Analysis of Rootkits: Attack Approaches and Detection Mechanisms

Alkesh Shah
Georgia Institute of Technology
salkesh@cc.gatech.edu
Advisor: Prof. Dr. Jonathon Giffin

Abstract

Attackers have increasingly started deploying toolkits, which compromise the system such that their malicious code is hidden from the users, the standard system tools as well as various malicious code detection tools. These toolkits, called as Rootkits, use different mechanisms to achieve this kind of stealth. Some of the mechanisms that rootkits have used include replacing system binaries, replacing standard system libraries with corresponding trojanised versions and subverting the kernel data structures. This report explores the various approaches that have been used by the rootkits, which are in existence today as well as the difference combating mechanism that have been used by the detection tools. The report also gives some future approaches that can be used for detecting rootkits.

1. Introduction

Software today continues to have bugs in it and the attackers exploit these vulnerabilities to gain access to the system. Also, additionally in order to hide their activities, the attackers have started injecting toolkits, which replace or modify the system binaries, replace system libraries with trojanised versions, modify the kernel data structures to provide the stealth. These toolkits are called as Rootkits. The threat of rootkits is even more since the actions of the attacker can go undetected by many detection tools. Attackers have used different ways of writing rootkits for hiding malicious code. Recently, rootkits having the capability of subverting the kernel have also come into fore. This kernel level rootkits provide the attacker with all the power in the system. At the same time, specialized rootkit detection tools have been developed over the past few years that detect such rootkits using different approaches. However, all the tools have not been able to identify all the types of kernel rootkits. In this report, we first give a broad classification of the different rootkits. We then describe the attack approaches taken by kernel rootkit writers in Section 3. In Section 4, we describe some of the detection tools and the mechanisms used by them. Finally, we conclude by giving future directions for rootkit detection mechanism.

2. Rootkit Classification

iDEFENSE Labs have classified rootkits into three primary classes -

1. Binary Rootkits
2. Library Rootkits
3. Kernel Rootkits

Binary Rootkits:
This type of rootkits replace the binaries of executable system utilities like ps, ls, netstat, … This trojanised binaries effectively hide the processes, files, ports opened by a backdoor, etc.

Kernel Rootkits:
This type of rootkits hook into the kernel and modify system calls. They can also modify the kernel memory image, system call table and other parts of the kernel. To inject such kind of rootkits Loadable Kernel Modules is an effective mechanism. Another way to inject rootkits is by exploiting some vulnerability in the system the attacker can gain root access, which allows him to hook into the various parts of the kernel. The kernel level rootkits are more dangerous as they compromise the very core of the operating system. As a result, some of the security packages, which rely on the kernel data structures for detecting malicious code will not be effective.

Library Rootkits:
This type of rootkits replace the standard system libraries (for ex. libproc.a, which is used for conveying information from the /proc kernel space to user space processes like ps, ls, netstat). Another way library rootkits can infect is by providing a custom library, which is added in files like /etc/ld.so.preload. It can ensure that the custom library, which provides system library functionality, is loaded before any other library is loaded.

3. Attack Approaches

In this section, we describe the different approaches that the kernel rootkits use to hook into the kernel. For the kernel rootkits to be able to modify the data structures of the kernel access to the kernel data is required. In other words, they need to execute from the kernel mode or they need to have root access. Drivers (in case of Windows) or Linux Kernel Modules (LKM) provide the one of the paths for the rootkits to be introduced into the kernel. The LKMs allow the running operating system to be extended dynamically and can also modify the kernel data structures. Once introduced rootkits modify the kernel data structures like system call table, task lists, etc. for hiding the modules. For gaining root access the attackers exploit the vulnerabilities that exists in the various software tools available today. After gaining the root access the attackers download the rootkit and install (or inject them in the kernel).

From observing the characteristics of the rootkits in wild today we have identified some of the means used by the rootkits (as noted below) –

A. Modifying of data structures, which display the processes currently running on the system
B. System call interception
   - modifying the system call table
   - modifying the system call handler code.
C. Interrupt Hooking
   - modifying the interrupt descriptor table
   - modifying the interrupt handler (in particular for the system calls)
D. Modifying the kernel memory image (/dev/kmem)
E. Intercepting calls handled by the VFS.
F. Virtual memory subversion

A. Process Hiding:

It has been observed that the kernel uses different data structures for displaying a list of processes that are executing on the system and for scheduling the processes. Rootkits have used this to their

advantage by providing the capability to hide the malicious process from the display list. In case of a Windows system there exists an EPROCESS structure (process descriptor) for storing the information of a process that is to be displayed. All the processes are linked using a doubly linked list. The Windows traverses this doubly-linked list while displaying a list of active processes (in Task Manager). Butler et. al [1] showed that by modifying the pointers of this list and EPROCESS structure can be unlinked from the list and thus hide the malicious process. As separate data structures are used in the kernel for scheduling the processes, this modification of pointers does not affect the scheduling of the malicious process. In case of Linux a similar mechanism is employed. The process descriptor, in case of Linux, is identified by the 'task_struct' structure. A 'task_array' structure, which contains a pointer to each process descriptor and a doubly linked list is used. Detailed explanation of dynamically modifying kernel objects to hide the process can be found in the paper written by Butler et. al [1] .

B. System Call Interception:

There are generally two modes of execution in most modern processors, viz. User mode and Kernel Mode. The user mode of execution cannot execute privileged instructions and also does not have access to the kernel data. The kernel mode of execution executes at the highest privilege generally and has the capability to access the entire address space. In order to perform some privileged operation, a program in the user mode executes a system call, which executes in the kernel mode. The system calls can be intercepted in two ways -

i. *Modifying the entry for the system call handler's address in the system call table*
   System call table contains addresses for the system call handler for each system call. When a system call is to be executed this system call table is used to jump to the code corresponding to the service requested. A rootkit can modify the entry in the system call table for a particular service to make it execute the malicious handler code.

ii. *Modifying the system call handler code*
   The first few instructions of system call handler for most of the services are similar. A rootkit can rewrite this first few instructions of the handler code such that it jumps it's execution to the rootkit's handler and from there it may invoke the actual system call handler. Thus the rootkit's handler has the ability to intercept data provided to the service's handler as well as to filter out the data returned by the service's handler.

An example of a rootkit that modifies the system call table entries is Creed's Knark rootkit, which redirects the system call table entries for fork, read and execve.

C. Interrupt Hooking:

Whenever an interrupt is raised in the system an interrupt handler is invoked, which does certain types of operations depending on the interrupt. For each interrupt there is a unique handler the address of which is stored in the Interrupt Descriptor Table (IDT). To invoke a system call, the system call number and the arguments to the system call are set up in the registers. A processor software interrupt is then raised, which results in switching from the user mode to the kernel mode and executing of the interrupt handler. The interrupt handler then reads the system call number and the corresponding index in the system call table to invoke the system call handler's code. Thus, similar to system call interception, the rootkit code can be executed by -

i. *Modifying the entry in the IDT*
   Similar to the system call table hooking, the IDT entry containing the address of the interrupt handler can be modified to execute the rootkit code. This is possible because even though the IDT is initialized at the boot time by the BIOS, Linux does it once again and takes control of the IDT.

ii. *Modifying the first few instructions of the interrupt handler*
   In most of the interrupt handlers, the first few instructions push the handler code's address into the stack and then jump to a error_code routine from where the handler is invoked. This first few instructions can be modified such that the rootkit's code is executed and from the rootkit code the real handler is invoked

The process of hooking the IDT is explained in an article by kad[13]

D. Modifying kernel memory image:

In Linux, the special file (/dev/kmem) contains the running kernel's memory image. This special file provides an interface for writing into the kernel. A kernel can be subverted by using such an interface to modify the current kernel. This method will not survive a reboot but most of the server class machines are generally not rebooted frequently. An example of a rootkit that works in this way is the SucKIT rootkit [2]. This rootkit works even if the kernel module support is disabled since it uses the 'kmem' interface. It works by replacing the reference to the system call table with a reference pointing to a new system call table. This new system call table contains entries for malicious system call handlers along with the legitimate ones. The original system call table (whose address can be obtained from the System.map file) still exists in the memory and its address has also not changed. Also, unlike the system call interception method, none of the system call handler's first few instructions have been replaced.

E. Intercepting calls handled by VFS

Another point of attack for the kernel rootkits is the Virtual File System (VFS) layer. The VFS layer in the kernel handles system calls related to the standard Unix file system. It also handles calls related to other special file systems like /proc. The 'adore-ng' rootkit compromises the system at the VFS layer. It redirects the reference to *proc_root_lookup* to a malicious lookup function and this redirection takes place in the kernel dynamic data section [2]. As can be seen, rootkits can replace the handler routines at the VFS layer with malicious routines, which can hide information by filtering the data. Also, this type of technique does not involve any system call hooking.

F. Virtual Memory Subversion:

Shadow Walker, by Butler and Sparks, targets the virtual memory subsystem of the kernel to compromise the system [3]. This proof of concept implementation raises the level of stealth achieved by rootkits and has been used in conjunction with the 'FU' rootkit to develop a modified 'FU' rootkit. Shadow Walker uses the approach of trapping the memory accesses by hooking the Page Fault handler. In other words, it can transparently control the contents of the memory as viewed by other applications (even scanners/detectors) and kernel drivers. When a detector tries

to read a region of memory modified by the rootkit, the rootkit traps the access and provides the detector with a 'normal' view of the memory.

As can be seen from the above examples, kernel rootkits have been developed to attack system call table, IDT, VFS and the virtual memory subsystem. There are a huge number of data structures that exist in the kernel and subsequently rootkits can have a wide number of targets to compromise the kernel. We now proceed to the next section, which describes different approaches that have been used by a number of rootkit detectors that have been developed over the recent times.

4. Detection Mechanisms

Different approaches have been devised for dealing with kernel rootkits depending on the attack approach taken into consideration. However, not all the attacks are covered by a single mechanism and it is believed that the best way to approach rootkit detection is to use different methods for detecting if the system has been compromised. We outline some of the mechanisms that have been used to detect the attack approaches described above.

A. Detour Functions

This approach is directed towards detecting hidden processes. It works by introducing detour functions for intercepting arbitrary Win32 binary functions (in case of linux the library used is Injectso). The detour functions replace the first few instructions of the target function and unconditionally jump to the detour function. This detour function provide checks (for EPROCESS blocks in case of Windows) to ensure that the forward (FLINK) and the backward (BLINK) pointers are properly pointing. This detour functions are inserted in execution time so the image of the binary functions on the disk is not modified [1]. In general, this approach should be possible for detecting modifications to dynamic kernel data structures. However, the checks in the detour functions will have to be customized depending on the data structures monitored.

B. Diff-based approach

This approach is used by Blacklight[4] and IceSword[5] to detect hidden processes. This approach uses kernel data structures to obtain a view of the processes running in the system. It then invokes a standard API function to get the API view of the processes running in the system and compares the two results. A difference in the results indicates a possible hidden process. In case of Windows, the tools mentioned use PspCidTable, which is a handle table for process and thread client IDs. Blacklight detects hidden processes by looping through a valid range of process ids (0-0x41dc) and retrieve the handles for the process that exist. It then compares the list with the list retrieved by calling CreateToolhelp32Snapshot. This is a Win32 API function that takes a snapshot of all the running processes in the system. Any differences between the two lists indicate that there is a hidden process [6]. In general, diff-based approach involves traversing the data structures to get a view of the system and then comparing that view with the view obtained from system utilities.

C. Comparing symbol address

This approach is directed towards detecting system call interception events. If an entry to the system call table is modified to point to the malicious code, it is possible to detect that by comparing the current system call table with the original map of kernel symbols which was

generated at the compile time. If there is any difference then some modifications have taken place. Tools like kstat [7] compare the system call addresses in system call table with known good value.

D. Binary Analysis

Kruegel et al. have used a binary analysis approach to detect kernel-level rootkits. They observe that rootkits perform writes to a number of locations in the kernel address space, which is generally not touched by the regular system modules. For example, rewriting the first few instructions of a system call handler to make it unconditionally jump to another code is something, which a regular module would not do. The binary analysis method checks if data transfer instructions do not perform writes to an illegal area [8].

E. Execution Path Analysis

Another method that has been suggested by Rutkwoski [9] to detect rootkits is do an execution path analysis. In this method, it has been observed that the rootkits change the execution path of the normal system call. This results in a different number of instructions getting executed when the rootkit code is executed. The kernel is a huge body of code having many different execution paths. However, if an histogram is calculated of the number of instructions executed by a system call it is generally constant. If a rootkit has hooked that system call, there will be instances when the number of instructions executed will be more. Rutkowski showed in his paper how an instruction counter can be used to detect rootkits.

F. Virtual Machines

Virtual machine technology can also be leveraged to detect kernel level rootkits. Paladin [10] is an example of a package, which uses the VMware virtual machine to detect rootkits. The methodology used by them is to define protected zones (memory protected zones and file protected zones) and guard the access to those zones. The memory image of the kernel and various jump tables are considered to be part of the memory zones. They also define dependency trees between different OS objects. One such example is to track the parent-child relationship between processes.

As can be seen, from the above approaches, not all the rootkits are covered by all the approaches. Detour functions fail to provide detection of rootkits applying system call interception techniques. Diff-based approaches also are catered towards detection of changes to kernel data structures, which provide information about the system. A system call interception involving overwriting of handler's first few instructions would be not detected by that approach. The approach followed by kstat would fail in case of rootkits like SucKIT, which do not modify the system call table and the original system call table still exists in the memory. The binary analysis and execution path analysis methods fail in case of rootkits involving dynamic kernel data structure manipulations (also called as Dynamic Kernel Object Manipulations - 'DKOM'). The approach used by Paladin is a very promising approach as it is quite general in it's approach. The containment algorithm used by them provides a good approach towards preventing illegal access to protected zones and thus preventing further damage to the system. At this moment, their approach is not able to discover rootkits, which follow the DKOM approach.

Another type of rootkits that have been recently discussed are Hardware Virtualization Rootkits [14]. This rootkits use the features of the newer Hardware Virtualized architectures like Intel-VT and AMD Pacifica. These new architectures provide a new VMX instruction set geared towards virtualization. Examples of rootkits that have been devised for them are SubVirt by Samuel King et al., BluePill by Joanna Rutkowska and Vitriol by Dino Dai Zovi. These rootkits start running in

kernel at ring 0, effectively installing a rootkit hypervisor and then migrate the running OS into a VM.

5. Future Directions

We believe that a good approach towards detecting kernel rootkits is to be able to work at a layer below the kernel. Virtual machines provide the right technology for working at a layer below the kernel. The approaches used by Paladin[10] and VMI[15] provide a good direction towards further investigations in this area.

Our work is focused on investigating in trying to get useful kernel information from the VMM layer so that compromises in the kernel structures can be detected. We are working towards developing a general solution such that it encompasses all the attack approaches that have been identified above. One approach targeted towards system call interception and interrupt hooking is to get the kernel symbol addresses before the kernel is compromised (just after installing) and keep track of any changes to the kernel symbol addresses. Also, similar to the Paladin approach we plan to define kernel code as a read only section. Another aspect that we are investigating into is using the VMM ability to be able to monitor the guest OS architectural (registers) state. We are looking towards using this guest OS information towards detection of rootkits. We are using the Xen Hypervisor for our investigation purposes. Also, while investigating another attack approach that we would consider is the usage of combination of mechanisms to achieve stealth. An example of this attack is the modified 'FU' rootkit that uses the Shadow Walker's memory hook engine and FU rootkit's methodology of hiding processes. We hope to achieve a mechanism, which has the ability to detect the current rootkits and also be general enough to identify future rootkits.

References

[1] Hidden Processes: The implication for intrusion detection - Butler, Undercoffer, Pinkston
[2] Detecting and categorizing kernel level rootkits to aid future detection – Levine, Grizzard, Owen
[3] Shadow Walker: Raising the bar for windows rootkit detection – Sparks, Butler
[4] Blacklight Homepage: http://www.f-secure.com/blacklight
[5] IceSword Homepage: http://www.xfocus.net/tools/200505/1032.html
[6] FUTo – Silberman and C.H.A.O.S, http://www.uniformed.org/?v=3&a=7&t=txt
[7] kstat - http://www.s0ftpj.org/en/tools.html
[8] Detecting kernel level rootkits through binary analysis – Kruegel, Vigna
[9] Advanced Windows 2000 Rootkit Detection (Execution Path Analysis) - Rutkowski
[10] Paladin: Automated Detection and Containment of Rootkit Attacks - Baliga, Chen, Iftode
[11] An overview of Unix rootkits - Anton Chuvakin, iDEFENSE labs
[12] Kernel rootkits - SANS - Dino Dai Zovi
[13] Handling Interrupt Descriptor Table for fun and profit - kad. Phrack Vol. 0x0b, Issue 0x3b
[14] Hardware Virtualization rootkits - Dino Dai Zovi
[15] A virtual machine introspection based architecture for intrusion detection - Garfinkel and Rosenblum