ATRA: Address Translation Redirection Attack against Hardware-based External Monitors

Daehee Jang[1], Hojoon Lee[1], Minsu Kim[1], Daehyeok Kim[2], Daegyeong Kim[1], Brent Byunghoon Kang[1]

[1]Graduate School of Information Security
[2]Cyber Security Research Center
KAIST, Daejeon, Korea
{daehee87, hjlee228, pshskms, dhkim7, daegyeong.kim, brentkang}@kaist.ac.kr

ABSTRACT

Hardware-based external monitors have been proposed as a trustworthy method for protecting the kernel integrity. We introduce the design and implementation of Address Translation Redirection Attack (ATRA) that enables complete evasion of the hardware-based external monitor that anchors its trust on a separate processor. ATRA circumvents the external monitor by redirecting the memory access to critical kernel objects into a non-monitored region. Despite the seriousness of the ATRA issue, the address translation integrity has been assumed in many hardware-based external monitors and the possibility of its exploitation has been suggested yet many considered hypothetical. We explore the intricate details of ATRA, explain major challenges in realizing ATRA in practice, and address them with two types of ATRA called *Memory-bound ATRA* and *Register-bound ATRA*. Our evaluations with benchmarks show that ATRA does not introduce a noticeable performance degradation to the host system, proving practical applicability of the attack to alert the researchers to seriously address ATRA in designing future external monitors.

1. INTRODUCTION

Kernel rootkit is a severe security threat to a system, since they are capable of subverting the operating system itself, compromising kernel objects, hijacking the kernel control flow by overwriting function pointers, and remaining undetectable from in-host security measures. Prior research efforts to mitigate this can be classified into two categories based on their root-of-trust: hypervisor-based approaches [1–6] and hardware-based approaches [7–11]. However, hypervisor-based approaches are known to have a limitation; since hypervisors are also a software layer, they can be exposed to software vulnerabilities. Recently, a number of vulnerability in commodity hypervisors have been reported [12–16]. Also, the monitors impose a burden on top of the overhead of the virtualization itself, as they leverage the virtualization functionalities to acquire monitoring capabilities.

Alternatively, a few hardware-based external monitors such as [7–11] have recently been introduced. In these approaches, the kernel integrity monitor runs on an independent processor, which is external to the processor that the monitored host runs on. This architectural isolation guarantees that any potential compromise on the monitored host would not affect the trustworthiness of the execution environment for the monitor. Since the monitor does not share its processor with the host, the monitor code can run as often as needed, not taking up CPU cycles from the host.

Note that the Intel SMM (Systems Management Mode) or ARM TrustZone can also provide a trustworthy execution environment for the monitor code. However, such hardware-assisted approach shares the same processor with the monitored host, which would entail a context-switching overhead for mode transition, and the monitor code runs by taking up CPU cycles on the host processor. The normal operations on the host must wait until the monitor code yields the shared processor.

Thus, as shown in recent burgeoning works [8–11], the hardware-based external monitors are considered as efficient kernel integrity monitors. However, the current hardware-based external monitor that runs on an independent processor is architected not to access the CPU states of the host's processor, which allows the adversary to completely evade the monitoring. Their monitoring schemes are based on the assumption that the virtual to physical memory mappings of their targets of monitoring are intact throughout the host system operation. This assumption exposes monitors to the critical security threat that allows kernel rootkits to compromise paging data structures and relocate important kernel objects without incurring visible effects on the virtual memory address space of the host.

Although hardware-based external monitors can try to defend this attack by additionally observing such data structures in the physical memory, the attacker can still bypass this additional effort because hardware-based external monitors cannot observe the CPU states of the host such as control registers. Despite the gravity of this problem, a number of studies on external processor-based approaches which do face this threat either underrated this limitation as hypothetical [8, 11] or seemed not aware of it [9, 10].

In this paper, we present *Address Translation Redirection Attack (ATRA)* which can entirely subvert the effectiveness

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
ACM 978-1-4503-2957-6/14/11.
http://dx.doi.org/10.1145/2660267.2660303 .

of existing hardware-based external monitors. Specifically, by presenting the design and implementation of ATRA, we show that it is a practically feasible attack. In order to enable ATRA in practice, we found a few challenges including the following:

- The attack should successfully circumvent hardware-based external monitors while it manipulates the address translation in the target system. By monitoring additional known memory regions that contain the kernel data structures needed for the address translation, external monitors can be enhanced to detect ATRA that touches monitored memory regions.
- The attack should not introduce a noticeable performance degradation. Since ATRA modifies some of the core system functionalities such as the interrupt handling, ATRA could induce a system-wide performance degradation. Considering that one of the main purpose of rootkit is to hide the existence from the system administrator, a noticeable performance degradation is a critical issue for the attacker.

We propose two types of ATRA: (i) Memory-bound ATRA, (ii) Register-bound ATRA to address these challenges. Memory-bound ATRA relocates important kernel objects and makes the entire system refer to the copy by attacking the page table data structures of the OS kernel. In addition, Register-bound ATRA manipulates the page table related CPU states of the system. In Register-bound ATRA, we devised a technique called *Inter-Context Register Modification* where a victim process is caused to update its own CPU state with an attacker's modified value either through reloading the previously saved context or through executing the attack code that modifies a register. As a consistent entry point for persisting the ATRA effect, we also engage a few CPU registers that do not change across context-switching between processes. We show that Register-bound ATRA can completely circumvent the existing hardware-based external monitors due to their limitations. We also reported that they cannot reliably detect Memory-bound ATRA due to the race condition for protecting dynamically allocated kernel objects including page tables and to the complexity of observing all possible attack vectors that are available throughout the multiple steps involving address translation for every pointer traversal and memory access.

Our contributions are summarized as below.

Despite the seriousness of the ATRA issue, previous descriptions of the related concepts were a few sentences [2, 8–11, 17–19]. To the best of our knowledge, this paper is the first work to explore the intricate details of ATRA and *thoroughly* describes the ATRA attack, which covertly controls the virtual address translation mechanism by manipulating CPU registers as well as page table related data structures, and it also demonstrates that ATRA can make all the existing hardware-based external monitors ineffective.

We implement two ATRA-enabled kernel rootkit attacks — system call table hooking attack and the LKM hiding attack — without being detected by hardware-based external monitors. Our performance evaluation with STREAM [20] and UnixBench [21] show that ATRA does not induce a noticeable performance degradation to the host system.

By showing the effective and complete evading method against existing hardware-based kernel integrity monitors, we emphasize the importance of exploring more solid kernel integrity monitoring schemes that are resilient to the ATRA-based attacks, and hope that the researchers will pay more serious attention to the ATRA threat, which have been often ignored or considered as hypothetical.

2. BACKGROUND AND ATTACK MODEL

Before we explain the ATRA attack in detail, it is necessary that the operating mechanism and the underlying assumption of the existing hardware-based external monitors are explained along with the attack model.

2.1 Attack Model

The objective of the adversary described in this paper is to subvert the operating system under the presence of a hardware-based external monitor. The attacker is assumed to have obtained a root privilege of a victim system and tries to deploy a rootkit which manipulates paging data structures to hijack the control, or hide itself from the entire system and the hardware-based external monitor. The adversary is assumed to have the capability to modify any kernel object in memory. Note that the attacker is capable of modifying CPU register values whereas existing hardware based external kernel integrity monitors are incapable of observing any changes in such registers.

2.2 System Assumptions

We assume that a target system is protected by a hardware-based external kernel integrity monitor that has the capability of introspecting the target system's physical memory regions where kernel data structures are located. It is important to note that the hardware-based external monitor detects the existence of any write traffic or modification destined for the monitored memory region so that it can protect any attempt to modify the important kernel data structures stated above. If the regions to be monitored are static (such as kernel codes and the system call table), the address locations to be monitored can be set in advance. However, for the dynamic kernel data structures, the address locations to be monitored cannot be set in advance, it must be determined after the dynamic memory object is allocated.

ATRA is applicable to any multi-paging system, however, for the brevity of ATRA discussion, we assume that the target system is based on the x86-32bit architecture with two-level paging for its virtual memory management mechanism and adopts Linux operating system. In the x86 architecture, the MMU performs address translation by referencing the processor's CR3 register and traverses in-memory data structures such as the page tables. The CR3 is a base register which contains the physical address of root page table so called Page Global Directory (PGD). The MMU uses this CR3 register to locate the physical address of the PGD and traverses the page tables depending on the level of paging. PGD holds 1024 entries that store the physical address of page tables. These page tables, pointed by PGD entries, also have 1024 entries of physical address of a page. Each page table is referred to as PTE (Page Table Entries).[1] MMU obtains the physical address of the top-most entry in the

[1]In Linux kernel source code convention and its documentations, PTE also signifies a single entry in the page table. For clarity of discussion in this paper, we use 'PTE' to refer the page table, and 'PTE entry' for the entry.

PGD by referencing the CR3 register and begins page table walking for the requested translation.

2.3 Operations of Existing Hardware-based External Monitors

To the extent of our knowledge, the current hardware-based external monitoring technology is limited to host memory monitoring; they are not capable of monitoring CPU states such as registers and flags. Also, the *semantic gap* between the host machine and external monitor introduces a significant challenge especially in locating the objects that need to be monitored in the host's virtual memory. These monitors focus on detecting kernel rootkits in a possibly compromised kernel. However, they assume the integrity of the kernel address translation [7–11] and this assumption persists even in the recent works.

In case of Copilot [7], it relies on the addresses of kernel symbols generated at compile time listed in the *System.map* file. Also, it calculates the physical addresses of the monitored regions by simply using the constant offset between the virtual addresses of the linear-mapped kernel region and their corresponding physical addresses. That is, the physical address of the kernel symbol in a linear-mapped region can be calculated by subtracting the linear mapping offset (i.e., *0xC0000000*). Vigilare [8] inherits this technique to determine the physical address of the monitored area that supposedly contains the important static kernel code and data. KI-Mon [9] and MGuard [10] also monitor kernel static region in a similar fashion.

Besides the page table integrity issues, the CPU registers play a critical role in the virtual address translation of the host. In fact, the registers serve as the root of page table walking. CR3, for example, stores the physical address location of the page directory for the process context that is currently being executed by the CPU [22].

3. ADDRESS TRANSLATION REDIRECTION ATTACK

We categorize ATRA into two types: *Memory-bound ATRA* and *Register-bound ATRA*. Memory-bound ATRA modifies the data structures related to the page table, while Register-bound ATRA achieves the translation redirection by directly or indirectly compromising the values related to the CPU registers such as CR3.

3.1 Memory-bound ATRA

Memory-bound ATRA targets the PGD and PTE in order to change the address translation mapping. While the PGD and PTE are essentially identical data structures, there is an important difference: PGD exists for each process while the kernel PTE is shared globally across processes. Due to this difference, the detailed application of ATRA against PGD (PGD-ATRA) requires more sophisticated method than PTE (PTE-ATRA). We first explain PTE-ATRA, then PGD-ATRA.

Method for PTE-ATRA: PTE-ATRA can be accomplished by simply modifying an entry in PTE where each PTE entry maps a page frame in physical memory (a 4KB page in conventional x86 systems). Assume that the kernel data structure (i.e., system call table) of the host kernel resides in a page that corresponds to a virtual address range from *0xC0001000* through *0xC0002000*. An adversary has a new page allocated using a function from the kernel memory allocation API such as *kmalloc*, then copies the content of the page with original kernel data structure to the new one (with kernel privilege, the attacker can also copy the page properties as well). The adversary can modify this relocated copy instead of the original one. Lastly, the attack is completed by overwriting the PTE entry which corresponds to *0xC0001000* with the physical address of the kernel data structure copy created on the newly allocated page frame.

Since PTEs that correspond to the kernel virtual memory space are shared among all processes, this modification affects the entire system. Figure 1 shows the three steps in PTE-ATRA.

Method for PGD-ATRA: PTE-ATRA can be easily applied, however, the detection also can be easily applied by monitoring the PTE. To avoid the detection against PTE manipulation, PGD-ATRA can be performed in a similar manner. However, unlike PTE-ATRA, PGD-ATRA needs to be launched for each process to make a system-wide effect. This is because each process has its own copy of PGD which maps the globally shared kernel PTEs [23]. In Linux, a simple way of accessing the entire PGDs in the system is traversing the linked list of *task_struct*s. Steps in PGD-ATRA are depicted in Figure 1.

After PGD-ATRA comes into effect, PTE-ATRA can be launched without being detected. However, PGD-ATRA can be detected by extending the PTE-ATRA detection scheme. As long as the adversary employs memory modification as the technique to achieve ATRA, the arms race of the attacker and the defender will be iterated as the level of paging increases.

We have discussed ATRA that exploits in-memory paging data structures of address translation. However, the mitigation methodology of Memory-bound ATRA that manipulates the in-memory component of the address translation seems to be evident; external monitor should check the integrity of the in-memory paging data structures. For instance, a simple write detection scheme on a single kernel PTE which contains the mapping for the target kernel data structure can be a mitigation method for PTE-ATRA. In the case of PGD-ATRA, the detection scheme becomes more complicated since the PGDs are dynamically allocated as a new process is created. However, the basic strategy for mitigation would be the same as PTE-ATRA.

3.2 Register-bound ATRA

The *Register-bound ATRA* exploits the fact that all existing hardware-based external monitors are incapable of monitoring CPU states. A concept of this attack was briefly mentioned in previous work [7, 11, 18, 19]. They considered this attack to be impractical or hypothetical. Our Register-bound ATRA proves otherwise. Register-bound ATRA targets the base register (CR3 in case of x86) of virtual address translation. As previously explained, the CR3 register is used by the MMU as the root of page table walking. By modifying the register, we can induce the MMU to walk our malicious page table instead of the original. This means that the virtual address space mapping of the process victimized by ATRA can be arbitrarily manipulated by attacker.

Saved-CR3-ATRA: A straightforward way to achieve CR3 modification would be overwriting the saved-CR3 values from memory (i.e., *task_struct→mm→pgd* in the case of Linux). We mention this ATRA application method as

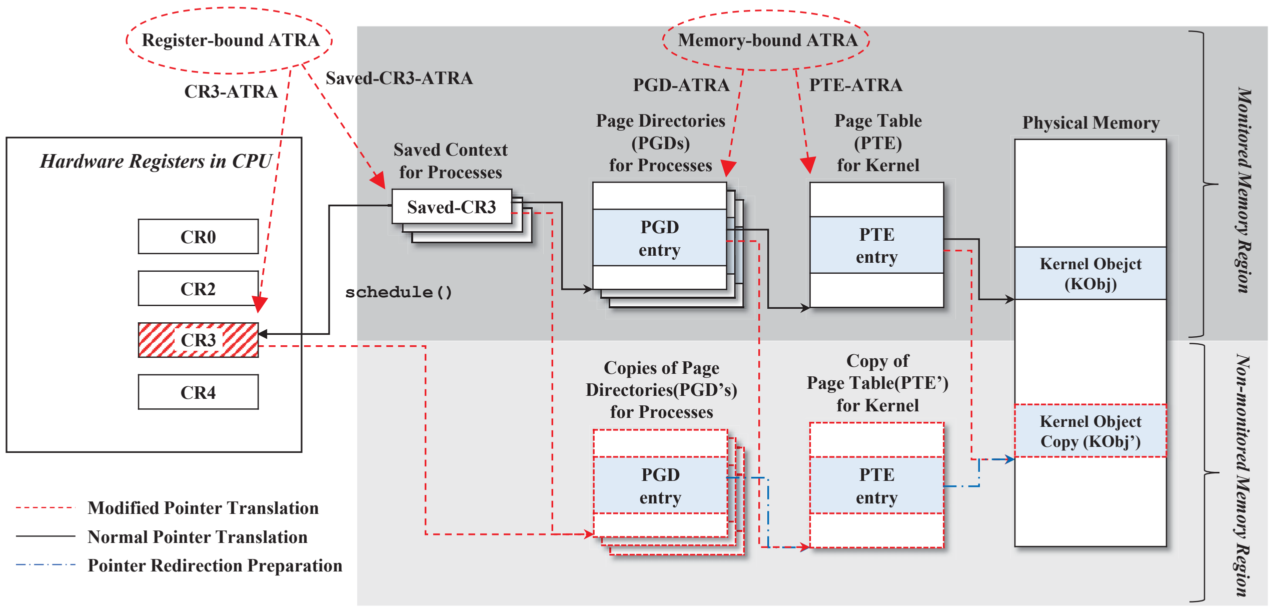
Register-bound ATRA
Memory-bound ATRA
CR3-ATRA
Saved-CR3-ATRA
PGD-ATRA
PTE-ATRA
Hardware Registers in CPU
CR0
CR2
CR3
CR4
Saved Context for Processes
Saved-CR3
schedule()
Page Directories (PGDs) for Processes
PGD entry
Page Table (PTE) for Kernel
PTE entry
Physical Memory
Kernel Obejct (KObj)
Kernel Object Copy (KObj')
Copies of Page Directories(PGD's) for Processes
Copy of Page Table(PTE') for Kernel
Monitored Memory Region
Non-monitored Memory Region
Modified Pointer Translation
Normal Pointer Translation
Pointer Redirection Preparation

Figure 1: Overview of Address Translation Redirection Attack (ATRA): ATRA can exploit the multiple steps in the virtual to physical address translation. PTE-ATRA modifies the page table entry that points to the original kernel object (KObj), so that the access is redirected to the copy (KObj'). PGD-ATRA manipulates the page directory entry that points to the original PTE. Saved-CR3-ATRA modifies the CR3 value saved in memory so that the context switch restores the modified value into the CR3 register when the targeted process is scheduled back. Finally, CR3-ATRA directly modifies the content of CR3 register so that the address translation starts with the manipulated PGD copy (PGD'). Note that CR3-ATRA directly updates CR3 register in CPU, not modifying any known memory-component.

Saved-CR3-ATRA, which provides an easy way to manipulate the entire CR3 of processes under context switching environment as shown in Figure 1.

Although Saved-CR3-ATRA is straightforward and easy to implement, it involves modification of an in-memory component since the saved-CR3 value resides in a memory region which the external monitor can be configured to observe. Figure 1 also shows ATRA attacks that achieve redirection attack by modifying in-memory components. They are PGD-ATRA, PTE-ATRA and Saved-CR3-ATRA. PGD and PTE ATRA manipulate the paging data structures on memory, and Saved-CR3-ATRA modifies the saved-CR3 value (via *task_struct→mm→pgd*), which is also located in a known memory region. Since these three ATRAs leave traces in memory, the external monitors can check the integrity of the known memory components. However, the next version that we will introduce does not manipulate any previously known memory component.

CR3-ATRA via Inter-Context Register Modification: Modifying an arbitrary register value of another process is not a straightforward task since the modification must not involve any change to the known memory regions which the external monitor could protect. In order to change the CR3 value of another process without modifying previously known memory region, attacker must induce the target process to update the CR3 register value from its running context. To overcome this challenge, we sought to find a register based system-wide hooking point which the attacker can inject his code into another running process across context switching. Good examples of this hooking points are *IDTR*, *SYSENTER_EIP_MSR* register.

We adopt the widely-known *Interrupt Descriptor Table (IDT)* hooking [24] to induce the target process to execute the code that modifies the CR3 register. More specifically, we force the victim process to invoke the attacker's code before the victim process enters the interrupt handler. The attacker's code will load the physical address of the relocated PGD into the CR3 register directly (Note that the attacker's code relocates entire chain of the paging data structures starting with the PGD as well). We refer this technique as *Inter-Context Register Modification*. We also describe the use of SYSENTER_EIP_MSR for enabling the *Inter-Context Register Modification* in Section 7.

The *Inter-Context Register Modification* technique enables the attacker to manipulate the CR3 register of an arbitrary process by changing the control flow (without modifying known in-memory component) before it accesses any kernel object. Using this technique, CR3-ATRA is launched every time a process enters kernel mode. This can be guaranteed because a process must raise an interrupt before entering the kernel address space (exceptional case will be further discussed in Section 7). System calls are the most common example of such interrupts; a non-privileged process raises software interrupt (e.g., *INT 0x80*) to temporarily enter the kernel space. Similarly, other cases such as page fault, signal handling and hardware interrupt would require the interrupt handling [23].

By hooking the IDTR, we avoid direct manipulation of the IDT entries which the memory access could be monitored.

Note that the initial starting point of an interrupt handling is the physical address value stored in a system-wide global CPU register that points the IDT (IDTR). Unlike the CR3, IDTR is not bounded to any process context. As a result, we can easily relocate the IDT out of the monitor's view by overwriting the IDTR just one time. Consequently we can manipulate any IDT entry which has the address of the interrupt handler without alarming the external monitor.

To sum up, the *Inter-Context Register Modification* can be achieved by hooking one of the system global registers (e.g., IDTR, SYSENTER_EIP_MSR) which cause other processes to execute the attacker's code when they run.

One may monitor the use of the IDTR updating instruction ($LIDT$) after booting as an indicator of IDT hooking attempt. However, existing hardware monitors cannot monitor the instructions that are being executed inside host CPU, thus they cannot determine whether LIDT is being executed or not.

4. CHALLENGES IN MITIGATING ATRA

In this section, we will discuss the mitigation issue regarding the two types of ATRA in detail.

4.1 Memory-bound ATRA Mitigation

To mitigate Memory-bound ATRA against an object, the external monitor should guarantee the address translation integrity of all virtual addresses that are related. Since the external monitor is capable of monitoring an arbitrary memory region, mitigating this threat seems to be possible by checking the integrity of related PGD entries and PTE entries. However, we found that there are some challenges regarding this mitigation methodology.

Race condition: In order to monitor PGDs and PTEs, external monitor must locate them from the physical memory. Note that once the external monitor locates PGD, finding physical memory location of PTE will be a trivial task since the PGD contains the physical address of entire PTEs (not vice versa).

Locating PGDs that are created for a newly forked process is a non-trivial task, since it is a dynamically allocated data structure which the memory location cannot be determined in advance. In Linux kernel, dynamic objects typically form a linked-list structure which can be traversed using pointers from a fixed entry point. Therefore, in order to locate them, external monitor could traverse the pointers of such linked list data structures inside the host memory at runtime. For example, KI-Mon [9] achieved this capability by implementing a technique so-called *Address Translation Engine*, which translates the host virtual address into physical address in order to traverse the linked list data structures inside the host memory. With these capabilities of the recent external monitors, we believe that locating the dynamic data structure such as PGD is not a challenge.

The challenging part is that locating these dynamic objects creates a race condition between the external monitor and the adversary. Consider a situation where a new kernel object that is of interest to both adversary and external monitor has been created during runtime. If the external monitor catches this event and locates the newly created object earlier than the adversary, the object will be successfully monitored without any problem. However, if the adversary's manipulation attack precedes the adding of the object to the monitored regions for the external monitoring, the further integrity protection for the newly created object might be rendered useless.

In case that the integrity of a dynamically created object can be verified by comparing it to a known-good-value or analyzing the semantic consistency invariant of the memory contents, the race condition may not be a problem. Consider a situation where an attacker hides a process by unlinking the data structure (e.g., *task_struct*) from its linked list which is used for the process enumeration. An external monitor can verify this attack by comparing the linked list of *task_struct* to the scheduler's run-queue that contains scheduled *task_struct*. In this case, the timing of the process hiding attack is not an issue for integrity verification, consequently race condition does not need to be considered. However, if the external monitor must enforce that the arbitrary initial value of the object not to be changed, the integrity verification can be unreliable due to the possibilities of race conditions.

The *Master Kernel Page Directory* contains the *untampered* original of the PGDs [23]. Thus the PGD contents regarding the specific target object's virtual address could be previously defined, which creates a semantic invariant. However, to mitigate Memory-bound ATRA, the external monitor should, in fact, check the integrity of additional data structures that are subject to the race condition (we will discuss this in the following paragraph). Therefore, we argue that the race condition is a challenge for mitigating Memory-bound ATRA.

Enumerating intermediate pointers: Another challenge in mitigating Memory-bound ATRA is the large attack surface where ATRA can be applied besides the PGD itself. This is because the kernel scheduler references a list of intermediate pointer values to reach the PGD of the next scheduled process. If the attacker is able to manipulate any of these pointers or change the virtual to physical address mapping of these pointer values, the kernel scheduler can be redirected to reference the maliciously crafted copy of the PGD located in a non-monitored region.

Before we discuss in more details, we should note some related facts regarding the PGD and kernel scheduler. When the context switch occurs due to system events such as interrupt and preemption, the scheduler selects next process to be scheduled. This selection is determined by the scheduling algorithm being used in the system (i.e., Completely Fair Scheduling, O(1) Scheduling and so on). Once the scheduler selects the next process, the scheduler references the PGD of corresponding process from memory and loads the PGD's physical address into the CR3 register (*switch_mm()*) [23].

As previously mentioned, there can be a significant number of intermediate pointer paths that the kernel scheduler can traverse to reach the PGD. For example, the value of the saved-CR3 is always referenced via *struct mm* (mm$\rightarrow$pgd) by scheduler. Similar to the case of PTE-ATRA, the attacker can copy the saved-CR3 value into another memory location and change the *mm* pointer instead. In addition, there exists a set of paging data structure entries that are referenced by the MMU for each pointer traversal. Hence, these paging data structures need to be monitored along with the pointers themselves to detect Memory-bound ATRA. Figure 2 depicts such an example.

We used LXR [25] tools to inspect the kernel source code for enumerating every pointer paths and the associated data structures between kernel scheduler and the saved-CR3. We,

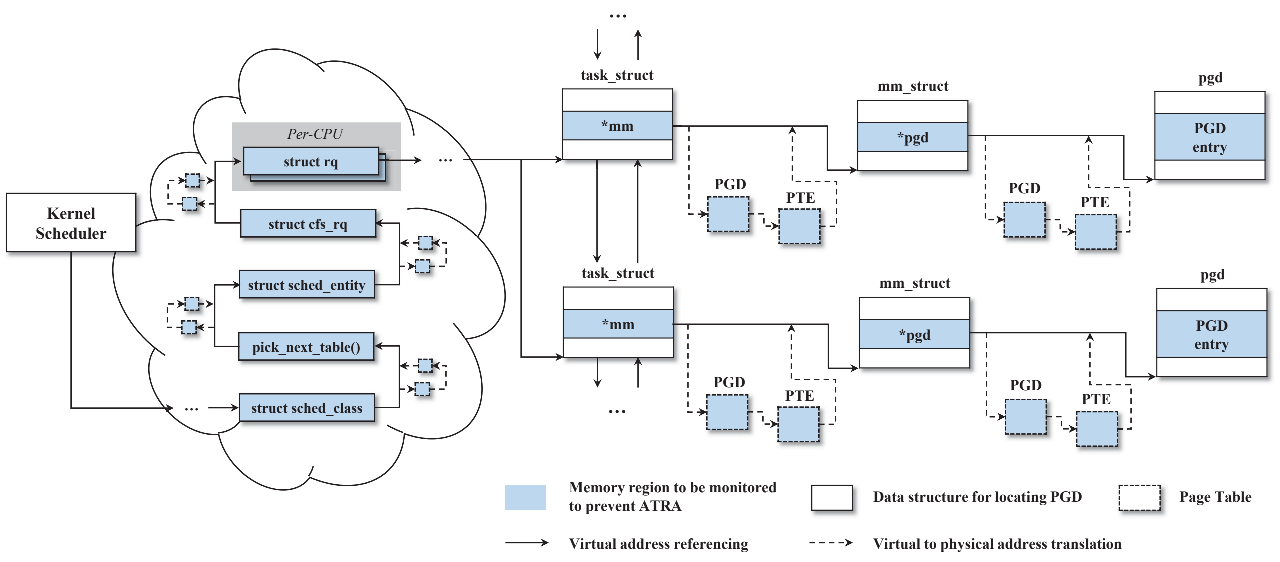

Figure 2: Additional Regions to be Monitored for Memory-bound ATRA Mitigation: The intermediate pointers between the kernel scheduler and the PGD form a convoluted linked list structure depending on the scheduling algorithm implemented in the kernel scheduler. Each of the PGD and PTE needed for translating virtual address of these pointers should also be monitored for mitigating Memory-bound ATRA. The solid arrow line represents the virtual address pointer referencing. The dotted arrow indicates the virtual to physical address translation.

then, checked each pointer manually and verified the exploitability for ATRA. Table 1 shows the profiled results regarding these data structures and pointers, types of which are categorized into global, or per-CPU or per-process specific.

Global data structures uniquely exist throughout the entire system, whereas per-CPU, per-process data structures exist for each CPU core and processes respectively. Global and per-CPU data structure are statically allocated during boot time, and they can be profiled beforehand so that they can be monitored. On the other hand, per-process data structures are created as a new process or thread is created, hence locations of these per-process data structures cannot be known in advance. Note that all these data structures are subject to the aforementioned race condition issue between the monitor and the attacker as the monitor scans the currently present processes and their data structures. Therefore, monitoring per-process data structures is a nontrivial task at best.

4.2 Register-bound ATRA Mitigation

As discussed, the mitigation of Memory-bound ATRA seems to be difficult, but there is no architectural limitation. That is, all modifications are made to the memory region where the location is in the view of the external monitor. However, this is not the case for Register-bound ATRA because the modifications on the registers are invisible to the hardware-based external monitor.

There has been a theoretical suggestion in the previous work for possible mitigation of relocation attacks. The idea is to scan the pages from the physical memory and search for the duplicate copy of kernel data structure which should uniquely exists throughout the entire memory [11]. For example, if there exists more than one system call table data structure, such case can be suspected as result of a relocation attack. However, the use of such heuristics would be limited and unreliable for many reason. First, it requires a signature for identifying the data structure from memory dump, since the semantic is not known to the memory scanner. Also, the heuristics can only be used for data structures whose number of instances are fixed. If a certain type of data structures are constantly allocated and deallocated, as the *task_struct* data structure for each process in Linux, such heuristics are simply not applicable.

One may argue that analysis of memory access pattern can be used as a countermeasure. As ATRA is applied to the victimized process, the original page tables of the process and the physical page frames that contain the target data structures are no longer referenced by the system. Hence, a sudden disappearance of memory traffics on the abandoned page tables and the original memory regions can be observed by hardware-based external monitors. A sudden drop in read accesses on the monitored critical kernel memory would certainly seem abnormal. Similarly, a discontinuity in bursty write traffic to dynamic kernel regions can also be distinguished from the normal kernel memory access patterns.

Nevertheless, such memory access pattern analysis cannot be a fundamental mitigation for ATRA. Register-bound ATRA can be selectively applied to victim processes, thus it is rather an uncomplicated task for the attacker to *synthetically* generate memory access patterns to the monitored regions. The memory pattern analysis will be completely hindered, if the attacker creates some non-ATRA-affected dummy processes that use the abandoned page tables and perform meaningless system calls to generate both read and write traffic to the once abandoned page tables and the memory regions. Likewise, even without a dummy process, non-

Source File	Data Structure	# of Instance	Property	Description
include/asm/page.h	pgd_t	per process	dynamic	saved-CR3 value used by scheduler
include/linux/mm_types.h	struct mm	per process	dynamic	contains pointer of PGD
include/linux/sched.h	struct task_struct	per process	dynamic	contains pointer of mm_struct
include/linux/sched.h	struct sched_entity	per process	dynamic	member of task_struct, referenced by scheduler
kernel/sched/core.c	struct sched_class	global	static	contains function pointer of enqueue_task()
kernel/sched/core.c	struct rq	per cpu	static	selects next task_struct to be scheduled
include/asm-generic.h	__per_cpu_offset	per cpu	static	contains the offset of percpu area which has rq

Table 1: The potentially exploitable data structures for ATRA from Linux kernel 3.8.1. If any of these data structures can be maliciously modified, the kernel scheduler will use attacker's saved-CR3 value. Note that each PGD, PTE entries for translating the virtual address of these data structures are also exploitable.

ATRA-affected processes will continue to access the original data structures, making the pattern analysis difficult. Moreover, the ATRA code when invoked can also be used to generate synthetic read/write traffic to the original memory region.

Also remind that the addresses of the regions to which the data structures are relocated by ATRA cannot be known, and they are *anonymous* among many other seemingly benign memory regions. Therefore, we conclude that the current hardware-based external monitors that cannot access the CPU processor states of the host are vulnerable to Register-bound ATRA.

5. IMPLEMENTATION

In this section, we present our implementation of ATRA. The current prototype is implemented and evaluated on a system running 32-bit x86 Linux operating system. We utilized two representative known rootkits which perform *system call table hooking* and *loadable kernel module (LKM) hiding attack*. The functionality of ATRA is implemented on top of these rootkits for demonstrating the capabilities and strengths against both static and dynamic kernel data structures.

We implemented ATRA attack as a *Loadable Kernel Module (LKM)* type of rootkit running in Linux kernel 2.6.28-11. The pseudocode of the ATRA attack are shown in Algorithm 1. The implementation mainly consists of two parts. First part is the installation of *ATRA_ISR* through IDT hooking by IDTR modification. By modifying the IDTR register to point to a copy of IDT which contains ATRA_ISR, we let all existing and newly spawning processes to execute our code in ATRA_ISR. The second part of the implementation in ATRA_ISR is invoked by the victim process, as it enters kernel mode. ATRA_ISR induces the victimized process to modify the CR3 value within its own context so that ATRA will be in effect every time the process enters kernel mode.

The attack process consists of three phases: (i) Attacker's preparation; (ii) Victim's execution of the attacker's prepared ATRA code; (iii) Victim's access to the relocated kernel data structure. Note that in Step 5, the attacker needs to flush the corresponding TLB entry to force the MMU to walk the relocated page table. Otherwise, the MMU will keep using the original address mapping that is recorded in the TLB cache.

Step 1 - Relocate KObj: Relocate the original kernel object (KObj) into non-monitored memory region, then manipulate the copied data structure. The *relocate* routine allocates a new page from non-monitored memory region with

Algorithm 1 Pseudocode for ATRA

```
Global: Interrupt Descriptor Table (IDT), IDT Register (IDTR)
Input: Kernel Object (KObj) to be manipulated
  ▷ Step 1
  KObj' ← relocate(KObj)  ▷ Copy KObj into a non-monitored memory region
  manipulate(KObj')
  ▷ Step 2
  IDT' ← relocate(IDT)
  redirect_pointer(IDTR, IDT')  ▷ Replace the IDTR value with the address of IDT'
  overwrite_IDT_entry(original_ISR, ATRA_ISR)
  handler ATRA_ISR
      ▷ Step 3 - PTE-ATRA
      PTE' ← relocate(PTE)
      redirect_pointer(PTE', KObj')  ▷ replace the entry of the PTE' which points the original KObj with the address of the KObj'
      ▷ Step 4 - PGD-ATRA
      PGD' ← relocate(PGD)
      redirect_pointer(PGD', PTE')  ▷ replace the entry of the PGD' which points the original PTE with the address of the PTE'
      ▷ Step 5 - Register-bound (CR3) ATRA
      redirect_pointer(CR3, PGD')  ▷ Replace the CR3 value with the address of PGD'
      flush_tlb()
      original_ISR()
  end handler
```

alloc_page() then obtains the corresponding virtual address via *page_address()*. Then, it copies the original page into the newly allocated page using *memcpy()* function. After this relocation is complete, the *manipulate* routine modifies the relocated kernel data structure as a typical rootkit does (in case of the *system call table*, we can hook the function pointer).

Step 2 - Relocate IDT: Relocate the original interrupt descriptor table into non-monitored memory region, then manipulate the copied table. The *relocate* routine duplicates the IDT using the *SIDT* instruction to fetch the virtual address of the IDT. Then, the *redirect_pointer* routine updates the IDTR to point the relocated IDT by using the *LIDT* instruction. Lastly, the *overwrite_IDT_entry* routine overwrites the IDT entries to point the manipulated handler called *ATRA_ISR*.

Step 3 - PTE-ATRA: Launch PTE-ATRA. The *relocate* routine duplicates the page table which contains the virtual to physical address mapping of the KObj. It can obtain the virtual address of page directory from *current→mm→pgd* and the physical address of the page table by indexing the page directory with upper 10 bits of the virtual address of

KObj. Then it converts the physical address of the page table into virtual address by using *__va()* macro. The *redirect_pointer* routine manipulates the virtual to physical address mapping in the relocated page table (*PTE'*) to map the physical address of relocated KObj (*KObj'*).

Step 4 - PGD-ATRA: Launch PGD-ATRA. Details of this step is the same as Step 3, except for relocating PGD and making it to point the manipulated PTE instead of the original.

Step 5 - CR3-ATRA: Replace the value of CR3 with the address of PGD'. The attacker needs to flush the corresponding TLB entry to force the MMU to walk the relocated copy of the page table. Any further memory access to the original kernel object will be redirected to the relocated one. After this step, it releases the execution flow to the original *Interrupt Service Routine (ISR)*.

Immediately after the attacker inserts the ATRA-enabled rootkit module, all processes including newly created ones will be affected each time when an interrupt is triggered. More precisely, the effect of ATRA persists while a process is in the kernel mode and vanishes when the process is scheduled out (and the changed TLB entry is flushed out). The kernel preemption could rarely affect this situation. We will discuss this further in Section 7.

It is important to note that when a process is scheduled out, the changed value in CR3 register is not saved back to the process's memory descriptor (e.g., *mm_struct*). This is because the kernel expects that the CR3 value is not modified while the process is running. When a context switch occurs, the value of CR3 is simply replaced with the saved-CR3 value of the next scheduled process.

In addition, since Register-bound ATRA includes additional steps in the interrupt handler, it can degrade the system performance when a process enters the kernel mode. In fact, the external monitor may not be able to measure the host system's performance change induced by ATRA, however if this change is noticeable enough for the system administrator we cannot say ATRA is practical attack. We will discuss this issue in Section 6.

To demonstrate ATRA, we implemented two ATRA rootkits. Each rootkit relocates the target kernel data structures (*system call table*, *LKM linked list*) before manipulating them. Description of these two rootkits are presented in Appendix A.1.

6. EVALUATION

We evaluate the correctness and performance impact of the ATRA implementation for the x86 Linux system. The real world rootkits such as *adore-ng* [26] manipulate various kernel objects such as system call table, LKM linked list, etc. By applying ATRA, we have shown that such important kernel objects can fall as a victim to the ATRA attack. After a successful ATRA, we verified that the system call table and LKM linked list are now relocated to a non-monitored physical memory area, and then we modified these relocated data structures to launch *system call table hooking attack* and *LKM hiding attack*. These two attacks were successfully performed while ATRA was in effect system-wide.

Since ATRA only manipulates the host states, not modifying any state on external monitors, the implementation specifics of the external monitors will have no bearing with the applicability of the attack. Any external monitors that inspect host memory via physical address for integrity-

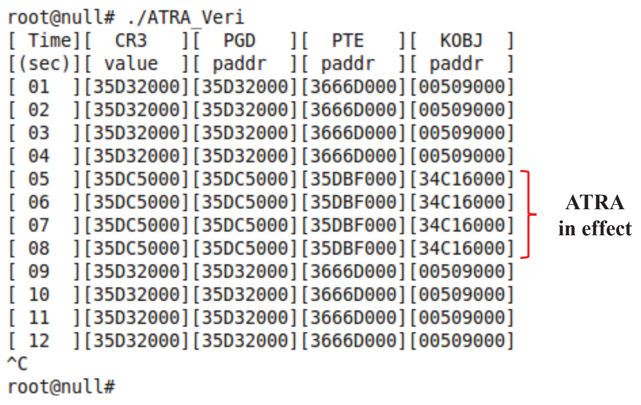
```
root@null# ./ATRA_Veri
[ Time][  CR3   ][  PGD   ][  PTE   ][  KOBJ  ]
[(sec)][ value  ][ paddr  ][ paddr  ][ paddr  ]
[ 01  ][35D32000][35D32000][3666D000][00509000]
[ 02  ][35D32000][35D32000][3666D000][00509000]
[ 03  ][35D32000][35D32000][3666D000][00509000]
[ 04  ][35D32000][35D32000][3666D000][00509000]
[ 05  ][35DC5000][35DC5000][35DBF000][34C16000]
[ 06  ][35DC5000][35DC5000][35DBF000][34C16000]
[ 07  ][35DC5000][35DC5000][35DBF000][34C16000]
[ 08  ][35DC5000][35DC5000][35DBF000][34C16000]
[ 09  ][35D32000][35D32000][3666D000][00509000]
[ 10  ][35D32000][35D32000][3666D000][00509000]
[ 11  ][35D32000][35D32000][3666D000][00509000]
[ 12  ][35D32000][35D32000][3666D000][00509000]
^C
root@null#
```

Figure 3: Screen capture of the ATRA-verification tool: ATRA_Veri program takes the virtual address of KObj as input and relocates KObj using ATRA. The numbers in each column corresponds to the physical address of PGD (obtained from CR3), PTE and KObj respectively. Note that we manually inserted the ATRA rootkit into kernel at Time 5, and removed it at Time 8. During Time 5 - 8 sec, the entire system accessed the kernel object at physical address *0x34C16000* which is a relocated copy of the original KObj.

check are subject to ATRA. Although no implementations of hardware-based monitors are publicly available for testing, one can determine ATRA applicability by checking if the external monitor design employs ATRA defense or not.

To confirm that ATRA has indeed relocated the entire chain of PGD, PTE and target kernel object (KObj), we made a simple verification tool (called ATRA_Veri) that takes a virtual address as input and enumerates the physical address of each paging component. The infected ATRA_Veri process repeatedly enters kernel mode and retrieves CR3 register value and calculates the physical address of PGD, PTE, and KObj. Using this tool, we have listed the physical addresses of aforementioned kernel data structures and compared them before and after ATRA rootkit insertion. We confirmed that the relocation attack was successfully applied upon loading the ATRA rootkit into the kernel. Figure 3 shows the screenshot of ATRA_Veri verification result where the ATRA attack was launched at Time 5 and removed at Time 8.

For evaluating the performance impact of ATRA on the system, we launched ATRA system-widely against the system call table and ran Unixbench 4.1 [21]. Unixbench performs a variety of system operations such as process creation, system calls, and so forth. We ran 3 trials, which took around 2 hours.

Figure 5 shows that the performance of the *execl* and the *system call* has degraded. We believe that the additional ATRA code that was planted into the interrupt handler was the main cause for the degradation. However, even though the additional code affects both the *execl* and the *system call*, the *execl* shows relatively higher performance degradation. We suspect that this is due to the initial page allocations made by ATRA. If the ATRA code is invoked for the first time after a process is created, the code executes the

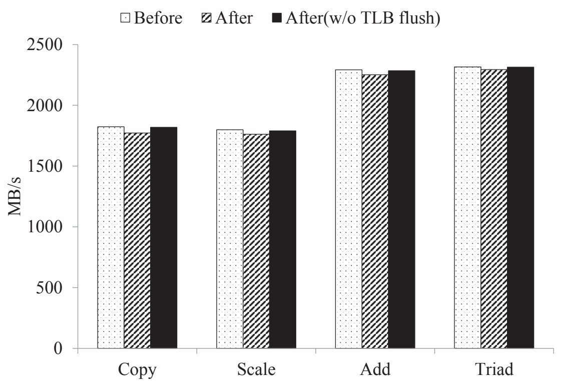

Figure 4: STREAM benchmark result: The memory bandwidth has been decreased after ATRA is in effect. However, if we launch ATRA without flushing the TLB (although the attack may not be effective), the performance degradation becomes negligible.

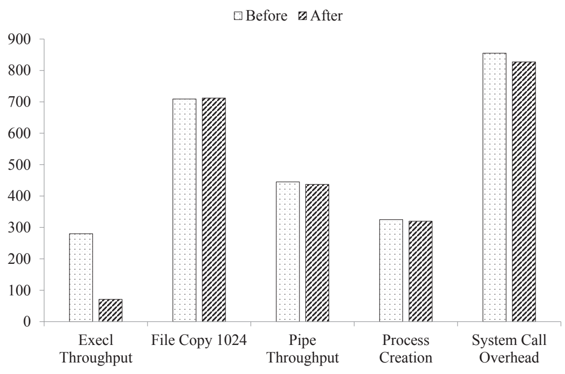

Figure 5: UnixBench score results before/after ATRA. The score unit for *File Copy and Pipe Throughput* is KB/s, and the rest is Loops Per Second (L/s). Note that the scores have been normalized by UnixBench baselines. The most noticeable differences between two results are the performance degradation of *Execl Throughput* and the *System Call Overhead* (Since the unit is L/s, the actual meaning of this category is system call throughput.) The degradation results can be explained by the additional ATRA code inside the interrupt handler.

initial routine, which allocates new pages to be used as the landing site for the relocation.

Since the *execl* (wrapper function for *execve* system call) is generally the first function invoked by a newly created process, (thus causing the initial page allocation) the *execl* function takes all the performance hit induced from the initial page allocation. However, in *system call* benchmark, the *getpid* system call is used for performance measure. Since the *getpid* system call will always be invoked after the *execl*, the initial overhead of the ATRA code will not be reflected in the *system call* benchmark. This explains the difference between the *execl* and *system call* benchmark result.

Other than the UnixBench performance experiment, we also ran the *STREAM* [20] benchmark to measure the memory I/O performance since ATRA is expected to incur additional TLB flushing. Flushing the TLB is known to incur a significant performance overhead since it requires page table walking instead of referencing the TLB cache. Note that the TLB flush induced by context switching does not flush the kernel mappings from TLB as the *page global bit* in the page table entry is set. Thus, the address translation mappings for kernel regions are rarely flushed on a special occasion (i.e., TLB is full). However, it is necessary for ATRA to immediately invalidate the TLB entries for the kernel regions, making the MMU to reference the modified mapping. Figure 4 shows the STREAM benchmark results where about 3 to 4% of performance overhead were observed when ATRA is in effect. However, ATRA without TLB flushing feature (although the attack becomes ineffective) showed nearly identical performance as the case without ATRA. This result indicates that the TLB flushing is the major cause for the memory I/O degradation.

The UnixBench and STREAM has shown that ATRA incurs additional performance overhead regarding process creation, interrupt handling and memory I/O. Nonetheless, we believe the slight difference of the system performance shown from the experiments is not a noticeable amount of change from the system administrator's view. Moreover, it would be difficult for the external monitor to measure the precise overall performance of the host system.

The benchmarks performed also served as an evaluation for the correct implementation of ATRA against victimized kernel object. The most anticipated error is improper setting of page table entry. Attacker should consider various possibilities which would create the segmentation fault while modifying the page table entry. However, the benchmark test proved the implementation caused no such errors.

Note that even if the ATRA implementation is correct, additional error could occur if the attacker naively launch ATRA for dynamic KObj against partial set of victim processes. This is because the content of shared dynamic kernel data structure could become inconsistent between two processes. Depending on the situation, this inconsistency might cause an unexpected error. This is rather a general synchronization issue as we often experience from multithreaded programming which the programmer (in this case, the attacker) should carefully consider.

7. DISCUSSION

Inter-Context Register Modification via INT 0x80 or SYSENTER: The Intel Pentium 2 processor has introduced a faster system call invoking instruction referred to as SYSENTER. When the SYSENTER instruction is issued, the *CS, EIP, SS, ESP* registers are automatically loaded with the values saved in the corresponding *Model Specific Registers (MSR).* System call invocation using SYSENTER bypasses the interrupt service routine in traditional invocations for better performance. The address of the SYSENTER handler is stored in the *SYSENTER_EIP_MSR* field of the MSR register. The SYSENTER handler can also be relocated in a similar fashion by modifying the *SYSENTER_EIP_MSR* field. In both cases – whether it is a *INT 0x80* system call or a SYSENTER system call – our relocation attack would be feasible.

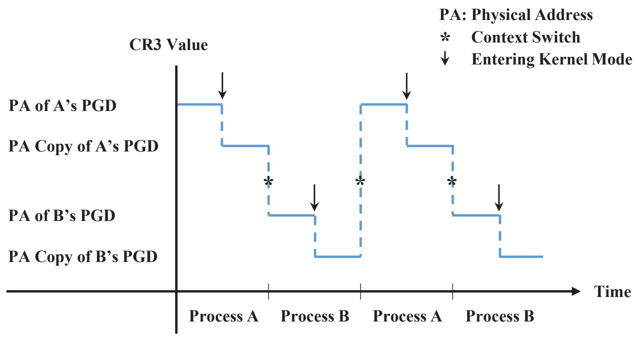

Figure 6: Transient effect of CR3-ATRA. The ATRA effect set by CR3-ATRA persists while the victimized process is in kernel mode until the process is scheduled out due to context switching. Note that when the victim process is scheduled back, the CR3 register will be restored with the saved-CR3 value in memory, which is the original value that has not been modified by CR3-ATRA.

Transient CR3-ATRA: Launching this attack modifies the CR3 register of the process directly, however, the register value saved in memory remains unchanged. Therefore, the effectiveness of CR3-ATRA becomes transient. The ATRA effect caused by CR3-ATRA persists while the victimized process is in kernel mode until the process is scheduled out. This is because when the victim process is scheduled back, the CR3 register will be restored with the previously saved one in memory. Figure 6 describes this transient effect of CR3-ATRA from the perspective of time and CR3 value.

Kernel Threads: Kernel threads, such as *kthreadd* or *ksoftirqd*, differ from conventional Linux processes; The kernel threads always run in kernel mode, so they do not need to invoke interrupt handler for accessing kernel objects. Therefore, the current methods of CR3-ATRA cannot be applied against kernel threads.

While the current implementation of CR3-ATRA leaves out the kernel threads, it would be possible to apply ATRA to kernel threads by injecting CR3-modifying code via other interrupt handlers (e.g., timer). It should be noted that the majority of processes are user-space ones, including the likely targets of rootkits such as *ps*, *ls*, and *bash*.

Register-bound ATRA in Preemptive Kernel: The transient effect of Register-bound ATRA begins each time a process enters kernel mode, and the effect ends when the global TLB entry is flushed. If the kernel is non-preemptive, this transient effect guarantees the ATRA-affected process to access the relocated kernel object. However, if the kernel is preemptive, on a rare occasion, the transient effect might vanish before the target kernel object is accessed.

For example, consider the following scenario: *Suppose the global TLB entry is flushed then a process enters kernel mode and invokes Register-bound ATRA code that updates the CR3 with the physical address of the relocated PGD. Immediately after this, the process is preempted and scheduled out by timer interrupt. When the process is scheduled back, the CR3 that contains the address of the relocated PGD will be replaced with the unmodified saved-CR3 value (on memory that can be protected by monitor). This makes Register-bound ATRA effect disappear before the target kernel object is accessed. However, the ATRA effect will be restored again when the process enters into kernel mode next time.*

To verify the likelihood of such occurrence, we tested Register-bound ATRA on preemptive Linux kernel and measured the number of such occurrence by repeatedly checking if the relocated kernel object was accessed or not. In our experiment where the victim process accessed the kernel object over billion times, no such case occurred — the original kernel object was never accessed. Therefore, we believe that the natural likelihood of this scenario is extremely low, although it is possible in theory.

8. RELATED WORK

The existing independent-processor-based kernel integrity monitors rely on an assumption that relocation attacks on their monitored kernel components are extremely difficult. Copilot [7] monitors a fixed range of physical memory address. The range is calculated by adding the fixed linear offset to the kernel symbols generated at compile time. The underlying assumption in this scheme is that the linear mapping between the virtual and physical addresses is intact. Vigilare [8] adapts the same method to identify the regions to be monitored, and the corresponding physical address range will be monitored throughout its operation. The authors explained that relocating a large portion of kernel code would generate an abnormal traffic in the host system bus.

The LLM [11] introduced a secure kernel integrity monitoring scheme which can be utilized for multi-core system. Their system is composed of a dedicated core and isolated memory for kernel integrity monitoring. They also sketched the possibility of *memory shadowing attack* to evade their monitoring system because the dedicated core was not able to access the context of the other core. They claimed that such hypothetical attack can be detected by heuristics such as identifying the free pages with kernel data structure.

A few hardware-based external monitors [7, 8, 11] have briefly mentioned potential ATRA-like attacks and sketched some heuristic methods for mitigation. The heuristics, however, might also be evaded or only covered the specific kernel data structure. For example, the heuristic that detects unusual copying of large portions of kernel code objects [8], can be circumvented by copying a small snippets of the kernel memory at a time. Furthermore, the heuristic, which identify duplicated kernel data structures in freed pages [11], cannot be applied to the dynamic kernel data structures because the legitimate kernel data structures as well as duplicated ones also remain in raw physical memory, which may cause numerous false alarms.

These existing works, including MGuard [10] and KI-Mon [9] that did not mention about the relocation attack at all, are vulnerable to the ATRA attack since their external processors are not equipped for monitoring CPU registers.

Unlike the hardware-based external monitors, in the hardware-assisted approaches such as Hypercheck [17] that uses SMM (System Management Mode) to run monitor code can be readily architectured to monitor the host's CPU registers since the SMM handler code for monitors shares the same processor with the monitored host. However, sharing the same processor entails a context-switching overhead between mode changes and imposes performance overhead on the host's normal operations. Each time the monitor code runs, the normal operation running on the same processor must yield and wait until the monitor code completes its inspection routines. Hypercheck also described the ATRA-like issue as *copy-and-change attack* and showed how the well-known IDTR hooking attacks can be detected.

Some of the prior works on hypervisor-based VMI were aware of the possibilities of ATRA-like monitoring evasion attacks. Sharif et al. [19] mentioned that any code with kernel privilege could relocate a page table by modifying CR3 register value. Also, Payne et al. [2] claimed that the relocation attack on the dispatcher component of their monitoring scheme would require *a considerable, if not impossible effort* since the dispatcher is in a 4MB page with the Windows XP kernel components. Although the possibility of attacks like ATRA has been mentioned in the previous studies [2, 19], there has not been a practical implementation and evaluation on the effectiveness of ATRA. We point out that ATRA might also affect hypervisor-based VMI tools, hoping that the mitigation of ATRA (described in Section 4) is considered in the design of future VMI tools. Further discussion is in Appendix A.2.

9. CONCLUSION

We presented Address Translation Redirection Attack (ATRA), which exploits the limitation in the existing hardware-based external monitors to completely circumvent all existing monitoring schemes. We showed the implementation of two types of ATRA called: (i) Memory-bound ATRA, (ii) Register-bound ATRA. We illustrated the gravity of the attack by first providing a set of possible ATRA mitigations, and then proving that ATRA foils all such countermeasures including even theoretical one. In our implementation, we successfully manipulated the address translation mechanism in Linux kernel without touching the memory regions monitored by hardware-based external monitors. Our evaluation with benchmarks showed that ATRA does not induce any noticeable performance degradation of OS. As long as this limitation of the hardware-based external monitors remains unresolved, any future advancement in their monitoring capability will be fruitless. It is our hope that this work will spur researchers to design a more trustworthy hardware-based external monitor, addressing the ATRA mitigation challenges presented in the paper.

Acknowledgements

This research was supported by MOTIE (The Minister of Trade, Industry and Energy), Korea, under the Brain Scouting Program (HB609-12-3002) by the NIPA (National IT Promotion Agency). This research is also based on work supported by the Software R&D Center, Samsung Electronics.

10. REFERENCES

[1] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09, 2009, pp. 545–554.

[2] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08. Washington, DC, USA: IEEE Computer Society, pp. 233–247.

[3] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11. New York, NY, USA: ACM, pp. 279–290.

[4] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, pp. 335–350.

[5] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, pp. 103–115.

[6] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07, 2007, pp. 128–138.

[7] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, pp. 13–13.

[8] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, pp. 28–37.

[9] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object," in *Proceedings of the 22nd USENIX conference on Security*, ser. SEC'13, 2013, pp. 511–526.

[10] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, "Cpu transparent protection of os kernel and hypervisor integrity with programmable dram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013, pp. 392–403.

[11] Y. Kinebuchi, S. Butt, V. Ganapathy, L. Iftode, and T. Nakajima, "Monitoring integrity using limited local memory," *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 7, pp. 1230–1242, 2013.

[12] A. T. Rafal Wojtczuk, Joanna Rutkowska. Xen 0wning trilogy.

[13] Xen: Security vulnerabilities. [Online]. Available: http://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html

[14] Vmware: Vulnerability statistics. [Online]. Available: http://www.cvedetails.com/vendor/252/Vmware.html

[15] Vulnerability report: Xen 3.x. [Online]. Available: http://secunia.com/advisories/product/15863

[16] Vulnerability report: Vmware esx server 3.x.

[17] J. Wang, A. Stavrou, and A. Ghosh, "Hypercheck: A hardware-assisted integrity monitor," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, S. Jha, R. Sommer, and C. Kreibich, Eds. Springer Berlin / Heidelberg, pp. 158–177, 10.1007/978-3-642-15512-3-9.

[18] S. Jin and J. Huh, "Secure mmu: Architectural support for memory isolation among virtual machines," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, 2011, pp. 217–222.

[19] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09, 2009, pp. 477–487.

[20] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[21] Byte-unixbench: A unix benchmark suite. [Online]. Available: http://code.google.com/p/byte-unixbench/

[22] *Intel 64 and IA-32 Architectures Software Developer's Manual*, INTEL, Aug 2012.

[23] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 2nd ed. O'Reilly and Associates, Dec. 2002.

[24] Idt hooking. [Online]. Available: http://resources.infosecinstitute.com/hooking-idt/

[25] The lxr project. [Online]. Available: http://lxr.sourceforge.net/en/index.shtml

[26] Stealth. the adore rootkit version 0.42. [Online]. Available: http://teso.scene.at/releases.php

[27] System calls and rootkits. [Online]. Available: http://lwn.net/Articles/297500/

[28] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*, 2011, pp. 297–312.

[29] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory," in *Proceedings of the 13th international conference on Recent advances in intrusion detection*, ser. RAID'10, 2010, pp. 178–197.

[30] B. Payne, M. de Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 2007, pp. 385–397.

[31] A. Lanzi, M. I. Sharif, and W. Lee, "K-tracer: A system for extracting kernel malware behavior." in *16th Symposium on Network and Distributed System Security*, ser. NDSS '09, 2009.

[32] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012, pp. 586–600.

[33] A. Srivastava and J. Giffin, "Efficient protection of kernel data structures via object partitioning," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, 2012, pp. 429–438.

[34] M. Grace, Z. Wang, D. Srinivasan, J. Li, X. Jiang, Z. Liang, and S. Liakh, "Transparent protection of commodity os kernels using hardware virtualization," in *Security and Privacy in Communication Networks*. Springer, 2010, pp. 162–180.

APPENDIX

A. APPENDIX

A.1 ATRA Enabled Rootkit Examples

System Call Table (SCT) hooking rootkit: System call table hooking is a common type of rootkit attack in the wild [27]. If a rootkit manipulates the *system call table*, it can achieve various types of malicious activities such as bypassing an anti-virus software and hiding its existence from the process list of the system. Specifically, to hide its own process information, it needs to manipulate related system calls that are used for retrieving the process information. The process status reporting tool such as *ps* reads */proc* directory to retrieve PIDs of all processes running in the system. By hijacking *read* system call, the attacker is able to hide information of its own process by deleting its PID from the retrieved result.

One may think that system call table manipulation attack can be easily mitigated by making the system call table immutable after the system's boot process, and this immutable memory region can be simply protected by hardware-based external monitors. However, we implemented the rootkit with ATRA that successfully manipulates the system call table without involving any changes in protected memory regions and subverts the naive defense mechanism. In our implementation, it first launches ATRA on the system call table and then hooks the *sys_getuid* and *sys_geteuid* system call entries into its internal function. Similarly, the rootkit can also manipulate other kernel status information such as network connection and file system information.

Loadable Kernel Module (LKM) hiding rootkit: The LKM hiding technique shown in *adore-ng* [26] is another typical type of real-world rootkit example. A rootkit can achieve this attack by removing metadata from doubly linked list of kernel modules, while keeping the actual data to reside in memory. The LKM hiding rootkit can hide a kernel module from reporting tools such as *lsmod*.

Hardware-based external monitors which is capable of tracing dynamic kernel data structure can detect this attack by observing the linked list in memory page. However, by launching LKM hiding with ATRA, this event becomes invisible from the monitor. We implemented an LKM hiding rootkit with ATRA, which manipulates the linked list of *struct module* kernel data structure containing metadata of inserted kernel modules. In our implementation, the rootkit launches ATRA for the targeted *struct module* kernel data structure object, which leads the subsequent LKM hiding attack to occur outside the view of the monitor.

A.2 Hypervisor-based Virtual Machine Introspection

There have been a few studies on hypervisor-based *virtual machine introspection (VMI)* for protecting OS kernels from untrusted codes such as rootkits. Such hypervisor-based VMI schemes can be classified into two approaches by their objectives. First, prior works on detecting and analyzing behaviors of rootkits by extracting semantics of kernel objects [3,6,28–32] emulate the MMU for the virtual address translation. Such MMU emulations use the page table of either guest OS or hypervisor to retrieve the corresponding host-physical address of the object. Second, prior studies on protecting the code and data of OS kernel [1, 2, 4, 19, 33, 34] utilize the hypervisor's page protection mechanism.

Meanwhile, we surmise that ATRA can affect some of the hypervisor-based VMIs under certain circumstances. For instance, ATRA might be able to evade the monitoring of VMI tools that depend on the guest page table for virtual to physical translation of the monitored objects, given that they do not monitor CR3 [6,30]. The monitor would not be aware of the relocated page tables and will be referencing the unused old page tables. However, if the guest page table walking is implemented such that the walking starts from the monitored context's CR3 value, ATRA might be detected.

VMI tools built on a hypervisor that employs *Shadow Page Table (SPT)* have the necessary capabilities to implement countermeasures for ATRA. More specifically, the hypervisors are capable of trapping register changes, and SPT operates by write-protecting the guest page tables and trapping all modification attempts. However, we accentuate that VMI tools should consider a countermeasure against ATRA for reliable introspection. On the other hand, a new memory virtualization technology often referred to as *Nested Paging* has arisen to ameliorate the performance overhead of SPT. Nested Paging, also known as *Extended Page Tables (EPT)* in Intel's terminology and *Nested Page Tables (NPT)* in AMD's, allows guest kernel to modify its page tables. While we conjecture that the removal of the write-protection in guest page tables would make the mitigation of ATRA more difficult, further investigation of the issue seems necessary.