Multi-Aspect Profiling of Kernel Rootkit Behavior

Ryan Riley
Purdue University
rileyrd@cs.purdue.edu

Xuxian Jiang
North Carolina State University
jiang@cs.ncsu.edu

Dongyan Xu
Purdue University
dxu@cs.purdue.edu

Abstract

Kernel rootkits, malicious software designed to compromise a running operating system kernel, are difficult to analyze and profile due to their elusive nature, the variety and complexity of their behavior, and the privilege level at which they run. However, a comprehensive kernel rootkit profile that reveals key aspects of the rootkit's behavior is helpful in aiding a detailed manual analysis by a human expert. In this paper we present PoKeR, a kernel rootkit profiler capable of producing multi-aspect rootkit profiles which include the revelation of rootkit hooking behavior, the exposure of targeted kernel objects (both static and dynamic), assessment of user-level impacts, as well as the extraction of kernel rootkit code. The system is designed to be deployed in scenarios which can tolerate high overheads, such as honeypots. Our evaluation results with a number of real-world kernel rootkits show that PoKeR is able to accurately profile a variety of rootkits ranging from traditional ones with system call hooking to more advanced ones with direct kernel object manipulation. The obtained profiles lead to unique insights into the rootkits' characteristics and demonstrate PoKeR's usefulness as a tool for rootkit investigators.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Invasive software

General Terms Security

Keywords Kernel Rootkit, Malware, Profiling

1. Introduction

Targeting operating system (OS) kernels, kernel rootkits are considered one of the most stealthy types of computer malware and pose a significant threat to the integrity of computer systems. They run at the highest level of privilege within the victim machine, hijack control of the OS kernel, and provide "value-added" services to allow other malicious activities or unauthorized accesses to occur – all hidden from the system administrator and users. For example, kernel rootkits have been used to hide bot programs or other backdoor software with the intention of maximizing the life time of a botnet.

Despite recent research efforts in kernel rootkit detection [Garfinkel 2003, Petroni 2004; 2006; 2007] and kernel rootkit prevention [Seshadri 2007, Riley 2008], less attention has been given to *kernel rootkit profiling*– the revelation of key aspects of a kernel rootkit's behavior. It is further desirable that such profiles be generated on-the-fly in "live" systems such as honeypots. Kernel rootkit profiles are valuable in the design of effective solutions to kernel rootkit detection, damage mitigation, and kernel integrity protection. In this paper, we define a kernel rootkit profile as be comprised of the following four aspects:

- *Hooking behavior*: the way the kernel rootkit hijacks the kernel's control flow, if any, during the rootkit's installation. Typically, such hijacking is done by modifying hooks (e.g., function pointers) in the kernel. Note that it is not uncommon for rootkits to install hooks within various kernel objects, including kernel code or dynamically allocated kernel objects [Hoglund 2006].
- *Targeted kernel objects*: the kernel objects accessed by the rootkit, such as those read or modified by the rootkit. Similar to hooking behavior, a targeted kernel object may be dynamic. A classic example is the *all-task* list, maintained by the OS kernel for accounting purposes but often manipulated by rootkits for hiding purposes.
- *User-level impacts*: the affected user-level applications whose execution may be directly affected by the execution of rootkit code. Note that we do not aim to derive a complete list of affected applications. Instead, we focus on a corpus of commonly-used system utilities (e.g., `ps`, `ls`, `netstat`, etc.) that retrieve important system information and are therefore often targeted by kernel rootkits.
- *Injected code*: the kernel rootkit code injected into the kernel memory address space for execution. The injected code should be accurately located at runtime and extracted for later forensic analysis.

A number of recent efforts have been reported towards kernel rootkit profiling [Yin 2007; 2008, Wang 2008, Lanzi 2009]. Despite their usefulness, the current approaches leave more to be desired in their capabilities: (1) Some approaches

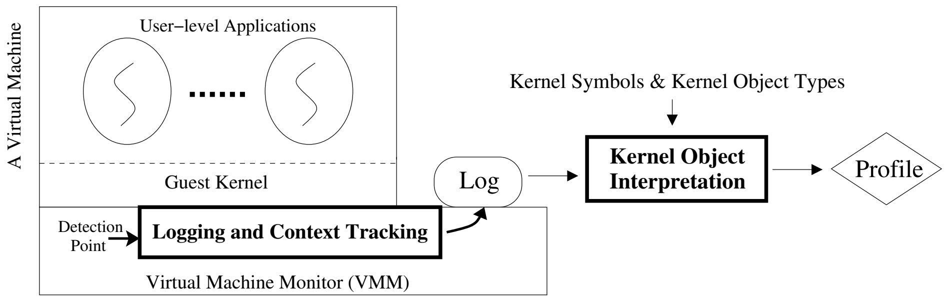
A Virtual Machine
User−level Applications
......
Guest Kernel
Detection Point
Logging and Context Tracking
Virtual Machine Monitor (VMM)
Log
Kernel Symbols & Kernel Object Types
Kernel Object Interpretation
Profile

Figure 1. VMM-based PoKeR architecture

require prior availability of the kernel rootkit code and knowledge that the rootkit attack is going to occur. However, such requirement makes it difficult to profile zero-day kernel rootkits live. (2) The current profiling techniques only focus on one aspect of rootkit behavior (e.g., hooking behavior) or on one stage of a rootkit's life cycle (e.g., installation or execution but not both). (3) The key techniques used in the existing approaches such as system-wide tainting or slicing have well-known limitations and challenges that are hard to overcome. For example, taint-based information flow tracking can be circumvented by various control-flow evasion schemes [Cavallaro 2008].

To overcome the above limitations we present PoKeR (*P*rofiler *o*f *Ke*rnel *R*ootkits), a virtualization-based kernel rootkit profiler that generates multi-aspect kernel rootkit profiles during rootkit execution. PoKeR is designed to be deployed in a system that can tolerate high performance overhead, such as a honeypot which is subject to rootkit attacks in the wild. A PoKeR-enabled system executes normally until a kernel rootkit is installed and ready to execute malicious code injected into the kernel. At that point, PoKeR switches the system (a virtual machine or VM) to a rootkit profiling mode and applies a strategy called "*combat tracking*"[1] to automatically track and determine the kernel objects – either static or dynamic – that are being targeted by the kernel rootkit. In addition, when the targeted kernel objects are being manipulated, PoKeR records the relevant system call contexts and infers potential effects on user-level applications.

We have developed a prototype of PoKeR and used it to profile 10 representative real-world kernel rootkits that exhibit a broad range of attack methodologies. This includes basic system call table hooking, the more advanced technique of direct kernel object manipulation [Silberman 2006], manipulation of function pointers inside dynamic kernel data objects [Hoglund 2006], and others. The profiles generated by PoKeR capture multiple aspects of the rootkit's behavior and permit unique insights into each rootkit's characteristics. We have also measured the performance of our QEMU-based prototype and found that it degrades virtualization system performance between 3x and 6x during profiling, with the virtualization system itself adding an additional slowdown of 3.8x above and beyond that of the physical host.

The contributions of our work are as follows:

- We identify four key aspects of kernel rootkit behavior and use them to characterize and profile existing kernel rootkits.
- We define the concept of an *instantaneous rootkit detection system* and discuss how an existing kernel rootkit prevention system can be transformed into one so that a *detection point* can be generated to trigger rootkit profiling.
- We propose a technique called *combat tracking* to determine the identity and type information of rootkit-targeted kernel objects, even if they are dynamically allocated from the kernel heap.
- We develop a PoKeR prototype and present the evaluation results with 10 representative real-world rootkits. The obtained rootkit profiles provide useful insights into rootkit behavior, some of which are difficult to obtain without PoKeR, despite in-depth analysis.

2. Assumptions

In this work we assume that a kernel rootkit has the same memory access privileges as the OS kernel itself. If the OS can read from or write to a memory location, so can the rootkit. This also means that the rootkit does not have privileges higher than that of the OS, such as those of a virtual machine monitor (VMM). The rootkit is free to modify any kernel objects, whether static or dynamic.

We also assume that the rootkit requires the execution of *injected code* at the kernel's privilege level. We do not, however, require that the injected code be persistent throughout the life cycle of the rootkit attack. We refer to a kernel rootkit that requires the execution of injected code at the kernel's privilege level as a *code injection kernel rootkit*. For ease of presentation, throughout this paper we will use the term kernel rootkit to refer to a code injection kernel rootkit. This assumption is realistic. Petroni et al. [Petroni 2007] surveyed 25 kernel rootkits and none of them violate our assumptions. In particular, all 25 rootkits make use of injected code in the

[1] Combat tracking, in war, is the art of hunting the enemy by following the signs he leaves behind as he moves. In PoKeR, we intend to follow the trail the rootkit leaves behind.

kernel space, and 24 of them require injected code to be persistent throughout their lifetime.

With regards to PoKeR itself, we assume that it has access to the OS kernel source code for static analysis, or to debugging symbols and type information for an already compiled kernel binary. We also assume that the system PoKeR is running on can tolerate high performance overhead during profiling.

3. Design

Figure 1 shows the overall architecture of PoKeR. As highlighted in the figure, PoKeR has two main components:

- The *Logging and Context Tracking* module resides inside the VMM and, once activated, collects runtime execution traces of malicious rootkit code. The execution trace is saved outside the target VM and contains information such as rootkit instructions executed, corresponding memory reads and writes, and associated execution context. The logging of execution context will be helpful later in assessing the user-level impacts of the rootkit attack. Note that the activation of this module requires a detection point and we will discuss it shortly in Section 3.1.
- The *Kernel Object Interpretation* module processes the collected execution trace and resolves read and write target addresses into the kernel objects read or manipulated by a rootkit. The dynamic nature of certain kernel objects significantly complicates the interpretation procedure.

There are three key challenges and techniques associated with the design of PoKeR. They will be presented in the following three subsections.

3.1 Switching to Profiling Mode

As mentioned in Section 1, PoKeR is primarily designed to be used in environments which can tolerate high overhead. A PoKeR-enabled system has two modes of operation. The first mode, *detection mode*, is its initial state. While in this mode, an instantaneous rootkit detection system (defined below) watches for kernel rootkit execution. Most of PoKeR's rootkit profiling features are disabled during detection mode. The other mode, *profiling mode*, starts right at the *detection point*, when the instantaneous rootkit detection system reports that a kernel rootkit attack is about to occur. In profiling mode, PoKeR enables its profiling features and logs the rootkit's actions at a fine granularity, such as instruction execution, system calls, and memory reads and writes. PoKeR will then generate the rootkit's profile according to the four aspects defined in Section 1.

To ensure that all of a rootkit's actions are profiled properly, the detection point must be generated before the very first rootkit instruction is about to execute in the kernel. We refer to a detection system capable of meeting this strict time constraint as an *instantaneous detection system*. Existing work in the area of rootkit prevention, such as Livewire [Garfinkel 2003], SecVisor [Seshadri 2007], and our prior work – NICKLE [Riley 2008], can be used as instantaneous rootkit detection systems. These systems are developed based on various virtualization techniques. For example, SecVisor makes use of hardware virtualization support to prevent malicious kernel code from executing while Livewire and NICKLE leverage software virtualization to ensure only legitimate kernel code will be running in the kernel space. The design of PoKeR will allow it to make use of any of these systems to generate rootkit detection points.

3.1.1 NICKLE as Instantaneous Detection System

In this work, we leverage NICKLE to serve as the instantaneous detection system that generates kernel rootkit detection points for PoKeR. In the following, we will give a brief overview of NICKLE. Interested readers are referred to our previous paper [Riley 2008] for more details.

In short, NICKLE operates inside a VMM and protects commodity guest OSes. NICKLE maintains two separate memory spaces for a running VM. One, the standard memory, functions just like the normal memory space: It stores code and data for both kernel and user levels. The other, the shadow memory, stores only kernel code that has been authenticated by NICKLE. This is done via an on-the-fly technique that uses hashes of known good kernel code for authentication and copies only authenticated kernel code from the standard memory to the shadow memory. At run time, all kernel instruction fetches issued from the guest OS are transparently routed to the shadow memory while all other memory accesses are routed to the standard memory. As a result, a kernel rootkit that is attempting to execute its injected, unauthorized code in the kernel space would be unable to do so. Failing to go through NICKLE's kernel code authentication, the injected code will remain in the standard memory and cannot be fetched from the shadow memory. This is all done in a manner that is transparent to the guest OS, which does not need to be modified.

Turning the original NICKLE into an instantaneous rootkit detection system for PoKeR is a natural next step. Instead of simply blocking rootkit code execution, the system will allow the code to be executed unhindered from the standard memory. During a guest kernel instruction fetch the contents of the standard and shadow memory are compared to determine if the same instruction exists in both. If a kernel instruction that is about to be fetched exists in the standard memory but not the shadow memory (or if the contents simply differ) then unauthorized code is about to be executed at the kernel level. This serves as PoKeR's detection point, and the system can be switched to profiling mode.

Given that we know an instruction is malicious prior to executing it, we have the unique opportunity to identify and extract the malicious rootkit code. It can then be analyzed further later on, such as by static analysis. To aid in this, we also record the order in which the instructions were executed. In addition, the malicious code identification capability may allow profiling mode to turn on and off during profiling – on when rootkit instructions are executed and off when authenticated kernel instructions are executed. The dynamic toggling between detection mode (faster) and profiling mode (slower) may result in better rootkit profiling efficiency.

3.2 Tracking Targeted Kernel Objects

Once kernel rootkit execution is detected and the profiling mode of PoKeR is switched on, it is necessary to keep track of all kernel objects manipulated by the kernel rootkit. The rootkit may, for example, traverse the entire process list looking for an entry with a specific PID to remove. Or, it may change key values in a TCP data structure within the kernel to mask the sending of data to a remote location. It is important that PoKeR be able to determine, upon the execution of a rootkit instruction, which kernel object is being read or modified. This is challenging because PoKeR operates at the VMM level, which does not directly provide a semantic view of the guest kernel objects. Unfortunately, current virtual machine introspection techniques [Garfinkel 2003, Jiang 2007, Payne 2007] do not support such a "reverse lookup" (namely, given a memory address, identify the corresponding kernel object).

A list of the rootkit's reads and writes is simple to obtain using PoKeR's logging and context tracking module, as it simply logs all reads and writes performed by the rootkit code. However, determining which kernel objects a rootkit is modifying is complicated by the fact that a large number of kernel objects are dynamically allocated. For example, we may know that a rootkit is modifying memory at address `0xc6600856`, but if that address is located within the kernel's heap there is no simple way to determine what object it is. (This is one reason that a simple symbolic debugger cannot be used to track kernel objects.) This is in contrast to statically allocated kernel objects, whose addresses can be easily determined at compile time. In order to handle dynamically allocated kernel objects, we need to create an *address-to-dynamic object map* that can be used to translate memory addresses into the kernel objects they are a part of.

One key observation that helps in creating this address-to-dynamic object map is that all dynamically allocated kernel objects must be accessible in some way from global variables or CPU registers. If one imagines kernel objects as a graph where the edges are pointers, then all objects will be transitively reachable from at least one global variable. If an object is not reachable in this way, then the kernel itself will not be able to access it and the object cannot be used. A similar observation has also been made in previous work on both garbage collection [Boehm 1988] and state-based control-flow integrity [Petroni 2007]. A brute force approach for mapping an address to a dynamic object would be to search the entire memory graph. This would be extremely inefficient and thus undesirable.

Algorithm 1 Combat Tracking Algorithm

```
Requires: addr: Address of read.
          val: Value read.

if addr in static map then
    // Query the static data for type information of the address
    type ← static_objects(addr)
else
    if addr in dynamic map then
        // Query the dynamic map instead
        type ← query_dynamic_map(addr)
    else
        // No type information known
        return
    end if
end if
if type is a pointer then
    // If we have a pointer, val is the address of a kernel object
    d_type ← dereference(type)
    add_dynamic_map(val, d_type)
end if
```

To support the address-to-dynamic object mapping in a more efficient way, we propose a technique called "combat tracking." The key observation in our combat tracking technique is that in order for a kernel rootkit to find the address of a dynamically allocated kernel object, it will first traverse to it from a statically allocated one. The rootkit, much like PoKeR, is naturally ignorant of the layout of dynamic kernel objects, and therefore will do a series of reads of kernel memory in order to reach the objects. By tracking a rootkit through its series of reads, we can dynamically build up an address-to-dynamic object map for PoKeR to look up a corresponding dynamic kernel object when given a memory address.

Algorithm 1 shows the combat tracking algorithm executed by PoKeR's kernel object interpretation module. The algorithm assumes the availability of an initial map of static objects and uses that, combined with the rootkit's reads, to build the map of dynamic objects on the fly. (In our prototype, the static kernel object map as well as the object type definitions come from a copy of the kernel compiled with debug symbols.) The first step in the algorithm is to determine what type of data the address being read is. We first query the static object map to see if the object is a global object and if that fails then we check our dynamic object map to see if we have previously added this address to the map. Once we find the type of the object being read, we determine if it is a pointer. We care about pointers because if a read occurs on a pointer object, then the value of the read corresponds to the address of a kernel object. This may be a kernel object we haven't seen before, and it can be used to further build the dynamic map. Given this, in the event the rootkit did read a pointer, we determine the value read by the rootkit (the address of the new object) as well as the de-referenced type of the pointer (the type of the new object) and we add this information to the dynamic map. In this way we progressively build up the address-to-dynamic object map based on the rootkit's reads.

To illustrate combat tracking, let us consider an example. Figure 2 is a simplified representation of the process list maintained in the Linux kernel. There is one global data structure at address `0xc0300000`, `init_task`, which forms the head of a linked list of dynamically allocated `struct task_struct` structures. If a rootkit were to try and find the `task_struct` for pid 3, it would do the following. First, it would read address `0xc0300004` in order to find the `next_task` pointer in the global `task_struct`. It would receive back `0xc11a0000`, the address of the next structure. Next, it would read the pid of that next structure at address `0xc11a0000`, and when it found that it was not 3, it would read `0xc11a0004` to find the next `task_struct` to search on. It would repeat this procedure until it found pid 3 in the `task_struct` at address `0xc11c0000`. From there it may modify a variable in that data structure (say at address `0xc11c0008`) in order to perform some sort of kernel object manipulation.

Without combat tracking, we would only know that the rootkit did a write at address `0xc11c0008` and we would have no way of knowing what kind of data was at the address. With combat tracking, given the entire chain of reads, the dynamic map would be built: When the rootkit first reads the `next_task` element of `init_task`, a query of the initial static map tells us that the read corresponds to an object of type `struct task_struct *`. Given this knowledge, combined with the fact that the rootkit reads `0xc11a0000` from that location, we know that address `0xc11a0000` contains a `struct task_struct` and add it to our dynamic map. When the rootkit later reads the `next_task` pointer from that dynamic data structure, we know (thanks to what we learned from the previous read) that the read is for another pointer of type `struct task_struct *` and can add that element of the linked list to our dynamic map as well. We continue on in this fashion until we have a map of all the data structures the rootkit has read. Later, when the write to address `0xc11c0008` occurs, we can check the dynamic map to know that the address is part of a `task_struct` and determine which element of the data structure is being modified.

We do not keep track of a kernel object's lifetime and remove its entry from the dynamic map right after its deallocation. The entry will still exist in the map despite there being no object at that location. Such a "stale entry" does not matter, however, because the rootkit should not access a deallocated kernel object. (If it does, it is most likely a programming error.) If a new object is ever allocated at a previously used address, then the chain of rootkit reads to the new object will result in the stale entry being replaced by a new entry that corresponds to the new object.

3.3 Discovering Rootkit Hooking and User-Level Impacts

For many kernel rootkits, one key reason for manipulating a specific subset of kernel objects is to eventually hijack the kernel's control flow so that the rootkit can somehow affect the execution state of the running kernel. The hijacking behavior is typically accomplished by modifying function pointers, many of which may be stored in dynamically allocated objects within the kernel's heap. To reveal a rootkit's hooking behavior, it is vital that we be able to find these hooks as they are being installed. It is also possible for the rootkit to directly modify legitimate code to force a call to the rootkit code. Fortunately, both types of changes can be thought of as a subset of the kernel object tracking problem (Section 3.2). Tracking modifications to existing code is similar to tracking modifications to static objects; whereas tracking function pointer modifications is simply a part of tracking object modifications using combat tracking – the main reason being that the modified function pointers belong to certain kernel objects.

As an example, consider a Linux kernel module (LKM) based rootkit[2] with the goal of ensuring that files that end in the extension "`.hacker`" are never visible to a user. The attacker installs this malicious rootkit as a kernel module using the `insmod` command. The system copies the malicious module into memory and then runs the module's `init()` function. Before the first instruction from `init()` is executed, the instantaneous rootkit detection system generates a detection point which turns on PoKeR's profiling mode. Next, the rootkit's initialization function modifies the system call table so that the system call originally used to retrieve a directory listing is changed to point to a malicious function that ensures files ending in `.hacker` do not appear in the listing. The write to the system call table is logged and interpreted. Thus the code's hooking point is discovered and the control flow modification made by the rootkit is profiled.

In addition to determining which function pointers get hijacked by a kernel rootkit, it is also desirable to determine how the modified kernel control flow will impact system calls made by user-level programs. This may help ascertaining which user-level programs are being targeted by a specific rootkit as well as giving us a general understanding of what the rootkit is trying to hide. For kernel rootkits that modify the system call table, such impact is fairly obvious: explicitly modified table entries will result in hijacked control flow when the corresponding system calls are made. For rootkits that do not directly modify system call table entries, however, determining which system calls will be affected is less obvious.

To determine which system calls get their control flow hijacked at runtime, we need to be able to correlate the execution of malicious rootkit code with the execution of

[2] An LKM based rootkit is a kernel module that implements rootkit functionality. It can be loaded into the kernel like a normal driver.

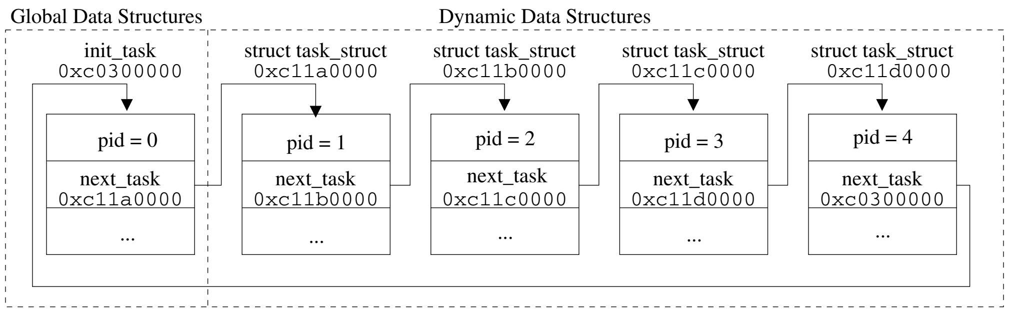

Figure 2. A simplified example of a Linux process list

the system call that led to it. To accomplish this, PoKeR will track the execution of system calls and apply a virtual machine introspection technique [Jiang 2007] to determine the current process context, namely which process is making the system call. Note that by logging the starting point when a system call is made and the ending point when the system call returns, PoKeR can effectively keep track of the lifetime of the system call. If malicious code execution is detected, PoKeR will infer the current process context of the malicious code execution and determine if any ongoing system call has the same process context. If so, the control flow of that system call is hijacked.

4. Implementation

To validate our design, we have developed a prototype of PoKeR. In this section we will describe its implementation.

4.1 Instantaneous Rootkit Detection

As mentioned in Section 3.1.1, NICKLE is our instantaneous rootkit detection system. NICKLE has been implemented and tested in multiple VMM platforms such as QEMU [Bellard 2005], VirtualBox [Innotek], and VMware Workstation [VMware]. In this work, we have chosen the QEMU port of NICKLE for implementation convenience.

4.2 Logging and Context Tracking

Once NICKLE signals the detection of malicious kernel rootkit code, PoKeR enters profiling mode. In profiling mode all kernel instructions are interpreted using the built-in dynamic re-compiler (a virtualization technique based on efficient, dynamic translation of guest code into host code) in QEMU so that the rootkit's actions can be logged at a fine granularity.

A sample of the log is shown in Figure 3. It shows the seven different types of log entries. The `R` and `W` log lines (lines 2 and 4) signify that the malicious rootkit code is now reading or writing. The reads and writes are caught by extending the QEMU-translated VM memory access instruction to include a check on whether the instruction issuing the access is malicious. The first argument on the line is the memory address being read or written and the second argument is the corresponding memory content. The `E` log line

```
1: M - 0xc883d000
2: R - 0xc1548054 - 0xc154a000
3: C - 0xc6706000
4: W - 0xc6707f24 - 0xc6707f3c
5: SC - 619 - insmod - sys_write
6: E - 0xc883ea28 - 619
7: SR - 619 - insmod
```

Figure 3. Sample log entries generated by PoKeR

(line 6) signifies the execution of rootkit code and is generated by PoKeR while a malicious instruction is being translated for execution. The arguments are the address of the malicious instruction and the pid of the process context it is running in, respectively. The `M` log line (line 1) is emitted whenever a kernel module is loaded – as seen by virtual machine introspection that is a part of NICKLE – and signifies the base address of that module's kernel data structure. (The `M` log line is the one item logged before a detection point is raised.) The `C` log line (line 3) is used to signify the address of the task structure of the currently running process (`current` in Linux) and is output preceding a read or write from that task structure.[3] The `SC` and `SR` log lines (lines 5 and 7) signify the start and end, respectively, of a system call. The `SC` log line includes information about the pid, program name, and system call made and is generated by extending the specific binary translation of the system call interrupt (`int 0x80` and `sysenter`). The `SR` log line only conveys the pid and program name, and is generated during the kernel-to-user mode switch.

The `SC`, `SR`, and `E` log entries allow us to determine which system calls have their control flow hijacked by a kernel rootkit. This is done by correlating the system call log entries with the rootkit code execution entries via the process context information. We parse through the log file and track currently running system calls (they begin with an `SC` and end with an `SR`) for running processes. In the event an `E` log

[3] We would like to point out that `current` in Linux is not an actual variable, it is instead a macro that derives the address of the task structure for the currently running process based on the runtime stack. The address of `current` cannot be determined by static analysis and this hint is needed by the object tracker later on. We output `current` during rootkit reads and writes that involve the task structure for the currently running process.

line for a given process occurs while there is an open system call in that process, we know that the system call's control flow has been hijacked.

As mentioned earlier, the malicious rootkit instructions executed are logged along with the order in which they are executed. Later, a customized disassembler [libdisasm] is used to combine these two pieces of information and produce a copy of the rootkit's executed code annotated with its order of execution.

4.3 Kernel Object Interpretation

Once the log file of memory accesses is available, it is important to translate these accesses into names and types of the corresponding kernel objects. To track both static and dynamic kernel objects as described in Section 3.2, static analysis must be performed on the kernel itself. PoKeR can then use this information in conjunction with the rootkit's memory reads to instantiate our combat tracking technique.

The Linux kernel is a large, complicated code base that makes traditional static analysis difficult. However, by compiling a copy of the kernel with debug symbols (the `-g` flag to `gcc`) the GNU debugger (`gdb`) [Free Software Foundation] can be used to extract the types, names, and locations of all static kernel objects. We modified `gdb` to facilitate easier access to this information and query for static kernel object information.

PoKeR's kernel object interpretation module is written in Python and implements combat tracking. It uses `gdb` for static type information and progressively builds its own internal map of dynamic kernel objects by processing rootkit reads using the algorithm in Section 3.2. The rootkit's kernel object manipulation profile can then be produced by querying the static and dynamic kernel object maps in interpreting the rootkit's memory writes. Our implementation also facilitates manual type annotation to accommodate *union* types. For the current prototype *union*s are handled by having a human user decide before hand which type should be used when that specific union is encountered. Another possibility would be to bifurcate union decisions by inserting all possibilities into the dynamic map. This could, however, result in an explosion of search space in the map. We look to emerging work in the area of automatic type determination [Cozzie 2008] to eventually automate the handling of unions.

5. Evaluation

In this section we present the results of using PoKeR to profile 10 real-world kernel rootkits and give a brief evaluation of PoKeR's performance. In our experiments, the host machine is an Intel Core 2 - 2.4GHz desktop running Ubuntu 8.10. The VMM is a modified version of QEMU 0.9.0 running with KQEMU enabled[4]. Our guest OS is Red-Hat 8.0[5] running a recompiled version of its stock kernel, Linux 2.4.18-14. The recompilation is needed to produce a version with debug symbols (Section 4.3.)

Table 1 shows an abbreviated summary of the profiling results. For each kernel rootkit, its profile consists of the four aspects described in Section 1. The first aspect, hooking behavior, is revealed by the modified function pointers in certain kernel objects shown in Table 1. The second aspect of the profile, targeted kernel objects, indicates which objects are of interest to a rootkit. Kernel objects read but not modified are part of this aspect of the profile, but are not shown in the table due to the sheer quantity of them and lack of space.

The third aspect of the profile is the potential impact on user-level programs. Given that most rootkits have a primary goal of altering a system administrator's view of the OS, we ran a corpus of 10 system utility programs that retrieve system information that kernel rootkits tend to hide. Four of them, `w`, `who`, `uptime`, and `finger` are capable of showing information related to currently logged-in users. Two, `netstat` and `ifconfig`, reveal information about network usage. Another pair, `ls` and `bash`, can reveal the existence of files. Information about running processes can be obtained by `ps`. Finally, `lsmod` shows the list of installed kernel modules.

These 10 programs were run and tested to see how many of the system calls they made resulted in the execution of rootkit code. They do not, however, represent the execution of all possible system calls. While a program could be written to exercise all system calls, the enormous variety of arguments and the control paths that those arguments could trigger would make it infeasible to ensure that the program would follow all hooked rootkit code paths. By using programs that a rootkit tends to hide information from, we expect that at least a portion of the malicious rootkit code will be triggered. During the execution of those 10 utility programs, 39 different system calls got executed and those that led to rootkit code execution are shown in Table 1. The last aspect of the profile is the extracted kernel rootkit code shown in Table 1 only by the number of rootkit instructions extracted. This is useful for determining the approximate size of a kernel rootkit, and the code is made available by PoKeR for further analysis, as shown in Section 5.2.2.

5.1 Profiling-based Study of Rootkit Behavior

As a kernel rootkit investigation tool, PoKeR allows a human expert to quickly ascertain and classify a rootkit's attack methodology without solely relying on manual analysis of the rootkit's binary, source code, or the compromised OS. In the following, we summarize the findings that generalize across the rootkits we have profiled using PoKeR.

[4] KQEMU is a host kernel module to enhance QEMU's performance by running some guest code natively on the host processor. It was disabled for the SucKIT experiments because it interferes with an instruction related to the interrupt descriptor table.

[5] We choose this version of Linux because it allows all the rootkits we experimented with to work properly.

Name	Code	Kernel Objects Modified: Kernel Object	Kernel Objects Modified: Note	User-Level Impacts	Attack Type

SucKIT 1.3b	1687 instr	`sys_call_table[59]`
		`system_call` at offset 47
		`tracesys` at offset 27
		`current->addr_limit`
		`current->flags`
			Function Pointer
			Code
			Code
			Data Object
			Data Object
				2 - fork
				3 - read
				4 - write
				5 - open
				6 - close
				11 - execve
				85 - readlink
				195 - stat64
				196 - lstat64
				220 - getdents64
					code change, syscall hook

rial	475 instr	`sys_call_table[3,5,6,141,167]`	Function Pointers	3 - read 5 - open 6 - close 167 - query_mod	syscall hook
rkit 1.01	12 instr	`sys_call_table[23]`	Function Pointer		syscall hook

knark 0.59	490 instr	`sys_call_table[2,3,11,37,54]`
		`sys_call_table[79,120,141,220]`
		`current->flags`
			Function Pointers
			Function Pointers
			Data Object
				2 - fork
				3 - read
				11 - execve
				54 - ioctl
				220 - getdents64
					syscall hook

kbdv3	30 instr	`sys_call_table[30,199]`
		`current->uid`
		`current->euid`
		`current->gid`
		`current->egid`
			Function Pointers
			Data Object
			Data Object
			Data Object
			Data Object
				199 - getuid32	syscall hook, DKOM

adore 0.42	770 instr	`sys_call_table[2,4,5,6,18,37,39,84,106]`
		`sys_call_table[107,120,141,195,196,220]`
			Function Pointers
			Function Pointers
				2 - fork
				4 - write
				5 - open
				6 - close
				195 - stat64
				196 - lstat64
				220 - getdents64
					syscall hook

adore 0.53	733 instr	`sys_call_table[1,2,6,26,37,39,120,141,220]`
		`proc_net->subdir->next->(...)->next->get_info`
		`proc_root_inode_operations->lookup`
			Function Pointers
			Function Pointer
			Function Pointer
				1 - exit
				2 - fork
				3 - read
				5 - open
				6 - close
				85 - readlink
				195 - stat64
				220 - getdents64
					syscall hook, data hook

adore-ng 0.56	785 instr	`proc_net->subdir->next->(...)->next->get_info`
		`proc_root_inode_operations->lookup`
		`proc_root_operations->readdir`
		`ext3_dir_operations->readdir`
		`ext3_file_operations->write`
		`unix_dgram_ops->recvmsg`
			Function Pointer
			Function Pointer
			Function Pointer
			Function Pointer
			Function Pointer
			Function Pointer
				3 - read
				5 - open
				85 - readlink
				195 - stat64
				220 - getdents64
					data hook

linuxfu	117 instr	`init_task->next_task->(...)->prev_task->next_task`
		`init_task->next_task->(...)->next_task->prev_task`
			Data Object
			Data Object
					DKOM

hp 1.0.0	100 instr	`pidhash[600]`
		`pidhash[600]->pid`
		`pidhash[600]->prev_task->next_task`
		`pidhash[600]->next_task->prev_task`
		`pidhash[600]->p_osptr->p_ysptr`
		`pidhash[600]->p_ysptr->p_osptr`
			Data Object
			Data Object
			Data Object
			Data Object
			Data Object
			Data Object
					DKOM

Table 1. Summary of kernel rootkit profiling results using PoKeR

From the “hooking behavior” aspect, we can generalize the rootkits’ profiles to three hooking strategies: modifying existing kernel code, hooking system call entries, and hooking function pointers in data structures. For example, one rootkit that we profiled, SucKIT, modifies existing kernel code. Five rootkits (rial, rkit, knark, kbdv3, and adore 0.42) use syscall hooking as their primary attack vector, with two others (SucKIT and adore 0.53) employing it in addition to other attack techniques. Two rootkits (adore 0.53 and adore-ng 0.56) hook function pointers in both static and dynamic kernel objects.

From the “targeted kernel objects” aspect, we can identify those kernel objects that are more likely to be manipulated by rootkits that manipulate kernel data structures directly. (This is also known as direct kernel object manipulation or DKOM). For example, some critical fields in the process control block (e.g., `uid`, `euid`) can be targeted (e.g., by the kbdv3 rootkit) for escalating the privilege of the process under which the rootkit code runs. The task list is often manipulated (e.g., by the linuxfu and hp rootkits) for process hiding purposes. Moreover, the semantics associated with the function pointers hijacked by kernel rootkits also reveal the rootkits’ intentions. For example, function pointers `get_info` and `lookup` can be hijacked (e.g., by adore 0.53 and adore-ng 0.56) to filter out “sensitive” information so that a rootkit can remain invisible in the compromised system.

Another interesting benefit of PoKeR’s rootkit profiles is that they reveal the changes made between various versions of the same rootkit. Consider the three different adore rootkits in Table 1. Version 0.42 relies solely on a system call hooking attack. A later version, 0.53, lessens its reliance on system call hooking and hooks two kernel objects instead. Once adore becomes adore-ng, it moves to entirely relying on hooks in kernel objects. Such an evolution of adore’s attack behavior is clearly illustrated by PoKeR’s profiles.

5.2 Detailed Results for Three Representative Rootkits

When conducting in-depth analysis of kernel rootkits, PoKeR is especially helpful in providing a human expert with information related to *what* a kernel rootkit did so that the expert can more quickly ascertain *why* the rootkit did it. In this section, we describe detailed profiling results for three kernel rootkits which each display different attack methodologies. In the following descriptions, we will present (1) an analysis of each rootkit based only on general knowledge of Linux and PoKeR’s multi-aspect profile and (2) a manual analysis based on the rootkit’s source code (which we have for the experiments.) Observations and explanations from analysis (1) are presented in normal text; while interpretations based on analysis (2) are indented and preceded by MANUAL INSPECTION RESULTS. Our intention is to show how a human expert can use PoKeR to quickly understand a rootkit’s behavior without its source code. The descriptions also highlight how closely the results from analysis (1) and (2) match and in particular, what PoKeR is or *is not* able to capture.

5.2.1 adore-ng 0.56

Hooking Behavior The hooking profile for adore-ng is quite interesting because it does not hook any system calls. In addition, one of its hooks requires combat tracking (Section 3.2) to reveal. The rootkit modifies six function pointers in various kernel objects. It is particularly interested in the `proc` file system, modifying three function pointers there. One of those pointers, `proc_net->(...)->get_info`, is located in an object that was dynamically allocated on the kernel’s heap (and was found by combat tracking.) The other two, `proc_root_inode_operations->lookup` and `proc_root_operations->readdir` are related to file operations on `proc`. The `proc` file system exports information from kernel-space to user-space and is used by applications that retrieve system information. `ps`, for example, retrieves the process list and `netstat` gets information about open network connections. The hiding of processes and network connections is the most likely reason for hooking `proc`.

> MANUAL INSPECTION RESULTS: A quick search of the adore-ng source code confirms that the `proc_net` hook is there to hide the existence of network connections on certain ports and the `readdir` hook is used to hide running processes. The `lookup` hook, however, is used to signal information to adore-ng’s kernel component. The analysis above did not catch this.

Adore-ng also impacts the main `ext3` file system. The first of these functions, `ext3_dir_operations->readdir`, is used to generate directory listings. The second function, `ext3_file_operations->write`, is used to write to files. The most obvious reason to hook `readdir` on the main file system would be to hide the existence of certain files. The `write` operation instead is to perform one layer of filtering so that rootkit-related information will not be visible.

> MANUAL INSPECTION RESULTS: The source code review confirms that `readdir` is used to hide files. `write` is hijacked to ensure that hidden processes do not write to any of the system wide log files in `/var`.

Lastly, adore-ng hijacks the `unix_dgram_ops->recvmsg` function pointer, which would allow it to intercept UNIX domain socket messages, a type of inter-process communication. This one is quite puzzling. Inter-process communication seems like a very strange thing to intercept and potentially stop.

> MANUAL INSPECTION RESULTS: The source code analysis reveals that it is used to intercept and delete messages to the system logging daemon.

Targeted Kernel Objects Based on PoKeR profiling results, it seems that adore-ng does not modify any kernel objects outside of function pointers and instead does its work by hijacking the control flow. In this respect the rootkit is not any more advanced than many system call hooking rootkits.

Address	Order	Instruction
C72EC40B	22	`lcall 0x00000414`
C72EC410	DATA	
C72EC414	23	`pop %eax`
C72EC415	24	`ret`
...		
C72EE0CB	1	`push %ebp`
C72EE0CC	2	`mov %esp, %ebp`
C72EE0CE	3	`sub $0x0C, %esp`
C72EE0D1	4	`mov $0x00001000, %ecx`
C72EE0D6	5	`push %edi`
C72EE0D7	6	`push %esi`
C72EE0D8	7	`push %ebx`
C72EE0D9	8	`movl 0x14(%ebp), %eax`
C72EE0DC	9	`mov $0x0804EF39, %ebx`
C72EE0E1	10	`sub $0x0804D040, %ebx`
C72EE0E7	11	`movl 0xC(%ebp), %edx`
C72EE0EA	12	`movl %eax, 0xEC(%edx)`
C72EE0F0	13	`movl 0x8(%ebp), %esi`
C72EE0F3	14	`leal 0x400(%esi,%ebx), %esi`
C72EE0FA	15	`movl %esi, -0x4(%ebp)`
C72EE0FD	16	`mov $0x00, %dl`
C72EE0FF	17	`mov %esi, %edi`
C72EE101	18	`mov %dl, %al`
C72EE103	19	`repz stosb %al, %es:(%edi)`
C72EE105	21	`lcall 0xFFFFE40B`

Table 2. Excerpt of SucKIT code extracted by PoKeR

However, it is important to note that while it does not modify any other kernel objects, its malicious code may still be modifying the system call results returned to user-level programs.

MANUAL INSPECTION RESULTS: The source code review confirms these results.

User-Level Process Effects Without modifying any system call table entries directly, adore-ng still manages to execute its malicious payload during system calls. This is logical, considering that the function pointers it modified would be called during various system calls. Our results show that five system calls from our corpus executed adore-ng code.

Extracted Code Adore-ng results in 785 instructions extracted.

5.2.2 SucKIT 1.3b

Hooking Behavior SucKIT is another interesting rootkit mainly because it only modifies one entry in the system call table, 59. This isn't even an interesting entry; it corresponds to `oldolduname`. (Which, as one can imagine, is deprecated and not frequently used.) We hypothesize that this system call is used by a user-space control program to invoke certain kernel-level functions.

MANUAL INSPECTION RESULTS: The source code review reveals that the above hypothesis is mostly correct. SucKIT makes use of that system call entry to make the kernel function `kmalloc` callable from user-space. This allows it to allocate a place for its kernel component from user-space and then install it via `/dev/kmem`.

Targeted Kernel Objects The targeted kernel objects are very interesting, leading to a few important observations. First, PoKeR's memory read log indicates that SucKIT reads in the entire system call table. Second, it modifies the code of two kernel functions, `system_call` and `tracesys`. These two functions can be used to dispatch system calls. For example, when a software interrupt `0x80` is received, the `system_call` function directs the system call to the proper kernel handler by reading the function pointer from the system call table. These two observations lead us to hypothesize that SucKIT makes a copy of the system call table and modifies the dispatcher functions to use the new table instead of the old one.

MANUAL INSPECTION RESULTS: The source code review confirms that the above hypothesis is correct.

User-Level Process Effects In SucKIT's profile, we observed no modifications to relevant function pointers other than the one to the strange system call. However, since SucKIT directly overwrites kernel code in the Linux system call dispatcher, it still hijacks the control flow of key system calls using its alternate table. In our test suite, we find that SucKIT manages to hijack 10 of the 39 system calls.

Extracted Code One tidbit from the extracted code was interesting enough to warrant inclusion here. Table 2 shows the first few dozen instructions executed by the SucKIT rootkit. The table shows the virtual address where the code was located, the order in which the instructions were executed, and the extracted instructions themselves – all provided by PoKeR.

One unique property of SucKIT that can be seen from these instructions is that it has a tricky way of creating a global variable. SucKIT installs itself into the kernel by writing its malicious kernel payload directory into a piece of memory specially `kmalloc`'d and then executing it. The specific address of kernel memory where SucKIT will reside is not known at compile time. Global variables (the addresses of which must be known at compile time) are not available to the rootkit author. Rootkits that install as kernel modules do not have this problem as the kernel will dynamically relocate their code and data prior to execution. Given that SucKIT does not have the benefit of dynamic relocation, a trick is used to permit the use of global variables when their addresses cannot be known a priori. Instruction 21 in the table (`lcall 0xFFFFE40B`) makes a function call to an offset of the current page, in this case a negative number. This call causes instruction 22 (near the top of the table) to execute. The memory layout starting at instruction 22 is quite interesting. One can see the layout is: instruction 22 followed by 4 bytes of data followed by instructions 23 and 24. When instruction 22 executes (another local call) the address of the memory immediately following the `lcall` is pushed onto the stack. This is the return address, but here it corresponds to the address of the 4 bytes of data. The `pop` instruction that runs next moves that address into register `eax` and then issues a `ret` that returns control flow back to the main code. At this point register `eax` contains the address of the 4 bytes of data. This mechanism allows the attacker to achieve the functionality of global variables without having to worry about dynamic relocation.

MANUAL INSPECTION RESULTS: The source code review confirms the above analysis.

5.2.3 hide pid (hp) 1.0.0

Hooking Behavior The hp rootkit modifies no function pointers and, in fact, does not hijack control flow at all. It also does not install persistent code. This is drastically different from the previous two rootkits.

MANUAL INSPECTION RESULTS: The source code review confirms these statements.

Targeted Kernel Objects The kernel object accessed by hp is the pid hash table. (`pidhash` is basically a table of task structures hashed by pid. It allows kernel functions to search for a process by pid without needing to traverse the entire process list. Entries in the hash table are still part of the process list, however.) It is possible to see the rootkit's intentions using the following excerpt from its object access log:

```
R - 0xc03a61a0 (0xc677c000): pidhash[600]
R - 0xc677c078 (0x0000025a): pidhash[600]->pid
R - 0xc677c054 (0xc6780000): pidhash[600]->prev_task
R - 0xc677c050 (0xc76d8000): pidhash[600]->next_task
R - 0xc677c050 (0xc76d8000): pidhash[600]->next_task
R - 0xc677c054 (0xc6780000): pidhash[600]->prev_task
W - 0xc76d8054 (0xc6780000): pidhash[600]->next_task->prev_task
R - 0xc677c054 (0xc6780000): pidhash[600]->prev_task
W - 0xc6780050 (0xc76d8000): pidhash[600]->prev_task->next_task
```

As can be seen from the log, the rootkit reads `pidhash[600]` in the table, verifies it is the correct entry by checking the pid, and then proceeds to remove that entry from the process list by modifying the previous and next pointers of its neighbors. We point out that these task structures are dynamically allocated, yet our combat tracking technique is able to identify them accurately.

MANUAL INSPECTION RESULTS: The source code review confirms that the above analysis is correct.

User-Level Process Effects As mentioned above, the hp rootkit did not execute any malicious code during the execution of our corpus. As a result, we can infer that it does not install persistent code into the kernel and thus has no impact on the system calls.

MANUAL INSPECTION RESULTS: The source code review confirms that the above analysis is correct.

Extracted Code The extracted code of hp is extremely small – only 100 instructions. This makes sense considering that all it seems to do is remove an item from the process list, and lends evidence to the idea that it may not do anything else.

MANUAL INSPECTION RESULTS: The source code review confirms that the above analysis is correct.

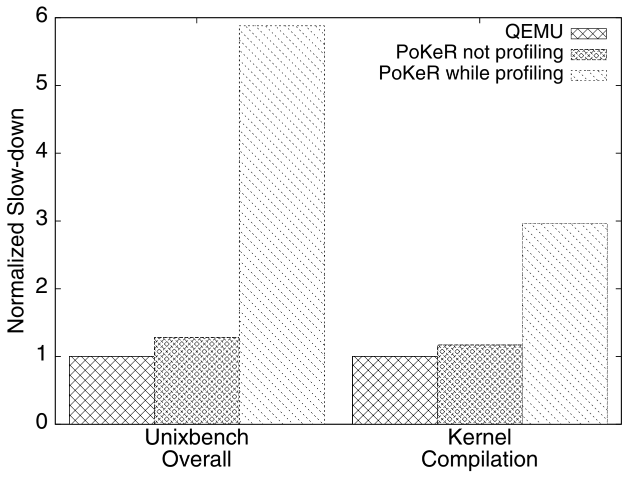

Figure 4. PoKeR performance results

5.2.4 Summary

For the three kernel rootkits analyzed above, the PoKeR-based analysis leads to 13 statements about the rootkits' behavior and 12 of them are confirmed by the analysis based on rootkit source code. (With 1 being mostly correct.) We highlight that the high accuracy of PoKeR is achieved based solely on one execution session and neither the source code nor the original binary of the rootkit was manually consulted. Our comparison also indicates that even when the rootkit source code is available, it would be technically convenient to first use PoKeR – instead of starting right from the source code – to achieve faster revelation and understanding of the rootkit's behavior.

5.3 Performance

While performance is not always a significant concern for a honeypot system, we feel it necessary to give a general idea of the speed at which the various components of PoKeR run.

Runtime PoKeR Module We ran two basic tests to determine the performance of PoKeR's runtime component that generates the log entries. All tests were run under the system as described at the beginning of Section 5. We ran Unixbench 4.1.0 as well as a test timing kernel compilation under standard QEMU, QEMU + PoKeR without a rootkit being profiled, and QEMU + PoKeR while profiling the `adore-ng` rootkit. The results, normalized to the speed of standard QEMU, are shown in Figure 4. (Lower is better.) While in the profiling mode, PoKeR is 2.96x slower than standard QEMU for the kernel compilation test and about 5.88x slower under the Unixbench test. While not profiling (but simply waiting to detect an attack), the slowdown is significantly less, 1.17x for the kernel compilation case and 1.28x for the Unixbench case.

QEMU QEMU itself contributes a noticeable amount of overhead to our PoKeR prototype. Thoroughly benchmarking QEMU is outside the scope of this work, however a basic benchmark is helpful for understanding PoKeR's overall performance. To test the overhead of QEMU we took version 0.9.1 (the latest release) and compared the performance of a native install of Ubuntu 8.10 to a QEMU+KQEMU virtualized copy. Both had access to the same amount of memory (512MB) and one processor core. A kernel compilation benchmark revealed an overhead of 3.8x. Given the portability of NICKLE to other dynamic translation-based VMMs such as VMware [Riley 2008], we believe that this portion of overhead could be reduced by making use of a more efficient VMM platform.

Log Processing To demonstrate the efficiency of log processing, the amount of time taken to process the log entries for each of the 10 rootkits in Table 1 was measured. The longest processing time was for rial: 3 minutes and 36 seconds. The shortest time was for rkit: 0.7 second. The average time across all 10 rootkits was 37 seconds.

6. Discussion

In this section we will discuss potential attacks against PoKeR as well some of its limitations and future improvements.

6.1 Attacks

There exist a number of potential attacks that a rootkit may employ to evade PoKeR.

Our current prototype relies on NICKLE to signal the execution of kernel rootkit code. However, if a kernel rootkit modifies kernel data directly from user-space using a memory access device such as `/dev/kmem`, PoKeR will not be able to profile it. We have synthesized such a rootkit, although it has limited functionality as it cannot execute its own kernel code. A related attack is one that uses only existing kernel code as in an advanced type of return-to-libc attack [Shacham 2007, Buchanan 2008] for the kernel. NICKLE would fail to generate the needed detection point for PoKeR. Existing approaches such as control-flow integrity [Abadi 2005] are able to detect these types of attacks and PoKeR could be engineered to use them to generate detection points.

Combat tracking implicitly relies on the fact that a rootkit must obtain dynamic kernel objects' addresses by starting a chain of reads at a static data object. A rootkit may not need to do this, however. It may, for example, call existing kernel code to retrieve the address of a data structure. In this case, the chain of reads would occur from legitimate kernel code and hence would not be logged. PoKeR can handle this situation by simply tracking all kernel reads instead of just rootkit reads, but at an increased performance penalty. Another potential approach would be to have PoKeR monitor all kernel reads as long as there is a pointer to malicious code on the current kernel stack. This pointer is likely a return address to the rootkit code, which has called the valid kernel code.

Another situation is one where a rootkit installs a code hook and uses it to walk the stack and find kernel object addresses on it. (If the rootkit author knows what functions have already been called prior to his hook, he can easily derive the type information for function arguments on the stack.) In this case, combat tracking would not be able to properly identify the types of data being read. PoKeR could be extended to monitor type information for items on the stack, similar to the way `gdb` does.

Finally, a rootkit may be able to scan kernel memory and guess at the identity of kernel objects, and do so with a high probability of success. One possible approach to combating this attack would be to periodically build a complete map of kernel objects (similar to SBCFI [Petroni 2007]). Assuming that this periodic map building occurred at regular intervals, PoKeR would be able to identify any dynamic kernel object with high probability, even without a chain of reads.

6.2 Limitations

There are some limitations to our current PoKeR prototype. First, our current profiling results are not complete for fully and provably determining all aspects of a given rootkit. Instead, we are only focusing on four specific aspects of the rootkit's behavior. Our lack of completeness is a trait shared by other dynamic analysis based systems [Moser 2007, Lanzi 2009].

Second, the current prototype is still limited in revealing the context in which the rootkit-manipulated kernel objects were used. For example, in the `adore-ng` experiment we noticed that the IPC datagram receive function was hijacked. However the derived profile could not tell us why. Manually inspecting the `adore-ng` source code indicated that this was used to filter messages being sent to `syslogd`. Thus, it would be a huge advantage if PoKeR could be improved to automatically reveal that. In the meantime, we also recognize that PoKeR's user-level impact metric is still simplistic and we plan to extend it to determine the complete set of system calls that may get hijacked at runtime. Correlating modified kernel objects with a static analysis of the kernel's call graph as well as multiple path exploration [Moser 2007] are potential avenues of research in this area.

Finally, a rootkit may be able to detect virtualization or PoKeR's profiling mode and alter its actions accordingly. Note that as virtual machines become more prevalent, they are quickly becoming valid targets for attacks and rootkit authors are losing their incentive to avoid them. While efforts could be made to mask the presence of virtualization from the attacker, it is considered an unsolvable problem in the general sense [Garfinkel 2007].

7. Related Work

Analyzing Kernel-level Malware The first area of related work includes recent efforts in investigating and understanding kernel malware behavior. For example, based on taint analysis, Panorama [Yin 2007] performs system-wide information flow tracking to understand how sensitive data (e.g., user keystrokes) are stolen or manipulated by malware. Unfortunately, the underlying taint-based information flow techniques fundamentally suffer from control-flow evasion attacks [Cavallaro 2008] that directly break taint propagation. From another perspective, K-Tracer [Lanzi 2009] combines backward and forward slicing techniques to understand kernel rootkit behavior. However, the slicing operation requires prior determination of the sensitive data on which to perform the slicing analysis. As a result, although it is capable of dealing with regular kernel rootkits that hijack system call table entries, it becomes less efficient to handle advanced ones such as DKOM-capable rootkits. In comparison, with the capability of tracking both static and dynamic kernel objects, PoKeR does not rely on such prior knowledge and can work with DKOM-capable rootkits (e.g., the hp rootkit in Section 5.2.3) as well.

Several other approaches have recently been proposed to understand rootkit hooking behavior. HookFinder [Yin 2008] analyzes a given rootkit sample and reports a list of kernel hooks that are being used by the rootkit. HookMap [Wang 2008] instead aims to systematically enumerate all of the kernel hooks that can be hijacked for rootkit-hiding purposes. These approaches mainly focus on one aspect of rootkit behavior, i.e., the hooking behavior. However, they miss other aspects that are also important for rootkit profiling purposes.

Detecting Rootkit Presence The second area of related work is the detection of kernel rootkits based on certain rootkit-related characteristics. For example, Copilot [Petroni 2004] leverages a trusted piece of hardware to collect the runtime OS memory image and infers possible rootkit presence by detecting any kernel code integrity violations. That concept has been further extended to identify other types of violations regarding semantic integrity of dynamic kernel data [Petroni 2006] and state-based control-flow integrity of kernel code [Petroni 2007]. Other solutions such as VMwatcher [Jiang 2007] and Strider GhostBuster [Wang 2005] exploit the self-hiding goal of rootkits and infer their presence by detecting differences between the views of the same system from different perspectives. Note that all the above approaches detect rootkit presence based on certain symptoms exhibited by rootkits. However, these systems are not designed to profile kernel rootkit behavior. Some of these could, in principle, be used as PoKeR's detection point generators. However, given that they detect rootkit presence sometime after the attack is under way, some of the rootkit actions may have been missed.

Preventing Rootkit Execution There are also efforts that aim to prevent kernel rootkit execution. For example, Livewire [Garfinkel 2003] is a software virtualization-based intrusion detection system that aims to protect the guest OS kernel code and critical data structures from being modified. SecVisor [Seshadri 2007] enforces kernel code integrity by leveraging hardware virtualization support. NICKLE [Riley 2008] proposes a memory-shadowing scheme that ensures only authenticated kernel code be fetched and executed in the kernel space. Other approaches such as driver signing [Microsoft] and various forms of driver verification [Kruegel 2004, Wilhelm 2007] have also been proposed to protect kernel integrity. Interestingly, though these systems are primarily developed to enforce kernel integrity, they might be adapted to serve as instantaneous rootkit detection systems. The concept of on-the-fly emulation of malicious code has been studied at the user-level [Portokalidis 2008]; in this work we apply the concept to kernel-level rootkit profiling.

8. Conclusion

We present the design, implementation, and evaluation of PoKeR, a kernel rootkit profiler that produces multi-aspect rootkit profiles which include hooking behavior, targeted kernel objects, user-level impacts, and executed rootkit code. In particular, via the combat tracking technique, PoKeR maintains a map of dynamic kernel objects, which allows it to accurately determine which kernel objects are modified by a rootkit. PoKeR is also able to extract the executed rootkit code and infer the potential impact the rootkit might have on user-level programs. PoKeR is evaluated using 10 real-world kernel rootkits, the profiles of which reveal a variety of attack methodologies and demonstrate PoKeR's effectiveness as a rootkit analysis aid.

Acknowledgments We would like to thank our shepherd, Úlfar Erlingsson, and the anonymous reviewers for their numerous, insightful comments that helped improve the presentation of this paper. We would also like to thank Junghwan Rhee for providing us with the code for his linuxfu rootkit. This work was supported in part by the US National Science Foundation (NSF) under Grants 0546173, 0716376, 0716444, 0852131, and 0855297. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[Abadi 2005] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*, November 2005.

[Bellard 2005] Fabrice Bellard. QEMU: A Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[Boehm 1988] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software, Practice and Experience*, 18(9):807–820, September 1988.

[Buchanan 2008] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In Paul Syverson and

Somesh Jha, editors, *Proceedings of CCS 2008*. ACM Press, October 2008.

[Cavallaro 2008] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2008.

[Cozzie 2008] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *OSDI*, pages 255–266, 2008.

[Free Software Foundation] Free Software Foundation. GDB: The GNU Project Debugger. *http://www.gnu.org/software/gdb/*. Last accessed October 2008.

[Garfinkel 2007] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[Garfinkel 2003] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium (NDSS 2003)*, February 2003.

[Hoglund 2006] Greg Hoglund. Kernel object hooking rootkits (KOH rootkits). *http://www.rootkit.com/newsread.php?newsid=501*, 2006. Last accessed November 2008.

[Innotek] Innotek. Virtualbox. *http://www.virtualbox.org/*. Last accessed January 2009.

[Jiang 2007] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy Malware Detection through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007)*, October 2007.

[Kruegel 2004] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 91–100, 2004.

[Lanzi 2009] Andrea Lanzi, Monirul Sharif, and Wenke Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Network and Distributed System Security Symposium*, February 2009.

[libdisasm] libdisasm. x86 Disassembler Library. *http://bastard.sourceforge.net/libdisasm.html*. Last accessed September 2008.

[Microsoft] Microsoft. Driver Signing for Windows. *http://technet.microsoft.com/en-us/library/cc784714.aspx*. Last accessed November 2008.

[Moser 2007] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[Payne 2007] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[Petroni 2004] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot: A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[Petroni 2006] Nick L. Petroni, Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[Petroni 2007] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007)*, October 2007.

[Portokalidis 2008] Georgios Portokalidis and Herbert Bos. Eudaemon: involuntary and on-demand emulation against zero-day exploits. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 287–299, 2008.

[Riley 2008] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, pages 1–20, September 2008.

[Seshadri 2007] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian 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)*, October 2007.

[Shacham 2007] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007)*, October 2007.

[Silberman 2006] Peter Silberman and C.H.A.O.S. FUTo. *Uninformed*, 2006. `http://uninformed.org/?v=3&a=7&t=sumry`.

[VMware] VMware. Vmware workstation, multiple operating systems including linux on windows. *http://www.vmware.com/products/ws/*. Last accessed January 2009.

[Wang 2005] Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, and Chad Verbowski. Detecting Stealth Software with Strider GhostBuster. In *Proc. IEEE International Conference on Dependable Systems and Networks (DSN 2005)*, pages 368–377, 2005.

[Wang 2008] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering Persistent Kernel Rootkits through Systematic Hook Discovery. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, pages 21–38, September 2008.

[Wilhelm 2007] Jeffrey Wilhelm and Tzi-cker Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proc. Recent Advances in Intrusion Detection (RAID 2007)*, pages 219–235, September 2007.

[Yin 2007] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, 2007.

[Yin 2008] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.