Barrier: カーネルモジュールの分離を通してカーネルの完全性保護に対する軽量ハイパーバイザ

華　景煜 †　　　　櫻井 幸一 †

† 九州大学大学院　システム情報科学府
〒 819-0395 福岡市西区元岡 744
{huajingyu, sakurai}@itslab.csce.kyushu-u.ac.jp

あらまし カーネルモジュールは、同じアドレス空間で実行しており、カーネルモジュールに対するアクセス制御や隔離がなく，カーネルルートキット (Kernel Rootkit) の根本的な発生原因となっている。この論文で専用ハイパーバイザー (Barrier) 設計し、カーネルモジュールを分離する。信頼性と重要性に基づき異なるアドレス空間内にカーネルモジュールを分割し、それぞれに割り当てられたメモリページを追跡する。実行時には、Barrier は現在のアドレス空間のためにホストレベルのページテーブル (HPT) 内に最小のメモリマッピングのみを提供する。結果として、カーネルモジュールは、秘密裏に自分のアドレス空間外のメモリにアクセスすることはできない。

Barrier: A Lightweight Hypervisor For Protecting Kernel Integrity via Kernel-Module Isolation

Jingyu Hua *†　　　　Kouichi Sakurai†

†Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
{huajingyu, sakurai}@itslab.csce.kyushu-u.ac.jp

Abstract The root-cause of kernel-rootkits is that current OSes lack of memory access control within the kernel space: once a kernel module is loaded into the kernel, it is granted the highest privilege and can access the whole kernel memory without any limitation. Targeting this problem, we present Barrier, a lightweight hypervisor designed for enhancing the kernel integrity by isolating the kernel modules. Specifically, it leverages the hardware-supported memory virtualization to isolate the memory pages of kernel modules with different importance and trustiness. All the cross-address-space interactions have to go through a strict mediation according to some access control policies, which will greatly increase the attacker's hardness to compromise the kernel integrity.

1 Introduction

Operating system (OS) is the core of a computer system. However, these years the OS security is seriously threatened by a class of malware that directly run in the kernel space. They are known as Kernel Rootkits. The hackers make use of them to hide their malicious processes and files, steal private data and even play the role of back doors to wait and execute their commands from the remote.

* Supported by the governmental scholarship from China Scholarship Council for his Ph.D. course.

Because the kernel rootkits directly comprise the kernel, it's hard to defend them with traditional antiviruses or IDSes that rely on the information provided by the kernel. In fact, no security tools within or above the OS kernel can well counter the kernel rootkits. The reason is simple: if such a tool can disable a rootkit, the rootkit can also disable this tool because they have the same privilege. So a better way is to move the countering measures into a more privileged layer that is independent of the OS kernel. By doing this, they are protected from being affected by the kernel rootkits. A good place to realize this idea is in the Hypervisor (i.e., Virtual Machine Monitor), which is an additional layer between the OS kernel and the hardware. It can help us intercept all the hardware accesses including memory operations from the OS. We may place some prevention logic here to safely capture those illegal memory writings inside the kernel that indicate the existence of kernel rootkits. Since the hypervisor is of higher privilege compared with the OS kernel, the kernel malware have no rights to disable the prevention logic on this layer. This method has become one of the major approaches to protect the OS security. In this research, we present Barrier, a lightweight hypervisor designed for enhancing the kernel integrity by isolating the kernel modules.

2 Related Works

The current hypervisor-based kernel protection approaches can be mainly divided into three categories.

The first category works by preventing critical kernel data in the memory from being tampered with [6, 7, 14]. They mark those memory pages containing critical data read-only in the page tables that are maintained by the hypervisor. By doing this, any attempt to modify these read-only pages will raise page faults and be further verified by the hypervisor. These systems suffer two problems: firstly, it is hard to collect all the critical kernel data scattered across the kernel pages as well as their access profiles. Secondly, many critical data are co-located together with frequently modified data. If marking them as read-only, many unnecessary page faults will greatly slow down the execution of the guest system.

The second category works by preventing executing unapproved kernel codes in the kernel space[3, 4, 5]. For example, in the SecVisor system [3], the hypervisor assigns the rights of read and execute to the memory pages occupied by the approved kernel codes while marks all the other pages non-executable. As a result, malicious codes out of the original kernel are prevented from being executed. This approach also has several critical disadvantages: firstly, it's hard to deal with those mixed pages containing both code and data. Secondly, some kernel malware such as the return-oriented rootkits [1] do not load any new codes but just utilize existing kernel codes to launch attacks. This approach is helpless to such malware.

The last category including HUKO [9] and Gateway [10] works by isolating untrusted kernel extensions from the original kernel. However, they only isolate dynamically loaded extensions but not other modules in the kernel. In addition, neither HUKO nor Gateway isolate untrusted extensions themselves from each other. As a result, they can freely attack each other.

3 Barrier Design

3.1 Overview

The root cause of kernel malware is the lack of memory access control (MAC) within the kernel space: each kernel module including the user drivers is granted the highest privilege and can access the whole memory space without any limitation. So, Barrier aims to enhance the integrity of the kernel by isolating the kernel modules and preventing them arbitrarily access each other's address space.

Specifically, Barrier implements the MAC policies defined in Table 1 for different kernel modules. We classify the kernel modules into three types based on their trustiness and importance: Trusted & Critical (T & C) modules, Non-Trusted & Critical (NT & C) modules and Non-Critical (NC) modules. We say a module is trusted when it is verified benign and contains few vulnerabilities that can be exploited by the attackers. So trusted modules are allowed to read and write the whole guest memory. However, they are prevented from arbitrarily executing code of non-trusted modules because this will damage their trustiness. We say a module is important (critical) when its code and data are critical, and must be protected from being accessed from non-trusted modules. On default, in Barrier, core-modules are considered trusted and critical (T & C), while all the other modules are considered critical but non-trusted (NT & C). Since the NC modules are not critical, we exclude them from our protection. Fig.1 shows the basic architecture of Barrier. We will describe how it enforces the MAC policies defined in Table 1.

表 1: MAC Policy used by Barrier within kernel space

Targets	T&C Modules			NT&C Modules			NC Modules		
	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
T&C Modules	allow	allow	allow	deny	deny	audit	deny	deny	audit
NT&C Modules	allow	allow	audit	deny	deny	audit	deny	deny	audit
NC Modules	allow	allow	audit	allow	deny	audit	allow	allow	allow
User Space Content	allow	allow	audit	allow	deny	audit	allow	deny	audit

3.2 Barrier Hypervisor

Barrier Hypervisor is the core of our system. Most components of Barrier are located on this layer. This prevents them from being affected by the malicious codes in the OS. We can directly use commodity hypervisors such as Xen and VirtualBox. However, these general-purpose hypervisors are designed for servers that have the requirement to run multiple operating system instances concurrently on a single computer. They spend great efforts on the isolation of different OS instances and the coordination of concurrent hardware accesses. If we directly construct Barrier on these commodity hypervisors, we have to suffer unnecessary performance overheads because Barrier is mainly designed for enhancing the security of personal computers, which rarely run multiple OSes but just an exclusive OS. Thereby, we decide to construct a specialized lightweight hypervisor for Barrier. Nevertheless, Barrier can be easily portable to the commodity hypervisors.

We make use of hardware virtualization support, which is offered by both recent Intel processors (Intel-VT [13]) and AMD processors (AMD-SVM [15]), to implement the full virtualization. When this hardware feature is turned on, the guest OS becomes running in an additional VM mode. In this mode, the page tables (PTs) in the OS can be only used to translate linear addresses (LAs) to guest physical addresses (GPAs), which are not machine addresses (MAs) used by CPU. MMU has to utilize another layer of PTs that are created and maintained by the hypervisor to translate the intermediate GPAs to the final MAs. These hypervisor PTs (HPTs) are invisible to the guest OS, and only the hypervisor has the rights to update them. Since each page table entry contains several bits that can control the accesses to the corresponding memory page, HPTs become a good place for Barrier to perform module-level MAC provided that it can accurately locate memory pages occupied by each kernel module.

We reserve a region of x MB in the high memory for the Barrier hypervisor. The remained memory is used by the Guest system. From the perspective of the gust, the size of the physic memory is (m - x) MB but not the original value m. The HPTs created by the hypervisor for the guest system are located within the hypervisor region. We fill them with identity mappings, which means the final MAs are identical to their GPAs after the translation. Because the HPTs do not contain the mappings for the VMM region, the guest system is prevented from accessing this region. Any attempts to violate this rule will raise page faults and then be trapped into the hypervisor. So the hypervisor is completely isolated from the guest system.

3.3 Address Space Isolation

In this section, we describe the isolation component in Barrier. It is responsible for isolating the address spaces of different modules so as to prevent one malicious module affecting modules in other address spaces.

3.3.1 T&C Address Space

All the T&C modules are isolated into one address space called T&C Address Space. Barrier creates a separate set of HPTs (A-HPTs) for this address space. The A-HPTs are filled in identical mappings for the whole machine memory owned by the guest system. However, different type of pages are granted different access permissions in their corresponding page table entries: code pages occupied by the T&C modules are marked executable and readable, while all the other pages are marked writable but no-executable. This can ensure the following issues when the host CR3 is made to point to A-HPTs: (1) only the codes belonging to the T&C modules have the right to execute. This protects the T&C modules from being modified by malicious codes in other modules because they are prevented from executing. (2) The T&C modules can freely read and write the memory of other modules. We can find that they are consistent with the MAC policies defined for T&C modules in Table 1.

3.3.2 NC Address Space

Similar with the T&C modules, all the NC modules are also isolated into a unified address space called NC Address Space. They use another set of HPTs named C-HPTs to access the machine memory. However, different from A-HPTs, C-HPTs are only filled in identical mappings for the memory pages occupied by the NC Modules as well as their guest PTs. All the other entries are left blank. As a result, the NC modules are limited to access NC modules. And attempts to access other critical modules will raise page faults and be trapped into the hypervisor for mediation. This is also consistent with the MAC policies defined for NC modules in Table 1.

3.3.3 NT&C Address Spaces

NT&C modules are different from other modules: they cannot be isolated into a unified address space because according to Table 1, they have to be isolated from each other, i.e., one NT&C module cannot freely access the code or data of another NT&C module. Therefore, each NT&C module should be put in a separated address space. For efficiency, all these address spaces are still made to share a common set of HPTs, called B-HPTs. At first, B-HPTs are left empty. At runtime, when the system control comes to a specific NT&C, Barrier dynamically creates the corresponding mappings for memory pages occupied by this module: its code pages are marked executable and readable, while its data pages are marked readable and writable. Therefore, while a NT&C module gains the CPU, it can only access its own code or data. So it is completely isolated from any other modules, which is consistent with the MAC policies defined in Table 1.

3.3.4 Address Space Profiling

From the above, we can find that in order to create dedicated HPTs for an address space, Barrier has first to accurately locate the memory pages occupied by its modules in the kernel memory. This task is not hard for the static modules because their locations are fixed at the compiling time. However, for the dynamic extensions, since their locations are never fixed until they are loaded into the kernel at runtime, the situation becomes much more complex. Barrier has to intercept the module loading events so as to associate the memory pages with the dynamic modules in time. In addition, because both static and dynamic modules may dynamically allocate or deallocate memory, Barrier is also required to trace these events so as to extend or shrink the corresponding address spaces. Because the page limitation, we will not present the detailed approach here. Based on the location data, Barrier is made to maintain a profile for each address space, which is composed of (1) the type of the address space, (2) the machine address range of each code and data block, and (3) the page number range of each code and data block. These profiles are used by the isolation component to create and update the HPTs.

3.4 Address Space Switching

As the execution of the guest system, the control is keeping switching among different address spaces. In general, Barrier has two kinds of address space transitions: one involves the switching of HPTs and another involves the dynamic updating of B-HPTs. Now, let's discuss their details.

The first type of address space switchings happens when a kernel module jumps to execute the codes in another module with different trustiness or importance. For example, assume the guest system is currently running in a T&C module M1, and at some position, it calls a function in another module M2, which is a NT&C or NC module. In this case, since the A-NPTs, which are currently being used by MMU, do not grant execute permission to the code pages of M2, a host-level page fault is raised, and the guest system exits and traps into the hypervisor. If this transition passes the mediation in the hypervisor world, Barrier reloads the hCR3 with the base address of the target HPTs, which immediately switches the address space. Then, Barrier can resume the execution of the guest system in M2.

The second type of address space switchings is due to the control transitions within NT&C modules. Assume the guest system is running in M1, which is a NT&C module, and at some position, it calls a function in another NT&C module, M2. Then, the control flow has to transfer from M1 to

M2, which will trigger the address space switching: (1) Since the current B-HPTs do not contain the mappings for the memory pages of M2, a host-level page fault is raised by the CPU and then the system traps in to the Barrier hypervisor. (2) Barrier checks whether this interaction is legal. If legal, Barrier quickly decides the target module based on the address of the target function. And then, it modifies the B-HPTs to make them map the address space of M2. This mainly involves two things: firstly, the corresponding entries for the memory pages of M2 should be filled in; secondly, the original mappings for M1 should be erased so as to isolate these two modules. (3) Go back to the guest system and begin to execute the function in M2.

3.5 Mediation

With the above isolation component, the execution of instructions corresponding to the "Audit" and "Deny" actions in Table 1 will cause page faults and then be trapped into the hypervisor. The goal of mediation is to validate these events and then take appropriate actions.

When a hypervisor-level page fault is captured, Barrier first decides which event in Table 1 occurs by examining the following information: (1) the qualification bits which reveal the actual type of the violation, (2) the current module, and (3) the target module. If it is a "Deny" event, Barrier directly denies it and triggers a protection alarm. However, if it is "Audit", a more careful examination is needed before making any further decision.

The basic rule to approve those "Audit" events is that the fault addresses must point to some objects that are exported by their owner modules. For example, if the instruction is a function call, then the target function must be exported and exist in the symbol table of the kernel. This prevents one module from accessing unauthorized functions or jumping to arbitrary positions in another module belonging to a different address space.

4 Evaluation

4.1 Protection Effectiveness

In this section, we evaluate the effectiveness of Barrier for the kernel integrity protection against the threats described in Sec. 2. We do this with four representative real-world kernel rootkits [1] and two self-designed malicious LKMs. When these malware are loaded into the kernel, they are regarded as the only non-trusted modules and are isolated into their own NT&C address spaces. The results are presented in Table 1.

We evaluated the performance overheads of Barrier by running a set of programs to compare their performance with and without Barrier. All the programs and their workloads are presented in Table 2. The first three are I/O-bound (Disk and Network), while the last three are CPU-bound. They can test the performance of the guest system in different aspects. For the *cat* and *gzip*, we marked the concrete file system *ext*4 and the disk driver as the NT&C modules and isolated them into their own address spaces. The *ext*4 is a static module. We located and isolated it based on the *built-in.o* file in *linux-src-dir/fs/ext*4. The disk driver on our system is composed by two dynamic modules, *ahci* and *libahci*, which are both located in *linux-src-dir/drivers/ata*. Because the workloads of *cat* and *gzip* are disk I/O-bound, these modules are accessed most frequently and cause the highest rates of address-space transitions. Hence, we can get the worst-case performance of Barrier by isolating them. For the *ncft*, since its workload is network I/O-bound as well, we also isolated the network interface card driver, which is the dynamic module *r*8169 on our system. For the three CPU-bound tests, since they only involve the core-kernel, it is trivial to isolate any other kernel modules.

We did experiments on an HP notebook with an AMD 1.66 processor, 2GB of memory and a 100Mbps ethernet card. The experiment results are shown in Table 3. We ran each program for five times and all the results are reported as their average values. We can find that the performance overheads introduced by Barrier in these programs vary from 17% to 0% compared with their native speeds. The performance overheads are proportional to the frequency of address space transitions. The three CPU-bound tests show a good result because they involve no address space transitions. This also demonstrates the high efficiency of the memory virtualization based on AMD-SVM. The overheads in the other three cases are much higher because all of their workloads are (disk or network) I/O-intensive, which brings the highest frequency of address space transitions. Given that they are the worst cases, they are acceptable.

[1]Note that the original versions of these rootkits cannot be installed on the high version of Linux kernel including 2.6.31. So we modified them according to our kernel, but their basic ideas remain the same.

表 2: Protection effectiveness of Barrier against a set of kernel malware

Kernel Malware	**Damaged Integrity Property**	**Barrier**	**HUKO**	**Gateway**
EnyeLKM	Code Integrity of T&C AS	yes	yes	yes
all-root	Non-Control Data Integrity of T&C AS	yes	yes	yes
adore-ng	Control Data Integrity of T&C AS	yes	yes	yes
lvtes	Code Integrity (call unauthorized function)	yes	yes	yes
MalLKM1	Data Privacy	yes	no	no
MalLKM2	Data Integrity of NT&C AS	yes	no	no

表 3: Performance results of benchmark programs

Benchmark	**# of Transitions**	**Native Performance**	**Barrier Performance**	**%**
Cat	231,327	14.7s	16.53s	88%
gzip	101,308	14.2s	15.87s	89%
ncftp	2,521,674	36.86s	44.27s	83%
Dhrystone 2	N/A	6,829,814lps	6,676,941lps	97%
Whetstone	N/A	1351$mwips$	1350$mwips$	100%
Process Creation	N/A	322ms	327ms	98%

5 Conclusion

We have presented the design and implementation of Barrier: a lightweight hypervisor designed to enhance the kernel integrity. It leverages hardware virtualization technology to isolate the kernel modules into different address spaces so as to capture the malicious interactions among them caused by the kernel malware. Since Barrier takes all the kernel modules into consideration, it better protects the kernel integrity compared with those proposals just isolate the dynamic extensions. Our evaluations on Linux show that Barrier brings acceptable performance overheads to the running of the protected system.

参考文献

[1] T. H. R. Hund and F. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. USENIX Security 2009.

[2] M. I. Sharif, W. Lee, and et al. Secure in-vm monitoring using hardware virtualization. CCS 2009.

[3] A. Seshadri, M. Luk, and et al. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. SOSP 2007.

[4] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries.USENIX Security 2008.

[5] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. RAID 2008.

[6] A. Srivastava, I. Erete, and J. Giffin. Kernel data integrity protection via memory access control. Technical Report, Georgia Institute of Tech, 2009.

[7] Z. Wang, X. Jiang, and et al. Countering kernel rootkits with lightweight hook protection. CCS 2009.

[8] E. Witchel, J. Rhee, and K. Asanovi. Mondrix: memory isolation for linux using mondriaan memory protection. SOSP 2005.

[9] X. Xiong, D.Tian, and P.Liu. Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions. INDSS 2011.

[10] A. Srivastava and J. Giffi. Efficient Monitoring of Untrusted Kernel-Mode Execution. NDSS 2011.

[11] A. Nguyen, N. Schear, and et al. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. ACSAC 2009.

[12] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. ACSAC 2008.

[13] Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide.

[14] J. Rhee, R. Riley, D. Xu. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring. ARES 2009.

[15] AMD64 Technology AMD64 Architecture Programmer's Manual Volume 2: System Programming.