Chapter 7

A METHOD FOR DETECTING LINUX KERNEL MODULE ROOTKITS

Doug Wampler and James Graham

Abstract Several methods exist for detecting Linux kernel module (LKM) rootkits, most of which rely on *a priori* system-specific knowledge. We propose an alternative detection technique that only requires knowledge of the distribution of system call addresses in an uninfected system. Our technique relies on outlier analysis, a statistical technique that compares the distribution of system call addresses in a suspect system to that in a known uninfected system. Experimental results indicate that it is possible to detect LKM rootkits with a high degree of confidence.

Keywords: Linux forensics, rootkit detection, outlier analysis

1. Introduction

The primary goals of an intruder are to gain privileged access and to maintain access to a target system. A rootkit is essentially a set of software tools employed by an intruder after gaining unauthorized access to a system. It has three primary functions: (i) to maintain access to the compromised system; (ii) to attack other systems; and (iii) to conceal or modify evidence of the intruder's activities [5].

Detecting rootkits is a specialized form of intrusion detection. Effective intrusion detection requires the collection and use of information about intrusion techniques [21]. Likewise, certain *a priori* knowledge about a system is required for effective Linux rootkit detection. Specifically, an application capable of detecting unauthorized changes must be installed when the system is deployed (as is typical with host-based intrusion detection), or system metrics must be collected upon system deployment, or both. But the time, effort and expertise required for these activities is significant. It should be noted that some Linux rootkit detection methodologies are not very effective when installed on an infected system. On the other hand, rootkit detection applications for Microsoft Windows are typically based on heuristics; these applications may be installed to detect kernel rootkits even after infection has occurred.

This paper focuses on the detection of Linux kernel rootkits. The methodology engages a statistical technique based on knowledge about the operating system and architecture instead of *a priori* system-specific knowledge required by most current rootkit detection techniques.

2. Background

This section presents an overview of rootkits, an analysis of rootkit attack techniques, and a summary of existing rootkit detection techniques.

2.1 Rootkits

The earliest rootkits date back to the early 1990s [20]. Some components (e.g., log file cleaners) of known rootkits were found on compromised systems as early as 1989. SunOS rootkits (for SunOS 4.x) were detected in 1994, and the first Linux rootkits appeared in 1996 [5]. Linux kernel module (LKM) rootkits were first proposed in 1997 by Halflife [5]. Tools for attacking other systems, both locally and remotely, began appearing in rootkits during the late 1990s.

In 1998, Cesare [4] proposed the notion of non-LKM kernel patching. He discussed the possibility of intruding into kernel memory without loadable kernel modules by directly modifying the kernel image (usually in `/dev/mem`) [5]. The first Adore LKM rootkit, which was released in 1999, altered kernel memory via loadable kernel modules. The KIS Trojan and SucKit rootkits were released in 2001; these rootkits directly modified the kernel image. In 2002, rootkits began to incorporate sniffer backdoors for maintaining access to compromised systems [5].

2.2 Rootkit Classification

Rootkits are generally classified into three categories. The first and simplest are binary rootkits, which are composed of Trojaned system binaries. A second, more complex, category includes library rootkits – Trojaned system libraries that are placed on systems. These two categories of rootkits are relatively easy to detect: either by manually inspecting the `/proc` file system or by using statically-linked binaries.

The third, and most insidious, category of rootkits constitutes kernel rootkits. There are two subcategories of kernel rootkits: (i) loadable kernel module rootkits (LKM rootkits), and (ii) kernel patched rootkits that directly modify the memory image in `/dev/mem` [22]. Kernel rootkits attack system call tables by three known mechanisms [12]:

- **System Call Table Modification:** The attack modifies certain addresses in the system call table to point to the new, malicious system calls [8].

- **System Call Target Modification:** The attack overwrites the legitimate targets of the addresses in the system call table with malicious code, without changing the system call table. The first few instructions of the system call function are overwritten with a jump instruction to the malicious code.

- **System Call Table Redirection:** The attack redirects all references to the system call table to a new, malicious system call table in a new kernel address location. This attack evades detection by many currently used tools [12]. System call table redirection is a special case of system call target modification [1] because the attack modifies the `system_call` function that handles individual system calls by changing the address of the system call table in the function.

2.3 Rootkit Detection

The first kernel rootkits appeared as malicious loadable kernel modules (LKMs). UNIX processes 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 needs to read from a disk, it uses the `open()` system call to request the kernel to open a file. Loadable kernel modules run in kernel space and have the ability to modify these system calls. If a malicious loadable kernel module is present in kernel space, the `open()` system call will open the requested file unless the filename is "rootkit" [5, 20]. Many systems administrators counter this threat by simply disabling the loading of kernel modules [20].

Host-based intrusion detection systems, e.g., Tripwire and Samhain, are very effective at detecting rootkits [20]. 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 using *a priori* knowledge about a specific system in rootkit detection.

The Linux Intrusion Detection System (LIDS) is a kernel patch that requires a rebuild (recompile) of the kernel. LIDS can offer protection against kernel rootkits through several mechanisms, including sealing the kernel from modification; preventing the loading/unloading of kernel modules; using immutable and read-only file attributes; locking shared memory segments; preventing process ID manipulation; protecting sensitive `/dev/` files; and detecting port scans [6].

Another detection method is to monitor and log program execution when `execve()` calls are made [6]. Remote logging is used to maintain a record of program execution on a system, and a Perl script implemented to monitor the log and perform actions such as sending alarms or killing processes in order to defeat the intruder [6].

Several applications are available for detecting rootkits (including kernel rootkits). These include `chkrootkit` [13], `kstat` [2], `rkstat` [23], St. Michael [23], `scprint` [19], and `kern_check` [17]. `chkrootkit` is a user-space signature-based rootkit detector while others, e.g., `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/or compare them with the entries in the `System.map` file [18]. This approach relies on a trusted source for *a priori* knowledge about the system in question in that the systems administrator must install these tools before the system is infected by a rootkit. Since `chkrootkit`, `kstat`, `rkstat` and St. Michael are signature-based rootkit detectors, they suffer from the usual shortcomings of signature-based detection. The remaining two tools, `scprint` and `kern_check`, are utilities for printing and/or checking the addresses of entries in system call tables.

Some rootkit detection techniques count the numbers of instructions used in system calls and compare them with those computed for a "clean" system [16]. Other detection techniques involve static analysis of loadable kernel module binaries [11]. This approach leverages the fact that the kernel exports a well-defined interface for use by kernel modules; LKM rootkits typically violate this interface. By carefully analyzing the interface, it is possible to extract an allowed set of kernel modifications.

Until recently, rootkit detection involved software-based techniques. Komoku Inc. now offers a low-cost PCI card ("CoPilot") that monitors a system's memory and file system [10, 14]. However, CoPilot uses "known good" MD5 hashes of kernel memory and must be installed and configured on a "clean" system to detect future deployments of rootkits [15].

Spafford and Carrier [3] have presented a technique for detecting binary rootkits using outlier analysis on file systems in an offline manner. Our technique is unique in that it permits the real-time detection of kernel rootkits via memory analysis.

3. LKM Rootkit Detection Technique

Our technique for detecting LKM rootkits does not require the prior installation of detection tools or other software. Trojaned system call addresses (modified addresses) are identified without using any *a priori* knowledge about a system. In particular, rootkits are detected by comparing the distribution of system call addresses from a "suspect" system with the distribution of system call addresses from a known "good" (uninfected) system. Outlier analysis is used to identify infected systems. This method not only identifies the presence of a rootkit, but also the number of individual attacks on a kernel and their locations.

In the following, we demonstrate that the distribution of system call table addresses fits a well-known distribution for more than one architecture. Also, we show how to detect Trojaned system call addresses (the result of rootkit activity) using outlier analysis on the underlying distribution of table addresses from an uninfected system.

3.1 Model Stability

A fundamental assumption of our statistical approach to rootkit detection is that the distribution of system call addresses for a specific kernel version is similar across architectures. This is a necessary condition if analysis is to occur without *a priori* knowledge of the system. We tested this hypothesis by conducting preliminary experiments on a 32-bit Intel machine and a 64-bit SPARC machine with different kernel compilation options. The experimental results are presented in Tables 1 and 2. The two tables show the Anderson-Darling (AD) goodness of fit scores for various distributions. The better the goodness of fit of a distribution, the lower its AD score.

The results show that, while the *Largest Extreme Value* distribution best fits the system call addresses for the 32-bit Intel machine (Table 1), it is not the best fit for the 64-bit SPARC machine (Table 2). However, the *Largest Extreme Value* is still a good fit (and a close second) for the SPARC machine. Although more observations are required to make claims about the goodness of fit of system call addresses for different computer systems, our preliminary results suggest that the claims may be justified, especially for architectures that use the same kernel version.

3.2 Experimental Results

When an LKM rootkit is installed, several 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 no longer has such a good fit. Because of the Linux memory model and the method of attack, the outliers are on the extreme right side of the distribution [1]. If these outliers are eliminated one by one, the

Table 1. Distribution fits for a 32-bit Intel machine (kernel 2.4.27).

Distribution	**AD Score**
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 2. Distribution fits for a 64-bit SPARC machine (kernel 2.4.27).

Distribution	**AD Score**
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

distribution slowly moves from an AD score near 100 to very close to the original AD score of approximately five for the 32-bit Intel machine (Table 1).

This idea is the basis of our technique for detecting LKM rootkits. The rootkits modify memory addresses in the system call table, which originally fit the *Largest Extreme Value* distribution very well (AD score of approximately five). The results appears to hold for multiple architectures. In fact, our experiments on the Intel 32-bit and SPARC 64-bit architectures yield similar results.

In the first experiment, we installed the Rkit LKM rootkit version 1.01 on a 32-bit Intel machine with Linux kernel version 2.4.27. We chose Rkit 1.01 because it attacks only *one* system call table entry (`sys_setuid`). If only one outlier can be detected using this method, rootkits that attack several system call table entries may be detected more easily.

The 32-bit Intel computer running Linux kernel 2.4.27 has a 255-entry system call table that fits the *Largest Extreme Value* distribution with an AD goodness of fit score of 5.038 (Table 1). When Rkit 1.01 is installed, the AD score changes to 99.210 (Table 3). Clearly, an outlier is present in the form of the `sys_setuid` system call table entry with a much higher memory address. In fact, the rootkit changes the `sys_setuid` system call table entry address from `0xC01201F0` (good value) to `0xD0878060`. The decimal equivalents are 3,222,405,616 (good value) and 3,498,541,152 (approximately 8.5% higher than the good value).

Table 3. Rkit 1.01 results.

System	**AD Score**
Clean	5.038
Trojaned	99.210
Trojans Removed	4.968

As shown in Table 3, when one system call table address is Trojaned, the AD goodness of fit score for the *Largest Extreme Value* distribution changes from 5.038 to 99.210, an increase of approximately 1,970%. When the Trojaned `sys_setuid` memory address is removed, the AD score improves to 4.968, within 1.4% of the original score of 5.038.

Table 4. Knark 2.4.3 results.

System	**AD Score**
Clean	5.038
Trojaned	109.729
Trojans Removed	5.070

In the second experiment, we installed the Knark LKM rootkit version 2.4.3 on the same test system (32-bit Intel computer running Linux kernel version 2.4.27). The Knark 2.4.3 rootkit attacks nine different memory addresses in the system call table. As shown in Table 4, the

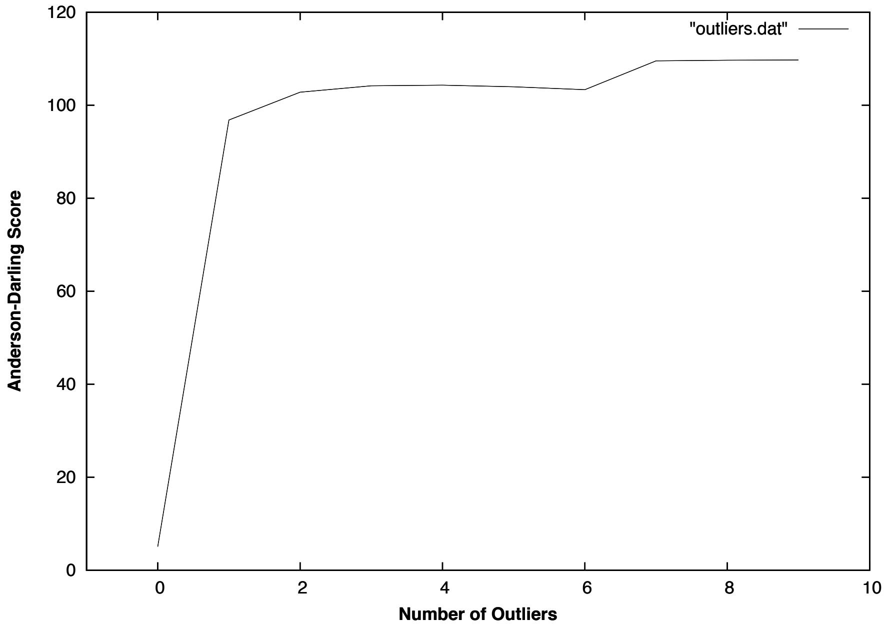

Figure 1. AD score improvement as outliers are removed (Knark 2.4.3).

results are similar to those obtained in the case of Rkit 1.01: a 2,178% decrease in the AD goodness of fit, followed by a return to within 0.7% of the original AD score when the outlying Trojaned addresses are removed.

Also in the second experiment, as the Trojaned system addresses are removed one by one, the AD score improves, but the improvements are not dramatic until the final outlier is removed (Figure 1). This is an example of the concept of "complete detection." Thus, a rootkit that Trojans only *one* system call table address can be successfully detected. Moreover, it is possible to detect not just some or most Trojaned system call addresses, but *all* Trojaned system call addresses.

4. Conclusions

This research was limited in that only LKM rootkits were investigated and only one operating system (Linux kernel version 2.4.27) was considered for two architectures (Intel 32-bit and SPARC 64-bit machines). Nevertheless, our experiments demonstrate that it is possible to detect these rootkits with a high degree of confidence using outlier analysis.

Our future work will evaluate the generality of the rootkit detection technique by testing it on other operating systems, architectures and

LKM rootkits. Also, we will attempt to verify the principal assumption that system call addresses in a system call table have (or closely fit) the same distribution for all architectures and kernel versions.

Other avenues for future research involve testing systems with security patches that could modify the kernel (and system call table entries), and developing techniques for detecting new kernel rootkits that modify system call table targets instead of addresses.

Acknowledgements

This research was supported in part by a grant from the U.S. Department of Homeland Security through the Kentucky Critical Infrastructure Protection Institute. The authors would also like to thank Drs. Adel Elmaghraby, Mehmed Kantardzic and Gail DePuy for their suggestions.

References

[1] M. Burdach, Detecting rootkits and kernel-level compromises in Linux (www.securityfocus.com/infocus/1811), 2004.

[2] A. Busleiman, Detecting and understanding rootkits (www.net-security.org/dl/articles/Detecting_and_Understanding_rootkits.txt) 2003.

[3] B. Carrier and E. Spafford, Automated digital evidence target definition using outlier analysis and existing evidence, *Proceedings of the Fifth Annual Digital Forensics Research Workshop* (www.dfrws .org/2005/proceedings/index.html), 2005.

[4] S. Cesare, Runtime kernel patching (reactor-core.org/runtime-kernel-patching.html).

[5] A. Chuvakin, An overview of Unix rootkits, iALERT White Paper, iDefense Labs (www.megasecurity.org/papers/Rootkits.pdf), 2003.

[6] D. Dittrich, Root kits and hiding files/directories/processes after a break-in (staff.washington.edu/dittrich/misc/faqs/rootkits.faq), 2002.

[7] Honeynet Project, Know your enemy: The motives and psychology of the black hat community (www.linuxvoodoo.org/resources /security/motives), 2000.

[8] P. Hutto, Adding a syscall (www-static.cc.gatech.edu/classes/AY 2001/cs3210_fall/labs/syscalls.html), 2000.

[9] Integrity Computing, Network security: A primer on vulnerability, prevention, detection and recovery (www.integritycomputing.com /security1.html).

[10] Komoku Inc. (www.komoku.com/technology.shtml).

[11] C. Kruegel, W. Robertson and G. Vigna, Detecting kernel-level rootkits through binary analysis (www.cs.ucsb.edu/~wkr/publications/acsac2004lkrmpresentation.pdf), 2004.

[12] J. Levine, B. Grizzard and H. Owen, Detecting and categorizing kernel-level rootkits to aid future detection, *IEEE Security & Privacy*, pp. 24–32, January/February 2006.

[13] M. Murilo and K. Steding-Jessen, `chkrootkit` (www.chkrootkit.org), 2006.

[14] R. Naraine, Government-funded startup blasts rootkits (www.eweek.com/article2/0,1759,1951941,00.asp), April 24, 2006.

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

[16] J. Rutkowski, Execution path analysis: Finding kernel based rootkits (doc.bughunter.net/rootkit-backdoor/execution-path.html).

[17] Samhain Labs, `kern_check.c` (la-samhna.de/library/kern_check.c).

[18] J. Scambray, S. McClure and G. Kurtz, *Hacking Exposed: Network Security Secrets and Solutions*, McGraw-Hill/Osborne, Berkeley, California, 2001.

[19] SecurityFocus, `scprint.c` (downloads.securityfocus.com).

[20] E. Skoudis, *Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses*, Prentice-Hall, Upper Saddle River, New Jersey, 2001.

[21] W. Stallings, *Network Security Essentials*, Prentice-Hall, Upper Saddle River, New Jersey, 2003.

[22] R. Wichmann, Linux kernel rootkits (coewww.rutgers.edu/www1/linuxclass2006//documents/kernel_rootkits/index.html), 2002.

[23] D. Zovi, Kernel rootkits (www.sans.org/reading_room/whitepapers/threats/449.php), SANS Institute, 2001.