Detecting Kernel-Level Rootkits Through Binary Analysis

Christopher Kruegel
Technical University Vienna
`chris@auto.tuwien.ac.at`

William Robertson and Giovanni Vigna
Reliable Software Group
University of California, Santa Barbara
{`wkr,vigna`}`@cs.ucsb.edu`

Abstract

A rootkit is a collection of tools used by intruders to keep the legitimate users and administrators of a compromised machine unaware of their presence. Originally, rootkits mainly included modified versions of system auditing programs (e.g., `ps` *or* `netstat` *on a Unix system). However, for operating systems that support loadable kernel modules (e.g., Linux and Solaris), a new type of rootkit has recently emerged. These rootkits are implemented as kernel modules, and they do not require modification of user-space binaries to conceal malicious activity. Instead, these rootkits operate within the kernel, modifying critical data structures such as the system call table or the list of currently-loaded kernel modules.*

This paper presents a technique that exploits binary analysis to ascertain, at load time, if a module's behavior resembles the behavior of a rootkit. Through this method, it is possible to provide additional protection against this type of malicious modification of the kernel. Our technique relies on an abstract model of module behavior that is not affected by small changes in the binary image of the module. Therefore, the technique is resistant to attempts to conceal the malicious nature of a kernel module.

1. Introduction

Most intrusions and computer security incidents follow a common pattern where a remote user scans a target system for vulnerable services, launches an attack to gain some type of access to the system, and, eventually, escalates her privileges. These privileges are then used to create backdoors that will allow the attacker to return to the system at a later time. In addition, actions are taken to hide the evidence that the system has been compromised in order to prevent the system administrator from noticing the security breach and implementing countermeasures (e.g., reinstalling the system).

The tools used by an attacker after gaining administrative privileges include tools to hide the presence of the attacker (e.g., log editors), utilities to gather information about the system and its environment (e.g., network sniffers), tools to ensure that the attacker can regain access at a later time (e.g., backdoored servers), and means of attacking other systems. Common tools have been bundled by the hacker community into "easy-to-use" kits, called *rootkits* [3].

Even though the idea of a rootkit is to provide all the tools that may be needed after a system has been compromised, rootkits focus in particular on backdoored programs and tools to hide the attacker from the system administrator. Originally, rootkits mainly included modified versions of system auditing programs (e.g., `ps` or `netstat` for Unix systems) [9]. These modified programs do not return any information to the administrator that involves specific files and processes used by the intruder. Such tools, however, are easily detected using file integrity checkers such as Tripwire [7].

Recently, a new type of rootkit has emerged. These rootkits are implemented as loadable kernel modules (LKMs). A loadable kernel module is an extension to the operating system (e.g., a device driver) that can be loaded into and unloaded from the kernel at runtime. Solaris and Linux are two popular operating systems that support this type of runtime kernel extension.

By implementing a rootkit as a kernel module, it is possible to modify critical kernel data structures (such as the system call table, the list of active processes, or the list of kernel modules) or intercept requests to the kernel regarding files and processes that are created by an intruder [10, 14, 15]. Once the kernel is infected, it is very hard to determine if a system has been compromised without the help of hardware extensions such as the Trusted Platform Module (TPM) [17, 12]. Therefore, it is important that mechanisms are in place to detect kernel rootkits and prevent their insertion into the kernel.

In this paper, we present a technique for the detection of kernel-level rootkits in the Linux operating system. The technique is based on static analysis of loadable kernel module binaries. More precisely, the use of behavioral specifications and symbolic execution allow one to determine if the module being loaded includes evidence of malicious intent.

The contribution of this approach is twofold. First, by using static analysis, our technique is able to determine if a kernel module is malicious *before* the kernel module is actually loaded into the kernel and executed. This is a major advantage, because once the kernel image has been modified it may become infeasible to perform dynamic analysis of the module's actions in a reliable way. Second, the technique is applied to the binary image of a module and does not require access to the module's source code. Because of this, the technique is widely applicable and it is possible to analyze the behavior of device drivers and other closed-source kernel components that are distributed in binary form only.

The rest of the paper is structured as follows. Section 2 discusses related work on rootkits and rootkit detection. Section 3 presents our approach to the detection of kernel-level rootkits. Then, Section 4 provides an experimental evaluation of the effectiveness and efficiency of our technique. Finally, Section 5 discusses possible limitations of the current prototype, while Section 6 briefly concludes.

2. Related Work

Kernel-level rootkits have been circulating in the underground hacker community for some time and in different forms [6]. In general, there are different means that can be used to modify kernel behavior.

The most common way of modifying the kernel is by inserting a loadable kernel module. The module has access to the symbols exported by the kernel and can modify any data structure or function pointer that is accessible. Typically, these kernel-level rootkits "hijack" entries in the system call table and provide modified implementations of the corresponding system call functions [10, 14]. These modified system calls often perform checks on the data passed back to a user process and can thus efficiently hide information about files and processes. An interesting variation is implemented by the `adore-ng` rootkit [15, 16]. In this case, the rootkit does not touch the system call table but hijacks the routines used by the Virtual File System (VFS), and, therefore, it is able to intercept (and modify) calls that access files in both the `/proc` file system and the root file system.

A related technique injects malicious code directly into existing kernel modules instead of providing a complete rootkit module. While this solution is in principle similar to the insertion of a rootkit kernel module, it has the advantage that the modification will survive a kernel reboot procedure if the modified module is automatically loaded in the kernel standard configuration. On the other hand, this technique requires the modification of a binary that is stored on the file system, and, therefore, it may be detected using integrity checkers.

Another way to modify the behavior of the kernel is to access kernel memory directly from user space through the `/dev/kmem` file. This technique (used, for example, by `SucKIT` [13]) requires the identification of data structures that need to be modified within the kernel image. However, this is not impossible; in particular, well-known data structures such as the system call table are relatively easy to locate.

Kernel-level rootkits can be detected in a number of different ways. The most basic techniques include searching for modified kernel modules on disk, searching for known strings in existing binaries, or searching for configuration files associated with specific rootkits. The problem is that when a system has been compromised at the kernel level, there is no guarantee that these detection tools will return reliable results. This is also true for signature-based rootkit detection tools such as `chkrootkit` [11] that rely on operating system services to scan a machine for indications of known rootkits.

To circumvent the problem of a possibly untrusted operating system, rootkit scanners such as `kstat` [4], `rkscan` [2], or `St. Michael` [8] follow a different approach. These tools are either implemented as kernel modules with direct access to kernel memory, or they analyze the contents of the kernel memory via `/dev/kmem`. Both techniques allow the programs to monitor the integrity of important kernel data structures without the use of system calls. For example, by comparing the system call addresses in the system call table with known good values (taken from the `/boot/System.map` file), it is possible to identify hijacked system call entries.

This approach is less prone to being foiled by a kernel-level rootkit because kernel memory is accessed directly. Nevertheless, changes can only be detected after a rootkit has been installed. In this case, the rootkit had the chance to execute arbitrary code in the context of the kernel. Thus, it is possible that actions have been performed to thwart or disable rootkit scanners. Also, rootkits can carry out changes at locations that are not monitored (e.g., task structures).

3. Rootkit Detection

The idea for our detection approach is based on the observation that the runtime behavior of regular kernel modules (e.g., device drivers) differs significantly from the behavior of kernel-level rootkits. We note that regular modules have different goals than rootkits, and thus implement different functionality.

The main contribution of this paper is that we show that it is possible to distinguish between regular modules and rootkits by statically analyzing kernel module binaries. The analysis is performed in two steps. First, we have to specify undesirable behavior. Second, each kernel module binary is statically analyzed for the presence of instructions sequences that implement these specifications.

Currently, our specifications are given informally, and the analysis step has to be adjusted appropriately to deal with new specifications. Although it might be possible to introduce a formal mechanism to model behavioral specifications, it is not necessary for our detection prototype. The reason is that a few general specifications are sufficient to accurately capture the malicious behavior of all LKM-based rootkits. Nevertheless, the analysis technique is powerful enough that it can be easily extended. This may become necessary when rootkit authors actively attempt to evade detection by changing the code such that it does not adhere to any of our specifications.

3.1. Specification of Behavior

A specification of malicious behavior has to model a sequence of instructions that is characteristic for rootkits but that does not appear in regular modules (at least, with a high probability). That is, we have to analyze the behavior of rootkits to derive appropriate specifications that can be used during the analysis step.

In general, kernel modules (e.g., device drivers) initialize their internal data structures during startup and then interact with the kernel via function calls, using both system calls or functions internal to the kernel. In particular, it is not often necessary that a module directly writes to kernel memory. Some exceptions include device drivers that read from and write to memory areas that are associated with a managed device and that are mapped into the kernel address space to provide more efficient access or modules that overwrite function pointers to register themselves for event callbacks.

Kernel-level rootkits, on the other hand, usually write directly to kernel memory to alter important system management data structures. The purpose is to intercept the regular control flow of the kernel when system services are requested by a user process. This is done in order to monitor or change the results that are returned by these services to the user process. Because system calls are the most obvious entry point for requesting kernel services, the earliest kernel-level rootkits modified the system call table accordingly. For example, one of the first actions of the `knark` [10] rootkit is to replace entries in the system call table with customized functions to hide files and processes.

In newer kernel releases, the system call table is no longer exported by the kernel, and thus it cannot be directly accessed by kernel modules. Therefore, alternative approaches to influence the results of operating system services have been proposed. One such solution is to monitor accesses to the `/proc` file system. This is accomplished by changing the function addresses in the `/proc` file system root node that point to the corresponding read and write functions. Because the `/proc` file system is used by many auditing tools to gather information about the system (e.g., about running processes, or open network connections), a rootkit can easily hide important information by filtering the output that is passed back to the user process. An example of this approach is the `adore-ng` rootkit [16], which replaces functions of the virtual file system (VFS) node of the `/proc` file system.

As a general observation, we note that rootkits perform writes to a number of locations in the kernel address space that are usually not touched by regular modules. These writes are necessary either to obtain control over system services (e.g., by changing the system call table, file system functions, or the list of active processes) or to hide the presence of the kernel rootkit itself (e.g., modifying the list of installed modules). Because write operations to operating system management structures are required to implement the needed functionality, and because these writes are unique to kernel rootkits, they present a salient opportunity to specify malicious behavior.

To be more precise, we identify a loadable kernel module as a rootkit based on the following two behavioral specifications:

1. The module contains a data transfer instruction that performs a write operation to an illegal memory area, or
2. the module contains an instruction sequence that i) uses a *forbidden* kernel symbol reference to calculate an address in the kernel's address space and ii) performs a write operation using this address.

Whenever the destination address of a data transfer can be determined statically during the analysis step, it is possible to check whether this address is within a legitimate area. The notion of legitimate areas is defined by a white-list that specifies the kernel addressed that can be safely written to. For our current system, these areas include function pointers used as event call-back hooks (e.g., `br_ioctl_hook()`) or exported arrays (e.g., `blk_dev`).

One drawback of the first specification is the fact that the destination address must be derivable during the static analysis process. Therefore, a complementary specification is introduced that checks for writes to any memory address that is calculated using a forbidden kernel symbol.

A kernel symbol refers to a kernel variable with its corresponding address that is exported by the kernel (e.g., via `/proc/ksysm`). These symbols are needed by the module loader, which loads and inserts modules into the kernel address space. When a kernel module is loaded, all references to external variables that are declared in this module but defined in the kernel (or in other modules) have to be *patched* appropriately. This patching process is performed by substituting the place holder addresses of the declared variables in the module with the actual addresses of the corresponding symbols in the kernel.

The notion of forbidden kernel symbols can be based on black-lists or white-lists. A black-list approach enumerates all forbidden symbols that are likely to be misused by rootkits, for example, the system call table, the root of the `/proc` file system, the list of modules, or the task structure list. A white-list, on the other hand, explicitly defines acceptable kernel symbols that can legitimately be accessed by modules. As usual, a white-list approach is more restrictive, but may lead to false positives when a module references a legitimate but infrequently used kernel symbol that has not been allowed previously. However, following the principle of fail-safe defaults, a white-list also provides greater assurance that the detection process cannot be circumvented.

Note that it is not necessarily malicious when a forbidden kernel symbol is declared by a module. When such a symbol is not used for a *write* access, it is not problematic. Therefore, we cannot reject a module as a rootkit by checking the declared symbols only.

Also, it is not sufficient to check for writes that target a forbidden symbol directly. Often, kernel rootkits use such symbols as a starting point for more complex address calculations. For example, to access an entry in the system call table, the system call table symbol is used as a base address that is increased by a fixed offset. Another example is the module list pointer that is used to traverse a linked list of module elements until the one that should be removed is reached. Therefore, a more extensive analysis has to be performed to also track indirect uses of forbidden kernel symbols for write accesses.

A clever intruder could use an attack in which two modules cooperate to evade detection. In this attack, a first module only reads the sensitive address (e.g., the address of the system call table) and then it exports a function to access the address. A second module then reads the sensitive address indirectly from the first module and uses it for an illegal write access. To thwart this evasion, all symbols and return values of functions declared by other kernel modules are also marked as forbidden. Thus, when the second module accesses the function exported by the first module, the return value is tagged as forbidden and also subsequent write operations based on this value would result in an alarm.

Naturally, there is an arms-race between rootkits that use more sophisticated methods to obtain kernel addresses and our detection system that relies on specifications of malicious behavior. For current rootkits, our basic specifications allow for reliable detection with no false positives (see Section 4 for details). However, it might be possible to circumvent these specifications. In that case, it is necessary to provide more elaborate descriptions of malicious behavior.

Note that our behavioral specifications have the advantage that they provide a general model of undesirable behavior. That is, these specifications characterize an entire class of malicious actions. This is different from fine-grained specifications that need to be tailored to individual kernel modules. In addition, behavioral specifications have the potential to detect previously unknown rootkits. In contrast to approaches that rely on anti-virus-like pattern matching techniques, our tool can detect any kernel-level rootkit that satisfies our assumptions.

3.2. Symbolic Execution

Based on the specifications introduced in the previous section, the task of the analysis step is to statically check the module binary for instructions that correspond to these specifications. When such instructions are found, the module is labeled as a rootkit.

We perform analysis on binaries using symbolic execution. Symbolic execution is a static analysis technique in which program execution is simulated using symbols, such as variable names, rather than actual values for input data. The program state and outputs are then expressed as mathematical (or logical) expressions involving these symbols. When performing symbolic execution, the program is basically executed with all possible input values simultaneously, thus allowing one to make statements about the program behavior.

One problem with symbolic execution is the fact that it is impossible to make statements about arbitrary programs in general, due to the halting problem. However, when the completeness requirement is relaxed, it is often possible to obtain useful results in practice. Relaxing the completeness requirement implies that the analysis is not guaranteed to detect malicious instructions sequences in all cases. However, this can be tolerated when most relevant instances are found.

In order to simulate the execution of a program, or, in our case, the execution of a loadable kernel module, it is necessary to perform two preprocessing steps.

First, the code sections of the binary have to be disassembled. In this step, the machine instructions have to be extracted and converted into a format that is suitable for symbolic execution. That is, it is not sufficient to simply print out the syntax of instructions, as done by programs such as `objdump`. Instead, the type of the operation and its operands have to be parsed into an internal representation. The disassembly step is complicated by the complexity of the Intel x86 instruction set, which uses a large number of variable length instructions and many different addressing modes for backwards compatibility reasons.

In the second preprocessing step, it is necessary to adjust address operands in all code sections present. The reason is that a Linux loadable kernel module is merely a standard ELF relocatable object file. Therefore, many memory address operands have not been assigned their final values yet. These memory address operands include targets of jump and call instructions but also source and destination locations of load, store, and move instructions.

For a regular relocatable object file, the addresses are adjusted by the linker. To enable the necessary link operations, a relocatable object also contains, besides regular code and data sections, a set of relocation entries. Note, however, that kernel modules are not linked to the kernel code by a regular linker. Instead, the necessary adjustment (i.e., patching) of addresses is performed during module load time by a special module loader. For Linux kernels up to version 2.4, most of the module loader ran in user space; for kernels from version 2.5 and up, much of this functionality was moved into the kernel. To be able to simulate execution, we perform a process similar to linking and substitute place holders in instruction operands and data locations with the real addresses. This has the convenient side-effect that we can mark operands that represent forbidden kernel symbols so that the symbolic execution step can later trace their use in write operations.

When the loadable kernel module has been disassembled and the necessary address modifications have occurred, the symbolic execution process can commence. To this end, an initial *machine state* is created and execution starts with the module's initialization routine, called `init_module()`.

Handling Machine State The machine state represents a snapshot of the system during symbolic execution. That is, the machine state contains all possible values that could be present in the processor registers and the memory address space of the running process at a certain point during the execution process. Given the notion of a machine state, an instruction can then be defined as a function that maps one machine state into another one. This mapping will reflect the effect of the instruction itself (e.g., a data value is moved from one register to another), but also implicit effects such as incrementing the instruction pointer.

When complete knowledge about the processor and memory state is available, and given the absence of any input and external modifications of the machine state, it would be possible to deterministically simulate the execution of a module. However, in our case, the complexity of such a complete simulation would be tremendous. Therefore, we introduce a number of simplifications that improve the efficiency of the symbolic execution process, while retaining the ability to detect most malicious instruction sequences.

A main simplification is the fact that we consider the initial configuration of the memory content as unknown. This means that whenever a value is taken from memory, a special *unknown token* is returned. However, it does not imply that all loads from memory are automatically transformed into unknown tokens. When known values are stored at certain memory locations, these values are remembered and can subsequently be loaded. This is particularly common for the stack area when return addresses are pushed on the stack by a call operation and later loaded by the corresponding return instruction.

During symbolic execution, we can simulate the effect of arithmetic, logic, and data transfer instructions. To this end, the values of the operands are calculated and the required operation is performed. When at least one of the operands is an unknown token, the result is also unknown.

Another feature is a *tainting* mechanism that tags values that are related to the use of forbidden kernel symbols. Whenever a forbidden symbol is used as an operand, even when its value is unknown, the result of the operation is marked as tainted. Whenever a tainted value is later used by another instruction, its result becomes tainted as well. This allows us to detect writes to kernel memory that are based on the use of forbidden symbols.

For the initial machine state, we prepare the processor state such that the instruction pointer register is pointing to the first instruction of the module's initialization routine, while the stack pointer and the base (i.e., frame) pointer register refer to valid addresses on the kernel stack. All other registers and the entire memory is marked as unknown.

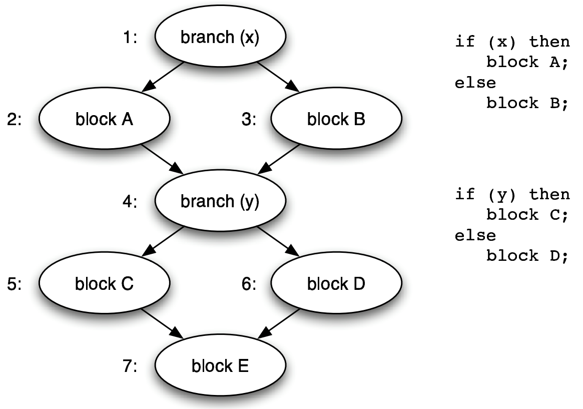
Then, instructions are sequentially processed and the machine state is updated accordingly. For each data transfer, it is checked whether data is written to kernel memory areas that are not explicitly permitted by the white-list, or whether data is written to addresses that are tainted because of the use of forbidden symbols.

The execution of instructions continues until execution terminates with the final return instruction of the initialization function, or until a control flow instruction is reached.

Handling Control Flow Control flow instructions present problems for our analysis when they have two possible successor instructions (i.e., continuations). In this case, the symbolic execution process must either select a continuation to continue at, or a mechanism must be introduced to save the current machine state at the control flow instruction and explore both paths one after the other. In this case, the execution first continues with one path until it terminates and then backs up to the saved machine state and continues with the other alternative.

The only problematic type of control flow instructions are conditional branches. This is because it is not always possible to determine the real target of such a branch operation statically. The most common reason is that the branch condition is based on an unknown value, and thus, both continuations are possible. Neither unconditional jumps nor call instructions are a difficulty because both only have a single target instruction where the execution continues. Also, calls and the corresponding return operations are not problematic because they are handled correctly by the stack, which is part of the machine state.

Because malicious writes can occur on either path after a conditional branch, we chose to save the machine state at these instructions and then consecutively explore both alternative continuations. Unfortunately, this has a number of problems that have to be addressed.



```
if (x) then
   block A;
else
   block B;

if (y) then
   block C;
else
   block D;
```

Figure 1. Example control flow graph.

One problem is caused by the exponential explosion of possible paths that need to be followed. Consider the case of multiple branch instructions that are the result of a series of if-else constructs in the corresponding source code (see Figure 1). After each if-else block, the control flow joins. In this example, the machine state needs to be saved at node 1, at the `branch(x)` instruction. Then, the first path is taken via node 2. The machine state is saved a second time at node 4 and both the left and the right path are subsequently executed (using the state previously saved at node 4). Then, the execution process is rewinded to the first check point, and continues via the right path (i.e., via node 3). Again, the machine state needs to be saved at node 4, and both alternatives are followed a second time. Thus a total of four paths have to be explored as a result of only two branch instructions.

Also, it is possible that impossible paths are being followed. If, in our example, both the `branch(x)` and the `branch(y)` instructions evaluated to the same boolean value, it would be impossible that execution flows through nodes 2 and 6, or through nodes 3 and 5. For our prototype, the path explosion problem and impossible paths have not caused any difficulties (refer to Section 4 for the evaluation of our system). This is due to the limited size of the kernel modules. Therefore, we save the machine state at every conditional branch instruction and explore both alternative continuations.

Another problem is the presence of loops. Because the machine state is saved at every branch instruction and both alternatives are explored one after another, the existence of a loop would prevent the execution process from terminating. The reason is that both continuations of the branch that corresponds to the loop termination condition are explored (i.e., the loop body and the code path after the loop). When the path that follows the loop body eventually reaches the loop termination condition again, the state is saved a second time. Then, as usual, both alternative continuations are explored. One of these continuations is, of course, the loop body that leads back to the loop termination condition, where the process repeats.

To force termination of our symbolic execution process, it is necessary to remove control flow loops. Note that it is not sufficient to simply mark nodes in the control flow that have been previously processed. The reason is that nodes can be legitimately processed multiple times without the existence of loops. In the example shown in Figure 1, the symbolic execution processes node 4 twice because of the joining control flows from node 2 and node 3. However, no loop is present, and the analysis should not terminate prematurely when reaching node 4 for the second time.

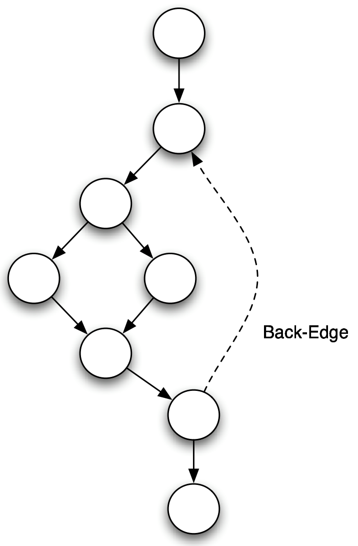


Figure 2. Control flow graph with loop.

Instead, a more sophisticated algorithm based on the control flow graph of the binary is necessary. In [1], a suitable algorithm is presented that is based on dominator trees. This algorithm operates on the control flow graph and can detect (and remove) the back-edges of loops. Simply speaking, a back-edge is the jump from the end of the loop body back to the loop header, and it is usually the edge that would be identified as the “loop-defining-edge” by a human looking at the control flow graph. For example, Figure 2 shows a control flow graph with a loop and the corresponding back-edge.

For our system, we first create a control flow graph of the kernel module code after it has been preprocessed. Then, a loop detection algorithm is run and the back-edges are detected. Each conditional branch instruction that has a back-edge as a possible continuation is tagged appropriately. During symbolic execution, no machine state is saved at these instructions and processing continues only at the non-back-edge alternative. This basically means that a loop is executed at most once by our system. For future work, we intend to replace this simple approach by more advanced algorithms for symbolic execution of loops. Note, however, that more sophisticated algorithms that attempt to execute a loop multiple times will eventually hit the limits defined by the halting problem. Thus, every approach has to accept a certain degree of incompleteness that could lead to incorrect results.

A last problem are indirect jumps that are based on unknown values. In such cases, it might be possible to heuristically choose possible targets and speculatively continue with the execution process there. In our current prototype, however, we simply terminate control flow at these points. The reason is that indirect jumps based on unknown values almost never occurred in our experiments.

4. Evaluation

The proposed rootkit detection algorithm was implemented as a user space prototype that simulated the object parsing and symbol resolution performed by the existing kernel module loader before disassembling the module and analyzing the code for the presence of malicious writes to kernel memory. The prototype implementation was evaluated with respect to its detection capabilities and performance impact on production systems. To this end, an experiment was devised in which the prototype was run on several sets of kernel modules. Detection capability for each set was evaluated in terms of false positive rates for legitimate modules, and false negative rates for rootkit modules. Detection performance was evaluated in terms of the total execution time of the prototype for each module analyzed. The evaluation itself was conducted on a testbed consisting of a single default Fedora Core 1 Linux installation on a Pentium IV 2.0 GHz workstation with 1 GB of RAM.

4.1. Detection Results

For the detection evaluation, three sets of kernel modules were created. The first set comprised the `knark` and `adore-ng` rootkits, both of which were used during development of the prototype. As mentioned previously, both rootkits implement different methods of subverting the control flow of the kernel: `knark` overwrites entries in the system call table to redirect various system calls to its own handlers, while `adore-ng` patches itself into the VFS layer of the kernel to intercept accesses to the `/proc` file system. Since each rootkit was extensively analyzed during the prototype development phase, it was expected that all malicious kernel accesses would be discovered.

The second set consisted of a set of seven additional popular rootkits downloaded from the Internet, described in Table 1. Since these rootkits were not analyzed during the prototype development phase, the detection rate for this group can be considered a measure of the generality of the detection technique as applied against previously unknown rootkits that utilize similar means to subvert the kernel as `knark` and `adore-ng`.

The final set consisted of a control group of legitimate kernel modules, namely the entire default set of kernel modules for the Fedora Core 1 Linux x86 distribution. This set includes 985 modules implementing various components of the Linux kernel, including networking protocols (e.g., IPv6), bus protocols (e.g., USB), file systems (e.g., EXT3), and device drivers (e.g., network interfaces, video cards). It was assumed that no modules incorporating rootkit functionality were present in this set.

Table 2 presents the results of the detection evaluation for each of the three sets of modules. As expected, all malicious writes to kernel memory by both `knark` and `adore-ng` were detected, resulting in a false negative rate of 0% for both rootkits. All malicious writes by each evaluation rootkit were detected as well, resulting in a false negative rate of 0% for this set. We interpret this result as an indication that the detection technique generalizes well to previously unseen rootkits. Finally, no malicious writes were reported by the prototype for the control group, resulting in a false positive rate of 0%. We thus conclude that the detection algorithm is completely successful in distinguishing rootkits exhibiting specified malicious behavior from legitimate kernel modules, as no misclassifications occurred during the entire detection evaluation.

```
scan: initializing scan for rootkits/all-root.o
scan: loading kernel symbol table from boot/System.map
scan: kernel memory configured [c0100000-c041eaf8]
scan: resolving external symbols in section .text
scan: disassembling section .text
scan: performing scan from [.text+40]
scan: WRITE TO KERNEL MEMORY [c0347df0] at [.text+50]
scan: 1 malicious write detected, denying module load
```

Figure 3. `all-root` **rootkit analysis.**

Rootkit	Technique	Description
adore	syscalls	File, directory, process, and socket hiding
		Rootshell backdoor
all-root	syscalls	Gives all processes UID 0
kbdv3	syscalls	Gives special user UID 0
kkeylogger	syscalls	Logs keystrokes from local and network logins
rkit	syscalls	Gives special user UID 0
shtroj2	syscalls	Execute arbitrary programs as UID 0
synapsys	syscalls	File, directory, process, socket, and module hiding
		Gives special user UID 0

Table 1. Evaluation rootkits.

Module Set	Modules Analyzed	Detections	Misclassification Rate
Development rootkits	2	2	0 (0%)
Evaluation rootkits	6	6	0 (0%)
Fedora Core 1 modules	985	0	0 (0%)

Table 2. Detection results.

To verify that the detection algorithm performed correctly on the evaluation rootkits, traces of the analysis performed by the prototype on each rootkit were examined with respect to the corresponding module code. As a simple example, consider the case of the `all-root` rootkit, the analysis trace of which is shown in Figure 3. From the trace, we can see that one malicious kernel memory write was detected at `.text+50` (i.e., at an offset of 50 bytes into the `.text` section). By examining the disassembly of the `all-root` module, the relevant portion of which is shown in Figure 4, we can see that the overwrite occurs in the module's initialization function, `init_module()`[1]. Specifically, the `movl` instruction at `.text+50` is flagged as a malicious write to kernel memory. Correlating the disassembly with the corresponding rootkit source code, shown in Figure 5, we can see that this instruction corresponds to the write to the `sys_call_table` array to replace the `getuid()` system call handler with the module's malicious version at line 4. Thus, we conclude that the rootkit's attempt to redirect a system call was properly detected.

```
00000040 <init_module>:
 40: a1 60 00 00 00        mov  0x60,%eax
 45: 55                    push %ebp
 46: 89 e5                 mov  %esp,%ebp
 48: a3 00 00 00 00        mov  %eax,0x0
 4d: 5d                    pop  %ebp
 4e: 31 c0                 xor  %eax,%eax
 50: c7 05 60 00 00 00 00  movl $0x0,0x60
 57: 00 00 00
 5a: c3                    ret
```

Figure 4. `all-root` module disassembly.

```
1 int init_module(void)
2 {
3   orig_getuid =
        sys_call_table[__NR_getuid];
4   sys_call_table[__NR_getuid] =
        give_root;
5
6   return 0;
7 }
```

Figure 5. `all-root` initialization function.

4.2. Performance Results

For the performance evaluation, the elapsed execution time of the analysis phase of the prototype was recorded for all modules, legitimate and malicious. Time spent parsing the object file and patching relocation table entries into the module was excluded, as these functions are already performed as part of the normal operation of the existing module loader. The goal of the evaluation was to provide some indication about the performance overhead introduced by the detection process in the loading of a module in a production kernel. Note that as mentioned previously, no runtime over-

1 Note that this disassembly was generated prior to kernel symbol resolution, thus the displayed read and write accesses are performed on place holder addresses. At runtime and for the symbolic execution, the proper memory address would be patched into the code.

head is generated by our technique after the module has been loaded.

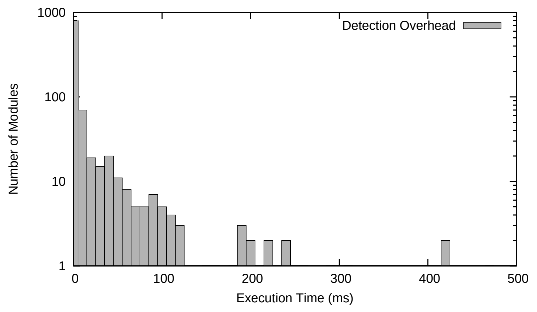


Figure 6. Detection overhead on module load.

Figure 6 shows the elapsed execution time of all evaluated modules, discretized into log-scale buckets with a width of 10 ms. As we can see, the vast majority of modules would experience a delay of 10 ms or less during module load. Several modules with more complex initialization procedures (and thus complex control flow graphs) required more time to fully analyze, but as can be seen in Table 3, the detection algorithm never spent more than 420 ms to classify a module as malicious or legitimate. Thus, we conclude that the impact of the detection algorithm on the module load operation is acceptable for a production system.

Minimum	Maximum	Median	Std. Deviation
0.00 ms	420.00 ms	0.00 ms	39.83

Table 3. Detection overhead statistics.

5. Discussion

Our prototype is a user-space program that statically analyzes Linux loadable kernel modules for the presence of rootkit functionality. These modules have to be ELF object files that are compiled for the Intel x86 architecture.

The limitation on the classes of modules that can be analyzed stems from the fact that a kernel module needs to be parsed and its code sections disassembled before the actual analysis can start. Therefore, additional parsing and disassembly routines would be necessary to process different object file formats or instruction sets. Because a vast majority of Linux systems run on Intel x86 machines, and because Linux kernel modules have to be provided as ELF object files, we developed our prototype for this combination. The analysis technique itself, however, can be readily extended to other systems.

Our tool is currently available as a user program only. In order to provide automatic protection from rootkits, it would be necessary to integrate our analyzer into the kernel's module loading infrastructure. As an additional requirement, the analyzer must not be bypassable when a process with root permissions attempts to load a module. The reason is that kernel modules can only be inserted by the root user. Thus, the threat model has to assume that the attacker has superuser privileges when attempting to load a kernel module.

Up until Linux 2.4, most work of the module loading process was done in user space, using the `insmod` program. In this case, adding our checker to `insmod` would not be useful because an attacker can simply supply a customized version without checks. The solution is to move the analyzer code into kernel space. Interestingly, starting from Linux 2.5, most of the module loading code has been moved into the kernel space, providing an optimal place to add our checks.

Unfortunately, mechanisms have been proposed to inject code directly into the kernel without using the module loading interface. These ideas originated from the fact that some system administrators disabled the module loading functionality as a defense against kernel-level rootkits. These mechanisms operate by writing the code directly into kernel space via the `/dev/kmem` device, completely bypassing the module loading code.

In our opinion, a sensible and secure solution would disallow modifications of kernel memory via `/dev/kmem`, a feature that is already offered by Linux security solutions such as grsecurity [5]. In addition, our kernel-level rootkit analysis system would operate in kernel context behind the module loading interface, thus having the opportunity to statically scan each module before it gets to run as part of the kernel.

A possible way for rootkits to evade the behavioral specification that is based on forbidden kernel symbols (see Section 3 for details) is to stop using these symbols. However, to perform the necessary modifications of the kernel data structures or function pointers, their addresses are needed. Therefore, alternative approaches to resolving these addresses are required. One option is to use a brute force guessing technique that works by scanning the kernel memory for the occurrence of "known content" that is stored at the target location. This is particularly effective for the system call table. The reason is that its content is known because system call table entries are pointers to handler functions whose symbols are exported.

Although a brute force guessing approach might not always be suitable, we propose the addition of a specification that considers the scanning of kernel memory as another indication of the presence of a rootkit. This specification checks for loops that, starting from any kernel symbol, sequentially read data and compare this data to constant values. Also, note that the specification that checks for illegitimate memory accesses based on actual destination addresses works independently of kernel symbols referenced by the module.

6. Conclusions

Rootkits are powerful attack tools that are used by intruders to hide their presence from system administrators. Kernel-level rootkits, in particular, directly modify the kernel, and, therefore, can intercept and prevent any attempt of an administrator to determine if the security of the system has been violated. Because of this, it is important to devise mechanisms that can protect the integrity of the kernel even in the aftermath of the compromise of the administrator account.

This paper presents a technique that is based on static analysis to identify instruction sequences that are an indication of rootkits. Informal behavioral specifications define such characteristic instruction sequences as data transfer operations that write to certain illegitimate kernel memory areas. Symbolic execution is then used to simulate the execution of the kernel module to detect instructions that fulfill these specifications. Through this method, it is possible to detect malicious behavior *before* a module is loaded into the kernel, and, in addition, it is possible to operate on closed-source components, such as proprietary drivers.

We implemented our technique in a prototype tool and we evaluated both the effectiveness and the performance of the tool with respect to nine real-world rootkits as well as the complete set of 985 legitimate kernel modules that are included with the Fedora Core 1 Linux distribution. The results show that all tested rootkits were successfully identified, and no false positives were raised on legitimate modules. We thus conclude that the technique can reliably detect malicious kernel modules and, therefore, it represents a useful tool to harden the operating system kernel. In addition, we show that detection can be done efficiently, despite the application of a potentially expensive static analysis technique.

Future work will be centered on devising a more formal description of the aspects that characterize rootkit-like behavior. In addition, we plan to study how attacks that attempt to bypass our detection procedures can be prevented. Finally, we intend to integrate the detection component into the kernel module loader infrastructure as a step towards preparing the system for general usage.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484 and by the National Science Foundation under grants CCR-0209065 and CCR-0238492.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques, and Tools*. World Student Series of Computer Science. Addison Wesley, 1986.

[2] S. Aubert. rkscan: Rootkit Scanner. `http://www.hsc.fr/ressources/outils/rkscan/index.html.en`, 2004.

[3] Black Tie Affair. Hiding Out Under UNIX. *Phrack Magazine*, 3(25), 1989.

[4] FuSyS. Kstat v. 1.1-2. http://s0ftpj.org/, November 2002.

[5] grsecurity. An innovative approach to security utilizing a multi-layered detection, prevention, and containment model. `http://www.grsecurity.net/`, 2004.

[6] Halflife. Abuse of the Linux Kernel for Fun and Profit. *Phrack Magazine*, 7(50), April 1997.

[7] G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. Technical report, Purdue University, Nov. 1993.

[8] T. Lawless. St. Michael and St. Jude. `http://sourceforge.net/projects/stjude/`, 2004.

[9] T. Miller. T0rn rootkit analysis. `http://www.ossec.net/rootkits/studies/t0rn.txt`.

[10] T. Miller. Analysis of the KNARK Rootkit. `http://www.ossec.net/rootkits/studies/knark.txt`, 2004.

[11] N. Murilo and K. Steding-Jessen. Chkrootkit v. 0.43. `http://www.chkrootkit.org/`.

[12] D. Safford. The Need for TCPA. IBM White Paper, October 2002.

[13] sd and devik. Linux on-the-fly kernel patching without LKM. *Phrack Magazine*, 11(58), 2001.

[14] Stealth. adore. `http://spider.scorpions.net/~stealth`, 2001.

[15] Stealth. Kernel Rootkit Experiences and the Future. *Phrack Magazine*, 11(61), August 2003.

[16] Stealth. adore-ng. `http://stealth.7350.org/rootkits/`, 2004.

[17] TCG. Trusted Computing Group Home. `https://www.trustedcomputinggroup.org/home`, 2004.