K-Tracer: A System for Extracting Kernel Malware Behavior

Andrea Lanzi[1,2] Monirul Sharif[1] Wenke Lee[1]

[1]School of Computer Science, College of Computing, Georgia Institute of Technology, USA
{msharif, wenke}@cc.gatech.edu

[2]Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano, Italy
andrew@security.dico.unimi.it

Abstract

Kernel rootkits can provide user level-malware programs with the additional capabilities of hiding their malicious activities by altering the legitimate kernel behavior of an operating system. While existing research has studied rootkit hooking behavior in an effort to help develop defense and remediation mechanisms, automated analysis of the actual malicious goals and capabilities of rootkits has not been adequately investigated. In this paper, we present an approach based on a combination of backward slicing and chopping techniques that enables automatic discovery of the system data manipulation behaviors of rootkits. We have built a system called K-Tracer that can dynamically analyze Windows kernel-level code and extract malicious behaviors from rootkits, including sensitive data access, modification and triggers. Our system overcomes several challenges of analyzing the Windows Kernel. We have performed experiments on several kernel malware samples and shown that our system can successfully extract all malicious data manipulation behaviors from them. We also discuss the limitations of our current system on newer rootkit strategies, and provide insight into how it can be extended to handle these emerging threats.

1 Introduction

In a system where malware programs reside in the same environment with anti-malware software, whoever operates at a lower-level in the system has the control and wins. Rootkits, programs that are designed to take control of systems by running with administrative level privileges, work as helper programs for other malware and provide additional illicit capabilities that can increase the difficulty of detection and clean-up. User-level malware generally use the help of kernel-level rootkits to hide processes and files, install backdoors, and maliciously manipulate other system level data. Kernel rootkits can easily hide activities from user-level programs and at the same time cripple kernel-level security programs.

Since rootkits are not used alone, discovering their malicious behaviors is essential in understanding the implications of malware that utilizes them. Determining that a rootkit has a backdoor capability that waits for network packets on a specific port with specific contents can enable the identification of infected hosts in a network by way of examining the network traffic. Network perimeter defense systems can also use this information to stop further malicious activities on infected hosts. Rootkits that hide processes with specific names can help identify the malicious programs that are being hidden by the rootkit. In order to conceal specific system level objects such as processes, files, etc., rootkits usually use conditions in the code before manipulating the data. Therefore, extracting triggers or conditions used on sensitive or critical data before modification reveals what specific malicious entities or data items the rootkit wants to conceal. With large, complex and closed-source kernels of operating systems such as Windows, manually understanding these behaviors of rootkits is a daunting and error-prone task. Thus, there is a dire need of an automated system for this purpose.

In order to intercept various events in the kernel, kernel-level rootkits implant *hooks* to divert execution flow to its own code. Recent research has focused on identifying the hooking behavior of kernel-level rootkits [26, 27] in an effort to understand the mechanism in which rootkits install their additional illicit behavior. This information can help identify potential locations in the kernel where hooks are placed as well as ways to remove rootkits from an infected system. However, such systems do not discover the actual malicious goals and capabilities of the rootkits. Moreover, a rootkit with the same malicious goal can use different methods to hook into the kernel. Some newer techniques such as *Direct Kernel Object Modification* (DKOM) and hardware

based approaches do not need to place hooks. While hooking behavior can be expressed as manipulation of kernel control-flow, the actual malicious goals of kernel rootkits can be expressed as *manipulation* of *system-level data*.

In this paper, we present an automated system that efficiently analyzes the data manipulation behaviors of rootkits. Our observation is that rootkits modify sensitive data accessed by user-level programs via system-calls or accounting data structures maintained by the kernel that reflect system state, by executing rootkit code during specific events in the kernel. In our approach, by stimulating potential events and performing data-flow analysis on sensitive data through the executed code paths in the kernel that service these events, we gather detailed information about the data access patterns, data modifications and used triggers of the rootkit.

We use derivatives of the *dynamic slicing* technique [2] for data-flow analysis. Our use of dynamic slicing differs from [26] in that we use a combination of backward and forward slicing on selected execution paths of the kernel. This combined approach is essential for identifying all possible data manipulation behaviors. The more traditional backward slicing phase identifies all potential data sources that can influence the sensitive data. The forward slicing technique is required for identifying all instructions that are influenced by the identified sources, including triggers used in the rootkit code. More precisely, our forward slicing technique is known as chopping [14]. Although many malware analysis techniques rely on dynamic taint propagation technique, we chose not to use it for our analysis purposes. Kernel-level code can read or write from anywhere in memory. Moreover, different kernel components share the same address space. Therefore, performing dynamic taint analysis on rootkits for identifying kernel data modifications may lead to an uncontrollable explosion of tainted code/data throughout the kernel, which may lead to a significant number of false positives. By using dynamic data-flow analysis on selective event-handling execution paths, our system is much more scalable and efficient than pure dynamic taint based approaches while reducing the false positives.

We have developed K-Tracer, an automated rootkit behavior analyzer for the Windows OS. It is developed using the whole system emulator, *QEMU* [4]. An important functionality of K-Tracer is to gather a dynamic trace of the execution path in the kernel that handles a specific stimulated event. In order to extract a precise dynamic trace, K-Tracer handles all Windows internal functionalities, including asynchronous or synchronous I/O events, user-space memory re-mapping of arguments during system calls, and interrupt handling [24]. Our system can identify several types of data manipulation behaviors, including access to sensitive data, modifications, and trigger conditions applied on the data before performing the data manipulation. We evaluated our system on a large number of rootkits and showed that we can extract information about the malicious capabilities of these rootkits. We summarize our contributions below:

- We designed an approach for analyzing kernel-level code that can extract information about the malicious goals and capabilities from the kernel rootkits through the stimulation of the data-flow events. Such rootkit behaviors are not revealed by hooking behavior analysis in existing research.
- We propose a new technique that is a combination of backward slicing and chopping techniques on selective stimulated execution paths in the kernel that identifies all possible data sources that the malware uses to influence sensitive data, and all possible instructions that are influenced by these sources, for example malicious triggers.
- To the best of our knowledge, we are the first to develop a dynamic kernel level analysis engine for the Windows OS. We have tackled several design and implementation challenges in order to make our system handle a large variety of rootkit techniques. We have performed experimental evaluation on different real-world rootkits, showing that our system is able to automatically extract detailed information about the malicious capabilities.
- We describe the limitations of our system against some newer rootkit techniques, and discuss how our system can be further improved to counter these new rootkit techniques.

We present background information related to rootkit behavior analysis in Section 2. In Section 3, we discuss the principles of K-Tracer's approach. Section 4 describes the architecture and details of K-Tracer. Experimental evaluation of our tool with various rootkits are discussed in Section 5. We provide a discussion of how our current implementation can be extended to handle other sophisticated rootkit techniques such as DKOM (Direct Kernel Object Modification) and hardware based techniques in Section 6. Related work is presented in Section 7. We conclude our paper in Section 8.

2 Background

The goal of rootkits can be generally expressed as attempts to manipulate information passed between user programs and the system. Usually, when user programs induce specific events in the system, rootkits intercept these events and perform manipulation on the sensitive data, which is the data being generated or utilized by the event. For example,

when user programs invoke system calls to enumerate the list of processes running on the system, rootkits can check for specific names and filter them, so that they become hidden.

In order to achieve the end-goal of data manipulation, kernel level rootkits use hooks to execute their own code in addition to the default kernel code. Hooks can be placed by inserting instructions in the control-path that jump to the malicious code, or they can be modifications to control-data, which is data used in control transfer instructions. Recent work such as HookFinder [27] and HookMap [26] focus on identifying and extracting hooks placed by rootkits. HookFinder performs impact analysis using dynamic tainting to identify hooks placed by a rootkit in the kernel execution paths. HookMap uses a more elaborate method of identifying all potential hooks in the execution path of kernel code that is induced by the execution of a specific security application. Although discovering the hooking behavior of rootkits can enable the creation of protection and remediation tools, it does not reveal any information about the true malicious capabilities and goals of the rootkits.

Discovering information about rootkit capabilities through its data manipulation behavior can help defend against rootkits as well as the user-level malware that gets help from them, regardless of the hooking mechanisms used by the rootkits. For example, discovering that a rootkit hides process names that start with "_root_" , reveals information that other malware programs working with this rootkit would have process names similar to this. If a rootkit has a backdoor capability and listens on the network for a particular type of message, signatures of a network perimeter defense system such as intrusion detection system (IDS) can be updated to include such messages. Analyzing kernel rootkits to understand the data manipulation behaviors is a complex task, given the complexity of today's operating systems. Moreover, the fact that all the kernel level code can access any data directly without requiring any well-formed interface such as system calls, makes such analysis more challenging than analyzing user-level malware code.

In contrast to recent work in automated kernel rootkit analysis, the overall goal of our proposed system is to analyze rootkits and extract its data manipulation behavior. In other words, given a rootkit, our system will answer three questions related to its malicious manipulation of system data - (1) *what* data items or types of data it accesses or modifies in the system, (2) *how* they are manipulated (whether they are accessed or modified, and if modified what the new values of the data item are), and (3) *when* these data manipulations are taking place, or what the trigger condition based on these data is, which the rootkit uses for activating malicious actions. By data we mean non-control data because control-data is handled by existing systems that identify the hooking behavior of rootkits [26, 27].

In general, the answers to the three questions above can express almost all data manipulation behaviors of rootkits. The information about what sensitive data a rootkit is manipulating is the first building block of the behavior. The method of data manipulation can be divided into two classes - *system data access* and *system data modification*. In the data access case, the rootkit code's purpose is to just read the sensitive data being utilized or generated by a particular event. Malicious goals that involve *stealing* data from a system can be expressed using this behavior. An example of such behavior is key-logging. In the data modification case, the rootkit code writes to the sensitive data. Most of the goals of *filtering* data to hide malicious activities involve data modification. For example, if a user program is scanning the filesystem using the `NtQueryDirectoryFile` system call in Windows, a rootkit that hides files will modify the results of the system call. In most cases, the data access and modification is preceded by *triggers* in the code that activate these actions when a specific condition is satisfied. Almost all filtering activities are preceded by triggers. As another example, backdoor capabilities may only be activated if the network packet received by the system has a specific value in a field or specific structure.

Dynamic tainting approach has been used in several malware analysis systems [18, 27, 28]. However, there are several issues with applying taint propagation to analyze kernel malware. First, kernel-level code has complete freedom in reading and writing anywhere in memory, enabling kernel malware to intentionally propagate taint to common data structures in memory by performing a series of write operations that do not change the data values. Legitimate code may access these common data structures and propagate taint further. Second, because of dynamic memory allocation and deallocation, the same memory address may be utilized by different kernel level components. If semantics of functions that manage heap memory in the kernel are not taken into account properly, taint propagation techniques may be exploited by malware to propagate taint to data managed by legitimate kernel code.

Performing dynamic taint analysis on rootkits for identifying kernel data modifications may lead to an uncontrollable explosion of tainted code/data throughout the kernel. For full-system taint propagation based approaches, if the malware attempts a taint explosion by modifying the kernel data, data manipulation analysis may produce false positive due to the propagation of the taint data to other kernel code path execution (interrupt handler, kernel thread etc.). Our approach is to perform analysis on specific kernel execution paths rather than the execution of the entire kernel, and then identify the direct actions of malicious code on the data accessed in those execution paths. Consequently, this reduces the false positives that may arise by the taint explosion performed by malware. On the other hand, the completeness

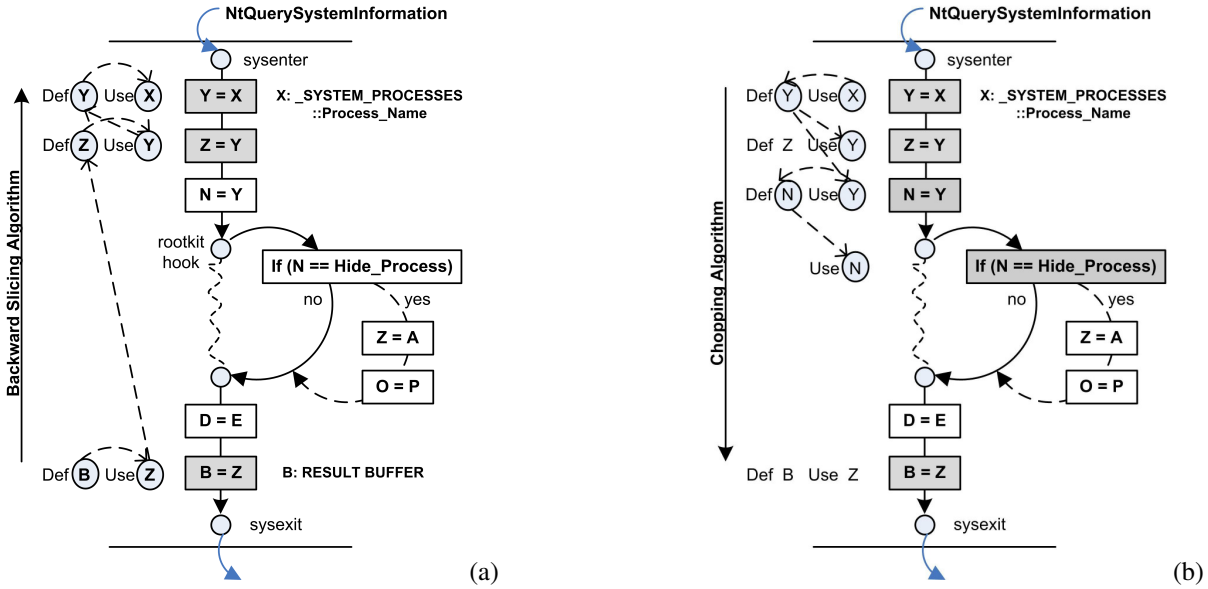

Figure 1. High-level overview of K-Tracer's approach.

of the information extracted by our approach depends on the selection of kernel execution paths.

Our approach of identifying the data manipulation behavior is to stimulate *sensitive events* and perform dynamic data-flow analysis on the related *sensitive data*. A sensitive event can be any software or hardware generated action such as system calls, hardware interrupts, etc., which may be used by rootkit for malicious purposes. The event handler is the part of the kernel code that is executed when the event occurs. The sensitive data items are the data that is provided during the event and the result generated by the event handler. The analysis technique we use is *dynamic slicing* [2]. Prior to our work, HookMap [26] has also utilized dynamic slicing in order to identify all potential hooking points in a kernel path. However, our technique differs by using a combination of *backward* slicing and *forward* slicing (chopping) techniques in order to identify all *data accesses*, *data modifications* and *triggers* of sensitive kernel data in the rootkit code.

3 Overview

We now briefly provide a high-level overview of our approach. In Figure 1, we depict the event handling of a typical Windows system call, `NtQuerySystemInformation`, that can be used to enumerate processes executing in the system. The resulting buffer, B, contains a list of processes executing in the system. Since the result of the system call is the buffer B, we consider it as the sensitive data. In the example, a kernel rootkit has placed a hook and changed the control flow path of the system call handling routine in order to execute its own code. For illustration purposes, we show several data copy operations between buffers from the initial data structure X, which contains the process name from the `_SYSTEM_PROCESSES` structure. The rootkit code contains a trigger-based data modification that checks whether the name of a process matches a specific string and updates data structures accordingly. In this example, we want our analysis to identify the condition as well as the malicious actions performed in the conditional code.

In order to perform dynamic slicing, we first gather a dynamic trace of the event handling code, which starts from the `sysenter` instruction invoking the system call to the execution of the `sysexit` instruction. The trace will contain the kernel code as well as the rootkit code that gets executed. We use a variation of the classic dynamic slicing algorithm presented in [2]. We perform slicing directly on the execution trace without building the program dependence graph (PDG), which requires static analysis. We rely on the dynamic data-flow characteristics of individual executed instructions to identify the data-dependency relations between them. Using the semantics of each instruction, we identify the *definitions* and the *uses* - the registers or memory location operands that are written to and read from, respectively. For example, in Figure 1, the instruction '$B = Z$' defines B and uses Z.

The first step is to perform *backward slicing* starting from the sensitive data. At the high-level the algorithm outputs a set of instructions that influence the computation of

the specified data. The algorithm starts from the last instruction and works iteratively on the trace in reverse order. A *working set* W is maintained that consists of the registers and memory locations whose definitions need to be tracked. Initially, W contains the sensitive data buffer. For each instruction, the algorithm checks whether the instruction defines any register or memory location in the working set. If it does, the instruction is added to the slice, and the data elements used in the instruction are added to the working set after removing the data elements that it defines. More precisely, if $d_1, d_2, ..., d_k$ are the definitions and $s_1, s_2, ..., s_l$ are the uses in an instruction, the working set is updated to W', where $W' = W \cup \{s_1, s_2, ..., s_l\} - \{d_1, d_2, ..., d_k\}$, if $d_i \in W$ for some $1 \leq i \leq k$.

Figure 1(a) illustrates the backward slicing algorithm on the example program. Initially, W is set to $\{B\}$. The algorithm starts from the last instruction in the trace, which is '$B = Z$'. This instruction defines B. Since it is in the working set, the working set is updated to $\{Z\}$ and the instruction is included in the slice. The next instruction included in the slice is '$Z = Y$', which defines Z. The instructions included in the slice at the end of the algorithm are highlighted in the figure. We call all the data accessed by the backward slice as *data sources*. In the example, X, Y and Z are included in the set of the data sources. Notice that the trigger, which is a use of N in a condition, is not identified by the backward slicing algorithm.

The next phase in our approach performs *chopping* analysis, applied on all data sources identified by the backward slicing algorithm. The algorithm starts with the sources in the working set and starts from the beginning of the trace, working iteratively on each instruction. For each instruction, if any of the operands used in the instruction is in the working set, the data defined by the instruction is added to the working set, and the instruction is added to the slice. Otherwise, the definitions are removed from the working set. More precisely, if $d_1, d_2, ..., d_k$ are the definitions and $s_1, s_2, ..., s_l$ are the uses of an instruction, the working set is updated to W' where $W' = W \cup \{d_1, d_2, ..., d_l\}$, if $s_i \in W$ for some $1 \leq i \leq l$. Otherwise, $W' = W - \{d_1, d_2, ..., d_l\}$. The algorithm has the same effect as taint propagation.

The chopping algorithm identifies all instructions that are influenced by the sources. Figure 1(b) depicts the application of the algorithm. As shown, this method identifies all possible uses of the sources, including N. Additional uses of N are identified, which includes the trigger condition "$N == Hide\_Process$".

In order to identify all types of malicious data manipulation, we need both slicing and chopping. Through backward slicing analysis, it is possible to identify *modifications* on sensitive data and find all the data sources from which the modified data are derived. Since the chopping algorithm identifies uses rather than definitions, it is not able to identify modifications to sensitive data that are generated from the system call event. However, backward slicing alone is not able to identify all instructions that *access* or use the sensitive data. The backward slicing algorithm's inability to identify the malicious trigger is an example of such cases. The chopping algorithm fills this gap. Moreover, by applying slicing before chopping, the chopping algorithm can identify not only uses related to sensitive data of the system call, but also the data sources from which sensitive data modifications may be derived.

In order to extract the malicious conditional code in the case where a malicious trigger is not exercised by our stimulation process, we perform graph dominance analysis on the statically identified code from the trigger instructions. Both the information of the malicious conditional code and trigger condition can be used for further analysis to discover the purpose of malicious activities. More details about these techniques are provided in Section 4.5.4.

4 System Design and Implementation

The principles provided in the previous section show how our slicing approach can be used to identify instructions that manipulate system-level data. There are several requirements for our analysis system: (R1) since the slicing algorithm identifies all instructions that manipulate sensitive data, the system must be able to differentiate between legitimate kernel code and malicious code in order to extract malicious data manipulation behaviors; (R2) the system needs to be able to stimulate as many events as possible that a rootkit may utilize to perform malicious actions; (R3) the system needs to map sensitive data specific to the stimulated sensitive event; (R4) for any stimulated event, the system needs to be able to precisely follow and trace the execution in the kernel that handles the event, so that no relevant instruction is missed; (R5) each instruction in the trace should have sufficient dynamic information about the operands for the analysis to work; (R6) the system needs to associate type information to the data buffers that are being manipulated by the rootkit to extract high-level information regarding its malicious activity; and (R7) the trace must contain the conditional code of all the triggers present in the trace.

These requirements have guided the design and implementation of K-Tracer. In this section, we first provide a high-level overview of the different phases involved in the analysis process, and present the architecture of K-Tracer. We then describe in detail how the architectural components work in each phase to accomplish the analysis goals.

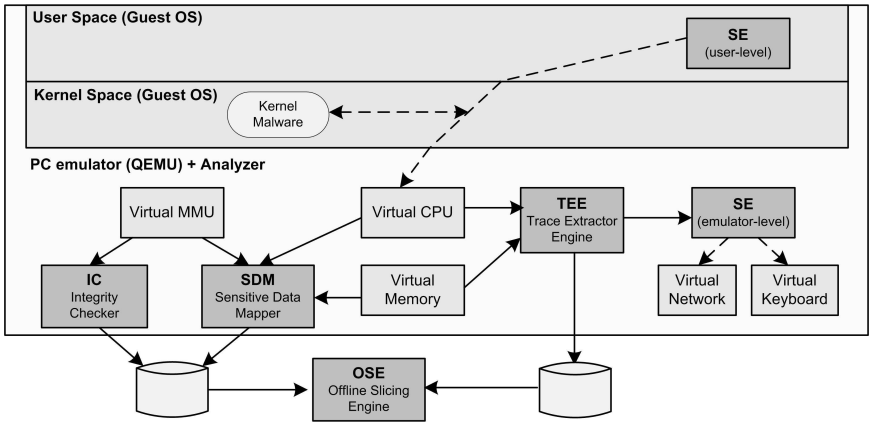

Figure 2. K-Tracer system architecture overview.

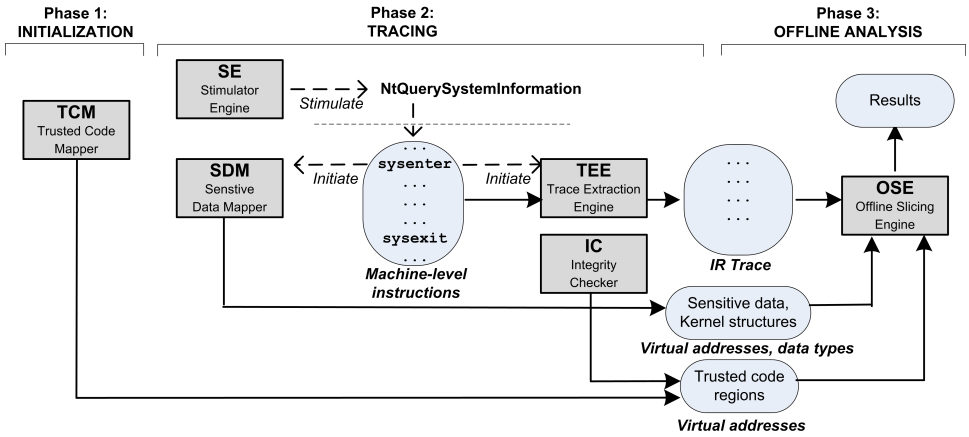

Figure 3. Rootkit analysis process in K-Tracer.

4.1 Overall Architecture and Analysis Process

K-Tracer is based on a whole-system emulator QEMU [4], which emulates an x86 computer system. For our analyzer, we chose the QEMU emulator environment for two main reasons. First, by using a whole-system emulator we are able to perform instruction-level execution tracing of any code running in the guest computer, including the guest operating system. It is possible to trace kernel thread execution as well as the interrupt handling code and monitor the Memory Management Unit (MMU) address translation. Second, the whole-emulator provides an isolation feature, consequently our analyzer is protected against techniques used by malware to disable the detected analyzers that run within the same operating system environment [25]. It is possible that malware may detect emulators to evade analysis [22]. However, our analysis approach is *orthogonal* to the underlying framework, and more transparent frameworks such as the hardware virtualization based *Ether* [9] may be used instead of emulation.

The overall architecture of K-Tracer is shown in Figure 2. The entire rootkit analysis process is performed in three main phases: *initialization*, *tracing* and *offline analysis*, as shown in Figure 3. We provide an overview of the phases and introduce the various components here. In the later sections, we describe the architectural components that perform different functionalities involved in the phases.

The main goal of the *initialization* phase is to build a knowledge base of legitimate code in the kernel. For the integrity of our analysis we assume that a clean boot of the OS can be performed and all the code belonging to the kernel, including any loaded driver, is trusted. This is a reasonable assumption because typically K-Tracer is run on a machine dedicated to malware/rootkit analysis rather than on a live/productivity machine. In this phase, relevant kernel data structures are traversed to extract the regions of code in the kernel address space. The *Trusted Code Mapper* (TCM) module, which is built into the QEMU code base, performs this operation.

After the initialization phase, during the *tracing* phase,

the rootkit driver is loaded into the OS and various sensitive events are stimulated one after another. For each stimulated event, the system first maps out the memory locations of the sensitive data involved in the event. Then, the system generates traces for code executed to handle the events in the kernel. The *Stimulator Engine* (SE) component is responsible for stimulating events in the guest OS. The *Sensitive Data Mapper* (SDM) component maps the virtual memory addresses of sensitive data items in memory when the event is stimulated. In addition, this module traverses kernel data structures related to the stimulated event with necessary type information. The *Trace Extraction Engine* (TEE) module performs the instruction level tracing and outputs an instruction trace relevant to the event. In order to detect whether any tampering of trusted code has occurred, a component called the *Integrity Checker* (IC) monitors memory writes performed during the rootkit execution to identify any modification to trusted code.

The final phase is the slicing analysis of the extracted instruction trace on the identified sensitive data for the stimulated event. The *Offline Slicing Engine* (OSE) is responsible for performing this analysis. If any malicious data modification is detected, this module outputs detailed information about the sensitive data access, modification and triggers.

4.2 Mapping The Trusted Code Regions

In order to satisfy requirement (R1), the *TCM* maps the trusted kernel code regions of the Windows OS. First, a clean version of Windows is booted as the guest OS in QEMU. After the boot is complete and before inserting the kernel rootkit code, the code regions of the kernel are mapped out. At the bottom layer of the Windows XP kernel is the HAL (Hardware Abstraction Layer). Above this layer is the Windows Kernel executive, which manages the core functionalities of the kernel and contains handling code of system calls. The code sections are contained in the files `HAL.DLL` and `NTOSKRNL.EXE`, respectively. Since the loaded base addresses of these components can change, we do not consider statically analyzing them. Rather, we observed that these components along with all loaded drivers are maintained in a linked list pointed by the symbol `PsLoadedModuleList`. By using Windows symbol information and a kernel debugger, we look up its base address. We then walk the list of modules and gather their information, which includes their base addresses. Since each module is a *PE* executable, we parse the headers to identify necessary code sections and stored their address ranges.

We consider any code that does not belong to the identified address regions as malicious. This method of identifying malicious code can create false positives if legitimate drivers are loaded after this initialization phase. However, our analysis environment is controlled and we ensure from our operating system image that no drivers are required to be installed or loaded during the analysis of the rootkit.

4.3 Stimulating Sensitive Events

In order to satisfy requirement (R2), our approach requires stimulation of sensitive events in the system whose behaviors may be maliciously modified by the kernel rootkit under analysis. One of the main issues of stimulating events is achieving completeness. Missing any event that is utilized by a rootkit may result in incomplete analysis. One way to achieve completeness is to stimulate all possible events so as to include all possible system calls or inputs to the system. However, this is not practical because the problem is similar to exhaustive program testing of the operating system kernel. We consider a more realistic and tractable approach of selecting the set of kernel events that can help manipulate data related to the high-level malicious goals of rootkits in general. We focus on three general malicious goals of rootkits: *hiding resources*, *stealing information*, and *enabling remote control*.

According to the classification provided in [3], we address the kernel-data attacks that fall into the *control interception* category, where the rootkits hijack the kernel control flow in order to perform the malicious goals. For our analysis, we consider two main categories of events that rootkits generally intercept control from - system calls and external input events. User-level programs utilize the system call interface to access sensitive system-level data, and in this case rootkits manipulate the arguments or results of system calls. In the case of external inputs to the system (we consider keyboard entry and network input), hardware interrupts are fired and handling of such events is through interrupt handlers.

We selected different system calls for each system resource category defined in [20]: process, token, timer, file and registry. First, for each category we chose the `ZwQuery*` native API, which is the system call used to retrieve the system information about that type of system resource. We then selected the system calls used for basic file and registry operations: `ZwCreateFile`, `ZwOpenFile`, `ZwReadFile`, `ZwDeletefile`, `ZwCreateKey`, `ZwOpenKey`, and `ZwDeleteKey`. Finally, we selected the system calls related to different I/O operations. We selected `NtRequestWaitReplyPort` for keyboard operations, `NtReadRequestData` for network request packet operation, and `NtIoDeviceIoControl` for I/O device operation.

Afterward, for each selected system call we select the parameters that represent the sensitive data and other information such as: size in byte of the sensitive data and I/O operation related to the stimulated system call. We store all

System Call	Sensitive Data	Size of Sensitive Data (byte)	I/O event
NtQuerySystemInformation	2 parameters	200	-
NtRequestWaitReplyPort	2 parameters	1	Keyboard I/O
NtReadRequestData	4 parameters	1	incoming network packet I/O

Table 1. Example of stimulated events file.

such information into a stimulated events file. In Table 1 we show an example of stimulated event information contained in the stimulated events file. It is worth noting that the selection process is based on the knowledge of security analyst and is performed manually.

In order to stimulate the different sensitive events, we use the stimulator engine (SE) shown in Figure 2. The SE has two counterparts - one is a user-level program residing in the guest OS and the other resides in QEMU's code base. The user mode program is responsible for invoking system-calls. The QEMU based component is used to stimulate external inputs into the guest OS by using the user-space drivers provided by QEMU.

The stimulation process works as follows. For each selected system call event, the user-space SE component invokes the appropriate system call. For stimulating an event that requires an I/O operation, the user-space counterpart invokes the QEMU counterpart to stimulate an external input such as a keystroke or a network packet.

It is worth noting that we do not stimulate different argument values of the system calls. Our assumption is that the rootkit does not know the value(s) prior to the control interception. Therefore, it has to apply the triggers on the specific argument values of the system call relevant to the malicious goals of the rootkit.

4.4 Identifying Sensitive Data and Relevant Kernel Data Structures

According to the requirement (R3), the slicing algorithm in K-Tracer requires the address of the sensitive data on which to perform the slicing algorithm. Therefore, for each stimulated event, the locations of relevant sensitive data needs to be extracted. In addition, in order to satisfy the requirement (R6), which is identifying the type information during the slicing algorithm, we need to propagate types from kernel data structures accessed by the kernel-level code. Since the backward slicing algorithm will identify all possible sources that are accessed in the event handling code, we need to map out relevant kernel-level data structures that may be accessed in the kernel-level code. Both of these tasks are performed by the Sensitive Data Mapper (SDM) module in K-Tracer, which is initiated when the handling of the stimulated sensitive event starts. For the example shown in Figure 3, the `sysenter` instruction initiates the activity of the SDM module.

4.4.1 Mapping Sensitive Data

We first describe how addresses of sensitive data related to the events are extracted. In the case of system calls, we mark the arguments and returned results as sensitive data. The problem is that these sensitive data buffers may not exist at the exact moment of stimulation. Rather, their locations can be identified when the event handling code starts in the kernel. Therefore, the addresses are extracted when the K-Tracer determines that the code for handling the event has started executing. In Section 4.5, which describes the tracing method of event handling code, we discuss how K-Tracer identifies the start of an event handler.

Since the kernel has complete control over the virtual memory management in the guest OS, one of the challenges in identifying virtual addresses of memory buffers is that kernel-level code may be accessing the same memory buffer with different virtual addresses. The Windows operating systems (based on the NT kernel that includes Windows 2000, 2003 Server, and XP) support three different methods of passing parameters from user space to kernel space during system calls - *Buffering I/O*, *Direct I/O*, and *Neither I/O*. These methods vary from system call to system call and are managed by the I/O manager [24]. For the case of Buffering I/O, the user space buffer is copied to a separate kernel space buffer. The kernel-level code directly accesses the user space address in the case of Neither I/O. However, when the I/O manager uses Direct I/O, which is mostly used for sending data to drivers, the addresses of the caller's buffer are re-mapped to separate virtual addresses belonging to the kernel. The user-level buffers are locked and the physical addresses information is saved into a structure called MDL and passed to the I/O driver.

In order to collect the addresses of the sensitive data for the *Buffering I/O* and *Neither I/O* method, it is sufficient to extract the virtual address that is passed to the kernel (generally from the stack layout based on the traced system call). For Direct I/O, the method we have developed is a new mechanism that is able to detect multiple virtual address mapping for the same buffer. First, when an instance of the sensitive data buffer is identified, its physical address is computed by traversing the page tables in the OS kernel. Later, while the *tracing* phase continues, the SDM module monitors all the memory operations and searches for new virtual address accesses that have the same physical address as the buffer. The extracted addresses of the sensitive data

are stored in files for use during the off-line analysis phase.

4.4.2 Mapping Kernel Data Structures

The goal of mapping kernel data structures related to a stimulated event is to enable identification of type information for a data buffer in memory that may be found as a source during offline slicing analysis. The SDM module maps relevant data structures in the kernel for the stimulated event in the same manner that we identified the list of kernel modules, and then stores the corresponding addresses to files for off-line analysis. For example, for the `NtQueryInformationProcess` system call, the list of all `EPROCESS` structures are traversed in memory. Since we have the type information of each field in the `EPROCESS` structure, we can identify and output the address of each field from the base address and also the type of the data. We later explain how this information is used in the offline analysis phase.

4.5 Tracing the Kernel

One of the most important tasks of K-Tracer is to precisely trace the event handling code in the kernel as dictated by requirement (R4). Kernel code handling of a specific event may be scheduled out-of-order and may also be preempted during execution due to other events. Moreover, some events such as user inputs generate interrupts that do not have any user process context to associate with it. We identified and overcame several challenges for precisely associating code that is executing in the kernel with the stimulated event.

The first challenge is managing delayed event-handling code execution. We classify the event-handling code in the kernel into two categories - *synchronous* and *asynchronous*. Synchronous events are events in which an application requests a service and the necessary data for the service is always available at the time of the request. In this case, the kernel returns immediately with the results while the program waits. For example, system-call invocation by an application to request the list of processes is a synchronous event. Asynchronous events occur when the application requests some data from the system that may not be immediately available and waits till the data is ready while the kernel continues with other activities. An example of such an asynchronous event is the following. In the first part, the user program invokes a system call requesting the I/O. The kernel schedules the application in the waiting queue. In the second part, when the data becomes available via an interrupt, the kernel sends an IRP packet (I/O request packet) that traverses a list of different drivers registered to service the I/O request, enabling them to manipulate the data. Finally, the I/O Manager asks the Windows Executive to wake up the application and service the data request. The starting point and ending point for these two types of events vary, and we need to consider managing both of these event types in our system.

The second challenge is identifying the context of code executing in the kernel, so that it can be precisely associated with an event. The user-mode stimulator program is used to stimulate synchronous events and the first part of asynchronous events. The CR3 register is widely used to identify the user-process context for a thread running in the kernel. However, since for any interrupt, the kernel ISRs (interrupt service routines) execute with the same CR3 value as the interrupted thread, relying on the CR3 value alone does not help separate the interrupt servicing code. We have developed a method called control-flow prediction (described in Section 4.5.1), which is used to detect and disregard interrupt driven code in cases where it is not relevant to the stimulated event. Another problem is that in the case of asynchronous events, the entire interrupt initiated second part can be executed in the context of any other process. Moreover, *Deferred Procedure Calls* (DPC) can be used for the second part, which also may not execute with the context of the stimulator program. DPCs are Windows supported methods that execute routines in the kernel-level to handle activities that have less priority than interrupts. Relying on the process context in these cases does not work. We describe in Section 4.5.2 our method of handling these cases.

4.5.1 Control Flow Prediction

As stated earlier, in order to be able to discard the code executed by the kernel after preempting a particular execution path that is being monitored, we develop a technique called *control-flow prediction*. The idea is to predict the address of the next instruction $i + 1$ from the instruction being executed i. This can be done by emulating sufficient semantics of the current instruction, so that the updated `EIP` can be computed without changing any processor state maintained by the emulator. For any instruction that is not a branch instruction, we compute the next instruction address by adding the size of the instructions to contents of the value of the `EIP` register. For unconditional/conditional branch instructions, we compute the virtual address of the target of the branch. Then, when the next instruction takes place we check the predicted address. If the prediction is correct, we can continue tracing the current flow, otherwise we save the address and stop tracing until the execution returns to it.

This check may be not sufficient in the case where another execution path (e.g. interrupt handler) executes the $i + 1$th instruction that belongs to the monitored execution path. In this case, the analyzer will resume tracing from the $i + 1$th instruction even though it is from another

thread. In order to solve this problem, we use a technique similar to the technique described in [10], along with the predicted `EIP` we save all the return addresses in all the frames from the call stack into a *virtual stack list*. Here, $S = \{r_1, r_2, ..., r_{n-1}\}$, where n is the number of frames in the call stack and r_{n-1} corresponds to the last invoked function. Every time that the analyzer detects a new execution path, different from the monitored path, it computes and stores the predicted `EIP` and the virtual stack list. Afterward, in order to determine if the control is returned to the monitored execution path, the analyzer, during a context switch, checks the predicted `EIP` and the virtual stack list. If both pieces of information match, the analyzer re-starts tracing. Using this method we are able to select and trace the appropriate control flow related to monitored event.

4.5.2 Handling Synchronous and Asynchronous Events

In order to manage the different asynchronous/synchronous flows, we define a set of *event handling parameters* by which we are able to select the appropriate instructions to trace. We describe them below:

- the *starting point* and the *ending point* of the trace: these identify the beginning and ending points of the tracing that will be performed on the kernel-level code.
- *process context* toggle: defines whether the execution of a particular process's context needs to be followed using the CR3 register value.
- *control-flow prediction* toggle: defines whether the analyzer has to follow the control-flow of the monitored event. The technique to follow the appropriate control flow is explained in section 4.5.1.

Tracing synchronous events is straightforward, we set both the control-flow prediction and process context. The tracing starts with the execution of the `sysenter` instruction and ends with the `sysexit` instruction execution. The method of handling asynchronous events is more complicated. In particular, we have two different settings for the asynchronous events. For the first part (I/O request), when the process requests the input, we use the same setting of the synchronous event - we enable the process context and control flow prediction. For the second part (I/O reply), where the input into the system causes an interrupt, we disable both the process context and control flow prediction. Process context is not useful in this case because the CR3 value may indicate any process in the system. Moreover, we need to disable the control flow prediction because we want to trace the driver code associated with the bottom half of the interrupt handling routine that belongs to another kernel execution flow. The starting point for second part (I/O reply) is specified as the interrupt handling routine related to the event that we want to trace and the ending point is the `sysexit` instruction with the context of the stimulated process. We report all the trace setting in Table 2.

4.5.3 Trace Extraction and IR Conversion

The Trace Extractor Engine (*TEE*) represents the core of our analyzer. The task of TEE is to trace the appropriate execution flow of the selected event and translates the x86 machine code instructions into Intermediate Representation (IR) form. It implements the control-flow prediction technique and the methods of handling synchronous and asynchronous events in the kernel. This module is informed by the SE component about the type of the event being stimulated. Based on the event type, the appropriate parameters for tracing is fixed. The example in Figure 3 shows the tracing of the `NtQuerySystemInformation` system call. Since it is a synchronous event, the analyzer enables control-flow prediction and the process context, and the tracing starts from `sysenter` and ends at `sysexit`.

We have developed our own intermediate representation (IR) representation for analysis. Besides simplifying analysis, we developed the IR to mainly satisfy requirement (R5). The backward slicing technique requires offline analysis after the complete trace is gathered. Our IR form is self-contained. Besides the opcode representation of the assembly instruction, the values of the operands involved in the instruction at the moment of its execution are also stored. For example, for register or memory based arguments, the contents of the register or memory location are also stored. Self-contained IR allows us to analyze the instructions trace offline even if the original machine instructions do not contain all necessary dynamic argument information.

4.5.4 Graph Dominance Analysis

In order to satisfy the requirement (R7), which requires that the conditional code of triggers are also extracted, we apply post-dominator analysis. By invoking this analysis at branches, we ensure that the trace produced by our analyzer contains all conditional code related to the triggers defined. In order to achieve this, we added the post-dominator analysis and a disassembling routine to the TEE engine.

The analysis process works as follows. During the instructions trace, every time the TEE component recognizes a branch instruction, the disassembling component extracts code from the `if` statement until the end of the block is reached. The code output from the portion of disassemble code is converted into a control-flow graph (CFG), so that the post-dominator algorithm [16] is applied. All the code defined between the `if` statement and the post-dominator is translated to IR form and stored along with the trace. Using this technique we can guarantee that our trace contains

Type of Event	Process Context	Control Flow Prediction	Starting point	Ending point
Synchronous	yes	yes	sysenter	sysexit
Asynchronous (I/O req.)	yes	yes	sysenter	-
Asynchronous (I/O reply)	no	no	interrupt event	sysexit

Table 2. Setting of tracing parameters.

the conditional code of every trigger defined in the dynamic trace.

4.6 Checking Integrity of the Trusted Code

In order to ensure that the trusted code of the kernel is not tampered with, the Integrity Checker (IC) component of K-Tracer monitors writes to memory during the tracing phase. The component hooks into the code in QEMU that emulates the MMU and Translation look-aside buffer (TLB) to intercept every memory operation performed on the system. If a write operation is detected on a memory address contained in the kernel code regions, it stores these addresses and marks them as untrusted. These untrusted addresses will be passed to the off-line slicer module that will perform the slicing analysis.

4.7 Off-line Analysis

Finally, the offline analysis phase is implemented in the OSE component of K-Tracer. For each stimulated event, it takes as input the IR trace generated by TEE, the virtual memory addresses of sensitive data, and the type information associated to memory addresses (belonging to kernel data structures) from the SDM module, and the trusted code region information from the TCM module. It then performs slicing and chopping as presented in Section 3. The component first performs backward slicing on the IR trace using the sensitive data buffers.

The component first starts from the bottom of the trace and looks for instructions that *define* sensitive data buffers. For each data source *used* in these instructions, the algorithm recursively finds out instructions that defined them as well, and so on and so forth. As a result all instructions and data sources that influence the values of the sensitive data are identified. At this point, type information is associated to the data sources that belong to the kernel data structures mapped during the event stimulation. Next, the chopping phase starts from the beginning of the trace, and iteratively identifies the instructions that are influenced by all the identified data sources.

We take the union of the sets of instructions identified in slicing and chopping phases. Using information of the trusted code ranges, we filter out all instructions that belong to the trusted code. It is worth noting that even if the rootkit hides its own code in some esoteric memory location [12, 13] (PCI card, graphics card, etc.) we are able to detect it. The OSE outputs all definitions (data modifications) and all uses (data accesses and triggers) in the remaining set of instructions with relevant type information. In general, type inference helps identifying the data structures and fields manipulated in these instructions by the rootkit.

5 Experimental Evaluation

In order to evaluate the effectiveness of our approach, we used K-tracer to automatically analyze the behavior of a dataset of Windows rootkits. We chose a representative set that contains a mix of malicious capabilities of rootkits found in the wild. We ensured that the selected rootkits utilize a variety of hooking mechanisms in order to show that our system is capable of identifying the actual malicious data manipulation behavior regardless of the hooking mechanism. Besides system call hooking, the selected rootkits used other techniques such as layered drivers and IRP function pointer modification to intercept inputs to the system. We describe in Section 6 how our system can be extended to handle rootkits that do not use any hooking mechanism.

As described earlier, K-Tracer stimulates a chosen set of events that include a set of system calls for process handling, file handling, network information, and various I/O related system calls. In addition, keystroke entry and network packets are injected into the system. For each rootkit, our analyzer identified the event for which malicious data manipulation was detected, the sensitive data that was being manipulated, as well as manipulation method indicating whether it was access or modification, and triggers (which indicates when data is manipulated). The results are shown in Table 3.

We first evaluated our tool on the HideProcessHookMDL [6] rootkit. Among all the stimulated events, our system identified data manipulation behavior for the information returned by the `NtQuerySystemInformation` system call that was called to extract the list of processes. During analysis, the SDM module detected two virtual addresses associated to the same buffer passed from user-space to the system call handler code (Direct I/O method) in which process information is returned. Both of these addresses were used by backward slicing algorithm from the end of the trace. The algorithm identified that the original system call handler code used the `EPROCESS`

Rootkit	Malicious detected event	Sensitive data	data manip.	# Trigger	active	passive
HideProcessHookMDL	NtQuerySystemInformation	buffer (2 par.)	modifying	1	1	-
TCPIRPHook	NtQuerySystemInformation	buffer (2 par.)	modyfing	1	1	-
Klog	NtRequestWaitReplyPort + key event	buffer (2 par.)	access	-	-	-
NTROOTKIT: filesystem	NtQueryDirectoryFile	buffer (6 par.)	modyfing	1	1	-
NTROOTKIT: keylogger	NtRequestWaitReplyPort + key event	buffer (2 par.)	access	1	0	1
NTROOTKIT: backdoor	NtReadRequestData+ network event	buffer (4 par.)	access/mod	20	2	18
SysEnterHook	NtWriteFile	buffer (6 par.)	access	1	0	1
MigBot	NtIoDeviceIoControl	buffer (7 par.)	access	1	0	1

Table 3. Evaluation of data manipulation and trigger analysis.

Rootkit	Trace IR size	Sensitive data size (byte)	Trace extracting time	Slicing Analysis time per trace
HideProcessHookMDL	7.2 Mb	200	50s	133 min.
TCPIRPHook	9.7 Mb	150	37s	100 min.
Klog	74 Mb	1	56s	1.6 min.
NTROOTKIT: filesystem	10 Mb	20	34s	26 min.
NTROOTKIT: keylogger	65 Mb	1	80s	1.2 min.
NTROOTKIT: backdoor	109 Mb	1500	58s	625 min.
SysenterHook	1.2 Mb	4	35s	2.5 min.
MigBot	10Mb	10	45s	6 min.

Table 4. Evaluation of space and time required for analysis.

data structures of different processes in the system as the source of data. The chopping algorithm identified a use of this data in a trigger condition inside the rootkit code. The type information gathered by the SDM module on the `EPROCESS` structure's fields were successfully propagated during chopping to the trigger condition. The trigger condition checks for a process name in the list of processes returned in the buffer to be equal to the string "_root_". We manually analyzed the rootkit and found that it hooks into the `NtQuerySystemInformation` by replacing its address in the `SSDT` (System Service Dispatch Table) and hides process information for processes with the name"_root_". Thus, our analysis system correctly identified the only malicious behavior contained in the malware.

We next experimented on the Klog rootkit [6], which mainly provides key-logging capability. Among all the stimulated events by the SE component, the analyzer was able to detect data manipulation behavior when keystrokes where stimulated. SE component stimulated the `NtRequestWaitReplyPort` system call event, requesting keyboard entry data (I/O request phase). During the external key-stroke stimulation, the keyboard interrupt was fired and our analyzer traced the execution as an asynchronous event following the code path starting from interrupt generation to the return of the original stimulated system call from the user-level program (I/O reply phase). Next, our analyzer performed backward slicing on the returned buffer of the `NtRequestWaitReplyPort` system call and all the sources of data. The slicing phase identified that there were no modifications involved with the keystroke data. The keystroke was read by the malicious code and finally used as an argument to the `ZwWriteFile` kernel function, which indicated that the keystroke was stored in a file by the rootkit. We manually analyzed the rootkit and confirmed this behavior, and also concluded that no other interesting data manipulation behaviors were present in this sample. As for the hooking mechanism, this rootkit inserted itself as a layered driver to handle the keyboard entry interrupt.

We next analyzed the TCPIRPHook [6] rootkit. While stimulating different events, K-Tracer was able to identify malicious behavior when network related information was queried from the system. In this scenario, the stimulation was done from the user space component by calling the `NtQuerySystemInformation` system call with the specific argument to query TCP or UDP ports open for a process. After performing backward slicing on the buffer returned by the system call, the chopping phase identified data returned in the buffer used in a trigger condition in the malicious code. The condition used the port number field in the returned information. Manual analysis confirmed that the main goal of this rootkit is to hide the network information related to the web connections at port 80 on the host. As a note, the rootkit hooked into the kernel by modifying the pointer to the `TCPIP.sys` IRP function related to the completion routine for incoming network packets.

We next applied K-Tracer's analysis on NTROOTKIT [1]. This rootkit is composed of several different modules, each of which is used to perform different malicious tasks. The data manipulation behavior output

by our system was grouped into three different classes: keyboard sniffing, file-hiding, and placing a network backdoor. The keyboard sniffing activity was identified in the same manner as described earlier for the Klog rootkit because NTROOTKIT uses the same layered driver technique for intercepting keystrokes. However, our analysis identified a trigger condition while performing chopping. The condition checks the scancode of an entered key to be between 0 and 64. The file-hiding capability was identified when the `NtQueryDirectoryFile` system call was stimulated. Similar to the technique employed by the HideProcessHookMDL, NTROOTKIT modified the function pointer to the handler of the `NtQueryDirectoryFile` system call to its own code. The chopping algorithm identified a trigger condition in the malicious code that checks for the filename or directory name field in the returned information that starts with the string "_root_". The control dependent code revealed that it updates the links to remove such file or directory objects from the list. The last NTROOTKIT component, which places a covert channel network backdoor, was detected by K-Tracer while stimulating incoming network packets. Tracing of this event was done using the method of asynchronous events. The backward slicing algorithm revealed triggers based on the incoming network packet. The malware contains conditions to check whether the incoming packet is a TCP packet, contains the IP source address 10.0.0.166 and the total length of the payload is 255 bytes. In order to further reveal the malicious data manipulation when these triggers are true, we constructed a packet with the similar information. During this phase, several triggers were identified by K-Tracer that check for commands in the incoming packets. The rootkit code contains a total of 15 additional triggers that work as commands, which check for the strings "ps", "help", "echo", "buffer", "hidedir", "hideproc", "sniffkeys", etc. Our analyzer identified the code that would execute if each of these triggers were true.

In order to cover the state-of-the-art rootkit hooking techniques, we developed two synthetic rootkits that perform *sysenter hooking* and *runtime code patching*. The `sysenter` hooking approach modifies the contents of a register `IA32_SYSENTER`, which contains the address of a system call handling routine. Once the register is modified, any system call can be redirected to a malicious function. The `sysenter` instruction is a faster method of calling system calls and is used by Windows operating systems after Windows XP SP2. In our synthetic rootkit, we intercepted calls to the `NtWriteFile` system call. We inserted a trigger inside the hooked malicious function that inspects the buffer that is written to a file for the pattern "password". If the pattern is found, the rootkit stores the entire data contained in the buffer into a file. Our analyzer was able to identify the malicious trigger when stimulating file related function calls. We next analyzed our system on the migbot rootkit. The technique used by the migbot rootkit is *runtime code patching*. In particular, we patched the `NtIoDeviceIoControl` function code and redirected its execution to the malicious function defined by the rootkit. Similar to the previous case, we inserted a passive trigger that inspects the data passed by the user-space application to the kernel driver to look for a password pattern. The ability of K-Tracer to monitor memory writes enabled the identification of the untrusted code regions that the rootkit patched. The backward slicing algorithm was able to successfully identify the trigger and reported the IR form of the conditional code.

Finally, in order to measure the efficiency of our approach, we collected data on the time and data memory required to perform complete analysis of each malware sample. As shown in Table 4, the bulk of the run-time is spent on performing the slicing and chopping analysis of different stimulated events. The use of selective dynamic tracing combined with off-line analysis also helped keep the usage of memory low.

6 Discussion

In this section, we discuss the implications of two new emerging rootkit techniques for modifying system data: Direct Kernel Object Modification (DKOM) and hardware-based techniques. Both techniques do not require hooking into the kernel execution path that handles the sensitive data. However, they allow the rootkit to manipulate only limited categories of data in the kernel. We describe how our analyzer can be extended to analyze these kinds of rootkits.

6.1 DKOM: Direct Kernel Object Modification Technique

Direct Kernel Object Modification (DKOM) DKOM is a technique that directly changes the information from the accounting tables in the kernel, avoiding the use of classic hooking techniques for intercepting events that access these accounting tables. For example, a rootkit can delete an entry from the `EPROCESS` table to hide a process from the user-space applications without affecting the execution of the system. However, this technique cannot be used to hide all types of resources. For example, to hide filesystem objects the rootkit still needs to use classic hooking techniques because the information about all files are not maintained in memory by the kernel. Another drawback is that, by using the DKOM mechanism there is always a time-window where detectors may be able to access the resource that is being hidden. In general, the DKOM rootkit can modify the accounting tables in three specific times - (1) at the driver

loading time, (2) on demand from the user-space component, or (3) at the creation time of the "target"[1] object (e.g. create process function for the process object). For the first two cases the object must be already present in the system. Therefore, some detectors may already log its presence before it is manipulated later. The last one assures stealthiness but is immediately detected by our analyzer (classic system call hooking technique) because it monitors the creation of target object event.

There are several challenges for using K-Tracer to analyze DKOM-based rootkits. The first issue is fixing the trace start and end point. If the rootkit changes the tables during loading time, it can be detected by changing the starting/ending point of tracing to be the kernel function used to initialize the driver (for example in Windows, we can intercept the `DriverEntry` function as a interface function). If the tables are modified on-demand, we first need to detect the communication between the user-space component and the rootkit driver. Then, the communication event needs to be used to trigger the analysis. Communication discovery can be done by monitoring system calls used for communication between drivers and user programs (e.g. `NTDeviceIoControl`).

The second issue is that we need to define the sensitive data on which to perform the slicing analysis. In order to collect the sensitive data, we can use the following strategy. When we detect the event to start tracing, we can dynamically walk the relevant accounting tables and collect their addresses. These addresses can then be used during slicing. We should be able to then detect all potential malicious manipulations of data and triggers used by the rootkit that relies on the DKOM technique.

6.2 Hardware-Based Technique

By directly accessing the hardware resources rather than relying on the OS, newer rootkit techniques can manipulate system state without requiring any hooking into the kernel code. The directly accessed hardware resources include peripheral hardware, disk controllers or USB device. Our current implementation fails to detect and analyze such rootkits because they do not hook any event handling code that manipulates the resources. For example the basic_keysniff rootkit [6] monitors the keyboard's serial port by using a polling technique.

In order to address this problem, since we are using a whole system emulator, we can utilize its facility to control every hardware device notification. The idea is to consider the hardware buffer belonging to the hardware device as the sensitive data. In this way, every time a hardware event occurs, (e.g. key ready into the serial buffer), we monitor the hardware buffer and collect the code that gets access to the buffer. We will then be able to detect and analyze most of the hardware rootkit code.

Another good example of a different hardware-based rootkit is Cloaker [8]. Cloaker exploits hardware features of the ARM architecture and implements a tiny operating-system. This rootkit is completely self-contained. That is, it does not use any code belonging to the OS. The hooking technique used by Cloaker is to define a new memory regions for the IDT table by flipping a bit on the ARM processor. In order to detect and analyze the Cloaker code, we do not need to modify our analyzer. In fact even if Cloaker takes control of the IDT table, by tracing the normal event code we are able to collect the instructions that belong to the flow of the hijacked code. Even in this case we are able to detect the rootkit code and extracts potential malicious triggers.

7 Related Works

Several approaches have been proposed for detecting [11, 23] and analyzing kernel malware. Kruegel et al. [15] proposed a system based on the static analysis to check the memory operation of the kernel driver at the loading time. In particular, the system checks using symbolic execution whether the driver can modify some sensitive kernel memory locations. Even though this approach can detect several malicious behaviors in kernel drivers with a low false positive, a limitation is that it cannot handle many obfuscation techniques [7, 17, 19] that can be used by the rootkit in order to impede static analysis.

Heng Yin et al. presented HookFinder [27], a fine-grained taint-analysis based system that is able to identify the hooking behavior of rootkits. More specifically, HookFinder considers any change made by the malware as tainted and recognizes the code and data locations that are modified to alter control flow to execute the rootkit code as hooks. Another hooking behavior identification system was proposed by Zhi Wang et al. [26]. The system extracts the trace of kernel instructions regarding a particular system call event, and then searches potential places for hooks in the instruction trace that can be used by a rootkit. After the system identifies the hooks, it uses dynamic slicing in order to recognize all memory locations that can be used by the rootkit to divert the control flow. Whereas HookFinder identifies hooks placed by a particular malware, HookMap identifies all potential hooks in the execution path of kernel code relevant to particular security applications. Even though these approaches are capable of extracting the hooking behavior of rootkits, they do not handle the behavior of maliciously manipulating non-control data in the kernel. An important reason why this type of analysis has become very important is because new rootkit approaches, such as

[1] By target object we mean the object that the rootkit wants to manipulate

DKOM and hardware based techniques [6], do not need to use hooking techniques to manipulate kernel data.

Petroni et al. proposed SBCFI [21] for kernel rootkit detection. SBCFI performs static analysis on the Linux kernel source code and builds a control-flow graph of the legitimate kernel-level code. Afterward, during system execution, the system checks the integrity of the control-flow at some specific fixed intervals. This approach is able to detect persistent control-flow manipulations only, and is not able to adapt to dynamic changes to legitimate control-flows in the kernel. Another tool, Panorama [28], proposed by the Heng Yin et al., uses dynamic taint analysis to detect malicious behavior that is primarily privacy-breaching. Although the system can be effectively used to analyze rootkits that steal data from the system, it cannot be used for analyzing other behaviors such as malicious triggers.

Several approaches have been proposed that identify trigger based behaviors in user-level malware programs. Bitscope [5] uses static analysis and symbolic execution to understand behavior of malware binaries and is capable of identifying trigger-based behavior. However, since it is static analysis based, it can be defeated by several known obfuscation techniques. Moser et al.'s multi-path exploration [18] was the first dynamic approach for identifying trigger-based behavior in malware. Given sufficient execution time, the technique can discover conditional code in malware. The system uses QEMU and dynamic tainting to identify conditions and construct path constraints that depend on inputs coming from interesting system calls. Once a conditional branch is reached, the system attempts execution on both of the branches after consistently changing memory variables by solving the constraints. This technique works well for the user-space malware but presents some issues when applied to discover triggers in the kernel space. First, kernel level analysis would require saving the entire machine snapshot for each potential branch, which may produce overwhelmingly large overhead for saving and restoring states. Second, the approach requires deterministic behavior, and ensuring that the kernel threads are scheduled and interrupts are triggered deterministically requires replaying all inputs to the system in a deterministic fashion. Finally, malware can intentionally propagate taint to many kernel data structures in an attempt to force exploration of many kernel execution paths, making constraint solving analysis and path exploration much more complex.

8 Conclusion

In this paper, we presented K-Tracer, a Windows OS kernel tracer that is able to extract the malicious goals and capabilities from the kernel rootkits through data-flow analysis of kernel execution of sensitive events. We implemented a new approach that is a combination of backward and forward slicing techniques on selective stimulated kernel events in order to identify the rootkit behaviors. We have applied our system, K-Tracer, to analyze several representative rootkit samples and have shown that our system is able to automatically extract detailed information about their malicious capabilities. We also discussed limitations of our current implementation and suggested how our system can be improved to handle new rootkit techniques.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. 0716570 and Grant No. 0831300, and the Department of Homeland Security under Contract No. FA8750-08-2-0141. 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 National Science Foundation and the Department of Homeland Security.

References

[1] NTROOTKIT: Kernel backdooring. http://www.rootkit.com/.

[2] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[3] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 246–251, Washington, DC, USA, 2007. IEEE Computer Society.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Usenix Annual Technical Conference*, 2005.

[5] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, S. D, and H. Yin. Bitscope: Automatically dissecting malicious binaries. In *CMU-CS-07-133*, 2007.

[6] J. B. by Greg Hoglund. *Rootkits: Subverting the Windows Kernel*. 2004.

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL98)*, January 1998.

[8] F. M. David, E. Chan, J. C. Carlyle, and R. H. Campbell. Cloaker: Hardware supported rootkit concealment. In *IEEE Symposium on Security and Privacy 2008*, 2008.

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *In Proceedings of The 15th ACM Conference on Computer and Communications Security (CCS 2008), Alexandria, VA, October 2008*, 2008.

[10] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[11] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium (NDSS 2003)*, 2003.
[12] J. Heasman. Implementing and detecting an acpi bios rootkit. In *In Black Hat Europe, Amsterdam*, March 2006.
[13] J. Heasman. Implementing and detecting a pci rootkit. In *Technical report, Next Generation Security Software Ltd*, November 2006.
[14] D. Jackson and E. J. Rollins. Chopping: A generalization of slicing. 1994.
[15] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *20th Annual Computer Security Applications Conference (ACSAC), IEEE Computer Society Press. USA, December 2004*, 2004.
[16] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.
[17] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.
[18] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium of Security and Privacy*, 2007.
[19] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC), IEEE Computer Society Press. USA, December 2007*, 2007.
[20] G. Nebbett. *Windows NT/2000 Native API Reference*.
[21] N. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.
[22] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *10th Information Security Conference (ISC), Lecture Notes in Computer Science, Springer Verlag. Chile, October 2007*, 2007.
[23] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008), Boston, MA*, September 2008.
[24] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals*. 2004.
[25] P. Szor. *The Art of Computer Virus Research and Defense*. Symatec Press, 2005.
[26] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)*, 2008.
[27] 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.
[28] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *14th ACM Conference on Computer and Communications Security (CCS), ACM Press. USA, October 2007*, 2007.