Chapter 1

A METHOD FOR DETECTING LINUX KERNEL MODULE ROOTKITS

Doug Wampler and James Graham

Abstract There are a variety of methods available to detect modern Linux kernel module rootkits. However, most existing methods rely on system specific *a priori* knowledge for full detection functionality. Either (a) some application must be installed when the system is deployed, as is typical with host based intrusion detection, or (b) system metrics must be saved to a secure location when the system is deployed, or (c) both of these actions must be performed. It is noted, however, that some of these methods do offer partial functionality when installed on an already infected system. This paper proposes a technique to detect Linux Kernel Module (LKM) rootkits that does not require system specific *a priori* knowledge, but rather just knowledge about the Linux operating system in general. This technique relies on outlier analysis and statistical techniques, is more formal and rigorous than most existing detection methods, and initial results indicate that Linux Kernel Module rootkit detection is possible with a high degree of confidence.

Keywords: Intrusion Detection, Operating System Forensics, Outlier Analysis, Real Time Forensic Analysis, Rootkit Detection

1. Introduction

Nearly everyone has observed the seemingly unlimited flaws and vulnerabilities inherent in the protocols, operating systems, applications, and other software that makes up modern computing environments. By exploiting these flaws, attackers are able to assume control of systems, steal private data, have these systems attack other systems, and generally wreak cyber-havoc. The rapid change of and development in computing technology has simply advanced too quickly for security technology to keep up, with the result that the majority of todays computing environments are inherently hackable [1].

Computer security mechanisms focus on three areas: prevention, detection, and recovery [2]. Prevention entails activities such as running secure versions of popular operating systems, disabling services with known vulnerabilities or weaknesses, and installing software or hardware designed to prevent successful attacks (including patches). Detection of successful or attempted attacks is covered in a broad field known as intrusion detection, which can further be divided into network intrusion detection and host based intrusion detection. Recovery from successful attacks usually entails restoring data and applications from tape backups [3].

A primary concern of attackers everywhere is not only how to gain privileged access to a system, but also how to keep it. A rootkit is a method by which hackers maintain control of a compromised system, attack other systems, destroy evidence, and decrease the chance of being detected by system administrators [4]. A rootkit is essentially a set of software tools employed by an intruder after gaining unauthorized, privileged access to a system. Rootkit software has three primary functions: (1) to maintain access to the compromised system; (2) to attack other systems; and (3) to conceal evidence of the attacker's activities [5].

Detecting rootkits is a specialized form of intrusion detection. Effective intrusion detection includes the collection of information about intrusion techniques that can be used to improve methods of intrusion detection [2]. In most current techniques for detecting Linux rootkits, some form of *a priori* knowledge about the specific system under observation is required for full detection functionality. Either (a) some application must be installed when the system is deployed, as is typical with host based intrusion detection, (b) some system metrics must be saved to a secure location when the system is deployed, or (c) both. In a perfect world, this would not present a problem. In reality, system administrators are busy people and the time, effort and expertise required for these activities is simply not available. However, it should be noted that some Linux rootkit detection methodologies do offer partial functionality when installed on an already infected system. It should also be noted that this research focuses on the detectection of Linux kernel rootkits. Furthermore, rootkit detection applications for Microsoft Windows are typically heuristic based, and can be installed and detect Windows kernel rootkits even after infection has occurred.

The purpose of this research is to develop a method to detect rootkits using a more mathematically and statistically rigorous method, that does not require system specific *a priori* knowledge but instead just general system knowledge for the operating system and architecture under observation.

2. Background

This section presents an overview and history of rootkits, an analysis of rootkit attack techniques, and a summary of existing rootkit detections techniques in Section 2.1, 2.2, and 2.3, respectively.

2.1 Rootkits

The earliest rootkits date back to the early 1990s [1], with some components (e.g., log file cleaners) of known rootkits found on compromised systems as early as 1989. Early SunOS rootkits (for SunOS 4.x) were detected in 1994, and in 1996, and the first Linux rootkits publicly appeared [5]. Linux Kernel Module (LKM) rootkits were first proposed in the hacker magazine Phrack by Halflife [5] in 1997. Tools for attacking other systems, both locally and remotely, began appearing in rootkits during the late 1990s.

In 1998, Non-LKM kernel patching was proposed by Silvio Cesare in his landmark paper *Runtime Kernel Patching* [6]. He pointed out the possibility of intruding into kernel memory without loadable kernel modules by directly modifying the kernel image (usually /dev/mem) [5]. In 1999, the first Adore LKM rootkit was released by TESO. This rootkit alters kernel memory via Loadable Kernel Modules.

In 2001, KIS Trojan and SucKit were released. These rootkits alter kernel memory not by using Loadable Kernel Modules, but by directly modifying the kernel image (usually in /dev/mem). In 2002, Sniffer backdoors begin to appear in rootkits. Maintaining access is typically accomplished using backdoors [5].

2.2 Rootkit Classification

There are three known categories of rootkits. The first and simplest type are binary rootkits, composed of trojaned system binaries that are placed on the host system. A logical second step in the evolution of the rootkit is the library rootkit, in which a trojaned system library is placed on the host system. These first two categories of rootkit are relatively easy to detect by manually inspecting the `/proc` file system or by using statically linked binaries.

The third, and most insidious, category of rootkit is the kernel rootkit. There are two subcategories of kernel rootkits, loadable kernel module rootkits (LKM rootkits) and kernel rootkits that directly modify the memory image in `/dev/mem` (*kernel patched rootkits*) [7]. Kernel-level rootkits attack the system call table by three known mechanisms [8].

System Call Table Modification. The attacker modifies the addresses stored in the system call table. The attacker, having written custom system calls [9] to replace several system calls within the kernel, changes the addresses in the system call table to point to the new, malicious custom system calls.

System Call Target Modification. In this case, the attacker overwrites the legitimate targets of the addresses in the system call table with malicious code. The system call table does not need to be changed. The first few instructions of the system call function is overwritten with a jump instruction to the malicious code.

System Call Table Redirection. In this type of rootkit implementation, the attacker redirects references to the entire system call table to a new, malicious system call table in a new kernel address location. This method can pass many currently used detection techniques [8]. Upon further investigation, it appears that the system call table redirection attack is simply a special case of the system call target modification attack [10], since the attacker simply modifies the *system_call* function, modifying the address of the system call table therein, which handles individual system calls.

2.3 Rootkit Detection

The first kernel rootkits appeared as malicious loadable kernel modules (LKM). Processes under UNIX run either in user space or kernel space. Application programs typically run in user space and hardware access is typically handled in kernel space. If an application wants to read from a disk, it uses the open() system call and asks the kernel to open a file. Loadable kernel modules run in kernel space and have the ability to modify these system calls. If there is a malicious loadable kernel module in kernel space, the open() system call will open the file requested unless the name of the file is rootkit [1, 5]. Many system administrators counter this threat by simply disabling the loading of kernel modules [1].

Host based intrusion detection systems (*Tripwire* and *Samhain* being the most well known) are still a relatively straightforward and effective way of detecting rootkits [1]. *Samhain* also includes functionality to monitor the system call table, the interrupt description table, and the first few instructions of every system call [5]. This is an example of acquiring *a priori* knowledge about a specific system for later use in rootkit detection.

The *Linux Intrusion Detection System (LIDS)* is a kernel patch that must be applied to kernel source code, and requires a rebuild of the ker-

nel. LIDS has the capability to offer protection against kernel rootkits through the following mechanisms: sealing the kernel from modification; prevent loading/unloading of kernel modules; immutable and read-only file attributes; locking of shared memory segments; process ID manipulation protection; protection of sensitive `/dev/` files; and port scan detection [11].

A past detection method proposed by Sebastian Krahmer from SuSE was to monitor and log any program execution when execve() calls were made. Combine this with remote logging, and one could maintain a record of program execution on a system. With a Perl script to monitor the log, one could perform actions such as sending alarms or killing processes in order to stop the intruder [11].

Applications do exist for the specific purpose of detecting rootkits (including kernel rootkits). These include several tools available for download including *chkrootkit*, *kstat*, *rkstat*, *St. Michael*, *scprint*, and *kern_check* [12–18]. *Chkrootkit* is a user-space signature based rootkit detector, while several others *(kstat, rkstat, and St. Michael)* are kernel-space signature based detectors. These tools typically print the addresses of system calls directly from `/dev/kmem` and compare them to the entries in the System.map file [19]. This approach relies upon some trusted source of *a priori* knowledge of the specific system in question, in that the system administrator must install these tools before the system is infected with a rootkit. *Chkrootkit*, *kstat*, *rkstat*, and *St. Michael*, as signature based detectors, suffer from the usual shortcomings of signature based detection. *Scprint* and *kern_check* are utilities for printing and/or checking the addresses of the entries in the system call table.

Other researchers have proposed to count the instructions used in system calls, comparing them to measurements taken from a clean system [20]. Further efforts in the field of rootkit detection include static analysis of loadable kernel module binaries [21]. This approach leverages the fact that the kernel exports a well-defined interface for use by kernel modules, and LKM rootkits typically violate this interface. By carefully analyzing this interface, one may extract an allowed set of kernel modifications. Until recently, efforts toward rootkit detection have been software based. College Park, Maryland based *Komoku Inc.* offers a low-cost, add-in PCI card ("CoPilot") that monitors a host systems memory and file system [22, 23]. However, Copilot uses known good MD5 hashes of kernel memory and must be installed and configured on a clean system in order to detect the future deployment of a rootkit [24]. Spafford and Carrier have presented a technique in which binary rootkits were detected using an outlier analysis technique on the file system in an offline forensic analysis situation [25]. The research presented in

this paper focuses on the real time detection of kernel rootkits through memory analysis.

3. Proposed Detection Technique

In this section, a new detection technique for LKM rootkits will be presented. This technique will not rely on (a) advanced installation of any rootkit detection or other software, or (b) any other *a priori* knowledge about the specific system under observation. Additionally, this new technique will be more rigorous than existing methods. Trojaned system call addresses (modified addresses) will be isolated and identified without using any *a priori* knowledge about the specific system under observation, and without the advance installation of any rootkit detection software.

It will be shown that rootkits may be detected statistically by understanding the underlying distribution of system call addresses of the various major kernel releases of Linux, and recording this *general* information and comparing it to *specific* systems infected with rootkits. More specifically, outlier analysis techniques will be used to perform this analysis. This method will not only indicate the presence of a rootkit, but also identify the number of individual attacks on the kernel and their locations. This information will aid other research efforts focusing on rootkit categorization and identification [8]. The results of this research may be used to not only detect the presence of Linux kernel rootkits, but to identify them based on the specific system calls being attacked.

The explanation of this new method will be organized as follows: Initially, it will be shown that the distribution of system call table addresses fits a well known distribution across more than one architecture. Finally, trojaned system call addresses (the result of rootkit activity) will be detected through experimentation based on changes to this underlying distribution.

3.1 Stability of Model Across Architectures

One consideration that is critical to the success of this research is that the distribution of system call addresses for a specific kernel version must be very close across various architectures. This is an absolute necessity if analysis is to occur without any *a priori* knowledge of the specific system under study. Preliminary experiments were conducted on a 32-bit Intel machine and a 64-bit SPARC machine with different kernel compilation options in order to test this hypothesis.

Table 3.1: Distribution fits from 32-bit Intel machine, kernel 2.4.27

Distribution	AD
Largest Extreme Value	5.038
3-Parameter Gamma	6.617
3-Parameter Loglogistic	7.022
Logistic	7.026
Loglogistic	7.027
3-Parameter Lognormal	10.275
Lognormal	10.348
Normal	10.350
3-Parameter Weibull	49.346
Weibull	49.465
Smallest Extreme Value	49.471
2-Parameter Exponential	81.265
Exponential	116.956

Table 3.2: Distribution fits from 64-bit SPARC machine, kernel 2.4.27

Distribution	AD
Loglogistic	10.599
Largest Extreme Value	11.699
Logistic	11.745
Lognormal	19.147
Gamma	20.460
Normal	23.344
3-Parameter Gamma	26.456
3-Parameter Weibull	32.558
3-Parameter Loglogistic	34.591
Weibull	36.178
3-Parameter Lognormal	37.468
Smallest Extreme Value	41.015
2-Parameter Exponential	52.604
Exponential	102.787

While the *Largest Extreme Value* distribution best fits the system call addresses from the 32-bit Intel machine, it was not the best fit for the system call addresses for the 64-bit SPARC machine used in preliminary testing. However, *Largest Extreme Value* is still a very good fit (a close 2nd) for the SPARC. While *many* more observations are necessary to make claims of goodness of fit for the system call addresses for various categories of computers, this preliminary result suggests that this may be possible, especially for machines of different architectures but having the same kernel version.

3.2 Experimental Results

As previously mentioned, there have been several attempts at preventing and detecting the deployment of rootkits, but they require some form of *a priori* knowledge (at least for optimum detection capability)

about the specific system under observation. This technique will employ the GNU debugger and other memory analysis tools, and possibly other techniques, to detect rootkits, through formal, rigorous analysis of the data.

When a Linux Kernel Module rootkit is installed, several of the entries in the system call table are changed to unusually large values (indicative of the system call table modification attack discussed previously). This changes the goodness of fit score for the *Largest Extreme Value* distribution the data is no longer such a good fit. Because of the Linux memory model and the method of attack, we know that the outliers are on the extreme right side of the distribution [10]. If these outliers are eliminated one by one, the distribution slowly moves from a score of approximately one hundred (100) back to very close to the original score of approximately five (5).

This new technique is a method for detecting Linux Kernel Module (LKM) rootkits. These rootkits modify memory addresses in the system call table, which originally fit the *Largest Extreme Value* distribution very well; the Anderson-Darling goodness of fit test yields a score of approximately five (5). This seems to hold across multiple architectures; experiments on Intel 32 bit architectures and SPARC 64 bit architectures yield similar results.

In experiment one, the Rkit Linux Kernel Module rootkit version 1.01 was downloaded and installed on a 32-bit Intel computer running Linux kernel version 2.4.27. Rkit 1.01 only trojans one entry in the system call table, *sys_setuid*. Rkit 1.01 was selected because (a) it is a LKM rootkit, and (b) it attacks only *one* entry in the system call table. If only one outlier can be detected using this method, rootkits that attack several system call table entries may be detected more easily.

From table 3.1, it is known that our test system a 32-bit Intel computer running Linux kernel 2.4.27 has a 255 entry system call table fitting the *Largest Extreme Value* distribution with an Anderson-Darling goodness of fit score of 5.038. When Rkit 1.01 is installed, the Anderson-Darling goodness of fit score changes to 99.210. Clearly, an outlier is present in the form of the *sys_setuid* system call table entry with a greatly increased memory address. The *sys_setuid* system call table entry address was changed from 0xC01201F0 (good value) to 0xD0878060. Converted to decimal, these values are 3,222,405,616 and 3,498,541,152 a difference of 276,135,536 and approximately 8.5% larger than the original value.

When one system call table address is trojaned, the goodness of fit score changes from 5.038 to 99.210, a change of approximately 1970%. When the trojaned *sys_setuid* memory address is removed from the data,

the Anderson-Darling goodness of fit score for the *Largest Extreme Value* distribution returns to 4.968, within 1.4% of the original score of 5.038.

Table 3.3: Results of Rkit 1.01 experiment

System	AD-Score
Clean	5.038
Trojaned	109.729
Trojans Removed	5.070

In experiment two, the Knark Linux Kernel Module rootkit version 2.4.3 was installed on the same test system, a 32-bit Intel computer running Linux kernel version 2.4.27. Knark is also a Linux Kernel Module rootkit, and attacks nine different memory addresses in the system call table. Experiment two yields similar results as experiment one, a 2178% decrease in goodness of fit, then a return to within 0.7% of the original score when the outlying trojaned addresses are removed.

Table 3.4: Results of Knark 2.4.3 experiment

System	AD-Score
Clean	5.038
Trojaned	99.210
Trojans Removed	4.968

Also in experiment two, as the trojaned system addresses are removed one by one, the Anderson-Darling goodness of fit score slowly improves, but does not show a dramatic or significant improvement until the final outlier is removed. The importance of this fact lies in the concept of *complete detection*. Through this method, a rootkit that trojans only *one* system call table address can be successfully detected. Figure 3.1, below, illustrates this finding. It is possible to not only detect most trojaned system call addresses, but *all* trojaned system call addresses.

4. Conclusions and Future Work

This research was constrained by the following: The only operating system under consideration was Linux Kernel version 2.4.27, running on Intel 32 bit and SPARC 64 bit architectures; only LKM rootkits were investigated; kernel versions may be expanded to include both 2.4 and 2.6 at a later time.

The two experiments discussed in this paper have shown that it is possible to detect with a high degree of confidence Linux Kernel Module rootkits using outlier analysis on these systems. This technique needs to be tested on as many additional LKM rootkits as possible. It is important to emphasize that many more experiments will be necessary to validate this finding.

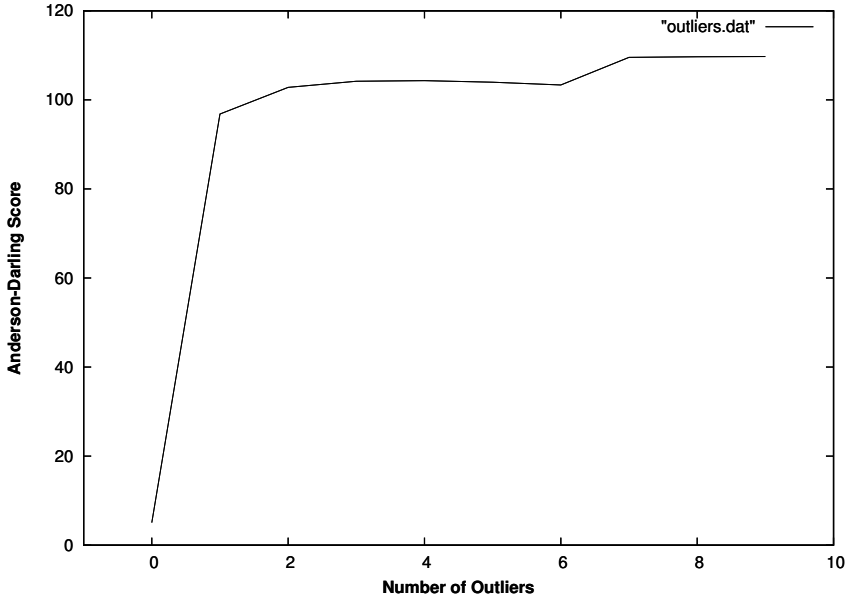

Figure 1. Figure 3.1: Anderson-Darling Score improvement as outliers are removed

Another assumption in this research is that the system call addresses within the system call table must closely fit the same distribution (or close to the same distribution) for all architectures and kernel versions. If this is not the case, it must at least be possible to categorize them into as few categories as possible. Further verification of this assumption will be addressed in planned future work where several more systems will be analyzed.

Additionally, some systems that an investigator will face may not be clean systems, and may have had several security update patches installed. Each of these patches will have the capability of modifying the system, the kernel, and therefore the system call table entries. Future research will address this possibility.

A newer class of kernel rootkits modify the system call table targets instead of the system call table addresses. The attacker simply overwrites the first few instructions of the system call target with a jump instruction to a location in memory containing the new, trojaned system call. The next step in this research will be to utilize this or similar techniques to detect rootkits using the system call table target modification attack. More specifically, it should be possible to disassemble all system calls and extract addresses from those instructions having memory ad-

dresses as operands. This information will be used with the technique discussed in this research, as well as approaches using additional outlier analysis techniques.

5. Acknowledgements

This research was supported, in part, by a grant from the Department of Homeland Security, through the Kentucky Critical Infrastructure Protection Institute. The authors would like to thank Doctors Adel Elmaghraby, Mehmed Kantardzic, and Gail DePuy for many helpful suggestions. The opinions and conclusions in this paper are solely those of the authors.

References

[1] E. Skoudis, Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses Prentice-Hall, 2002, pp. 399-445.

[2] W. Stallings, Network Security Essentials, 2nd ed 2003.

[3] Network Security: A Primer on Vulnerability, Prevention, Detection and Recovery, 9-1-2006, http://www.integritycomputing.com/security1.html

[4] Know Your Enemy: Motives, 9-1-2006, http://www.linuxvoodoo.org/resources/security/motives/

[5] A. Chuvakin, "An Overview of Unix Rootkits," iALERT White Paper, iDefense Labs, http://www.megasecurity.org/papers/Rootkits.pdf, February, 2003.

[6] Runtime Kernel Patching, 9-1-2006, http://reactor-core.org/runtime-kernel-patching.html

[7] Linux Kernel Rootkits, 9-1-2006, http://linuxcourse.rutgers.edu/documents/kernel_rootkits/index.html

[8] J.Levine, B.Grizzard, and H.Owen, "Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection," *IEEE Security & Privacy*, no. January/February 2006, pp. 24-32, 2006.

[9] Lab Exercise 2: Adding a Syscall, 9-1-2006, http://www-static.cc.gatech.edu/classes/AY2001/cs3210_fall/labs/syscalls.html

[10] Detecting Rootkits and Kernel-level Compromises in Linux, 9-1-2006, http://www.securityfocus.com/infocus/1811

[11] Root Kits and hiding files/directories/processes after a break-in, 9-1-2006, http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq

[12] Chkrootkit, 9-1-2006, http://www.chkrootkit.org

[13] Detecting and Understanding rootkits, an Introduction and just a little-bit-more, 9-1-2006, http://www.net-security.org/dl/articles/Detecting_and_Understanding_rootkits.txt

[14] kern_check.c, 9-1-2006, http://la-samhna.de/library/kern_check.c

[15] Kernel Rootkits, 9-1-2006, http://www.sans.org/reading_room/whitepapers/threats/449.php

[16] Rootkit levels of infection and mitigation, 9-1-2006, http://searchopensource.techtarget.com/tip/1,289483,sid39_gci1149598,00.html

[17] Scprint.c, 9-1-2006, http://jdoe.freeshell.org/howtos/ExitTheMatrix/misc/kernel_auditor/scprint.c

[18] Wikipedia: Rootkit, 9-1-2006, http://en.wikipedia.org/wiki/Rootkit

[19] J. Scambray, S. McClure, and G. Kurtz, Hacking Exposed: Network Security Secrets & Solutions, 2nd ed McGraw-Hill, 2001.

[20] Execution path analysis: finding kernel based rootkits, 9-1-2006, http://doc.bughunter.net/rootkit-backdoor/execution-path.html

[21] Detecting Kernel-Level Rootkits Through Binary Analysis, 9-1-2006, http://www.cs.ucsb.edu/~wkr/publications/acsac2004lkrmpresentation.pdf

[22] Komoku Inc., 9-1-2006, http://www.komoku.com/technology.shtml

[23] Government-funded Startup Blasts Rootkits, 9-1-2006, http://www.eweek.com/article2/0,1759,1951941,00.asp

[24] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor," *Proceedings of USENIX Security Symposium*, pp. 179-194, 2004.

[25] B. Carrier and E. Spafford, "Automated Digital Evidence Target Definition Using Outlier Analysis and Existing Evidence," *Proceedings of the 2005 Digital Forensics Research Workshop*, 2005.