Toward Revealing Kernel Malware Behavior in Virtual Execution Environments

Chaoting Xuan[1], John Copeland[1], and Raheem Beyah[12]

1 Georgia Institute of Technology, 2 Georgia State University

Abstract. Using a sandbox for malware analysis has proven effective in helping people quickly understand the behavior of unknown malware. This technique is also complementary to other malware analysis techniques such as static code analysis and debugger-based code analysis. This paper presents *Rkprofiler*, a sandbox-based malware tracking system that dynamically monitors and analyzes the behavior of Windows kernel malware. Kernel malware samples run inside a virtual machine (VM) that is supported and managed by a PC emulator. By building its monitoring component into the PC emulator, Rkprofiler is able to inspect each instruction executed by the kernel malware and therefore possesses a powerful weapon against the malware. Rkprofiler provides several capabilities that other malware tracking systems do not. First, it can detect the execution of malicious kernel code regardless of how the monitored kernel malware is loaded into the kernel and whether it is packed or not. Second, it captures all function calls made by the kernel malware and constructs call graphs from the trace files. Third, a technique called *aggressive memory tagging* (AMT) is proposed to track the dynamic data objects that the kernel malware visit. Last, Rkprofiler records and reports the hardware access events of kernel malware (e.g., MSR register reads and writes). Our evaluation results show that Rkprofiler can quickly expose the security-sensitive activities of kernel malware and thus reduces the effort exerted in conducting tedious manual malware analysis.

Key words: Dynamic Analysis, Rootkit, Emulator.

1 Introduction

When a attacker breaks into a machine and acquires administrator privileges, kernel malware could be installed to serve various attacking purposes (e.g., process hiding, keystroke logging). The complexity of attackers' activity on machines has significantly increased. Rootkits now cooperate with other malware to accomplish complicated tasks. For example, the rootkit Rustock.B has an encrypted spam component attached to its code image in memory. The initialization routine of this rootkit registers a notification routine to the Windows kernel by calling the kernel function PsCreateProcessNotifyRoutine. This notification routine is then invoked each time that a new process is created. When detecting the creation of Windows system process Service.exe, Rustock.B decrypts the spam components and injects two threads into the Service.exe process to execute the spam components [7]. Without understanding the behavior of the Rustock.B rootkit, it would be difficult to determine how the spam threads are injected into the Service.exe process. To fully comprehend malicious activitiy on a

compromised machine, it is necessary to catch and dissect key malware that attackers have loaded onto the machine. Thus, analyzing rootkits is an inevitable task for security professionals.

Most of the early rootkits were rudimentary in nature and tended to be single-mission, small and did not employ anti-reverse engineering techniques (e.g., obfuscation). These rootkits could be manually analyzed using disassemblers and debuggers. Since rootkit technology is much more mature today, the situation has changed. Rootkits have more capabilities and their code has become larger and more complex. In addition, attackers apply anti-reverse engineering techniques to rootkits in order to prevent people from determining their behavior. Rustock.C is one such example. The security company, Dr. Web, who claimed to be one of the pioneers that provided defense against Rustock.C, took several weeks to unpack and analyze the rootkit [8]. The botnet using Rustock.C was the third largest spam distributor at that time, sending about 30 million spam messages each day. This example illustrates how the cost incurred by the delay of analyzing kernel malware can be huge. As another example, the conficker worm that has infected millions of machines connected to the Internet was reported by several Internet sources [6][10] (on April 8th 2009) that a heavily encrypted rootkit, probably a keylogger, was downloaded to the victim machines. At the time of the initial submission of this paper for publication, which was three days later, no one had published the details of the rootkit. It is still unclear how severe the damage (e.g., economic, physical) will be as a result of this un-dissected rootkit. Accordingly, developing new approaches for quickly analyzing rootkits is urgent and also critical to defeating most rootkit-involved attacks.

Several approaches have been proposed to address the rootkits analysis problem to some extent. For examples, HookFinder [30] and HookMap [27] are two rootkit hooking detection systems. The former uses dynamic data tainting to detect the execution of hooked malicious code; and the latter applies backward data slicing to locate all potential memory addresses that can be exploited by rootkits to implant hooks. K-tracer [14] is another rootkit analysis system that uses data slicing and chopping to explore the sensitive kernel data manipulation by rootkits. Unfortunately, these systems cannot meet the goal of comprehensively revealing rootkit behavior in a compromised system. Meeting this goal requires answering two fundamental questions: 1) what kernel functions have been called by rootkits?; and 2) what kernel data objects have been visited by rootkits? In the paper, we present a proof-of-concept system, Rkprofiler, in attempt to address these two questions. Rkprofiler is built based on the PC emulator QEMU [5] and analyzes Windows rootkits. The binary translation of QEMU allows Rkprofiler to sandbox rootkits and inspect each executed malicious instruction. Further, Rkprofiler develops the memory tagging technique to perform just-in-time symbol resolving for memory addresses visited by rootkits. Combining deep inspection capability with the memory tagging capability, Rkprofiler is able to track all function calls and most kernel object accesses made by rootkits.

The rest of paper is structured as follows. We point out the technical challenges for completely revealing rootkit behavior in Section 2. Section 3 gives the overview of the Rkprofiler system, including its major components and malware analysis process. Section 4 presents the technical details of tracking rootkits. Then, we present several case studies in Section 5 and discuss the limitations of Rkprofiler in Section 6. Section 7 surveys related work and Section 8 gives the conclusion of the paper.

2 Challenges

Modern operating systems (OSs) like Windows and Linux utilize two ring levels (ring 0 and 3) provided by X86 hardware to establish the security boundary between the OS and applications. Kernel instructions and application instructions run at ring level 0 and 3 respectively (also called kernel mode and user mode). The execution of special system instructions (INT, SYSENTER and SYSEXIT) allows the CPU to switch between kernel mode and user mode. This isolation mechanism guarantees that applications can only communicate with the kernel through well-defined interfaces (system calls) that are provided by the OS. Many sandbox-based program analysis systems take advantage of this isolation boundary and monitor the system calls made by malware [1][3]. While this approach is effective to address user-space malware, it fails to address kernel malware. This is because there is no well-defined boundary between benign kernel code and malicious kernel code. Kernel malware possess the highest privileges and can directly read and write any kernel objects and system resource. Moreover, kernel malware may have no constant "identity" - that is, some kernel malware could be drivers and others could be patches to benign kernel software. So the first challenge is how to create a "virtual" boundary between kernel malware and benign kernel software. Rkprofiler overcomes this challenge by using the timing characteristic of malware analysis. Before loading kernel malware, all kernel code is treated as benign code; after loading kernel malware, newly loaded kernel code is considered malicious. Note this "virtual" boundary only isolates code, but not data. This is because the data created by malicious code can also be accessed by benign code, and Rkprofiler does not monitor the operations of benign kernel code for the purpose of design simplicity and better performance.

When monitoring a VM at the hypervisor layer, only hardware-level activities (e.g., memory reads and writes) are observed. To make these observations useful, it is necessary to translate the hardware-level activities to software-level activities. Here, software-level activities refer to using software terms to describe program activities. For example, "local variable X is modified." This translation requirement is also known as the semantic gap problem [9]. This problem can be expressed as the following: given a memory address, what is its symbol? Automatically finding the symbols for static kernel objects (global variables and functions) is straightforward, but automatically finding the symbols for dynamic kernel objects (data on stack and heap) is challenging. This challenge is not well addressed by previous work. In this paper, we propose a method called *aggressive memory tagging* (AMT) to overcome this challenge. The basic idea of AMT is to perform the symbol resolution at run time and derive the symbols of dynamic kernel objects from other kernel objects whose symbols have been identified. It should be pointed out that Microsoft does not publish all kernel symbols and we can only gather the kernel symbols that are publically available (Microsoft symbol server, DDK documents and some unofficial Internet sources). So the current implementation of Rkprofiler is not able to resolve many unpublished symbols. Nevertheless, we find that it identifies most sensitive available symbols in our evaluation.

3 System Description

Rkprofiler is composed of four software components: *generator*, *controller*, *monitor* and *reporter*. These software components operate in three phases temporally: *pre-analysis*,

analysis and *post-analysis*. In the pre-analysis phase, the generator collects symbols of native Windows kernel modules (e.g., ntoskrnl.exe, ndis.sys) from the program database (PDB) files available on the Microsoft symbol server [15] and header files in Microsoft's Driver Development Kit (DDK). Two databases are produced by the generator at the end of this stage: type graph and system map. The type graph database contains the data type definitions of native Windows kernel modules. There are six classes of data types: basic type, enum, structure, function, union, and pointer. The data types in the last four classes are considered as composite data types, indicating that a data type includes at least one sub data type. For example, the sub data types of a structure are data types of its data members. In the type graph database, Rkprofiler assigns a unique type ID to each data type. A data type is represented by its type ID, type name, size, class ID and class specific data (e.g., the number of sub data types and their type IDs). The system map database keeps the names, relative virtual addresses and type IDs of global variables and functions used by native Windows kernel modules. In addition, the names and type ID of parameters and the return value for each function are also stored in system map. The generator is comprised of several executables and Perl scripts.

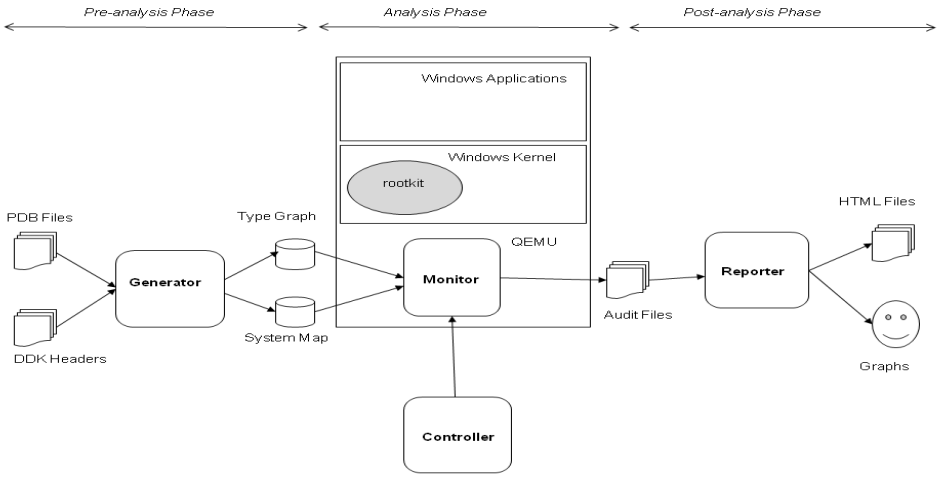

Fig. 1. Rkprofiler architecture and rootkit analysis process

Executing malware and monitoring its behavior are carried out in the analysis phase. Two components of Rkprofiler, controller and monitor, are involved in this phase. The monitor is built into QEMU. The controller is a standalone shell script that sends commands to the monitor via the Linux signal mechanism. Four commands are defined in their communication messages: `RKP_INIT`, `RKP_RUN`, `RKP_STOP` and `RKP_REPORT`, (which are explained shortly). First, a test VM is started and goes into a clean state in which no malware is installed and executed. Then, the controller sends a `RKP_INIT` command to the monitor. After receiving the command, the monitor queries the kernel memory image of the guest OS and creates a hash table of trusted kernel code. Next, the controller instructs the monitor to start monitoring through a `RKP_START` command.

At that point, Rkprofiler is ready for the monitoring task. For example, a user starts executing malware in the VM. Depending on the attack objectives of the malware, the user may run other applications to trigger more behaviors from the malware. For example, if the malware is intended to hide processes, the user may open the Windows task manager to induce the hiding behavior. Since the malware can be tested repeatedly, the attack objectives of the malware can be inferred from the analysis results of previous tests. To obtain the monitoring result or end the test, the user can have the controller issue `RKP_REPORT or RKP_STOP` commands to the monitor. The first command informs the monitor to write the monitoring result to local audit files; the second command prompts the monitor to stop monitoring and clear its internal data structures. Four audit files in CSV format are generated in the analysis phase: trace, tag trace, tag access trace, and system resource access trace. These files contain the functions called by the malware, their parameters and return values, kernel data objects visited by the malware and their values. In the post-analysis phase, the reporter is executed to create user-friendly reports. Using the audit files generated in the analysis phase, the reporter performs three tasks. First it builds a call graph from the call trace and saves the graph to another file; second, it visualizes the call graph and tag trace with open-source software GraphViz [11]; third it generates the HTML-formatted reports for call traces and tag traces (CSV format). The entire analysis process is illustrated in Figure 1.

The monitor component of Rkprofiler was built based on the open-source PC emulator QEMU. To support multiple CPU architectures, QEMU defines an intermediate instruction set. When QEMU is running, each instruction of a VM is translated to the intermediate instructions. Rkprofiler performs code inspection and analysis at the code translation stage. To improve the performance, QEMU caches the translated Translation Block (TB) so that it can be re-executed on the host CPU over time. However, this optimization approach is not desirable to Rkprofiler because an instruction can behave differently in varied machine states. For example, the instruction CALL, whose operand is a general-purpose register, may jump to diverse instructions depending on the value of that register. For each malicious TB that has been cached, Rkprofiler forces QEMU to always perform the code translation. But, the newly generated code is not stored in the cache and the existing cached code is actually executed. Another problem arises when a TB contains multiple instructions. In QEMU, VM states (register and memory contents) are not updated during the TB translation. Except for the first instruction, the translation of all other instructions in a TB could be accompanied by incorrect VM states, possibly resulting in analysis errors. Rkprofiler addresses this problem by making each malicious TB include only one instruction and disabling the direct block chaining for all malicious TBs.

4 Design and Implementation

Kernel malware could take the form of drivers and be legitimately loaded into the kernel. They can also be injected into the kernel by exploiting vulnerabilities of benign kernel software. Rkprofiler is designed to detect kernel malware that enter the kernel in both ways. Roughly speaking, before any malware is executed, Rkprofiler looks up the kernel memory image and identifies all benign kernel code in the VM. Then it groups them into a Trust Code Zone (TCZ) and a hash table is created to store the code addresses of the TCZ. When malware is started, any kernel code that does not belong to the TCZ is regarded as malicious and therefore is tracked by Rkprofiler.

Identification of the trusted kernel code is straightforward if the non-execute (NX) bit of the page table is supported by the (virtual) Memory Management Unit (MMU) of a (virtual) processor. In this case, the kernel code and data do not co-exist in any page of memory. Rkprofiler just needs to traverse the page table of a process to find out all the executable kernel pages. QEMU can provide a NX-bit enabled virtual processor (by enabling the PAE paging mechanism), but this system configuration is not common. Doing so may influence the malware behavior in an undesired manner. For example, the malware could stop running when it detects that the (virtual) CPU is NX enabled. So, the current implementation of Rkprofiler does not require enabling the NX-bit of the virtual CPU. Instead, it interprets all images of benign kernel modules and obtains the Relative Virtual Addresses (RVA) of the code sections. Then it computes their actual virtual addresses by adding the RVAs to the module base addresses, which is acquired by scanning the kernel memory of the VM. After that, Rkprofiler stores the TCZ addresses in a hash table. However, one common type of kernel malware attack is to patch the benign kernel code. To accommodate this type of attack, Rkprofiler excludes the patched code from the TCZ and revises the TCZ hash table at run time. Rkprofiler identifies the patched code by examining memory write operations and memory copy functions that the malware performs. Note, malware could escape this detection by indirectly modifying the TCZ code (e.g., tampering with kernel memory from user space). A more reliable method is to monitor the integrity of the TCZ as [20] does. Last, Rkprofiler determines whether a kernel TB is malicious or not right before it is translated. If the address of a TB is not within the TCZ, it is deemed as a malicious TB. The hash table implementation of the TCZ ensures that malicious code detection has a small performance hit on the entire system.

4.1 Malicious Code Detection

Kernel malware could take the form of drivers and be legitimately loaded into the kernel. They can also be injected into the kernel by exploiting vulnerabilities of benign kernel software. Rkprofiler is designed to detect kernel malware that enter the kernel in both ways. Roughly speaking, before any malware is executed, Rkprofiler looks up the kernel memory image and identifies all benign kernel code in the VM. Then it groups them into a Trust Code Zone (TCZ) and a hash table is created to store the code addresses of the TCZ. When malware is started, any kernel code that does not belong to the TCZ is regarded as malicious and therefore is tracked by Rkprofiler.

Identification of the trusted kernel code is straightforward if the non-execute (NX) bit of the page table is supported by the (virtual) Memory Management Unit (MMU) of a (virtual) processor. In this case, the kernel code and data do not co-exist in any page of memory. Rkprofiler just needs to traverse the page table of a process to find out all the executable kernel pages. QEMU can provide a NX-bit enabled virtual processor (by enabling the PAE paging mechanism), but this system configuration is not common. Doing so may influence the malware behavior in an undesired manner. For example, the malware could stop running when it detects that the (virtual) CPU is NX enabled. So, the current implementation of Rkprofiler does not require enabling the NX-bit of the virtual CPU. Instead, it interprets all images of benign kernel modules and obtains the Relative Virtual Addresses (RVA) of the code sections. Then it computes their actual virtual addresses by adding the RVAs to the module base addresses, which is acquired by scanning the kernel memory of the VM. After that, Rkprofiler stores the TCZ

addresses in a hash table. However, one common type of kernel malware attack is to patch the benign kernel code. To accommodate this type of attack, Rkprofiler excludes the patched code from the TCZ and revises the TCZ hash table at run time. Rkprofiler identifies the patched code by examining memory write operations and memory copy functions that the malware performs. Note, malware could escape this detection by indirectly modifying the TCZ code (e.g., tampering with kernel memory from user space). A more reliable method is to monitor the integrity of the TCZ as [17] does. Last, Rkprofiler determines whether a kernel TB is malicious or not right before it is translated. If the address of a TB is not within the TCZ, it is deemed as a malicious TB. The hash table implementation of the TCZ ensures that malicious code detection has a small performance hit on the entire system.

4.2 Function Call Tracking

Kernel malware often interacts with the rest of the kernel by calling functions exported by other kernel modules. In Rkprofiler, we use the terms I2E (Internal-to-External) and E2I (External-to-Internal) to describe the function-level control flow transferring between malicious code and benign code. Here, internal and external functions refer to the malicious function code and benign function code respectively. Function calls and returns are two types of events that Rkprofiler monitors. For example, *I2E call* indicates the event that an internal function invokes an external function; *I2E return* refers to the event that an internal function returns to its caller that is an external function. Capturing these function events is important for Rkprofiler to reveal the activity of the malware. Further, in an instance, the kernel malware may directly call the registry functions exported by ntoskrnl.exe like zwSetKeyValue to manipulate local registry entries. Rkprofiler is also designed to capture the I2I (Internal-to-Internal) call and return events. By doing so, Rkprofiler is able to construct (partial) call graphs of the kernel malware, which helps a security professional understand the code structure of the malware. This capability is important, especially when the malware is obfuscated to resist static code analysis. Note, E2E (External-to-External) function events are not monitored here because Rkprofiler does not inspect benign kernel code.

To completely monitor the function-level activity of malware, a data structure called *function descriptor* is defined to represent a stack frame (activation record) of a kernel call stack, allowing Rkprofiler to track the call stacks of the kernel malware. When a function that is called by malware is detected, Rkprofiler creates a new function descriptor object and pushes it to the stack. Conversely, when the function is returned, its function descriptor object is popped from the stack and is deleted. One function descriptor has a pointer that points to the function descriptor of the caller. This pointer is used by Rkprofiler to construct the caller-callee relationships in the post-analysis phase.

The method of detecting a function call event depends on the calling directions. For I2I and I2E calls, Rkprofiler monitors the CALL instructions executed by the malware. Further, it can obtain the function address from the operand of a CALL instruction and the return address that is next to the CALL instruction. For E2I calls, a CALL instruction belongs to TCZ and is not monitored by Rkprofiler. So, the detection point is moved to the first instruction of the callee function. To capture E2I calls, Rkprofiler adds extra data members to the TB descriptor *TranslationBlock*. The first data member indicates what the last instruction of this TB is: CALL, JMP, RET or others. If it is

a CALL instruction, the second data member records the return address of the call. Rkprofiler fills in the two data members of a TB when it is being translated. In addition, Rkprofiler creates a global pointer that points to the last TB descriptor whose code was just executed by the virtual CPU. Before translating a malicious TB, Rkprofiler queries the last TB descriptor to decide if it is an E2I call event. The decision is based on three criteria: 1) if the last TB is benign; 2) if the last instruction of the last TB is CALL; and 3) if the return address stored in the kernel stack is equal to the one stored in the last TB descriptor. The reason for criterion 3 is that the return address is always constant for both direct and indirect calls. On the other hand, Rkprofiler processes the function return events in a similar way to the call events: for I2I and E2I returns, Rkprofiler captures these events by directly monitoring the RET instructions executed by the malware; for I2E returns, Rkprofiler detects them at the instructions directly following the RET instructions and the criteria of the decision are similar to that for the E2I calls.

Two problems complicate the call event detection methods described above. The first one is a *pseudo function call*, which is caused by JMP instructions. When a kernel module attempts to invoke one function exported by another kernel module, it first executes the CALL instruction to invoke an internal stub function and the stub function then jumps to the external function by running the JMP instruction. Normally, the internal stub function is automatically generated by a compiler and the operand of the JMP function is an IAT entry of this module, whose value is determined and inserted by the system loader. Without recognition of these JMP instructions, Rkprofiler incorrectly treats an I2E call as an I2I call: labeling the new function descriptor with the internal stub function address. One example of such functions is DbgPrint. To address a pseudo function call, Rkprofiler first creates an I2I function descriptor and labels it with the internal stub function address. When detecting if an internal JMP instruction is executed in order to jump to an external address, Rkprofiler locates the I2I function descriptor from the top of the function tracking stack, and replaces the internal address with the external address. The second problem is an *interrupt gap*. This is where an interrupt is sent to the (virtual) CPU while it is executing an E2I CALL (or I2E RET) instruction. Consequently, some interrupt handling instructions are executed between the E2I CALL (or I2E RET) instruction and the subsequent internal instruction that Rkprofiler monitors. In this situation, the last TB descriptor does not record the expected CALL (or RET) instruction, so Rkprofiler is unable to track the E2I call (or I2E return) event and observes an unpaired return-call event. The solution to this problem is part of our future work. Fortunately, we did not see interrupt gaps in the experiments.

4.3 Memory Access Tracking

Rkprofiler observes the hardware-level activity of kernel malware, however it should be translated to software-level activity to be understandable to users. Thus, given a virtual address that the malware visits, Rkprofiler is required to find its symbols (e.g., variable name and type). In this paper, we name the process of finding symbols for kernel objects as memory tagging. A memory tag is composed of tag id, virtual address, type ID, variable name (optional) and parent tag id (optional). If a kernel object is owned by the malware, it is an internal kernel object; otherwise, it is an external kernel object. If a kernel object is located in the dynamic memory area (stack and heap), it

is a dynamic kernel object; otherwise, it is a static kernel object. Rkprofiler tags four types of kernel objects: static internal, dynamic internal, static external and dynamic external. Static external kernel objects include global variables and Windows kernel functions. Their symbols are stored in a system map. Tagging a static kernel object is straightforward. Rkprofiler searches the system map by its virtual address and the hit entry contains the target symbols. However, tagging a dynamic kernel object is challenging because its memory is dynamically allocated at run time and the memory address cannot be predicted. Attackers often strip off the symbols of their malware in order to delay reverse engineering, so Rkprofiler assumes that malware samples do not contain valid symbols.

Previous Linux rootkit detection systems [19][4] present one approach of tracking dynamic kernel objects. A rootkit detector first generates a kernel type graph and identifies a group of global kernel variables. At run time, it periodically retrieves the dynamic objects from the global variables based on the graph type. For example, if a global variable is a linked list head, the detector traverses the list under the direction of the data structure type of list elements. Unfortunately, this approach cannot be applied to the task of profiling kernel malware. First, it covers a limited number of kernel objects, and many other kernel objects such as functions and local variables are not included. Second, since the creation and deletion of dynamic kernel objects could occur at any time, the time gap between every two searches in this approach will produce inaccurate monitoring results. Last, this approach may track many kernel objects that the malware never visits. In this paper, we propose a new symbol exploration approach, *Aggressive Memory Tagging* (AMT), that can precisely find symbols for all kinds of static and dynamic kernel objects at a low computation cost.

AMT Description We define a kernel object as *contagious* if another kernel object can be derived from it. *Tag inferring* is a process where a kernel object (child object) is derived from another (parent object). Two types of kernel objects are considered contagious: pointers and functions. A pointer kernel object could be a pointer variable or a structure variable containing a pointer member. The child object of a pointer is the pointee object. For a function, its child objects are the parameters and return value of this function. AMT follows the principle of the object tracking approach described above: tracing the dynamic objects from the static objects. Specifically, Rkprofiler first tags all static kernel objects that the malware accesses (memory reads/writes and function calls) by querying the system map. Then, the child objects of the existing contagious tags are tagged via tag inferring. This process is repeated until the malware stops execution or the user terminates monitoring. Note, a tag could become invalid in two scenarios: 1) if when a function returns, the tags of its local variables are invalidated; and 2) if a memory buffer is released, the associated tag becomes out of date as well. Only valid tags can generate valid child tags.

Rkprofiler performs tag inferring through a pointer object at the time that the malware reads or writes the pointer object. The reason is as follows: when reading a pointer, the malware is likely to visit the pointee object through the pointer; when writing a pointer, the malware will possibly modify the pointer to point to another object if the new value is a valid memory address. Because the executions of benign kernel code are not monitored by Rkprofiler, both read and write operations over a pointer have to be tracked here. If only read operations are monitored, Rkprofiler cannot identify the kernel objects whose pointers are written by malicious code and

read by benign code. Many hooks implanted by rootkits fall into this scenario. Similarly, if only write operations are monitored, Rkprofiler can miss the reorganization of kernel objects whose pointers are written by benign code and read by malicious code. Many external kernel objects that are visited by rootkits fall into this scenario. The procedure of tag inferring through a pointer object is as follows: 1) Rkprofiler detects a memory read or write operation and searches the tag queue to check if the target memory corresponds to a contagious tag; 2) if yes, Rkprofiler obtains the up-to-date pointer value and verifies that it is a valid memory address; 3) Rkprofiler searches the tag queue to check if the pointee object is tagged; 4) if not, Rkprofiler obtains the symbols of the pointee object from the type graph and creates a new tag. On the other hand, when a recognizable function is called, tag inferring through the function object is carried out by identifying the function parameters. Input parameters are tagged when the function is called; output parameters are tagged when the function returns.

Implementation Rkprofiler creates a data structure called *tag descriptor* to represent memory tags. A tag descriptor includes the virtual address of the tag, type ID, a boolean variable, a num variable for memory type, one pointer to the parent tag and one pointer to the function descriptor. The Boolean variable indicates if a tag is contagious or not. The memory type member tells if the tagged object is on the stack, heap or another memory object. Rkprofiler monitors the kernel memory management functions called by malware and records it to a heap list (the memory buffers allocated to the malware). When a buffer is released, Rkprofiler removes it from the heap list. The function descriptor member of a tag helps identify which function is running when this tag is generated. Finally, Rkprofiler maintains a tag queue that contains all the tags that have been created. When a tag is created, its tag descriptor is inserted into the tag queue. The tag is removed from the tag queue after it becomes invalid. Because malware's memory accesses are frequent events, Rkprofiler needs to search the tag queue frequently as well. The tag queue describes a group of various-sized memory segments. If it is organized as a list structure like a linked list, its linear searching time is expensive. To address the problem, Rkprofiler applies the approach presented in [29] that converts a group of various-sized memory segments to a hash table. The basic idea is to break a memory segment into a number of fix-sized memory segments (buckets). A list structure is stored in one bucket to handle the case that some portions of the bucket should not be counted. In this way, the time for searching the tag queue becomes constant.

The Windows kernel provides built-in supports for linked lists via two data structures: `SINGLE_LIST_ENTRY` (for single linked list) and `LIST_ENTRY` (for double linked list). Several kernel APIs are available to simplify driver developers' tasks when managing linked lists (e.g., adding or removing elements). However, this support causes problems to the memory tagging process of Rkprofiler. For example, in a double linked list, each element contains a data member whose data type is `LIST_ENTRY`. Two pointers of this data member point to the `LIST_ENTRY` data members of two neighbor elements. When one list element is tagged and malware tries to visit the next list element from this one, Rkprofiler just tags the `LIST_ENTRY` data member of the next list element with the type `LIST_ENTRY`. This is not acceptable because what Rkprofiler wants to tag is the next list element with its type. In the pre-analysis stage, we annotated the `SINGLE_LIST_ENTRY` and `LIST_ENTRY` data members with the type names of list elements and their offsets. When parsing the type header file, the generator replaces

the SINGLE_LIST_ENTRY and LIST_ENTRY data members with pointers to list elements. The offset values are also stored in the type graph, allowing the monitor to find the actual addresses of neighbor elements. Another problem is relative pointers. The Windows kernel sometimes uses relative pointers to traverse a list in the following way: the address of the next element is computed by adding the relative pointer and the address of the current element. One example is the data buffer that contains the disk file query result by kernel function NtQueryDirectoryFile. Because these relative pointers are defined as unsigned integer, we also need to label the relative pointers in the kernel type header file such that Rkprofiler can recognize them and properly compute the element addresses.

Rkprofiler has to handle two ambiguous data types that the Windows kernel source uses. The first one is *union*. Union is a data type that contains only one of several alternative members at any given time, and the memory storage required for a union is decided by its largest data member. Unfortunately, guessing which data member of a union should be used at a given time depends on code context, which is hard to automate in Rkprofiler. The second one is generic pointer *pvoid*. Pvoid can be caste to another data type by developers. The actual data type that pvoid points to at a given time is context dependent too. Automatically predicting the pointee data type for pvoid is another challenge. The current default solution is to replace a union with one of its largest members and leave pvoid alone. While performing the analysis, a user can modify the kernel data type header file and change the definition of union or pvoid in terms of his understanding of their running contexts. An automated solution to this problem is part of our future work.

4.4 Hardware Access Monitoring

In comparison to user-space malware, kernel malware is able to bypass the mediation of the OS and directly access low-level hardware resources. In X86 architectures, in addition to the memory and general-purpose registers that kernel malware access through instructions like MOV and LEA, other types of system storage resources could also be visited and manipulated by kernel malware. CPU caches (e.g., TLB) dedicate registers and buffers of I/O controllers. Attackers have developed techniques that take advantage of these hardware resources to devise new attacks. For example, upon a system service (system call) invocation made by a user-space process, Windows XP uses instruction SYSENTER (for Intel processor) to perform the fast transition from user space to kernel space. The entry point of kernel code (a stub function) is stored in a dedicated register called IA32_SYSENTER_EIP, which is one of Model-Specific Registers (MSRs). When executing SYSENTER, the CPU sets the EIP register with the value of IA32_SYSENTER_EIP. Then, the kernel stub function is called and it transfers the control to the target system service. To compromise Windows system services, a rootkit could alter the system control-flow path by resetting the IA32_SYSENTER_EIP to the starting address of a malicious stub function, and this function can invoke a malicious system service. So, capturing the malware's accesses to these sensitive hardware resources could be essential to comprehend its attacking behavior. Currently, Rkprofiler monitors twenty system instructions that malware might execute. They are not meant to be complete at this point and can be expanded in the future if necessary.

5 Case Studies

5.1 FUTo

FUTo is an enhanced version of the Windows kernel rootkit FU, which uses the technique called Direct Kernel Object Manipulation (DKOM) to hide processes and drivers and change the process privileges. DKOM allows rootkits to directly manipulate kernel objects, avoiding the use of kernel hooks to intercept events that access these kernel objects. For example, a rootkit can delete an item from the `MODULE_ENTRY` list to hide a device driver without affecting the execution of the system. This technique has been applied to many rootkit attacks, such as hiding processes, drivers and communication ports, elevating privilege levels of threads or processes and skewing forensics [12]. In this experiment, FUTo was downloaded from [21] and it included one driver (msdirectx.sys) and one executable (fu.exe). The fu.exe was a command-line application that installed the driver and sent commands to the driver according to the user's instructions. During the test, we executed the fu.exe to accomplish the following tasks: querying the command options, hiding the driver (msdirect.sys) and hiding the process (cmd.exe). After that, we used Windows native system utilities (task manager and driverquery) to verify that the target driver and process did not show up in their reports. The test took less than 3 minutes.

We compared the call graph created by Rkprofiler with the call graph created by IDA-Pro (which uses the static code analysis technique). It was found that the former was the sub-graph of the latter, which is as expected. The tag trace graph of this test is shown in Table 1. The driver msdirectx was executed in four process contexts in the graph. Process 4 (System) is the Windows native process that was responsible for loading the driver misdirectx. The driver initialization routine (with `tag_id` 0) was executed in this process context. The other three processes were associated with FUTo.exe and they communicated with the misdirectx driver to perform the tasks of hiding the driver and process. One important observation is that the major attacking activities have been recorded by Rkprofiler and can be easily identified in the tag trace table by users. To hide itself, the driver msdirectx first reads the address of its module descriptor (with `tag_id` 9) from its driver object (with `tag_id` 1). Then it removes this module descriptor from the kernel `MODULE_ENTRY` list by modifying the Flink and Blink pointers in two neighbor module descriptors (`tag_id` 15 and 16). Similarly, to conceal process cmd.exe, msdirectx first obtains the process descriptor (with `tag_id` 2) of the current process by calling kernel function IoGetCurrentProcess. Starting from this process descriptor, msdirectx traverses the kernel EPROCESS list to find the process descriptor (with `tag_id` 4) of process csrss.exe. These two steps take place in the System process context. After receiving the command for hiding the cmd.exe process sent by one of the fu.exe processes, msdiretx searches the kernel EPROCESS list, beginning with the process descriptor of csrss.exe. When the process descriptor (with `tag_id` 32) of cmd.exe is found, msdirectx removes it from the kernel EPROCESS list by altering Flink and Blink pointers in two neighbor process descriptors (with `tag_id` 31 and 33). Furthermore, Flink and Blink pointers in the process descriptor of cmd.exe are also modified to prevent the random Blue Screen of Death (BSOD) when exiting the hidden process. To evade the detection of rootkit detectors, FUTo deletes the hidden process from the other three kernel structures: kernel handle table list, handle table of the process csrss.exe and PspCidTable. The first one is a linked list, and the DKOM behavior of FUTo over this kernel structure was captured and displayed in the tag trace

Table 1. FUTO tag trace table

Tag ID	Address	Type	Parent	Category	Size(bytes)	Process ID	Process Name
0	0xf6b7e7e6	FUNCT_0049_0953_DriverInit	n/a	function	n/a	4	System
1	0x825c3978	DRIVER_OBJECT	n/a	struct	168	4	System
2	0x827cba00	EPROCESS	n/a	struct	608	4	System
3	0x825991c8	EPROCESS	2	struct	608	4	System
4	0x825ce020	EPROCESS	3	struct	608	4	System
5	0xf7b0ec58	PVOID	n/a	pointer	4	4	System
6	0xf6b8b92	PDEVICE_OBJECT	n/a	pointer	4	4	System
7	0xf6b7e722	FUNCT_0049_095B_MajorFunction	1	function	n/a	4	System
8	0xf6b7d43a	FUNCT_00BC_0957_DriverUnload	1	function	n/a	4	System
9	0x8264600	MODULE_ENTRY	1	struct	52	4	System
10	0x82609f18	DEVICE_OBJECT	n/a	struct	184	1920	fu.exe
11	0x8266fc28	IRP	n/a	struct	112	1920	fu.exe
12	0x8266fc03	IRP	n/a	struct	112	1920	fu.exe
13	0x826bc118	IRP	n/a	struct	112	1952	fu.exe
14	0x826bc103	IRP	n/a	struct	112	1952	fu.exe
15	0x826d8288	MODULE_ENTRY	9	struct	52	1952	fu.exe
16	0x8055ab20	MODULE_ENTRY	9	struct	52	1952	fu.exe
17	0x826bc210	IRP	n/a	struct	112	1952	fu.exe
18	0x825d1020	EPROCESS	4	struct	608	1880	fu.exe
19	0x8273a7c8	EPROCESS	18	struct	608	1880	fu.exe
20	0x826eb408	EPROCESS	19	struct	608	1880	fu.exe
21	0x825d5a80	EPROCESS	20	struct	608	1880	fu.exe
22	x825e4da0	EPROCESS	21	struct	608	1880	fu.exe
23	0x825a9668	EPROCESS	22	struct	608	1880	fu.exe
24	0x82695180	EPROCESS	23	struct	608	1880	fu.exe
25	0x825a0da0	EPROCESS	24	struct	608	1880	fu.exe
26	0x82722980	EPROCESS	25	struct	608	1880	fu.exe
27	0x825c27e0	EPROCESS	26	struct	608	1880	fu.exe
28	0x82624bb8	EPROCESS	27	struct	608	1880	fu.exe
29	0x825de980	EPROCESS	28	struct	608	1880	fu.exe
30	0x8248bda0	EPROCESS	29	struct	608	1880	fu.exe
31	0x8264a928	EPROCESS	30	struct	608	1880	fu.exe
32	0x8263a5a8	EPROCESS	31	struct	608	1880	fu.exe
33	0x825d9020	EPROCESS	32	struct	608	1880	fu.exe
34	0xe13ed7b0	HANDLE_TABLE	4	struct	68	1880	fu.exe
35	0x82607d48	ETHREAD	32	struct	600	1880	fu.exe
36	0xe15ca640	HANDLE_TABLE	32	struct	68	1880	fu.exe
37	0xe10a8a08	HANDLE_TABLE	36	struct	68	1880	fu.exe
38	0xe1747cd0	HANDLE_TABLE	36	struct	68	1880	fu.exe

graph too (see `tag_id` 36, 37 and 38). The last two kernel structures are implemented as three-dimensional arrays, which is not supported by the current version of Rkprofiler. So, the tag trace graph does not include the modification of these two kernel structures.

Combining Rkprofiler's output with other reports, we discovered other interesting behavior of FUTo. First, FUTo employed an IOCTL mechanism to pass control commands from user space to kernel space. During the driver initialization, a device `\\Device\\msdirectx` was created by calling the kernel function IoCreateDevice. Then a dispatch function (data type `FUNCT_0049_095B_Majorfunction` and `tag_id` 7) was registered to the driver object (with `tag_id` 1) that was assigned to msdirectx by the Windows kernel. This dispatch function was invoked by the kernel I/O manager to process I/O requests issued by the fu.exe processes. By checking the parameters of this dispatch function, we found that the I/O control codes for process and driver concealment tasks are 0x2a7b2008 and 0x2a7b2020. Second, the kernel string function strncmp was called 373 times by one msdirectx function, implying a brute-force searching operation. The first parameter of this function was constant string "System" and the second parameter was 6 bytes of data within the process descriptor of the process System (with `tag_id` 2). Beginning with the address of the process descriptor, the address of the second parameter was increased by one byte each time this string function was called. The purpose of the search was to find the offset of the process name in the EPROCESS structure. This was confirmed by manually checking the FUTo source. It seems that the definition of EPROCESS structure has changed over the Windows versions and the brute-force searching allows FUTo to work with different Windows versions.

5.2 TCPIRPHOOK

Inserting hooks into the kernel to tamper with the kernel control-flow path is one major technique that attackers apply to rootkit attacks. A hooked function can intercept and manipulate kernel data to serve its malicious aims. TCPIRPHOOK is one such rootkit and it intends to hide the TCP connections from local users. Specifically, this rookit exploits the dispatch function table of the TCP/IP driver object (associated with driver TCPIP.sys) and substitutes a dispatch function with its hook. The hooked function registers another hook to the I/O request packets (IRP) such that the second hook can intercept and modify the query results for network connections. We downloaded the rootkit package from [21] which also included one driver file, irphook.sys. The rootkit was implemented to conceal all http connections (with destination port 80). Before installing the rootkit, we opened Internet Explorer to visit a few websites, and then ran the netstat utility to display the corresponding http connections. We loaded the irphook.sys to the kernel and used netstat to verify that all https connections were gone. In the end, we unloaded the irphook.sys. The test took less than 3 minutes.

The call graph of TCPIRPHOOK is shown in Figure 2. Function 0xf7ab8132 (irphook.sys) was the first hook that was inserted into the 14th entry (IRP_MJ_DEVICE_CONTROL,) of the dispatch function table in the driver TCPIP.sys. The replaced dispatch function was TCPDispatch (address 0xf726fddf) owned by driver TCPIP.sys. The first hook invoked TCPDispatch 15 times in the call graph. In fact, it is common for rootkits to call the original function in a hook, which reduces the coding complexity of the hook. Function 0xfa7b8000 (irphook.sys) was the second hook that was responsible for modifying the query results for network connections. Although the second hook seems to

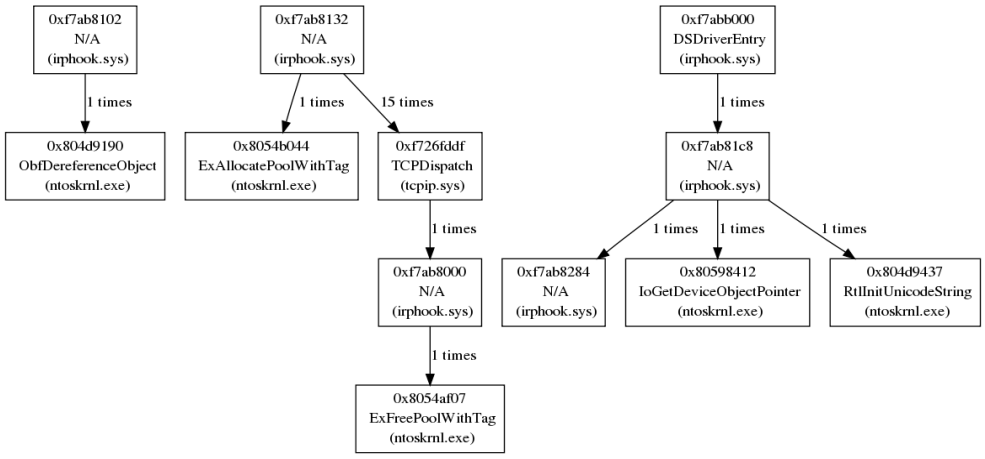

Fig. 2. TCPIRPHOOK call graph

be called by TCPDispatch in the call graph, the actual direct caller of the second hook was IopfCompleteRequest (ntoskrnl.exe). This is because Rkprofiler did not track the benign kernel code and had no knowledge of their call stacks. On the other hand, even the indirect caller-callee relation between TCPDisptch and the second hook can imply that the network connection query caused synchronous IRP processing and completion in the kernel, which is comparable to asynchronous IRP processing and completion. But this information cannot be inferred by simply looking at the IDA-pro's call graph, because IDA-pro cannot statically determine the symbol of function TCPDispatch and the calling path from the first hook to the second hook in Figure 3 is not presented in the IDA-pro's call graph.

Figure 3 is the tag trace graph of TCPIRPHOOK. Two hooking activities are illustrated in this graph. The first hook was installed at the driver loading stage. To hook the dispatch function table of the driver TCPIP.sys, TCPIRPHOOK first calls the kernel function IoGetDeviceObjectPointer with the device name `\\Device\\Tcp` to get the pointer (with `tag_id` 7) to the device object (with `tag_id` 8) owned by driver TCPIP.sys. Then, the device object was visited to get the address of the driver object (with `tag_id` 9) owned by driver TCPIP.sys. Last, TCPIRPHOOK carried out the hooking by accessing the 14th entry of the dispatch function table in the driver object: reading the address of the original dispatch function (with `tag_id` 10) and storing it to a global variable; writing the address of the second hook (with `tag_id` 11) to the table entry. The second hook was dynamically installed in the context of process netstat.exe. When netstat.exe was executed to query TCP connection status, the Windows kernel I/O manager created an IRP (with `tag_id` 12) for the netstat.exe process. This IRP was passed to the first hook (`function_id` 5 and `tag_id` 11) of TCPIRPHOOK. The first hook obtained the `IO_STACK_LOCATION` object (with `tag_id` 13) from this IRP and wrote the address of the second hook (with `tag_id` 14) to the data member CompletionRoutine of the `IO_STACK_LOCATION` object. Thus, being one IRP completion function, the second hook would be called by the Windows kernel to process the I/O return data for this IRP. Last, the tag trace graph also captures the manipulation of the I/O return data. The buffer of the I/O return data was pointed to by the data member

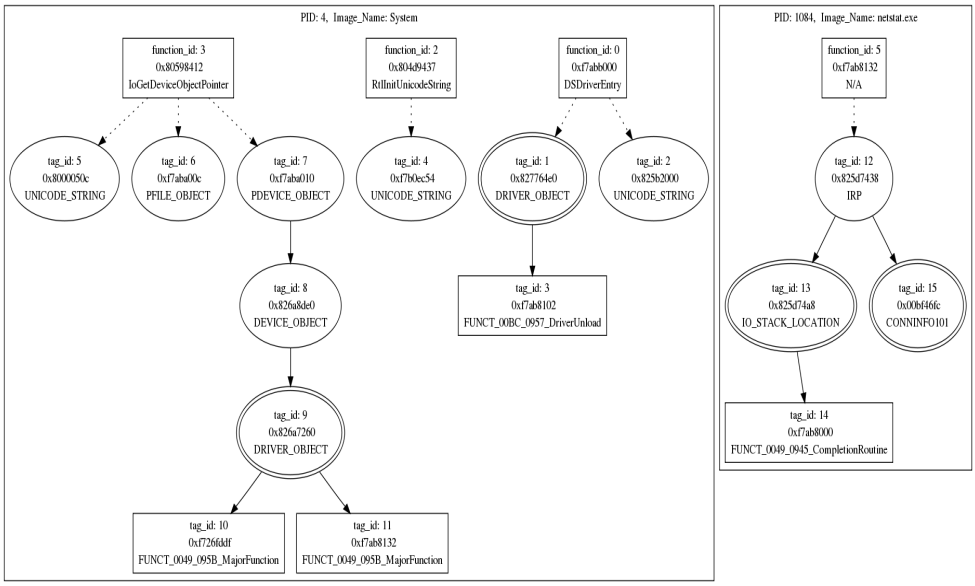

Fig. 3. TCPIRPHOOK tag trace graph

UserBuffer of IRP and it was an array of structure CONNINF101 (with `tag_id` 15). The size of the buffer was stored in the data member IoStatus.Information of the IRP. Clearly, the `tag_id` 15 was modified in the tag trace graph. By examining the tag trace table, we found that the status of all http connections in the buffer were changed from 5 to 0.

5.3 Rustock.B

Rustock.B is a notorious backdoor rootkit that hides malicious activities on a compromised machine. The distinguished feature of this rootkit is the usage of multi-layered code packing, which makes static analysis cumbersome [7]. Unlike the other two rookits described above, we did not have access to the source code of this rootkit. However, several analysis results on this rootkit published on the Internet helped us understand some behaviors of this rootkit. We downloaded Rustock.B from [20] as one executable. During the test, we just double-clicked the binary and waited until the size of the Rkprofiler log stop being populated. The test lasted about five minutes.

A malicious driver named system32:lzx32:sys was detected by Rkprofiler. 90857 calls and 2936 tags were captured in the test. The driver contained self-modifiying code and we found many RET instructions that did not have corresponding CALL instructions at code unpacking stages. This is because unpacking routines executed JMP instructions to transfer the controls to the intermediate or unpacked code. In addition, the driver modified the dedicated register `IA32_SYSENTER_EIP` through WRMSR and RDMSR instructions to hijack the Windows System Service Descriptor Table (SSDT). One hook was added to the dispatch function table of driver Ntfs.sys to replace the original `IRP_MJ_CREATE` dispatch function. This is similar to what TCPIRPHOOK does. We

compared the report generated by Rkprofiler with others on the Internet and they matched each other well. Table 2 lists the external functions and registry keys that were called and created by Rustock.B. Unfortunately, the full report of this test cannot be presented due to the space constraint.

Table 2. External functions and registry keys manipulated by Rustock.B

External Functions	ExAllocatePoolWithTag, ExFreePoolWithTag, ExInitializeNPagedLookasideList, IoAllocateMdl, IoGetCurrentProcess, IoGetDeviceObjectPointer, IoGetRelatedDeviceObject, KeClearEvent, KeDelayExecutionThread, KeEnterCriticalRegion, KeInitializeApc, KeInitializeEvent, KeInitializeMutex, KeInitializeSpinLock, KeInsertQueueApc, KeLeaveCriticalRegion, KeWaitForSingleObject, MmBuildMdlForNonPagedPool, MmMapLockedPages, MmProbeAndLockPages, NtSetInformationProcess, ObfDereferenceObject, ObReferenceObjectByHandle, ProbeForRead, PsCreateSystemThread, PsLookupProcessByProcessId, PsLookupThreadByThreadId, RtlInitUnicodeString, `_stricmp`, `_strnicmp`, swprintf, wcschr, wcscpy, `_wcsicmp`, `_wcslwr`, wcsncpy, `_wcsnicmp`, wcstombs, ZwClose, ZwCreateEvent, ZwCreateFile, ZwDeleteKey, ZwEnumerateKey, ZwOpenKey, ZwQueryInformationFile, ZwQueryInformationProcess, ZwQuerySystemInformation, ZwReadFile
Registry Keys	`HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\pe386` `HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_pe386`

6 Discussion

In addition to the incomplete kernel symbols provided by Microsoft, the current implementation of Rkprofiler suffers several other limitations that could be exploited by attackers to evade the inspection. First, attackers may compromise the kernel without running any malicious kernel code, e.g., directly modifying kernel data objects from user space or launching return-to-lib attacks without the use of any function calls [24]. Rkprofiler is not able to detect and profile such attacks. Instead, other defense approaches like control flow integrity enforcement [2] could be adopted to address them. Second, the instruction pair CALL/RET is used as the sole indicator of function call and return events. attackers can obfuscate these function activities to escape the monitoring. For example, JMP/JMP, CALL/JMP and JMP/RET can be employed to implement the function call and return events. Moreover, instead of jumping to a target instruction (either the first instruction of a callee function or the returned instruction of a caller function), a attacker could craft the code to jump to one of its neighbor instructions, while preserving the software logic intact. Defending against such attacks is part of our future work. Third, a attacker may deter the AMT method by accessing

dynamic objects in unconventional ways. For example, a rootkit can scan the stack of a benign kernel function to get the pointer to a desired kernel object. These attacks are very challenging, because building an accurate and up-to-date symbol table for all kernel objects is impractical. Last, malware may have the capability of detecting virtual machine environments and change their behavior accordingly. Exploring multiple execution paths [16] and static analysis could mitigate this problem to some extent.

7 Related Work

Many previous works have focused on run time rootkit detection [19][4][17][18][28] and prevention [29][22][25]. The main purpose of these mechanisms is to protect data and code integrity of the guest OS at run time. On the other hand, researchers have also applied program analysis techniques to create offline rootkit defense mechanisms with goals such as rootkit identification, hook detection and so on. Several works that fall into this category are discussed below.

Rootkit Identification: Kruegel [13] proposed a system that performs static analysis of rootkits using symbolic execution, a technique that simulates program execution with symbols. This system can only detect known rootkits. Moreover, anti-static-analysis techniques like code obfuscation can be used to defeat this system. Limbo [26] is another rootkit analysis system that loads a suspicious driver into a PC emulator and uses flood emulation to explore multiple running paths of the driver. Limbo has a low false positive rate, but it performs poorly when detecting unknown rootkits. Also, flood emulation makes rootkits behave abnormally in the emulator, possibly resulting in inaccurate detection. Panorama [31] uses dynamic taint analysis to detect privacy-breaching rootkits. Sensitive system data like keys and packets are tainted and system-wide taint propagation is tracked. A taint graph is generated to tell whether a target rootkit accesses the tainted data or not. Although this system is good at capturing data-theft rootkits, it cannot provide necessary behavior information (e.g., kernel hooking) associated with other types of rootkits.

Hook Detection: HookFinder [30] and HookMap [27] aim to identify the hooking behavior of rootkits. HookFinder performs dynamic taint analysis and allows users to observe if one of the impacts (tainted data) is used to redirect the system execution into the malicious code. On the other hand, HookMap is intended to identify all potential hooks on the kernel-side execution paths of testing programs such as ls and netstat. Dynamic slicing is employed to identify all memory locations that can be altered to diverted kernel control flow. Unfortunately, hooking is only one aspect of rootkit behavior and both systems cannot provide comprehensive a view of rootkit activities in a compromised system.

Discovery of Sensitive Kernel Data Manipulation: K-tracer [14] is a rootkit analysis system that automatically discovers the kernel data manipulation behaviors of rootkits including sensitive data access, modification and triggers. K-tracer performs data slicing and chopping on sensitive data in the rootkit trace and identifies the data manipulation behaviors. K-tracer cannot detect hooking behaviors of rootkits and is unable to deal with DKOM and hardware-based rootkits. In comparison, Rkprofiler can handle a broad range of rootkits, including DKOM and hardware-based rootkits, and provide a complete picture of rootkit activities in a compromised system.

Rootkit Profiling: PoKeR [23] is a QEMU-based analysis system that shares the same design goal as Rkprofiler: comprehensively revealing rootkit behavior. PoKeR is

capable of producing rootkit traces in four aspects: hooking behavior, target kernel objects, user-level impact and injected code. Similar to Rkprofier, PoKeR infers the dynamic kernel object starting from the static kernel objects. However, PoKeR only tracks the pointer-based object propagation, while Rkprofiler tracks both pointer-based and function-based object propagation. So Rkprofiler can identify more kernel objects than PoKeR. In addition, the function call and hardware access monitoring features of Rkprofiler are not offered by PoKeR.

8 Conclusion

In this paper, we present a sandbox-based rootkit analysis system that monitors and reports rootkit behavior in a guest OS. The evaluation results demonstrate the effectiveness of this system in revealing rootkit behavior. However, to strengthen the current implementation of Rkprofiler, we need OS vendors to provide the unpublished symbols, some of which may have been reversely engineered by attackers.

References

1. Anubis Project. http://anubis.iseclab.org/?action=home, 2009.
2. M. Abadi, M. Budiu, U. Erlingsson, and j. Ligatti. Control-flow integiryt: Principles, implementations, and applications. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2005.
3. BitBlaze Project. http://bitblaze.cs.berkeley.edu/, 2009.
4. A. Baliga, V. Ganapathy and L. Iftode. Automatic Inference and Enforcement of Kernel Data Structure Invariants. . In Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC), 2008.
5. F. Bellard. QEMU and Kqemu. http://fabrice.bellard.free.fr/qemu/, 2009.
6. CBS News. Conficker Wakes Up. http://www.cbsnews.com/stories/2009/04/09/tech/cnettechnews/main4931360.shtml, 2009.
7. K. Chiang and L. Lloyd. A case Study of the Rustock Rootkit and Spam Bot. In First workshop on hot topics in understanding botnets, 2007.
8. Dr.Web Company. Win32.Ntldrbot (aka Rustock.C) no longer a myth, no longer a threat. New Dr.Web scanner detects and cures it for real. http://info.drweb.com/show/3342/en, 2009.
9. T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In Proceedings of the Symposium on Network and Distributed System Security (NDSS), 2003.
10. Geeg Blog. The Conficker Worm Awakens. http://geeg.info/blog4.php/2009/04/the-conficker-worm-awakens, 2009.
11. GraphViz Project. http://www.graphviz.org/, 2009.
12. G. Hoglund, J. Butler. Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, August 2005.
13. B. C. Kruegel, W. Robertson and G. Vigna. Detecting Kernel-Level Rootkits through Binary Analysis. In Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC), 2004.
14. A. Lanzi, M. Sharif, W. Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In Proceeding of the Annual Network and distributed System Security Symposium (NDSS), 2009.
15. Microsoft Symbol Server. http://msdl.microsoft.com/download/symbols, 2009.

16. A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In proceedings of the IEEE Symposium on Security and Privacy, 2007.
17. N. L. Petroni, T. Fraser, J. Molinz, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In Proceedings of the USENIX Security Symposium, 2004.
18. N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In Proceedings of the USENIX Security Symposium, 2006.
19. N. L. Petroni, M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2007.
20. Offensivecomputing Website. http://www.offensivecomputing.net/, 2009.
21. Rootkit website. http://www.rootkit.com, 2009.
22. R. Riley, X. Jiang and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID), 2008.
23. R. Riley, X. Jiang, D. Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In Proceedings of the ACM SIGOPS European Conferecne on Computer Systems (EuroSys), 2009.
24. H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2007.
25. A. Seshadri, M. Luk, N. Qu and A. Perrig. SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2007.
26. J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID), 2007.
27. Z. Wang, X. Jiang, W. Cui, X. Wang. Countering Persistent Kernel Rootkits Through systematic Hook Discovery. In Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID), 2008.
28. X. Jiang, X. Wang, D. Xu. Stealthy Malware Detection through VMM-Based "Out-of-the-Box" Semantic View Recontruction. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2007.
29. C. Xuan, J. Copeland and R. Beyah. Shepherding Loadable Kernel Modules through On-demand Emulation. In Proceedings of the conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2009.
30. H. Yin, Z. Liang, and D.Song. Hookfinder: Identifying and understanding malware hooking behaviors. In Proceeding of the Annual Network and distributed System Security Symposium (NDSS), 2008.
31. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Captureing System-wide Information Flow for Malware Detection and Analysis. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2007.