Subverting Operating System Properties through Evolutionary DKOM Attacks

Mariano Graziano[1,3], Lorenzo Flore[2], Andrea Lanzi[2], and Davide Balzarotti[1]

[1] Eurecom
[2] Università degli Studi di Milano
[3] Cisco Systems, Inc.

Abstract

Modern rootkits have moved their focus on the exploitation of dynamic memory structures, which allows them to tamper with the behavior of the system without modifying or injecting any additional code.

In this paper we discuss a new class of Direct Kernel Object Manipulation (DKOM) attacks that we call *Evolutionary* DKOM (E-DKOM). The goal of this attack is to alter the way some data structures "evolve" over time. As case study, we designed and implemented an instance of *Evolutionary* DKOM attack that targets the OS scheduler for both userspace programs and kernel threads. Moreover, we discuss the implementation of a hypervisor-based data protection system that mimics the behavior of an OS component (in our case the scheduling system) and detect any unauthorized modification. We finally discuss the challenges related to the design of a general detection system for this class of attacks.

1 Introduction

Rootkits are a particular type of malicious software designed to maintain a hidden access to a compromised machine by targeting the running kernel. To mitigate this severe threat, several defense techniques for code protection and attestation have been proposed in the literature [27, 37, 39, 46]. These mechanisms try to protect the applications and the kernel code against any illicit modification of its instructions. This also prevents hooking techniques that attempt to divert the control flow to a routine controlled by the attacker.

However, while the code of the kernel is easy to protect, its dynamic data structures often remain outside the boundaries of traditional defenses. Left unprotected, they quickly became one of the main targets of modern *rootkits*, that manipulates their values to tamper with the behavior of the system without the need to modify the existing code. Even though these attacks are simple to understand and relatively easy to perform, protecting the dynamic memory structures of an operating system is a very difficult task. For instance, the classic example of Direct Kernel Object Manipulation (or DKOM) attack consists of hiding a running process by simply removing its corresponding element from the processes list (e.g., the `EPROCESS` structure in Microsoft Windows). Detecting DKOM attacks often rely on the assumption that even though some information can be modified, the original value can still be present in other OS

context. For example, even if an element is deleted from the `EPROCESS` linked-list, in order to be executed the process still needs to be present in the scheduling queue. Consequently, a common technique to detect DKOM attacks consists in cross-checking different sources of information to verify if their values are consistent. For instance, this is the approach adopted by the `psxview` Volatility plugin [45] to detect hidden processes. Researchers also proposed more sophisticated monitoring techniques that maintain a reference model of the running system to compare with the actual data structures. For example, Rhee et al. [36] proposed to use an *allocation driven mapping* to identify dynamic kernel objects by intercepting their allocations/deallocation operations, and use this information to maintain a precise model of the running kernel. This approach also included a hidden kernel object detector that uses this un-tampered view of kernel memory to detect DKOM data hiding attacks.

Despite the recent efforts in detecting DKOM attacks, all the proposed techniques are based on the assumption that during an attack there is always something *anomalous* in the *state* of the kernel dynamic data structures, typically in the form of a missing or modified element. However, a closer look at DKOM techniques reveals that there are two different ways to manipulate data to influence the behavior of the system. More precisely, from an attacker point of view, we can identify a *discrete* attack that only tampers with a dynamic structure at an isolated point in time, and an *evolutionary* attack that works by continuously tampering with the internal state of the system. In the first case, the objective of the attack is reached by changing some information stored in a data structure, by adding or removing elements, or by changing the pointer relationship between data structures. As we described above, this may leave the system in an inconsistent state, which can often be detected. In the second case, presented in this paper, the goal of the attack is instead obtained by influencing the behavior of the system by *continuously* modifying its memory and thus by affecting the evolution of its dynamic data structures.

Due to the nature of this attack, it is possible that every single snapshot of the system is indistinguishable from a clean state. Therefore, the attack only manifests itself in an anomalous *evolution in time* of a given *property* of the operating system. While this may seem just a minor variation of the original DKOM technique, in this paper we show that it has very severe consequences from a detection point of view. In fact, the only way to detect an evolutionary attack is to implement a detector that can verify if a certain behavioral property of the kernel is satisfied over time. This requires a very complex tool that continuously monitor the system, and replicates (or emulates) part of its behavior inside the detector.

The goal of this paper is twofold. First, we present the design and implementation of an *Evolutionary* DKOM (E-DKOM) attack, and show that it cannot be detected by any of the existing techniques. As a case study, we describe a novel attack against the OS scheduling algorithm. This attack can be used to silently block the execution of any critical security application, both in user- and kernel-space. The second contribution of the paper is to discuss the possible countermeasures. It is important to note that our goal is to detect the tampering of the operating system, and not the code of the rootkit itself.

At the moment, the only generic defense solution would be to use a reference monitor to trace all memory operations and enforce that only the authorized code can modify

a given critical structure. Unfortunately, this technique has two big limitations. First, it is likely to introduce a large computational overhead. Second, any memory access needs to be properly identified and attributed to the piece of code responsible for that operation. Unfortunately, a precise attribution in a compromised system is still an open problem – known as "confused deputy attack" [16].

As an alternative, we discuss a custom defense technique based on a monitor (implemented as a thin hypervisor) that can duplicate part of the behavior of the OS that needs to be protected (the scheduler's properties in our case), and guarantee that this behavior is respected by the running system. Unfortunately, this is not a general solution, as it would require a different monitor for every property that needs to be enforced.

The rest of the paper is organized as follows. In Section 2 and Section 3 we describe our attack and its own threat model, discuss its properties, and emphasize the differences with respect to traditional DKOM attacks. We then focus on a practical example in Section 4, in which we present the details of an attack against the Linux operating system scheduler. Section 5 shows the results of our attack tests, and Section 6 introduces our prototype hypervisor-based defense mechanism. Finally, Section 7 discusses the generality of the attack, its limitations and future work, Section 8 describes related work and Section 9 concludes the paper.

2 Evolutionary DKOM Attacks

There is a subtle difference between a traditional DKOM attack and its evolutionary counterpart that we present in this paper: in the evolutionary attack, the goal of the attacker is to affect the **evolution** of a data structure in memory, and not just its values. For instance, the two classic DKOM examples of privilege escalation and process hiding require the attacker to directly modify a number of kernel data structures to achieve the *desired state* (respectively remove an element from a linked list, or modify the UID of a process). In the more sophisticated version of DKOM attack we present in this paper, the *"desired state"* is replaced by a *"desired property"*. More in detail, the attack we present in the next sections affects the normal evolution of the red-black tree containing vital information for the scheduling algorithm. On the other hand, the traditional DKOM attacks change individual fields in data structures of interest, like the `task_struct` to unlink a task. The latter operation is discrete and does not affect the evolution of the `task_struct` list in any way.

This difference has a number of important consequences. First of all, while a traditional DKOM can be performed in one single shot, an evolutionary attack needs to continuously modify the kernel memory to maintain the target condition. Moreover, when the attacker stops his manipulation, the system naturally resumes its original operation. This fundamental difference seems to be in favor of traditional DKOMs, since a single memory change should be harder to detect that a continuous polling process. However, in this paper we show that in practice the result is the opposite of what suggested by common sense. In fact, from a defense point of view, it is easier to detect an *altered state* than to detect an *altered property*. To detect the latter, a monitor needs to record the evolution of the affected data structures over time, and also needs to replicate the logic of the kernel property that it wants to enforce.

While it is possible to implement such a detector (as we discuss in Section 6 for our attack), this needs to be necessarily customized for each property. As a result, it is difficult to propose a general solution for evolutionary attacks.

3 Threat Model

In this paper we assume a powerful attacker who is able to execute malicious code both at the kernel and at the user level, and who can modify any critical kernel data structures. Kernel-level access can be achieved via kernel-level exploits or social engineering the user to install a malicious kernel module. The attacker can also use sophisticated ROP rootkit techniques [20, 44] or other stealthy techniques [41, 42] in order to overcome existing code protection mechanisms. The attacker has the ability to make its malicious code undetected to any current state of the art anti-malware software.

However, since our defense solution is based on a custom hypervisor, we include both the hypervisor and the security VM as part of the trusted computing base (TCB). To focus only on the detection of E-DKOM attacks without replicating previous works, we also assume that the core kernel code of the user VM is protected and cannot be subverted by any malicious code. This can be achieved by making the kernel's code pages read-only [18, 38] or by using others code protection systems proposed in the past [10, 27, 39]. Existing protection techniques also ensure that the attacker cannot tamper or hook code of the OS, and cannot shutdown processes or kernel threads without the system notice [23, 27].

To summarize, our threat model covers an attacker that can run arbitrary code in the OS kernel and tamper with dynamic data structures, but that cannot modify the existing code or attack the hypervisor.

4 Subverting the Scheduler

In this section we first introduce the Completely Fair Scheduler (CFS) algorithm adopted by Linux-based OS. We then describe the principles of our attack to subvert the scheduling algorithm and present two different scenarios where our attack can be applied.

4.1 Goal

The goal of the attacker is to silently and temporarily stop the execution of a process without leaving direct evidences. This means the target process is no more able to run on the CPU but it is still visible and listed as a normal running application.

In a post-exploitation phase, this feature is a really valuable asset. Miscreants may disable security monitors and detectors so that system administrators or final victims do not notice any suspicious activity. A perfect target in this scenario is either an antivirus software or a network/host intrusion detection system. The desired result is to reach this goal without raising any warning or visible alarms. This can be achieved in several

ways in a modern operating system like Linux or Windows. The first idea that comes to mind is to kill the target application. However, this technique is easily detectable by the victim because the process (or processes) is no more listed in the list of the running applications. Several security applications have a watchdog specifically designed to detect these circumstances to restart the application. Another simple approach would be to suspend the process or turn it into a zombie. Unfortunately also this technique is not stealthy, and in a post-exploitation phase this cannot be tolerated. For example, it would be fairly easy to spot the anomaly by inspecting the output of a program like `ps`. Finally, another possible option could be to directly modify the code of the target application, for instance to inject an infinite loop or an attempt to acquire a lock on some unavailable resource. While this would be definitely more difficult to detect, security-critical applications often have kernel components to protect the integrity of their code.

Therefore, in order to reach our objective in a completely transparent way, a good target for the attacker would be to tamper with the scheduler implementation in the OS. This is a complex task and the implementation details may vary between different systems. For instance, a desktop machine has to be more reactive than a server. Indeed, it is clear the scheduling load may differ in a server spawning several tasks for all the incoming connections compared to a desktop machine used by an average secretary. All these differences affect the scheduler implementation. To perform the attack on the scheduler implementation the rootkit's author has to study in detail the inner mechanisms of the targeted component. We implemented this idea in a proof of concept attack against the current implementation of the Linux scheduler on a Debian "jessie" GNU/Linux distribution for both x86-32 and x86-64 systems. It is worth noting that our scheduler attack is able to stop the defensive mechanisms for an arbitrary amount of time. For example during an attack the intrusion detection system can be disabled, and then enabled again when the attack is terminated. We call such attack evolutionary transient attacks.

4.2 An Overview of the CFS Algorithm

As the name says, the main goal of the CFS algorithm used by the Linux kernel is to maintain a *fair execution* by balancing the processor time assigned to the different tasks of the system. The objective is to prevent one or more tasks from not receiving enough CPU time compared with the others. For this purpose, the CFS algorithm maintains the total amount of time assigned so far to a given task in a field called *the virtual runtime*. The smaller a task virtual runtime is in terms of execution, the higher the probability is to be the next being scheduled on the system. The CFS also includes the concept of *sleeper fairness*. This concept is used for the tasks that are not at the moment ready to run (e.g., those waiting for I/O) and it ensures that such tasks will eventually receive a comparable share of the processor when they are ready to execute. The CFS algorithm is implemented using a time-ordered red-black tree. A red-black tree is a tree with some interesting properties. First of all, it is self-balancing, which means that no path in the tree will ever be more than twice as long as the others. Second, any operation on the tree occurs in $O(logn)$ time – where n is the number of nodes in the tree.

4.3 CFS Internals

All tasks in Linux are represented by a memory structure called `task_struct` that contains all the task information. In particular, it includes information about the task's current state, the task stack, the process flags, the priority (both static and dynamic), and other additional fields defined by the Linux OS kernel in the `sched.h` file. It is important to note that since not all the tasks are runnable, the CFS scheduling fields are not included in the `task_struct`. Instead, the Linux OS defined a new memory structures called `sched_entity` to track all the scheduling information.

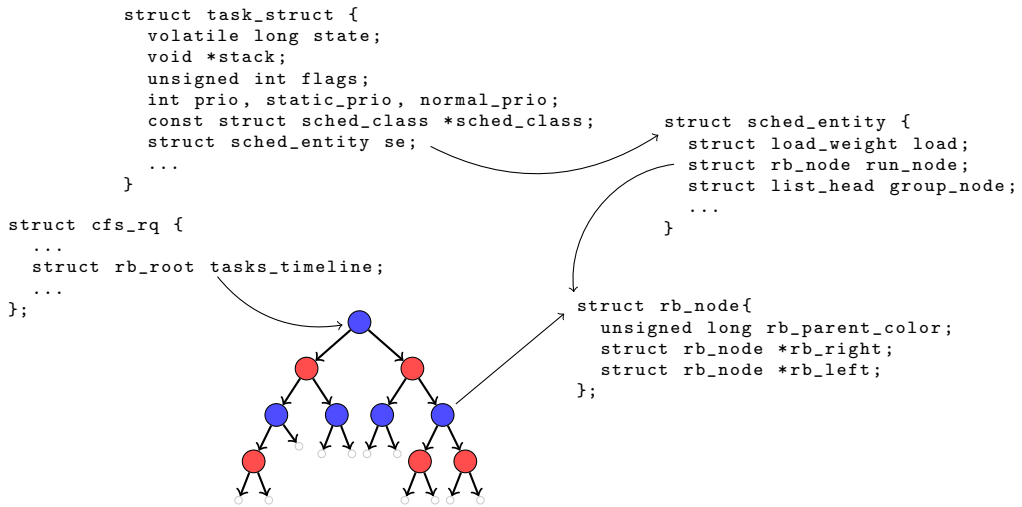

Fig. 1. CFS Black Tree Structures

The relationships between the various memory structures and the scheduling algorithm are summarized in Figure 1. At the root of the tree we have the `rb_root` element from the `cfs_rq` structure. Leaves in a red-black tree do not have any useful information; instead the internal nodes represent one or more tasks that can be executed. Each node in the red-black tree is represented by a `rb_node`. Such a node only contains the reference to the child and the color of the parent. The `rb_node` is defined into the `sched_entity` structure, which includes the `rb_node` reference, the load weight, and some data statistics. The most important field inside the `sched_entity` memory structures is the `vruntime`, which represents the amount of time the task has been running on the system. Such field is also used as the index for the red-black tree. The `task_struct` is at the top, and is responsible for describing the task and including the `sched_entity` structure.

The scheduling algorithm is quite simple and it is implemented inside the function `schedule()`. The first action of the function is to preempt the currently running task. Since for each task the CFS only knows the virtual running time, the algorithm does

not have a real notion of time slices for preemption, and therefore the preemption time is variable. After the scheduler interrupts the current running task, the task is put back into the red-black tree by calling the `put_prev_task` function. After that the scheduling function invokes the `pick_next_task` function that is in charge of selecting the next task to execute. This function simply takes the left-most task from the red-black tree and returns the associated `sched_entity`. By using the `sched_entity` and invoking the `task_of()` function the system returns the reference to the relative `task_struct`. At the end of this procedure the scheduler passes the task to the processor to execute it.

4.4 Scheduler E-DKOM Attack

In this Section we describe how an attacker can target the OS scheduler to suspend the execution of one or more of the processes running in a Linux system. Such an attack can be used in order to stop security applications such as antivirus software or Network Intrusion Detection System. Consequently, by using this technique the attacker is able to elude the system protection mechanisms without tampering with any OS code or modifying the control-flow of the running system.

Attack Principles From an architectural point of view, the attack requires a kernel module that executes code at regular time intervals (e.g., by registering a timer). The module walks the process list and identifies the process it wants to stop. It then collects the process descriptor and uses it to locate the corresponding node in the CFS *red-black* tree. Afterward, the attack alters the scheduling list by changing the *virtual runtime*'s value of the target process. In this way the attacker forces the scheduling algorithm to push the process at the end of the list and postpone its execution. By using this technique the attacker can stop any processes, thread, and kernel thread that are running on the system.

Implementation Details Our prototype first initializes a global kernel timer registering a call-back function to be executed at regular intervals. Then, the malicious module spawns two CPU-bound kernel threads to populate the scheduling list in case the queue is empty. This can be useful in cases when most of the processes are waiting for I/O operations, and the target process is the only one that requires to be executed. It is important to note that the kernel threads represent a normal task for the scheduling system, consequently the scheduler puts them in the same scheduling queue with the others user space processes and threads. Two is the minimum number to assure at least one predecessor and one successor of the target process.

After these two initial operations, the attack algorithm identifies the reference of the target process into the CFS red-black tree and queue it at the end of the scheduling list. This can be done by walking the `task_struct` looking for the element representing the target process. From there, the code can extract the `sched_entity` struct and use the `struct_rb_node` field to identify the corresponding element in the CFS red-black tree. In Figure 1 we show the link among the memory structures described so far. At this point the attack algorithm locates the rightmost element in the tree, which

correspond to the last task that will be scheduled for execution. Afterwards the kernel module changes the *virtual runtime* of the target process to a value higher than the value of the rightmost element. After this change, the scheduling algorithm, following the CFS policies, will move the target process at the end of the scheduling list. This operation is repeated every time the global kernel timer expires.

5 Attack Evaluation

In order to evaluate the real impact of our attack, we used it to stop two common security mechanism: a popular IDS, and the Inotify notification mechanisms [28] that is used by many programs to identify any modification on the files that occurs in the system. For instance, Inotify is used by several security applications, including Tripwire [1, 24] and by most of the antivirus systems.

Case Study I: Blocking an IDS In this experiment, a possible attack scenario is represented by a Linux router machine used to protect an intranet network. The router machine is equipped with an IDS and a system that verifies the code integrity of the kernel and user-space applications [27]. By using such defensive mechanisms, the attacker cannot modify any code running on the router and she cannot shutdown any user-space applications without the system noticing it. For our test we set up the IDS with a simple signature to detect a remote buffer overflow attack by looking at the presence of the shellcode pattern in the network packets.

Our experiment is divided in two parts. We first launched a simple buffer overflow attack over the network protected by the IDS , and we verified that the alert was correctly raised.

In the second test, we simulated that an attacker was able to install our rootkit into the router, using the IDS application as a target. While the IDS application was stopped by our rootkit, we run the network attack and double-checked that no alerts were generated. Meanwhile, Linux was reporting the targeted process as a running process. It is important to note that the kernel uses a circular memory buffer to store the network packets copied from the network card into the OS system before delivering them to the right application. Therefore, before resuming the execution of the IDS the attacker needs to generate benign traffic to force the queue to rotate and overwrite the network packets related to the attack.

Case Study II: Blocking Inotify *Inotify* is an inode monitoring system introduced in Linux 2.6.13. This API provides mechanisms to monitor filesystem events involving both files or entire directories. Most of the security applications, such as integrity checker (Tripwire) or antivirus software use such mechanism in order to detect any modification of the filesystem. For example, antivirus detectors re-scan any modified file in order to check whether a malicious infection occurred.

Inotify provides its own set of system calls: `inotify_init()` to create a new monitoring instance with its own file descriptor, `inotify_add_watch()` to add a file to the monitored group, and `inotify_rm_watch()` to remove the monitor. After the registration of the files and directories that the application wants to monitor, the code needs to

invoke the `poll()` API to be notified when the registered events happen. It is important to note that the inotify events are reported through a user-space device created as a communication channel between the user-space application and the kernel. This device is associated to a kernel buffer used to collect and temporarily store the filesystem events. By using the `read()` function, the application can retrieve information about which event have occurred.

For our evaluation we created a user space application that monitors a chosen group of files on our system. The goal of the attack is twofold. First, the attacker should be able to modify a file without the inotify-based application noticing the change. Second, the attack needs also to guarantee that after the attack phase, the inotify events should resume and correctly reach the application as if nothing has happened.

To this end our evaluation is composed by three steps. In the first step we run the inotify-based application and use our rootkit to temporarily stop its execution. In the second step, the attacker modifies one of the monitored file, and then forces a number of events (at least 1024*16) on other files with the goal to saturate the kernel buffer associated to the device. This way the event associated to the target file is overwritten by the new benign modifications. Finally, the rootkit wakes up the inotify application, and we verified that it did not receive any event about the attacker modification.

This can be quite severe in a number of scenarios. For instance, the Android system uses a similar inotify mechanism that is mainly adopted to build security monitors and detectors [12]. Our attack can temporarily disable them without leaving any trace in the system.

Attack Discussion: One may argue that a malicious kernel module could be detected by a simple detector that is able to find out in memory a footprint of the malicious code or detect any suspicious activities by monitoring the frequency of the interrupt timer issued at the kernel level (e.g., timing traces). Even if those techniques could be effective against our attack, the kernel module can hide its own timing activities and code in several sophisticated ways.

First of all it can hide the presence of the code just diverting the control flow of a benign timer kernel module by using dynamic hooking that targets transient control data as described in [43] and then perform a ROP attack for changing the time scheduling activity. By using these attack techniques the detector cannot see any suspicious kernel modules among the list of the registered kernel modules timer and the malicious code is reduced to a few ROP gadgets resulting in minimal memory footprint. A more resilient approach is called Address Translation Redirection Attack (ATRA) and is presented in [21]. By using such a technique the attacker can relocate important kernel objects (e.g., malicious kernel module) and makes the entire system refer to the copy by attacking the page table data structures of the OS kernel. Finally, as shown in [25], our malicious kernel module could be completely implemented in GPU space. A GPU-assisted malware binary contains code destined to run on different processors. When executing it, the malware loads the device-specific code on the GPU, allocates a memory area accessible by both the CPU and the GPU, initializes it with any shared data, and schedules the execution of the GPU code. Depending on the design, the flow of control can either switch back and forth between the CPU and the GPU, or separate tasks can run in parallel on both processors.

Other defense solutions to this attack could rely on a remote code attestation mechanism [7], a method to remotely check whether some security proprieties of the running application are preserved. In this case it is important to note that the attacker, as we can show in the previous section, can stop the defensive mechanism to be scheduled for the duration of the attack, and then restored it. By using code attestation method or any other watchdog mechanisms that check the status of the process (e.g., stack, registers, etc.) it is difficult to set up the right time to check since we do not know when the attack will happen. Remote attestation could be set to run constantly for the entire life of the process. Deploying this solution on real-time systems could be prohibitive in terms of performance overhead, and it could be difficult to use to monitor more than one precess at a time.

6 Mitigation

In this section we describe the design and implementation of a detection system that can be used to protect against the scheduler attack presented in Section 4. We start by presenting the idea behind our solution, we then describe our system architecture, and we finally evaluate our approach against some scheduling attack samples.

6.1 Defense Mechanism Principles

Our approach for the detection of scheduling attacks is to observe and mimic the behavior of the OS scheduler by intercepting events that occur in the OS context. More in details, in case of the scheduling subsystem, the idea is to monitor the execution time of all processes and check if the fairness property is preserved. In order to obtain the real execution time for each process/task we need to intercept some fundamental operations about the process activities such as the process creation and termination, the process execution, and the process I/O waiting. By using those operations our system can carefully estimate the execution time for each process and, by mimicking the behavior of a real scheduler, detect whether any anomaly (i.e., a process starvation) occurs in the system.

6.2 Defense Framework Architecture

Our defense mechanism is implemented as a custom hypervisor. This is required in order to obtain a resilient and robust reference monitor in presence of kernel-level attacks. Our anomaly detection mechanism is based on the assumption that the system should give the same amount of execution time to each process (fairness scheduling property). Consequently, if one process that is not blocked in I/O operations is not scheduled at least once for each quantum of time, the system raises an alarm. From an architectural point of view, our system consists of two main software components: (1) the *Task Tracer* and (2) the *Periodic Monitor*. Both components work together to simulate the fairness property and to reveal any anomaly on the system.

The main goal of the Task Tracer is to replicate the tasks information at the hypervisor level, storing them in a list of `task_struct` data structures. To this end, the

Task Tracer needs to intercept a number of process events. In particular it needs to detect four main events:

- **Process Creation**: This event happens when the *create process* system call is invoked.
- **Process Exit**: This event occurs when an exit system call or any process error exception is invoked by the system.
- **Process Execution**: This event occurs when a process is assigned to a given processor for its execution.
- **Queue Insertion** & **Removing**: These events happen when a task is inserted or removed from the scheduling queue (CFS red-black tree).

When a new process is created, the *Task Tracer* component allocates a new `task_struct` element to keep track of its information: name, process description etc. Moreover, for each new process, the system adds a life timestamp field named `last_seen`. This value represents the starting time of the process life, that will later be used to check the time spent by the process waiting on the scheduling queue. The queue insertion and removing operations are at the core of our detection mechanism. In fact they allow the system to set the starting and ending time for each process. The starting time begins when the process is inserted into the scheduling queue. In particular when a process will be inserted in the scheduling queue (CFS red-black tree), the hypervisor detects it and it sets the timestamp field for this particular task. In case the process is not scheduled for execution after a certain time (defined by a configurable scheduling threshold) the system reports an anomaly. The effect of the remove operation from the scheduling queue is to reset the timer associated to a particular process. It is important to note that intercepting the insert and remove operations is sufficient to monitor the execution time for all the processes of the system, since one of the main assumption of the Linux scheduling algorithm is that every process needs to be added to the scheduling list before it can be executed.

The goal of the other software component, the *Periodic Monitor*, is to periodically check the status of the execution time for each process and update their timestamps (`last_seen` fields). More in details, every time the timeout occurs, the *Periodic Monitor* goes through all the elements of the *task_list* created by the *Task Tracer* software component and checks among all the monitored processes the timestamp field reported in the `task_struct` element. If the difference between this timestamp field and the current timestamp is greater than the scheduling threshold the system reports an anomaly, otherwise it just update its value with the new timestamp.

6.3 Implementation Details

Our current prototype is implemented as an extension of HyperDbg, an open-source hardware-assisted hypervisor framework [11]. Typically, by monitoring low-level interactions between the guest operating system and the memory management structures on which the OS depends, a hypervisor can infer when a guest operating system creates processes, destroys them, or triggers a context-switch between them. These techniques can be performed without any explicit information about the guest operating system vendor, version, or implementation details [23]. Unfortunately, our detector needs

some information that cannot be inferred only by observing the interactions between the guest OS and the memory management structures. For example, insert and remove operations on the scheduling queue or the creation and destruction of userspace and kernel threads are fine-grained operations that cannot be identified by observing from outside the OS. Therefore, our framework needs to rely on a hooking mechanism that is specific for a particular operating system (Linux in our current prototype). In order to intercept each task creation event, we inserted a hook on the `wake_up_new_task` function. Such function is invoked the first time a new task is inserted into the scheduling queue after the system invokes `do_fork`. This is used to create a process on the system. We chose this function since the argument of the `wake_up_new_task` function is the `task_struct_element` that already contains all the process information that will be stored into the hypervisor memory. The system also needs to intercept a process or task termination for two reasons: (1) when a process explicitly call the `exit` function and (2) when it receives a signal or exception for its own termination. In both cases the function that is invoked is the `do_exit`. When such a function is called, by using the kernel macro `current` the system obtains the pointer to the `task_struct` related to the process to terminate. Consequently our hypervisor puts an hook on the `do_exit` function to intercept this information. Finally the system needs to intercept the queue operations: insert and remove. In particular when a process is inserted in the scheduler running queue (CFS black-red tree) a function called `enqueque_task` is invoked. This function is in charge for inserting the `task_struct` structures inside the CFS tree, and any information about the inserted process can be retrieved starting from `ecx` register. For removing elements from the scheduler queue, the operating system provides a function called `dequeue_task`. This function is called when the scheduler removes a task from the CFS tree and the reference to the task in this case is stored into `edx` register.

To implement the *Periodic Monitor* component inside the hypervisor we extended the core of HyperDbg. In particular, we created a time simulator inside the hypervisor by using the Timestamp Counter `TSC` register provided by the `x86` architecture. This register counts the clock cycle and it is independent from the processor frequency. In particular, the hypervisor core reads the value of TSC each time a `VMexit` occurs in the system. If the elapsed time reach the timeout set by the *Periodic Monitor*, the hypervisor invokes the periodic monitor component. It is important to note that the `VMexit` are very frequent in the system, consequently our timer simulator does not suffer from any considerable delay.

6.4 Evaluation

In this section we describe the experiments that we performed in order to test our defensive mechanism. The main goal of the experiments is to test the efficacy and the efficiency of the detection system.

Overhead In the first experiment we measure the overhead produced by our system. To this end we performed two main tests. In the first test we measured the execution

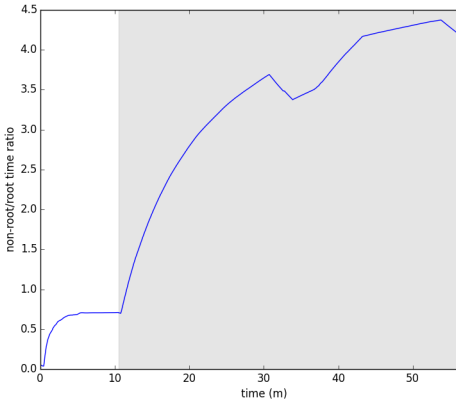

Fig. 2. Detection System Overhead During Normal Operation

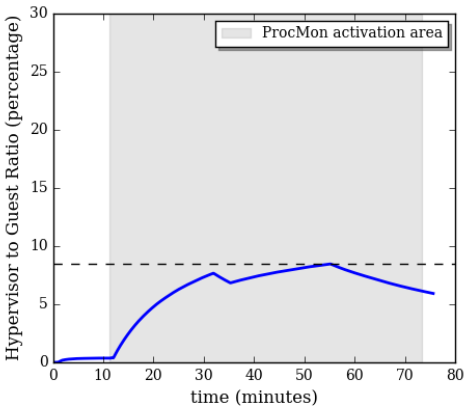

Fig. 3. Detection System Overhead Under an Artificial Stress

time with our detection framework enabled, while the user performs a number of normal operations – like browsing the web (e.g., Facebook, Google, etc.), reading PDF documents, and editing files for a total of 60 minutes. To compute the overhead we use the TSC timer provided inside the hypervisor. We compute the ratio between the time spent inside the hypervisor with respect to the time spent for the OS execution. We report the result in Figure 2. As we can see from the Figure, the gray area represents the window time where the detector is active. The line in the graph shows instead the ratio between the execution time spent into the hypervisor and the execution time spent into the OS. We can observe that overhead never goes above 5%.

Since during the normal operation the system overhead is low, we performed a second test where we stress the allocation/dis-allocation of the processes in order to measure the worst case scenario. For this test we used the stress suite to simulate a huge allocation/deallocation of the processes on a Linux system. The overhead we observed in this case was at most 9%. The test was run for 80 minutes and the final result is reported in Figure 3. Again, it is important to note the experiments performed in these tests produced a very intensive process allocation/deallocation and therefore it is not representative of the behavior during the normal process activities of the system.

Detection Accuracy In order to measure the detection accuracy of our system we tested the system while running some scheduler attacks. Since we have never observed such attacks in the wild, we used our artificial dataset to test the application. More in details, we again performed the experimental evaluation with a popular IDS and with *Inotify* (as explained in Section 4) but this time with our defensive mechanism enabled. In this test, our system was able to detect both attacks and correctly recognize the anomalous process that was under attack. We also performed an artificial experiment on kernel threads. In this case we first created some artificial kernel threads and we

then blocked their execution by using our attack. Also in this experiment, our system was able to detect all the attacks performed against the OS kernel.

6.5 False Positives & False Negatives

It is important to note that both false positives and false negatives can occur depending on the value of the detection threshold set by the system. In particular, if such a threshold is too low, and therefore close to the real waiting time for scheduled tasks, the system can raise false alarms. On the other hand, if the threshold is too large, the system can miss short attacks that fits into the time window. Therefore, the threshold should be tuned on the values of scheduler waiting times observed on the monitored OS. After a short training period, we set the threshold to 40ms. We then run our defensive system on our work computers for one week without observing any false alarm.

7 Discussion

In this section we discuss the generality of the proposed attack, the limitation of the defense solutions and possible future work.

7.1 Generality

In this paper we presented a new class of attacks. For the sake of simplicity we only described a single instance of E-DKOMs. In particular we chose to investigate the scheduler attack because it perfectly summarizes all the important key points of the evolutionary DKOMs attacks and it was relatively easy to implement.

The scheduler subsystem is a good candidate but it is not the only possible target. In fact, the operating system offers other interesting functional components to investigate such as the memory management, the network subsystem, and the I/O subsystem. A requirement for E-DKOM attacks is to tamper with dynamic data structures that contain fundamental information for controlling the OS behavior. The targeted data structure needs to contain information that defines an OS *property* along with an OS specific behavior. In the scheduler attack example the OS *property* to subvert was the execution fairness, every process defined into the run-queue structure need to be scheduled for running after a certain time window. The goal of the attacker was to create starvation for a select set of processes (e.g. AVs, IDSs). Another possible target for E-DKOM attacks can be the virtual memory subsystem. In this case the *property* is related to the memory pages replacement algorithm and the way the algorithm chooses the page to swap to disk (e.g., LRU or FIFO). The attacker can alter this *property* by changing the memory structure that contains the numbers of accesses received by the page. By altering this number an attacker can decide which page should be stored on disk and also on which disk location (e.g., filesystem inode), creating a potential data leakage among the applications.

We believe that the OS contains a significant number of sensitive memory structures that can be tampered by an attacker to consequently tamper a certain OS behavior

without being detected. Automatically discovering such memory structures along with the analysis of attack impact will be the task of our future research.

7.2 Limitations

The defensive solution described in the previous sections is based on a custom hypervisor that plays the role of an external agent able to monitor the execution of the guest operating system. Unfortunately, collecting information from outside the OS is not a trivial task, and requires to overcome the well-known problem of the *semantic gap* [6, 9, 22]. The Intel hardware support for virtualization simplifies only in part this issue, allowing the hypervisor to catch only low level events (e.g., writing attempts to control registers). Unfortunately, all the abstractions introduced by the operating system are lost and need to be reconstructed by the hypervisor code.

In the literature, several solutions have been proposed to detect hidden processes from a virtual machine monitor. These techniques typically intercept all the writing attempts to the `CR3` register by leveraging the Intel hardware support. This control register contains the base address of the *page directory*, a fundamental data structure to translate virtual to physical addresses. At every *context-switch*, the OS loads the right value of the `CR3` register to access the process's virtual address space. In this way, systems like *Antfarm* [23] are able to discover all the running processes by observing this low level event. Other systems, like *Patagonix* [27], achieve the same goal by setting the process's pages as non-executable (using the `NX` flag). In this way, every execution attempt is intercepted, allowing the hypervisor to discover all running processes.

Our scheduler attack introduces a new challenge: the hypervisor needs to identify the processes that are in the scheduled queue but are not executed in the system. If a process is not executed, then there is no access to its `CR3`, nor to its `NX` pages. In fact, the attack introduced in this paper may stop the process during its creation, so that the monitoring system would never observe the `CR3` associated to the program.

To make the problem worse, the granularity of this instance of E-DKOM is at the thread level, but the address space is shared among all threads of the same process – making an approach based on the monitoring of the `CR3` register too imprecise. For this reason, to implement a successful defense technique, the hypervisor needs to set breakpoints in the kernel code to extract threads information and to inspect the state of each tasks, (e.g. if it is in the running or waiting queue).

Moreover, the hypervisor has to mimic the OS scheduler component to guarantee the scheduling property and detect deviations from the expected behavior. In our example, this requires to follow over time the evolution of the scheduler data structures, in particular the evolution of the `runqueue` to spot any anomaly.

For all these reasons, we believe this instance of E-DKOM attack sheds light on several limitations of current solutions to address the *semantic gap*. Moreover, since each solution would need to be specifically tailored for the property tampered by the attack, this example also shows the challenge of developing a general solution for the detection of E-DKOM attacks.

8 Related Work

Over the years, operating systems have introduced several countermeasures to hinder the exploitation of userland applications. These protections have significantly raised the bar for the attackers, making it increasingly difficult to gain full control of a remote machine. As a consequence, it is now fundamental for criminals to gain a persistent and stealth access on a compromised target immediately after the breach. This is often achieved by installing a rootkit in the OS kernel. The role of a rootkit is to hide resources in the compromised machine, and this can be achieved either by using hooking techniques or by tampering with dynamic kernel data structures.

In the literature, several approaches have been proposed to protect the kernel from the malicious modifications introduced by rootkits. A first set of countermeasures was designed to guarantee the integrity of the kernel, in order to prevent attackers from modifying its code and introducing hooks [46]. There are two ways to achieve this objective: i) by introducing a self-defense mechanism in the kernel, such as PatchGuard [29] for Windows x86-64 or ii) by adopting an external monitor, such as a VMM-based system [14, 37, 38, 46] or a dedicated hardware coprocessor [26, 30, 32, 48]. For instance, SecVisor [38] and Nickle [37] are two hypervisor solutions that protect the integrity of the kernel code from unintended modifications. Unfortunately, this class of protections have been bypassed by DKOM attacks [19, 31] which target dynamic kernel data without the need to modify the kernel code.

A more complex and comprehensive defensive solution is to enforce the control flow integrity (CFI) of the kernel. CFI was initially proposed by Abadi et al [2] for userland applications and then extended and ported to the kernel by Petroni et al. [33]. The state-based CFI (*SBCFI*) proposed by Petroni is enforced by a hypervisor and periodically scans the kernel memory to detect deviations from the allowed control flow. *SBCFI* can detect persistent control flow changes but fails to prevent DKOM attacks.

To protect against DKOM, it was necessary to introduce new solutions to enforce the kernel *data* integrity. The most interesting approaches in this direction are based on invariants or on data partitioning. The first class can be split into two subgroups: external systems [3, 18, 36] and memory analysis [5, 8] techniques. External systems are implemented as either a virtual machine monitor [18, 36] or by using a separate machine [3]. The rationale behind these defensive techniques is to take an untampered view of the objects running in the target kernel and then compare this list with the invariants derived by walking the kernel data structures. Similarly, memory analysis solutions [5, 8] leverage memory snapshots to isolate kernel objects and then compare with a list retrieved directly from the live system. Unfortunately, invariants may not exist for some kernel data structures, thus a different approach has been proposed around the concept of object partitioning. For instance, Srivastava et al. [40] proposed *Sentry*, a hypervisor solution able to divide kernel objects fields in different memory regions depending on their security impact. Writes on these sensitive fields are then monitored and a strict access control policy is enforced to detect if the writer is legitimated. This approach has two main drawbacks: a large performance overhead and the complexity of the writer's identification process.

More formal architectures have been proposed to verify dynamic kernel structures as proposed by Petroni et al. [34]. These rule-based systems may be effective to detect advanced threats but they are error prone and depend on the astuteness of the rules writer. E-DKOM attacks are able to bypass these protections given the huge new attack surface exposed by this generic technique.

The solution we propose in this paper belongs to the class of mimic defensive solutions. Researchers have often proposed approaches to isolate a single OS component and emulate it outside the system to provide a ground truth to the analyst [15, 17]. In our case, a custom hypervisor reproduces the same scheduling algorithm (CFS) in a faithful step by step emulation. The drawback of these approaches is that they only solve a particular instance of the problem. In fact, we show how to protect the scheduler but an attacker can still exploit a different property of the kernel. Moreover, these defensive solutions are not ideal, as discussed by Garfinkel [13]. Specifically, the developers have to carefully think and manage all possible corner cases in order to avoid possible bypasses, making this process highly prone to errors.

To the best of our knowledge, E-DKOM attacks – as formalized in this paper – have never been discussed in the literature. The most complete overview of the DKOM's problem has been provided by Baliga et al. [4] as well as Rhee et al. [35]. They proposed a DKOM's taxonomy and investigated a novel data kernel attacks and possible POC solutions. Although they mention the huge attack surface exposed by modern kernels and the failing approach adopted by current detectors, they did not address our attack. In light of the current state of the art, it is clear that all the existing defense mechanisms are not able to detect this new class of attack and new comprehensive solutions are required to address this new and complex threat.

In our example of E-DKOM attack, we use soft timer interrupt requests (STIR) in order to perform polling tasks and modify the targeted dynamic memory structures. Even if the detection of malicious soft timer interrupt has been addressed in the literature [47], an attacker can use several stealthy techniques to hide the execution of malicious kernel code. For example, by using Address Translation Redirection Attacks (ATRA) [21], an attacker can hide memory pages along with kernel interrupt routines (e.g. code memory page). This would trick an integrity code checker to analyze the code of a benign timer routine. Finally, it is worth noting that in our threat model we consider an attacker equipped with state of the art offensive tools, that are not always detectable by the current defensive solutions.

9 Conclusion & Future Work

In this paper we discuss a new type of DKOM attack that targets the evolution of a data structure in memory, with the goal of tampering with a particular property of the operating system. Since at every single point in time the internal state of the OS is not anomalous, the detection of this type of attack, which we call evolutionary kernel object manipulation, requires a completely new approach as well.

We conducted a number of experiments to show the feasibility of an evolutionary attack against the Linux scheduler. Our attack is able to temporarily block any process or kernel thread, without leaving any trace that could be identified by existing DKOM detection and protection systems. Moving to the defense side, we then presented the

design and implementation of a hypervisor-based detector that can verify the fairness of the OS scheduler. While our prototype is able to detect all the attacks with zero false positives, the implementation needs to be customized on a case-by-case basis, and it also requires the hooking of a number of internal functions of the operating systems (making the technique harder to maintain and port to other systems). This shows that evolutionary attacks are very hard to deal with, and more research is needed to mitigate this threat.

As a future work we are now investigating other possible E-DKOM attacks that can be executed on some specific kernel subsystems. As we already discussed in the Generality Section, one example could be related to the virtual memory subsystem and in particular to the selection of the candidate memory page to swap. It would also be interesting to work on an automated analysis system that can autonomously inspect the OS kernel and identify possible candidate data structures that have an interesting time-evolutionary behavior – and that therefore could be targeted by future E-DKOM attacks.

References

1. Tripwire. `http://www.tripwire.com/`.
2. Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), CCS '05, pp. 340–353.
3. Baliga, A., Ganapathy, V., and Iftode, L. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference* (2008), ACSAC '08, pp. 77–86.
4. Baliga, A., Kamat, P., and Iftode, L. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007), SP '07, pp. 246–251.
5. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 555–565.
6. Chen, P. M., and Noble, B. D. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems* (2001), HOTOS '01.
7. Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., OHanlon, B., Ramsdell, J., Segall, A., Sheehy, J., and Sniffen, B. Principles of remote attestation. *International Journal of Information Security 10*, 2 (2011), 63–81.
8. Cui, W., Peinado, M., Xu, Z., and Chan, E. Tracking rootkit footprints with a practical memory analysis system. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 601–615.
9. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (May 2011).
10. Fattori, A., Lanzi, A., Balzarotti, D., and Kirda, E. Hypervisor-based malware protection with accessminer. *Computers & Security 52* (2015), 33 – 50.
11. Fattori, A., Paleari, R., Martignoni, L., and Monga, M. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)* (Antwerp, Belgium, Sept. 2010). `https://code.google.com/p/hyperdbg/`.
12. Fedler, R., Kulicke, M., and Schtte, J. An antivirus api for android malware recognition. In *MALWARE* (2013).

13. Garfinkel, T. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium* (February 2003).
14. Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium* (2003), pp. 191–206.
15. Grill, B., Platzer, C., and Eckel, J. A practical approach for generic bootkit detection and prevention. In *EuroSec 2014* (4 2014).
16. Hardy, N. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev. 22*, 4 (Oct. 1988), 36–38.
17. Haukli, L. Exposing bootkits with bios emulation. In *Blackhat US* (August 2014).
18. Hofmann, O., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with OSck. In *ASPLOS* (2011).
19. Hoglund, G., and Butler, J. Rootkits: Subverting the Windows Kernel. In *Addison-Wesley Professional* (2005).
20. Hund, R., Holz, T., and Freiling, F. C. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Presented as part of the 18th USENIX Security Symposium (USENIX Security 09)* (Montreal, Canada, 2009), USENIX.
21. Jang, D., Lee, H., Kim, M., Kim, D., Kim, D., and Kang, B. B. Atra: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 167–178.
22. Jiang, X., Wang, X., and Xu, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *In:Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2007).
23. Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX 2006 Annual Technical Conference (USENIX '06)* (Boston, MA, June 2006).
24. Kim, G. H., and Spafford, E. H. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2Nd ACM Conference on Computer and Communications Security* (1994), CCS '94, pp. 18–29.
25. Ladakis, E., Koromilas, L., Vasiliadis, G., Polychronakis, M., and Ioannidis, S. You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger. In *Proceedings of the 6th European Workshop on System Security* (April 2013), EuroSec, Prague, Czech Republic.
26. Lee, H., Moon, H., Jang, D., Kim, K., Lee, J., Paek, Y., and Kang, B. B. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Presented as part of the 22nd USENIX Security Symposium* (Washington, D.C., 2013), USENIX, pp. 511–526.
27. Litty, L., Lagar-Cavilla, H. A., and Lie, D. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proc. 17th Usenix Security Symposium* (San Jose, CA, July 2008).
28. Love, R. intro to inotify. `http://www.linuxjournal.com/article/8478`.
29. Microsoft. PatchGuard - Kernel Patch Protection. `https://technet.microsoft.com/en-us/library/cc759759%28v=ws.10%29.aspx`.
30. Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., and Kang, B. B. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 28–37.
31. Peter Silberman and C.H.A.O.S. FUTo. `http://uninformed.org/index.cgi?v=3&a=7&p=7`.
32. Petroni, J., Fraser, T., Molina, J., and Arbaugh, W. A. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (San Diego, CA, 2004/// 2004), SSYM'04, USENIX Association, USENIX Association, pp. 13 – 13.

33. Petroni, Jr., N. L., and Hicks, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (Oct. 2007), pp. 103–115.
34. Petroni Jr, N. L., Fraser, T., Walters, A. A., and Arbaugh, W. A. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th conference on USENIX Security Symposium* (2006/// 2006), pp. 20 – 20.
35. Rhee, J., Riley, R., Xu, D., and Jiang, X. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES 2009)* (Fukuoka, Japan, March 2009).
36. Rhee, J., Riley, R., Xu, D., and Jiang, X. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Proceedings of the 13th International Symposium of Recent Advances in Intrusion Detection (RAID 2010)* (Ottawa, Canada, September 2010).
37. Riley, R., Jiang, X., and Xu, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection* (2008), RAID '08, pp. 1–20.
38. Seshadri, A., Luk, M., Qu, N., and Perrig, A. SecVisor: A tiny hypervisor to guarantee lifetime kernel code integrity for commodity oses. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2007).
39. Seshadri, A., Perrig, A., Doorn, L. V., and Khosla, P. Swatt: Software-based attestation for embedded devices. In *In Proceedings of the IEEE Symposium on Security and Privacy* (2004).
40. Srivastava, A., and Giffin, J. Efficient protection of kernel data structures via object partitioning. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACSAC '12, pp. 429–438.
41. Srivastava, A., Lanzi, A., and Giffin, J. *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ch. System Call API Obfuscation (Extended Abstract), pp. 421–422.
42. Srivastava, A., Lanzi, A., Giffin, J., and Balzarotti, D. *Detection of Intrusions and Malware, and Vulnerability Assessment: 8th International Conference; DIMVA 2011, Amsterdam, The Netherlands, July 7-8, 2011. Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, ch. Operating System Interface Obfuscation and the Revealing of Hidden Operations, pp. 214–233.
43. Vogl, S., Gawlik, R., Garmany, B., Kittel, T., Pfoh, J., Eckert, C., and Holz, T. Dynamic hooks: Hiding control flow changes within non-control data. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 813–328.
44. Vogl, S., Pfoh, J., Kittel, T., and Eckert, C. Persistent data-only malware: Function hooks without code. In *Proceedings of the 21th Annual Network & Distributed System Security Symposium (NDSS)* (Feb. 2014).
45. Volatility Foundation. psxview Volatility command. `https://github.com/volatilityfoundation/volatility/wiki/Command%20Reference%20Mal#psxview`.
46. Wang, Z., Jiang, X., Cui, W., and Ning, P. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 545–554.
47. Wei, J., Payne, B. D., Giffin, J., and Pu, C. Soft-timer driven transient kernel control flow attacks and defense. In *ACSAC* (2008).
48. Zhang, X., van Doorn, L., Jaeger, T., Perez, R., and Sailer, R. Secure coprocessor-based intrusion detection. In *Proceedings of the Tenth ACM SIGOPS European Workshop* (Sept. 2002).