Towards a Tamper-Resistant Kernel Rootkit Detector

Nguyen Anh Quynh, Yoshiyasu Takefuji
Graduate School of Media and Governance,
Keio university
5322 Endoh, Fujisawa, Japan 252-8520
{quynh,takefuji}@sfc.keio.ac.jp

ABSTRACT

A variety of tools and architectures have been developed to detect security violations to Operating System kernels. However, they all have fundamental flaw in the design so that they fail to discover kernel-level attack. Few hardware solutions have been proposed to address the outstanding problem, but unfortunately they are not widely accepted. This paper presents a software-based method to detect intrusion to kernel. The proposed tool named *XenKIMONO*, which is based on Xen Virtual Machine, is able to detect many kernel rootkits in virtual machines with small penalty to the system's performance. In contrast with the traditional approaches, XenKIMONO is isolated with the kernel being monitored, thus it can still function correctly even if the observed kernel is compromised. Moreover, XenKIMONO is flexible and easy to deploy as it absolutely does not require any modification to the monitored systems.

Categories and Subject Descriptors

H.2.0 [**General**]: Security, integrity and protection; D.4.6 [**Security and Protection**]: Invasive Software

Keywords

Kernel Rootkit, Intrusion Detection, Xen Virtual Machine, Linux

1. INTRODUCTION

Detecting the security violation is the fundamental goal of all the Intrusion Detection System (IDS). Since the Operating System (OS) kernel provides the core functions for the whole system, it is very important for IDS to protect the kernel as well. The problem is that once the kernel is compromised, for example by kernel-level rootkits, all the information returned from kernel is no longer reliable. However, almost all the current approaches to discover and protect kernel-level attacks presuppose that the kernel is not breached. This assumption is obviously flawed, because if the attacker successfully penetrates the kernel, he might never fail to patch its code and data to evade the detection, or even disable it. To counter-attack, detection tools attempt to use the not-yet-broken part of kernel to look for rootkits evidence. But then the attacker can study their techniques to adapt his rootkits accordingly and goes on to disable the detection tool. Subsequently, the battle might never finish ([9]).

The root of this trouble lies in the architecture of most OS kernels: Modern kernels are typically designed in monolithic way, which means it is basically a huge program in a whole. As a result, there is no separation between kernel codes, as everything is in the same protection domain. Hence any kernel code can read from or write to any other part of kernel. Thus if a rootkit is inserted into the kernel, it can disable the related code of the detection tool running inside kernel. This problem is well-known for a long time, but cannot be completely fixed because of the in-use architecture in the current OSes.

Recently researchers proposed several hardware-based solutions to address the above problem: The system runs with special add-in hardware (for example a PCI card), and they proposed to do the detection from these hardware ([10], [26]). Because the OS has no way to tamper the hardware, it is supposed that the attacker cannot defeat the detection tool staying in these devices. However, this approach is quite impractical: it requires new and possibly expensive hardware, and it also needs special support from the OS. All these reasons make this approach hard to be deployed in large scale.

Another promising approach is to put the IDS into the Virtual Machine Monitor (VMM) to detect intrusion inside Virtual Machine running on top of VMM ([6]). The idea is to exploit the unique advantage of Virtual Machine (VM): while all the VMs are separated and in different protection domain (so they cannot interfere each other), we can inspect the memory of them from outside [1]. As a result, if we put the detection tool outside of the observed VMs, and inspect the memory of the VM at run-time, we can discover the security problems, while still guarantee that the detection tool cannot be harmed by the attacker inside. Unfortunately the authors only provided the high-level concept of their IDS, which is based on VMWare Workstation, and the resulted IDS was never released to the public.

Inspiring by the idea, our work examines in detail a software-based solution to monitor memory to detect kernel security violation on Virtual Machine (VM) environment. Based on the above principles, we have developed a kernel rootkit detector named XenKIMONO [2] for Xen Virtual Machine Monitor ([2]). In our experiments, XenKIMONO is able to detect all kind of kernel rootkits installed in Linux-based VMs run on Xen. While XenKIMONO is made to run on Xen platform, the principle works for all other hardware-level Virtual Machine Monitor as well.

The rest of this paper consists of 6 parts: part 2 briefly ana-

[1]Usually we can only do that with special access right, from a privileged VM.

[2]KIMONO stands for Kernel Integrity MONitOr.

lyzes few typical kernel rootkits and their infected techniques. Part 3 presents the architecture and implementation of XenKIMONO, while part 4 discusses the advantages together with shortcoming of our solution. Part 5 demonstrates the efficiency of XenKIMONO by evaluating it with several popular kernel rootkits available in the wild, then measures its performance impact. We summarize the related work in the part 6. Eventually we conclude the paper and outline future works in final section, part 7.

2. ROOTKIT OVERVIEW

Rootkits are malicious software that allow the attacker to keep controlling the infected machine, and hide their presence from the view of system administrator. Usually rootkits are installed into the system after the attacker obtains the privileged access of the host. Once running, rootkits modify the host to provide backdoors through which the attacker can spy on the system's activities, and regain access to system in the future without legal authentication. Rootkit is emerging as a major issue for computer users in the last few years, and unfortunately there is no evidence that the problem will stop very soon ([13]).

2.1 Rootkit Categories

Basically we can categorize rootkits into 2 groups:

1. **User-level rootkit**: This kind of rootkit is also called application-level rootkit. They replace critical system utilities like *ls*, *ps*, *netstat*, ... with modified versions that hide the existence of attacker's files, processes and network connections. Fortunately user-level rootkits are quite easy to uncover, because they do not modify the OS kernel: we can get information about files, processes and network connections from kernel, and compare them with the output from user-level utilities to detect the deviation.

2. **Kernel-level rootkit**: This kind of rootkit poses a lot of problem, and brings in the endless battle between anti-malware researchers and bad guys. Kernel rootkit modifies the OS kernel to provide faked information[3]. In that case, even if the system utilities are intact, the information they produce might still be falsified. As a result, the solutions used to track down user-level rootkits are all useless.

Because user-level rootkits are quite easy to deal with and, most importantly, can be technically detected, we do not try to counter them in this paper. Instead, we focus on the kernel-rootkit and attempt to address this kind of malware effectively.

2.2 Rootkit techniques

To understand about the applied techniques of kernel-rootkits, we analyzed some of the most popular ones available in the wild. While there are various other rootkits, they all use the same methods to infect the system and evade the current detection tools.

In general, to successfully install the kernel rootkit into kernel, the attacker must overcome 2 challenges:

1. **Insert rootkit into kernel**: To insert rootkits into kernel, the attacker usually puts the rootkit into a kernel module, then load the module to kernel. Various well-known rootkits use this technique, like *Knark* [14], *Adore* [20], *vlogger* [15] and *Sebek* [21] [4]. To thwart this problem, many systems choose not to support kernel module [5]. The attacker counters the issue by writing directly to kernel memory via memory devices such as */dev/*{*kmem,mem*} on Linux [17]. Unfortunately the administrator cannot always disable these devices, because they are necessary for some applications, such as X.

2. **Hide its existence**: The administrator of the system can detect the kernel rootkit if it is not well covered. In case the rootkit is a kernel module, this module must not be listed from any user-space tool such as *lsmod* [6].

   Besides covering itself, rootkit is usually employed to hide other evidences or activities of attacker, such as backdoor that opens secret ports for unauthorized remote access, or hidden rootkit files. Similar to the case of kernel module above, these activities must not be shown via usual administrative tools such as *ls*, *ps*, *netstat*.

   To hide the rootkit presence, attacker usually employs the following techniques: *Replace system-calls*, *patch jump-tables*, *patch kernel code* and *modify critical kernel objects*.

   (i) **Replace system-calls**: System-calls are kernel functions responsible for system services. To access system resource, user-space application must request services from kernel, and the actual job is done by system-calls. So obviously, if the attacker can replace the system-calls with his code, he can maintain the control on the whole system. In fact, this trick is widely used in many kernel rootkits such as Adore, Knark, vlogger and Sebek.

   (ii) **Patch other kernel jump-tables**: Jump-table is a widely-used technique in implementation of many OSes. Basically jump-table is a list of entry points, which serves as addresses of kernel functions. Reference to the entries in table can be done via a numbered index. Linux kernel provides some critical jump-tables, with notable examples are *Interrupt Descriptor Table* and *Page Fault Handler exception table*. Not surprisingly, these important tables are quickly abused by some sophisticated rootkits ([11], [1]).

   (iii) **Patch kernel code**: While replacing system-calls is a very popular trick in rootkit world, this might expose the rootkit itself. The reason is that the new system-call is put in unusual place, and that is suspicious ([7]). Hence another method is proposed: a system-call can be hijacked by patching the first 7 bytes of its code ([19]). This technique can defeat the rootkit detectors that only check the system-call table integrity.

   (iv) **Modify critical kernel objects**: Besides modifying code and jump-tables, another favorite target of rootkits is kernel objects. Usually this trick is employed to hide the existence of kernel module rootkits or user-space rootkit processes. Rootkits can modify related objects in some ways, so the whole system still function normally, but the objects are hidden from user-space. For example in Linux, kernel modules are declared in a structure of *module* type, and connected together by a double linked-list. The attacker can disconnect the rootkit module from the linked-list, so the module is

[3]Clearly the attacker must somehow gain privileged access before this step.

[4]Though Sebek is proposed to use for honeypot purpose, it embraces a lot of blackhat tricks to function stealthily, hence Sebek can be used as a rootkit.

[5]In Linux, this can be achieved by compiling Linux kernel with option *CONFIG_MODULES=n*.

[6]*lsmod* is a tool used to list kernel modules on Linux systems.

not shown when user-space tool list them[7] ([20], [15], [21]). The same method is applicable with the linked-list of *task_struct* structure in Linux systems to hide rootkit daemons ([23]).

As this technique does not require to patch kernel code or jump-tables, it can evade quite a few rootkit detectors. To detect them, we must foresee all the potential places that can be abused, which is unfortunately not easy in practice.

Simply put, all the rootkits either patch the kernel text or jump table or kernel objects to infect the system, as well as hide their presence. Thus if our IDS is able to detect the modification to these parts of kernel, we might be confident that we can discover various kind of kernel malware.

3. XENKIMONO SOLUTION

3.1 Xen Virtual Machine

The original meaning of VM indicates a number of different identical execution environments on a single computer, each of which runs an OS. The host software which provides this capability is often referred to as a Virtual Machine Monitor (VMM) or hypervisor.

Xen is such an open source and free VMM ([2]). Basically, Xen is a thin layer of software operating above the bare hardware, and Xen exposes a VM abstraction that is slightly different from the underlying hardware. Xen introduces a new architecture called *xen*, which is very similar to x86 architecture. The VM executing on Xen are modified (at kernel level) to work with Xen architecture.

Running on top of Xen, VM is called Xen domain, or domain in short. A special privileged domain named Domain0 (or Dom0 in short) always runs. Dom0 manages other domains (called User Domain, or DomU in short), including jobs like start, shutdown, reboot, save, restore and migrate them between physical machines. The privileged Dom0 can inspect and modify other DomU's memories. However, DomU cannot access memory of any other domain without permission explicitly granted by the corresponding domain.

3.2 XenKIMONO Goals

The design of XenKIMONO is driven by the following goals:

(1) **Tamper-resistant**: XenKIMONO must be able to reliably detect kernel rootkits, even if observed kernel is compromised. Moreover, XenKIMONO must be able to resist the attack from the attacker.

(2) **System independence and flexibility**: XenKIMONO must be able to work without any cooperation from the monitored machine, which means it can work with any system without having to install any special software, or modifying its internal functions. Specifically, for Linux-based VM, XenKIMONO should be able to work with any kernel version of any vendor, as well as any compiled option. If we achieve these goals, our IDS becomes portable and the maintenance cost is significantly decreased.

3.3 XenKIMONO Design

3.3.1 Overview

We put the XenKIMONO, which is implemented in a form of a daemon process named *xenkimonod*, into Dom0 and let it inspect the kernels of other DomUs to detect potential rootkits. Because Dom0 is separated from other domains, and in fact it is in different security protection domain, XenKIMONO is tamper-resistant with malicious attack from DomUs. With this strategy, we achieve the goal (1).

Having XenKIMONO in Dom0 gives us another advantage: our detection tool is invisible to the attacker in DomU, because there is no way to discover another process running in Dom0. Thanks to the strong isolation between the domains, the attacker cannot easily detect XenKIMONO presence, thus XenKIMONO might has a better chance to do its job without the enemy's awareness.

To inspect DomU, XenKIMONO must be able to access to the kernel memory of DomUs. Fortunately this can be done in Xen thanks to some Xen API: From Dom0, XenKIMONO can map the kernel memory of any DomU ([25]) and does all the processing, such as reading or writing, on the mapped memory .

Regarding the goal (2), XenKIMONO must overcome another hurdle: how to understand the "meaning" of the raw memory it has accessed to. Because all we have are only memory pages of the kernel, we must still analyze its state to interpret the data to a higher level, preferred at OS-level structures. For example, we might want to know where in the memory the information about kernel modules is located, and which kernel modules are loaded in the VM. Simply put, we must parse the raw memory to extract out meaningful information to understand what is going on in the observed VM.

All of these requirements can be done thanks to kernel symbols and other information extracted from DomU's kernel binary: We can extract the information about kernel objects from kernel binary, and obtain information about their places in the DomU's kernel thanks to the help of their kernel symbol map file. Given the detail information about kernel types and their addresses, XenKIMONO is able to pinpoint the concerned kernel objects in the protected VMs. Because we can have the necessary information independent of kernel versions or compiled options, XenKIMONO is able to achieve the goal (2).

3.3.2 Detection Methods

To detect the kernel rootkits in DomUs, XenKIMONO proposes 2 strategies: (1) *Integrity checking* to detect illegal changes to kernel code and jump-tables; (2) *Cross-view detection* to detect the malicious modifications to critical kernel objects.

(1) **Integrity checking**: XenKIMONO pays close attention to critical parts of observed kernels. If these parts are modified at run-time, it might indicate that the kernel is being attacked by a rootkit. XenKIMONO will fire alarms in that case, and let the system decides what to do to react.

On the critical parts of kernel, XenKIMONO calculates the hashes [8] of these memory areas at known-good time, when we can make sure that the kernel is "clean". Then at run-time it periodically recalculates these hashes and compare against to the saved values. As the hash algorithm guarantees the unique value for all these kernel memories, any difference indicates the monitored sections have been modified.

(2) **Cross-view detection**: The rootkits usually employ the "lie" strategy to evade the detection, in which the information re-

[7]On Linux, this can be done with the *lsmod* tool.

[8]we use the MD5 algorithm for hashing.

turned to user-space is modified to hide their presence (for example to hide rootkit processes or files). The kernel is thus modified by the rootkit, which leads to the inconsistency between the system layers.

To find the inconsistency in kernel, XenKIMONO inspects the VM's kernel state and compare them to the information got from user-level programs of that VM [9]. In addition, to defeat even more advanced attacks, XenKIMONO tries to collect the same information from different places in kernel. Any conflict between these views suggests the malicious tampering.

Besides the above tactics, XenKIMONO also tries to detect other evidence of attacks, so hopefully we can discover the nasty activities even before the kernel is compromised. XenKIMONO employs following methods: *monitor critical processes*, *detect suspicious activities* and *White-list based detection*.

(1) **Monitor critical processes**: One of the popular methods of rootkits to defeat the defense system is to shutdown the security critical daemons ([8]). XenKIMONO counters this tactic by continuously monitoring registered processes such as IDS, system logger, and other application that system relies on. To do that, XenKIMONO extracts the list of processes from the observed kernel, and then compares them to the registered list. If any process is absent, we can signal alarms.

    A simple and naive way to monitor processes is to rely on their file-paths. However, this solution can be defeated by some tricks: the attacker can replace the "good" executables with the "bad" executables of the same names. XenKIMONO defeat the problems by registering each critical process with several data such as file-path, text-size and hashed value of file text. As the hashed value is very hard to faked, XenKIMONO is not easily fooled.

(2) **Detect suspicious activities and evidences**: As we are able to parse the kernel memory, we can observe the suspicious evidences of intrusion in some kernel objects. XenKIMONO applies the following practices:

    - **Watching user privilege**: Normally the *uid* of a process remains unchanged throughout execution. Likewise other credentials such as *euid*, *suid* should be same as the *uid*. However some vulnerability in the kernel might allow the attacker to elevate his privilege, so his *uid* or *euid* becomes 0, thus effectively gives him root access ([24]). XenKIMONO tracks unregistered processes and alarms if any of these values changes unexpectedly.
    - **Watching network interface**: One of the popular tactics of the intruder is to install a network sniffer on the system to sniff pass-by network packets for valuable data like user-name and password. However, the sniffer would put the network interface into the *promiscuous* mode. XenKIMONO monitors the operation of network interface for such a suspect.

(3) **White-list based Detection**: To help the inspection more effectively, it is a good idea to make clear which actions are illegal on the monitored system. To realize that, XenKIMONO applies *white-list* technique to detect suspicious activities. Each VM registers the following *white-lists*:

    - **Process white-list**: List of applications that can have root access.
    - **Network white-list**: List of network ports that the applications can bind to.
    - **Kernel module white-list**: List of kernel modules that can be loaded into kernel.

    These *white-lists* are registered in a configuration file for each VM, and the lists are loaded into memory for detection procedure by XenKIMONO when the corresponding VM boots up. From the result of the inspection process, any activities not in the above *white-lists* can be considered malicious by XenKIMONO. We believe that this strategy is effective to defeat many evasion techniques emerging in the future,

[9]We get the user-level data via remote shell.

3.3.3 Response to Attack

Once XenKIMONO detects the intrusion, it can response with several methods, which is configurable for each protected VM:

- **Report problems**: XenKIMONO can fire alarm by writing report to logging file. The administrator can use any logging tool to analysis, and react accordingly.
- **Stop infected VM**: To mitigate damage to other systems, XenKIMONO can exploit the unique advantage introduced by Xen to stop or pause the VM, and have the administrator come to investigate the problem.
- **Checkpoint infected VM**: Xen allows to checkpoint a VM to a file, so the administrator can do forensic on the system image later with a tool like *crash-util* ([16]). XenKIMONO can ask Xen to do the job for it with *xm save* command.

3.4 XenKIMONO Implementation

At the moment XenKIMONO is only implemented in Linux. The reason is that other OSes (like FreeBSD and NetBSD) are not quite ready for Xen 3.0.2, the most advanced Xen version we are working on, yet. So in this part we will present XenKIMONO's implementation specifically for Linux domains. The same techniques can be applied for others, however.

3.4.1 Access DomU's Kernel Memory

To access to a specific virtual address of DomU, we must first translate it into physical address. Currently Xen support several kinds of architecture: *x86_32*, *x86_32p* and *x86_64*, and each of these platforms has different schemes of paging memory. Hence XenKIMONO must detect the underlying hardware, and then translates the virtual memory accordingly by traversing the page table tree.

To traverse the page table tree, it is imperative to know the physical address of the page directory. In Xen, we can have the virtual control register *cr3* of each virtual CPU of the VM by getting corresponding CPU context via Xen function *xc_vcpu_getcontext()* ([25]). Besides, as Xen supports several architectures such as *x86*, *PAE* and *x86_64* (thus different page-table formats), XenKIMONO must handle the page-table accordingly to convert the virtual address to physical address.

Afterwards, XenKIMONO accesses the memory of DomU by mapping the physical address with the function named *xc_map_foreign_range()* ([25]). Then it goes on reading or writing to the mapped memory [10]

For each DomU, XenKIMONO has a configuration file that specifies the interval time of checking (by default is 15 seconds). Accordingly, XenKIMONO is timed to periodically inspect the mapped memory of the protected VMs to find rootkits.

[10]This depends on the mapped right is *PROT_READ* (read) or *PROT_WRITE* (write).

3.4.2 *Parsing Kernel Objects*

A key challenge in inspecting the memory of VM is how to bridge the semantic gap between the raw memory and kernel objects. To do that XenKIMONO must be able to have a good knowledge about OS structure. And to understand in detail the layout of DomU's kernel and kernel objects, XenKIMONO must know exactly their address and structure.

* **Object's address**: Each object in the kernel is located at a certain memory address, and kept unchanged during its lifetime[11]. To watch the integrity of the object, it is mandatory to know its address. XenKIMONO finds the address of Linux kernel objects via the kernel symbol file *System.map* coming with the kernel binary.

* **Object structure**: To know only the object address is far from enough. For example, if we want to get the list of kernel modules, we can first reach the address of the first kernel module, the Linux variable *modules*. But then to get the next kernel module pointed by a field named *list.next* in the *module* structure, we must know the relative address of this field in module structure. This job is not trivial, as the *module* structure depends on kernel compiled option, and it might also change between kernel versions[12].

  To address this problem, we propose to exploit the kernel debugging information stored in kernel binary. If the kernel is compiled with debug option, the kernel binary stores detail information in *DWARF* format about all the kernel-types and variables ([3]).

  To extract data about kernel-types, we leverage part of code of LKCD project ([18]). LKCD is an open source tool to save and analyze the Linux kernel dump. LKCD can parse the dump thanks to an internal library *libklib*. This library parses the kernel symbols and extracts kernel-types from debugged kernel binary, then caches the data in the memory for its tool, *lcrash* to use. Besides, *libklib* also interprets *lcrash* user command, and serves as a disassembly engine for various hardware platforms. Because of these reasons, *libklib* is a very big and complicated code, thus cannot be employed as it is for XenKIMONO. Another problem is that *libklib* is designed to analyze kernel dump, but not to cope with hostile data. So if somehow the attacker modifies the kernel structure in malicious way, *libklib* might crash.

  In our implementation, we only reused part of *libklib*, in which we only keeps the code that extracts and parses kernel-type information from kernel binary. The library is also hardened to resist potential attacks. Finally, our kernel parse code is around only 14000 lines of C source code, which is about 30% size of the original *libklib*.

3.4.3 *Critical Kernel Areas*

Regarding the critical parts of kernel that XenKIMONO needs to pay attention to, we noted that the kernel rootkits usually attempt to patch the jump tables modify kernel text. Hence if XenKIMONO can detect the modifications to these parts, it can detect these rootkits.

In the current shape, XenKIMONO keeps close watch on the following parts of VM's kernel:

[11]Note that Linux kernel memory is never swapped out.

[12]Linux kernel never tries to keep compatible between different versions. The Linux kernel developers argue that backward compatibility might block its continuous innovation.

* **Kernel text**: To detect modification to kernel code, XenKIMONO monitors the memory within the range of [*_text*, *_etext*], and [*_sinittext*, *_einittext*]. The virtual address of these symbols can be found in the kernel symbol map file *System.map* accompanying the kernel.

* **Hypercall-page**: DomU requests service from VMM via hypercalls, which is similar to system-call in native OSes. All the hypercall addresses are put in a memory of *4KB* starting from kernel symbol *hypercall_page* . XenKIMONO observes for the change to this critical area to detect malicious modification to DomU's hypercalls.

* **Popular jump-tables**: XenKIMONO detects modification to following jump-tables, which can be addressed by the start address and length:

  – **System-call table**: System-call table is one of the most favorite targets of rootkits. This table starts at the kernel symbol *sys_call_table*, and spans in a range of 1240 bytes for Linux kernel *2.6.16*.

  – **Interrupt Descriptor Table**: This table saves the address of system interrupts, and can be abused by rootkits ([11]). This table starts at symbol *idt_table*, with length of 2048 bytes.

  – **Page-Fault Handler exception table**: Linux uses exception table to handle page fault in memory management. The attacker can modify the fix-up address in the table to redirect the handler to his malicious code ([1]). The exception-table is laid out in the range [*__start___ex_table*, *__stop___ex_table*].

  In all the above cases, the virtual addresses of the related kernel symbols can be found from the kernel symbol map (*System.map*) coming with the kernel binary of the corresponding VM.

  To detect the more advanced rootkits of the future, it is very important to keep tight control on the all the possible places in kernel that can be abused. We plan to support other jump-tables in the future ([4])

3.4.4 *Cross-view Detection*

As almost all the rootkits try to hide its presence of in the system, XenKIMONO can discover them with following cross-view inspections: *kernel module*, *user process* and *network socket*.

* **Kernel module**: Cross-checking kernel modules is a technique used to defeat the trick of removing modules from the linked-list to hide malicious module in kernel ([15]).

  XenKIMONO gets the list of kernel modules thanks to the kernel symbol *modules*, which points to the linked-list of modules. Knowing that kernel module has the type of *module* structure, XenKIMONO can walk the list and get all the modules loaded in the system. This list is checked against the list of modules get from user-space with the command *lsmod*.

* **User Process**: Rootkits usually run hidden processes on the system. To detect this kind of rootkit, XenKIMONO cross-checks the list of processes by comparing the list got from kernel and the list got from user-space. While the user-space view can be easily got by the command *ps*, the list of processes can be extracted from kernel via the kernel symbol *init_task*: XenKIMONO can walk the linked-list of processes

pointed to by this symbol and get all the information about processes, such as process name, pid.

- **Network socket**: Another favorite trick of rootkit is to run hidden daemon to provide remote access to outside. To cover the daemon, it needs to hide the network sockets and ports from user-space view.

  XenKIMONO detects this kind of hack by comparing the list of sockets and ports got from kernel and user-space. User-space can provide this information with the popular tool *netstat*. Whereas, the list of UDP sockets can be achieved by walking the hash table *udp_hash*, and the list of TCP sockets can be got through the *tcp_hashinfo* hash table.

3.4.5 Detect Suspicious Activities and Evidences

XenKIMONO detects suspicious problems by parsing the critical kernel objects and discover abnormal activities.

- **Watching user privilege**: To monitor the *uid*/*euid*/*suid* of system processes, XenKIMONO needs to access to the processes and their data fields in the kernel. The linked-list of processes start at the kernel symbol *init_task*, and the *uid*/*euid*/*suid* can be extracted out from the corresponding fields of the *task_struct* structure derived from the list.

- **Watching network interface**: XenKIMONO can monitor the operation status of network interfaces thanks to the kernel symbol *dev_base*, which points to the linked-list to all the interfaces available on the system. As each entry in the list is of *netdev* structure, XenKIMONO can track down to the *promiscuity* field in the structure to get the *promiscuous* mode of each interface.

4. DISCUSSIONS

To make XenKIMONO work, the VM's kernel (DomU's kernel) must be compiled with kernel debugging information. That is simply the only requirement for VM's kernel, and can be easily done by enable an option at compile time. In fact all Linux distributions provides the debugged kernels in addition to the normal kernels, so XenKIMONO users can avoid recompiling their kernels by installing the debugged kernels.

Though XenKIMONO can effectively detect all kind of rootkits, it might suffer from the *timing attack* limitation as follows: as XenKIMONO is scheduled to run periodically (the interval time is configurable for each VM), a smart attacker might quickly load then unload the rootkit in between the inspected time, so he can evade the detection. However, inconsistent operation might significantly impair rootkit's functionality. We can mitigate this problem by varying the interval checking time (in random way) to reduce the predictability, so it is hard to know when it is safe for the attacker to install the rootkit.

The cross-view detection has a potential flaw: XenKIMONO might produce fall-positive when it compares information collected from user-space and from different places in kernel at different time. To mitigate the issue, we can exploit another unique advantage of Xen: before doing the check, XenKIMONO pauses the VM to gather information from its kernel, and then resumes the machine after it finishes the job.

However, this trick does not help to fix the race problem when comparing the information from kernel and user-space: this is due to the fact that when the VM is paused, we cannot get data from user-space. To decrease this problem, we run the collection information tool in user-space 2 times: first time before pausing the VM, and second time after resuming it. If the gathered data is persistent, we can be quite confident to ignore the potential race problem.

Because all the access to the kernel needs to pause the VM, this solution causes negative impact on the performance. We address the problem by running the inspection in 2 phases:

1. **Phase 1**: Pause the VM to collect all the necessary information for *integrity checking* and *white-list based detection* from its kernel. This includes all the data about kernel modules, processes, network sockets, etc.

2. **Phase 2**: Resume the VM, and get necessary information for *cross-view detection* from user-space of that VM via remote shell.

This process repeats the in the next loop, and as a result each time of inspection requires pausing the VM only 1 time. Our experiments demonstrated that this method greatly reduces the overhead generated when running XenKIMONO, while the system becomes more scaleable even if we carry more checking procedures in the future for emerging rootkits.

5. EVALUATION

This section first presents the security evaluation results of XenKIMONO, then measures its performance impact.

5.1 Detect Kernel Rootkits

This section presents the security evaluation the efficiency of XenKIMONO: we test 5 well-known kernel rootkits to see if they are detected by XenKIMONO. These rootkits are installed on a VM, which is under the watched of XenKIMONO in Dom0.

We evaluated 5 popular rootkits as followings.

(i) **Adore**: Adore rootkit ([20]) functions as a Linux kernel module (LKM) and it can hide files and process specified by the attacker. It opens a backdoor for the attacker to connect to, so he can get in without having to authenticate. To keep controlling the system, Adore modifies the kernel by replacing 12 system-calls, such as *fork*, *read*, *write*, *stat*, *kill*, *getdent*, etc ...

(ii) **Knark**: Knark ([14]) employs the similar tactic to Adore to hide its own files put in */proc*. It replaces the *getdent* system-call and modifies the output, so file-system utilities such as *ls* cannot see the hidden files. Besides, Knark runs 3 backdoors and open 2 hidden ports for unauthorized remote access.

(iii) **vlogger**: vlogger ([15]) is a cool Linux keylogger, which is able to capture all the keystrokes and stealthily send them out to a remote machine via UDP protocol. Functioning as a hidden LKM, vlogger can cover itself quite well: it replaces the *open* system-call to hijack the TTY subsystem, and injects its code to gather keyboard data. vlogger even modifies the network stack to hide its own UDP traffic from the network sniffer tools, and to normalize the network statistics that can be exposed to the user-space.

  As the latest version of vlogger only works on Linux kernel 2.4, we had to modify its source code to have it run on 2.6 kernel.[13]

[13]vlogger 2 hooks into kernel 2.4 by patching system-calls thanks to the exported symbol *sys_call_table*, but that symbol is no longer exported in 2.6 kernel.

(iv) **Sebek**: Sebek ([21]) is a data capture tool of modern honeynet architecture. Running as a hidden LKM, Sebek tries to capture all the I/O system activities, including keystrokes, I/O at file-system level and network traffic. The latest version 3.1.2b of Sebek supports the Linux 2.6 kernel, thus available for Linux-based domains

(v) **SuckIT**: SuckIT ([17]) is one of the more advanced kernel rootkits available. It can function without LKM support by directly patching the OS kernel memory via */dev/kmem*. Instead of modifying system-call tables as others, it patches the Interrupt Descriptor Table handler, which is triggered whenever a system-call is executed. The pointer to the system-call table is changed and redirected to SuckIT's own table elsewhere.

All of the above rootkits are quickly detected by XenKIMONO when they are installed into the tested VM. Moreover, some of them are caught in more than 1 checking procedures, like below:

- Adore, Knark, vlogger and Sebek are detected when XenKIMONO checks the integrity of the system-call table.
- Adore, vlogger and Sebek are detected when XenKIMONO detects a hidden kernel module in the checking procedure of cross-view detection.
- Knark is detected because it opens 2 unregistered TCP ports (number 18667 and 31221), which violates the *White-list detection* of registered ports that processes on the VM can bind to.
- Knark is detected because when XenKIMONO detects 3 hidden processes (which are 3 backdoor daemons opening to outside for unauthorized access).
- SuckIT is detected by the integrity checking procedure on jump-tables (specifically, the IDT table).

Because other kernel rootkits more or less use the same methods we described above to infect system, we believe that XenKIMONO can effectively detect most (if not all) of the kernel rootkits.

5.2 Performance Overhead

To measure the performance penalty, we choose a classical benchmark: decompress the Linux kernel. The test chooses the Linux kernel 2.6.16.13, which is supported by Xen 3.0.2. We run this benchmark on a normal VM and compare the result with the VM protected by XenKIMONO installed. The benchmark is run 10 times, then we get the average results. The configuration of the domains in the benchmarks are as below:
Dom0: Memory: 600MB RAM, CPU: Pentium3 900MHZ, IDE HDD: 40GB, NIC: 100Mbps
DomU: Memory: 128MB RAM, file-backed swap partition: 512MB, file-backed root partition: 2GB

All the domains in the tests run Linux Ubuntu distribution (version Breezy Badger), with the latest updates.

Table 1 shows the result of the benchmarks (all the numbers are in seconds):

e user 1m25.540s

We can see that though the VM protected by XenKIMONO takes 52% time more than the normal VM to finish the work, the overall overhead is 6.92%. The reason of the slowdown is that each time XenKIMONO inspected for rootkits, it must pause the VM, then resumes it after the job is done.

All in all, the fact that XenKIMONO costs acceptable overhead makes it practical for production systems.

	Normal VM	**VM protected by XenKIMONO**
real	1m57.541s	2m5.680s
user	1m23.860s	1m30.541s
sys	0m15.390s	0m23.466s

Table 1: Measurements on unzip the Linux kernel

6. RELATED WORKS

T.Garfinkel et.al introduced the concept of Introspection IDS based on VM ([6]), which inspires our work. However, the proposed IDS was designed for VMWare Workstation, which is very different from Xen in architecture. One example is that their IDS is able to intercept system-calls of VMs, and that is feasible because on VMWare Workstation, the virtualization is done on top of the VMM's OS. Meanwhile, it is impossible to do that on Xen, as the virtualization is done at hardware level. Besides, the prototype of their concept is never released to the public, but our work XenKIMONO is planned to be published under the open source GPL license for everybody to use.

In another attempt to detect malicious activities on VM environment, M.Laureano et al. has also proposed a solution for User-Mode Linux (UML) ([12]). Unfortunately, like in the above case, we could not find any code of the paper released to the public. Besides, UML architecture is based on virtualizing Linux system-calls, and their IDS relies on this feature to intercept information for detecting intrusion. Clearly this cannot be done in Xen, because Xen is a hardware-level virtualization.

Copilot ([10]) is another solution close to our approach. Being a hardware-based solution, Copilot is in the form of a PCI card that is installed on the host being monitored for rootkit activity. The goal of the PCI card is to remain as independent of the potentially subverted operating system as possible. To do this, the PCI card has its own CPU and uses Direct Memory Access (DMA) to periodically scan the physical memory of the computer looking for rootkit behavior.

Because it is a hardware based solution, Copilot provides a high degree of assurance; however, this is not without a price: The firmware on Copilot card needs to have intimate knowledge of the kernel layout, thus requires updating whenever the kernel is recompiled. Last but not the least, any hardware based solution is going to be more costly to purchase and to maintain, hard to widely deploy. Moreover, a machine needs to be dedicated to be administrator host. All these problems make Copilot less attractive. In contrast, XenKIMONO is a totally software approach and requires no hardware to function.

Zhang et al. proposed using a secure coprocessor as an intrusion detection system for kernel memory ([26]). Specifically, the authors describe a method for kernel protection that consists of identifying invariants within kernel data structures and then monitoring for deviations. This strategy of interpreting and comparing kernel data structures is very similar to that of Copilot.

Kstat ([5]) is a kernel analysis tool that checks */dev/kmem* for information about LKMs and the status of the system-call table. It can slo be used to extract other information directly from the memory image, like the running processes or the network interface state. However, Kstat can be undermined if the attacker has access to kernel.

StMichael ([22]) is another tool to perform integrity checks on portions of Linux kernel, including *sys_call_table*. Same as Kstat, StMicheal can be defeated if the attacker gains privileged access to to the kernel memory and disables it.

LKCD ([18]) is an open source tool to save and analyze the

Linux kernel dump. To do that LKCD must have a good understanding about kernel architecture and its object structure. XenKIMONO reuses part of its library *libklib* to parse the VM kernel image.

7. CONCLUSIONS AND FUTURE WORKS

This paper proposes the design and implementation of XenKIMONO solution to detect the security violation to the Operating System kernel at run-time. XenKIMONO exploits the unique advantage of Virtual Machine to stay separately with the monitored systems. As a result, XenKIMONO is in completely different protection domain, and our detection is reliable even in the case the attacker takes over the kernel. XenKIMONO employs and combines different strategies to detect different kinds of kernel rootkits in a reliable way.

The security evaluation proves that XenKIMONO is able to detect all of 5 popular kernel rootkits in almost instant way (the window time is configurable). Because all other rootkits have the same methods of infection, we strongly believe that they all can be detected by XenKIMONO once they attempt to install themselves into the kernel. Though the solution is not perfect as we discussed in the section 4, we can mitigate the problems with different tactics.

The performance evaluation also demonstrates the small penalty of XenKIMONO on the monitored VMs. We believe that this is an advantage, and XenKIMONO can be launched into production systems without causing any negative impact.

When we implemented XenKIMONO, we studied and evaluated the most popular rootkits available to make sure that XenKIMONO can detect all of them. In the future, we will closely follow the emerging rootkit technique to extend XenKIMONO capability, so it can adapt to the new trick of rootkits.

8. REFERENCES

[1] buffer. Hijacking Linux Page Fault Handler Exception Table. `http://www.phrack.org/show.php?p=61&a=7`, August 2003.

[2] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[3] DWARF Workgroup. DWARF Debugging Format Standard. `http://dwarf.freestandards.org/Home.php`, January 2006.

[4] T. Fraser. Automatic discovery of integrity constraints in binary kernel modules. Technical report, University of Maryland Institute for Advanced Computer Studies, December 2004.

[5] FuSyS. KSTAT: Kernel Security Therapy Anti-Trolls. `http://www.s0ftpj.org/tools/kstat.tgz`, February 2002.

[6] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed Systems Security Symposium*, February 2003.

[7] T. Holz. Detecting honeypots and other suspicious environments. In *Proceedings of the 6th IEEE Information Assurance Workshop*, June 2005.

[8] S. Inc. W32/Sdbot-ADD Worm. `http://www.sophos.com/virusinfo/analyses/w32sdbotadd.html`, September 2005.

[9] Joris Evers. Rootkits get better at hiding. `http://news.com.com/2100-7355_3-6095762.html?part=rss&tag=6095762&subj=news`, July 2006.

[10] N. L. P. Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, August 2004.

[11] kad. Handling Interrupt Descriptor Table for fun and profit. `http://www.phrack.org/phrack/59/p59-0x04.txt`, December 2002.

[12] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *Procceedings of the 30th EUROMICRO Conference*, September 2004.

[13] McAfee Avert Labs. Rootkits, Part 1 of 3: The Growing Threat. `http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_newappleofmalwareseye_en.pdf`, April 2006.

[14] T. Miller. Analysis of the Knark rootkit. `www.ossec.net/rootkits/studies/knark.txt`, 2001.

[15] rd. Writing Linux kernel keylogger. `http://www.phrack.org/phrack/59/p59-0x0e.txt`, July 2002.

[16] Redhat Inc. Crash-util. `http://people.redhat.com/anderson/`, July 2006.

[17] sd. Linux on-the-fly kernel patching. `http://www.phrack.org/show.php?p=58&a=7`, July 2002.

[18] SGI Inc. LKCD - Linux Kernel Crash Dump. `http://lkcd.sf.net`, April 2006.

[19] Silvio Cesare. SysCall redirection without modifying the SysCall table. `http://vx.netlux.org/lib/vsc05.html`, 1999.

[20] stealth. adore-ng rootkit. `http://stealth.7530.org/rootkits/`, March 2004.

[21] The Honeynet Project. Know your enemy: Sebek. `http://www.honeynet.org/papers/sebek.pdf`, November 2003.

[22] Tim Lawless. StMichael: Kernel-level IDS. `http://sourceforge.net/projects/stjude`, December 2005.

[23] ubra. Process hiding and the Linux scheduler. `http://www.phrack.org/show.php?p=63&a=12`, August 2005.

[24] Wojciech Purczynski. Linux kernel ptrace/kmod local root exploit. `http://www.securiteam.com/exploits/5CP0Q0U9FY.html`, March 2003.

[25] Xen project. Xen interface manual. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html`, August 2006.

[26] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th ACM SIGOPS European workshop*, September 2002.