootkits. In addition, one rootkit from [15] is devised to attack the hardware resources (system BIOS). Unfortunately, the Qemu's BIOS is not updatable, so the rootkit cannot be successfully installed to the test VM. The other 17 rootkits are listed in table 6. A rootkit may have several operation modes and different modes may use different attack tactics. For example, with the technique described in [16],

Table 3. Detection Result of Dark

ROOTKIT	FUNCTION					TYPE	HIT KERNEL RULES		ACTION
	HID	PE	REE	REC	NEU		Load	Operation	
Adore	X	X				I	17	18	Reject
Adore-ng	X	X			X	II	7,12,13	18	Alarm
Adore-ng(hidden)	X	X			X	II	7,12,13	18	Alarm
Darklogger				X		II		15	Alarm
Exception		X			X	I	2	18	Reject
fileh-lkm	X					I	17		Reject
Hookstub		X				I	4	18	Reject
Hp	X	X				II	18	7,12,13	Alarm
KIS	X		X			I	17		Reject
Knark	X	X	X			I	17	18	Reject
Linspy2				X		I	16		Reject
Nfsniffer				X		II	9	16	Alarm
Nushu					X	II		16	Alarm
Pizzaicmp			X			II	9	16	Alarm
Prrf	X	X				II	11,12,13	18	Alarm
Sebek				X		I	7,17		Reject
Srookit	X					I	5		Reject
Vologger				X		I	17	14	Reject
Vologger(local)				X		II		1	Alarm

Adore-ng can optionally hide itself into a benign module, forming a "combo" module. We test the regular Adore-ng and hidden Adore-ng separately. To comprehensively understand the rootkits' behavior, we run several Linux utilities like *ls*, *ps*, *netstat* and *ssh* to verify whether a rootkit works as expected after its installation. Moreover, when a rootkit violates a reject rule, we intentionally instruct DARK not to shutdown the guest VM and make the rootkit continue to run until all testing utilities are finished. Thus, we can catch all security rules that the rootkit hits.

The test result in table 3 suggests that DARK is able to detect all the rootkits with the security rules in table 3. Some rootkits violate multiple rules at the loading stage and operation stage. System call table (rule 17) and task list (rule 18) are primary kernel objects that rootkits target on. Several type I rootkits hijack system call table to hide user-space objects or steal private data. IDT table and kernel exception table are another two static kernel objects that the rootkits tamper with in the test. To type II HID rootkits, proc file system provides exploitable kernel objects that are alternatives to system call table: two such rootkits (adore-ng and prrf) alter the relevant data objects of the proc system to hide processes and network connections. All the PE rootkits modify the user id and group id in the `task_struct` objects to raise a process' privilege level. Another observation is that all rootkits are captured at the loading stage except the Darklogger and Nushu. As we pointed out before, Darklogger is a non-integrity-violation rootkit and does not illegally change any kernel object in the kernel. It just creates a kernel thread and initializes some data structures at the loading stage. Yet, its reading of the PTS buffer is caught by DARK at the operation stage. Nushu manipulates the packets from/to local network adapters by indirectly registering hooks to the kernel through the function `dev_add_pack`. Because this function is not defined in table 3, Nushu escapes the loading-stage inspection. But DARK detects the intrusion when it reads socket buffers at the operation stage. Note that powerful kernel integrity verifiers are still likely to catch the Nushu due to its hooking behavior.

In the experiment, Adore-ng is embedded in the module `iptables_filter` to create a combo module. By comparing the hidden Adore-ng with the regular Adore-ng, we find that they hit the same set of rules. However, the combo module can not be unloaded from the kernel even after we flush the iptable rules and stop the iptable service. After further investigation, we found the reason. Both hidden Adore-ng and the regular Adore-ng modify the kernel module list, which is a list of `module` objects. The regular Adore-ng changes the next fields of the previous and next `module` objects with the purpose of hiding itself, while the combo module alters the `uc.usecount` field of the current `module` object to persist its existence in the kernel. Vlogger is also tested in two operation modes. Although the regular mode offers more powerful features than local mode, the latter turns out to be stealthier: it only alters the dynamic kernel objects and is a type II rootkit.

To estimate the false positive rate of the detection system, we choose 7 categories and total 20 drivers from the Linux source, and execute them in the DARK

system. When we test the network drivers, we deactivate the optional rules 9 and 16 to avoid the false alarms. The test result indicates that 19 of 20 drivers pass the test. The failed module is jdb and it is a journaling block device driver used by Ext3 file system for data recovery. This driver alters the `journal_info` filed of two process' `task_struct` objects, leading to the violation of rule 18. This false alarm implies that the rule 18 is too restrictive and should be revised to only include the sensitive fields that task list members. But, on the other side, this violation does not incur the system termination and we believe that overall quality of the security rules is good.

6.2 Performance

Performance evaluation is intended to measure the impact of on-demand emulation on overall system performance. The module `iptable_filter` from Linux source is chosen to be monitored. First, this module operates at the kernel network stack, which is one of major attacking targets of rootkits. Second, running this module in emulation mode is expected to only degrade the performance of the network subsystem in the kernel, and other subsystems should not be affected. `Iptable_filter` registers three hooks to netfilter and applies the iptable rules to network traffics at three guarding points of the netfilter: input, output and forward. We write a number of input and output iptable rules and neither of them actually blocks the network traffics during the test. Three benchmarks: bonnie [17], iperf [18] and lmbench [19], are performed to examine the performance of disk IO, network IO and the entire system respectively.

Comparing with VMM-only system (pure virtualization system), DARK's overhead comes from on-demand emulation, which is composed of two parts: 1. Context switch between virtualization and emulation; 2. Execution overhead in emulator, including binary translation, policy enforcement and execution of translated code sequences. To identify the contribution of each part to the overall cost, we devise another test system: DARK-CS. It does the context switch from virtualization to emulation when an `iptable_filter` function starts to run. Then, emulator returns the control back to VMM immediately and the `iptable_filter` function is actually executed over VMM. Therefore, context switch between virtualization and emulation is the only overhead of DARK-CS. In the experiment, we run each benchmark in DARK, DARK-CS and VMM-only system.Table 4 shows the test result of bonnie. It's observed that the three systems have little

Table 4. Bonnie test result for 100 M files

	SEQUENTIAL OUTPUT						SEQUENTIAL INPUT				RANDOM	
	Per Char		block		Rewrite		Per Char		block		Seeks	
	K/Sec	%CPU	K/Sec	%CPU	K/Sec	%CPU	K/Sec	%CPU	K/Sec	%CPU	/Sec	%CPU
VMM	8528	64	12755	45	19082	53.0	15805	75	129292	71	3515	84
DARK-CS	8038	61	11715	41	17402	48.2	16860	80	130266	74	4969	85
DARK	8168	67	13949	43	18742	49.8	14480	73	125493	72	5117	83

Table 5. Bonnie test result for 100 M files

	VM as Server (M/Sec)		**VM as Client (M/Sec)**	
	TCP	UDP	TCP	UDP
VMM	21.81.2	1.050.1	26.82.3	1.130
DARK-CS	19.730.5	1.010	23.991.4	1.080.1
DARK	19.600.6	1.000.1	24.051.0	1.080.1

performance difference when running bonnie. This is because bonnie just accesses the files on disk and `iptable_filter` is not being executed. Bonnie's test result suggests that DARK's overall performance is same as VMM-only system when on-demand emulation doesn't take place.

The iperf test result in table 5 reveals the impact of on-demand emulation on overall system performance. TCP and UDP throughputs of DARK-CS are slightly (about 10%) lower than VMM-only's, which indicates that the overhead of context switching is non-negligible but not significant. CPU state transferring, shadow page table synchronization and page fault handling are three main components of context switching in DARK. However, It is still unknown which component should take the responsibility of performance penalty at the moment. Further, it seems that neither component has much room left for performance improvement. Table 5 also suggests that DARK and DARK-CS have indistinguishable TCP and UDP throughputs. This result can be explained by the code caching technique introduced in section 5.2: to a block of module code, binary translation and policy enforcement are performed only at the first time this block of code is executed, and its translated code sequence in the code cache plays the primary role of deciding the performance in the long run. So code caching is effective to reduce the emulation overhead. We also did the performance test with the Lmbench, and test result confirmed the conclusions we draw above. We can not present the test result under the space constraint.

7 Related Work

Ho [20] proposed the concept of On-demand Emulation that can be used to solve the security problems. His system modified the emulator's hardware support to enable the data tainting at the system level. The system was built on Qemu and Xen VMM, and its main application is prevention of malicious code injection by tracking data received from the network as it propagates through the target VM. DARK does not tamper with any VM's hardware setting, and focuses on kernel rootkit detection.

Kernel Integrity Verification [4][5][21][22][24][37] is one popular rootkit detection approach that follows the spirit of Tripwire [23] in protecting the file systems. It builds a baseline database for the measurable objects (e.g., text, static data) of the target guest and periodically queries current states of those measurable objects to detect intrusions by comparing them with the baseline database. As we mentioned before, these integrity verification methods suffers

dealing with dynamic kernel objects and are also incapable of detecting non-integrity-violation rootkits.

Kruegel [27] and Limbo [26] use static and dynamic program analysis techniques to inspect the innocence of a driver off-line. Similar to DARK, both systems create a group of security policies and monitor module behavior. However, they are not run-time rookit detection system, so they suffer fundamental hurdles to static and dynamic program analysis, e.g., code obfuscation, or inaccurate and incomplete analysis result. HookFinder [38] and HookMap [39] are two systems that explore hooking behavior. The former employs the dynmic data tainting to caputer the hooks implanted by rookits, and the later uses the data slicing to identify all potential hooks on the kernel-side execution paths of testing programs such as *ls*, and *netstat*.

SecVisor [31] and NICKLE [32] are two rootkit prevention systems that rely on trusted VMM to enforce life-time kernel integrity. A trusted VMM ensures that only authenticated code can execute in kernel mode, which is a stronger security property than Vista's driver signing. Both systems can protect kernel from code injection attacks including zero-day kernel exploits. DARK is intended to handle the unauthenticated drivers. As long as these drivers follow the behavior specification (security policies) defined in DARK, they are allowed to run in the kernel. So, DARK is an enhancement to the existing prevention solutions.

8 Conclusion

In this paper, we presented a rootkit prevention system to dynamically monitor a suspicious module using on-demand emulation. In addition, we develop a group of security rules to effectively detect rootkits that we gathered, which was demonstrated in the security evaluation. In the end, we show that the performance of a VM is not affected for the majority of system operations. Context switches between emulator and VMM slightly decrease the system performance.

References

1. Rutkowska, J.: Subverting Vista Kernel for Fun and Profit (2006), `http://www.invisiblethings.org/papers.html`
2. Garfinkel, T., Rosenblum, M.: AVirtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the Symposium on Network and Distributed System Security, NDSS (2003)
3. Zhang, X., van Doorn, L., Jaeger, T., Perez, R., Sailer, R.: Secure Coprocessor-based Intrusion Detection. In: Proceedings of the ACM SIGOPS European Workshop (2002)
4. Petroni, N.L., Fraser, T., Molinz, J., Arbaugh, W.A.: Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In: Proceedings of the USENIX Security Symposium (2004)
5. Petroni, N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS (2007)

6. Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection through VMM-Based "Out-of-the-Box" Semantic View Recontruction. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS (2007)
7. Rutkowska, J.: Introducing Stealth Malware Taxonomy (2006), http://www.invisiblethings.org/papers.html
8. Baliga, A., Kamat, P., Iftode, L.: Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In: Proceedings of IEEE Symposium on Security and Privacy (2007)
9. BroFrancis, M.D., Ellick, M.C., Jeffery, C.C., Roy, C.: Cloaker: Hardware Supported Rootkit Concealment. In: Proceedings of IEEE Symposium on Security and Privacy (2008)
10. Heasman, J.: Implementing and Detecting a PCI Rootkit. Technical report, next Generation Security Software Ltd. (November 2006)
11. Heasman, J.: Implementing and Detecing an ACPI BIOS Rootkit. In: Black Hat Europe, Amsterdam (March 2006)
12. Bellard, F.: Qemu and Kqemu (2008), http://fabrice.bellard.free.fr/qemu/
13. King, S.T., Chen, P.M., Wang, Y.M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing malware with virtual machines. In: Proceedings of the IEEE Symposium on Security and Privacy, Washington, DC, USA, pp. 314–327. IEEE Computer Society, Los Alamitos (2006)
14. Blue Pill, http://bluepillproject.org/
15. Scythale. Hacking deeper in the system, http://www.phrack.com/
16. Truff. Infecting Loadable Kernel Module, http://www.phrack.com/
17. Bonnie, http://www.textuality.com/bonnie/
18. Iperf, http://dast.nlanr.net/Projects/Iperf/
19. McVoy, L.W., Staelin, C.: Lmbench: Portable Tools for Performance Analysis. In: Proceedings of the USENIX Annual Technical Conference, pp. 279–294 (1996)
20. Ho, A., Fetterman, M., Clark, C., Warfield, A., Hand, S.: Practical Taint-Based Protection using Demand Emulation. In: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (2006)
21. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In: Proceedings of the ACM Symposium on Operating systems Princeiples, SOSP (2005)
22. Microsoft. Windows Kernel Patch Protection (2008), http://www.microsoft.com/whdc/driver/kernel/64bitpatching.mspx
23. Kim, G., Spafford, E.: The Design and Implementation of Tripwire: A File system Integrity Checker. Technical report, Purdue University (1993)
24. Petroni, N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In: Proceedings of the USENIX Security Symposium (2006)
25. Wang, Y.M., Beck, D., Vo, B., Roussev, R., Verbowski, C.: Detecting Stealth Software with Strider GhostBuster. In: Proceeding of International Conference on Denpendable Network Systems, DSN (2005)
26. Wilhelm, J., Chiueh, T.: A Forced Sampled Execution Approach to Kernel Rootkit Identification. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 219–235. Springer, Heidelberg (2007)
27. Kruegel, B.C., Robertson, W., Vigna, G.: Detecting Kernel-Level Rootkits Through Binary Analysis. In: Proceedings of the 20th Annual Computer Security Applications Conference, ACSAC (2004)

28. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, Reading (2005)
29. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execurtion via program shepherding. In: Proceedings of the USENIX Security Symposium (2002)
30. Security-Ehanced Linux, http://www.nsa.gov/selinux/
31. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In: Proceedings of the ACM Symposium on Operating Systems Principles, SOSP (2007)
32. Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
33. Windows Vista Security Blog, http://blogs.msdn.com/windowsvistasecurity/archive/2007/08/16/driver-signing-kernel-patch-protection-and-kpp-driver-signing.aspx
34. Windows Driver Signing, http://www.microsoft.com/
35. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid. In: Proceedings of the 4th International Conference on Virtual Execution Environments (VEE) (March 2008)
36. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor Suppot for Idnetifying Covertly Executing Binaries. In: Proceedings of the USENIX Security Symposium (2008)
37. Baliga, A., Ganapathy, V., Iftode, L.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. In: Proceedings of the 24th Annual Computer Security Applications Conference, ACSAC (2008)
38. Yin, H., Liang, Z., Song, D.: Hookfinder: Identifying and understanding malware hooking behaviors. In: Proceeding of the Annual Network and distributed System Security Symposium, NDSS (2008)
39. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 21–38. Springer, Heidelberg (2008)