A Normality Based Method for Detecting Kernel Rootkits

Doug Wampler
Indiana Department of Correction
302 West Washington Street
Indianapolis, IN 46204
317.233.1022
dwampler@idoc.in.gov

James H. Graham
University of Louisville
J.B. Speed School of Engineering
Louisville, KY 40292
502.852.0475
Jhgrah01@louisville.edu

ABSTRACT
Rootkits are stealthy, malicious software that allow an attacker to gain and maintain control of a system, attack other systems, destroy evidence, and decrease the chance of detection. Existing detection methods typically rely on a priori knowledge and operate by either (a) saving the system state before infection and comparing this information post infection, or (b) installing a detection program before infection. This approach focuses on detection using reduced a priori knowledge in the form of general knowledge of the statistical properties of broad classes of operating system/architecture pairs.

A modified normality based approach proved effective in detecting kernel rootkits infecting the kernel via the system call target modification attack. This approach capitalizes on the discovery that system calls are loaded into memory sequentially, with the higher level calls, which are more likely to be infected by kernel rootkits loaded first, and the lower level calls loaded later. In the single case evaluated, the enyelkm rootkit, neither false positives nor false positives were indicated. The enyelkm rootkit was selected for analysis since it infects the Linux kernel via the system call target modification attack, which is the subject of this research.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection – Invasive software (e.g., viruses, worms, Trojan horses).

General Terms

Keywords
Intrusion Detection, Operating System Forensics, Outlier Analysis, Forensic Analysis, Rootkit Detection.

1. INTRODUCTION
A primary concern of attackers everywhere is not only how to gain privileged access to a system, but also how to keep it. In order to keep privileged access, the attacker must conceal his, or her, activities from the system administrator and other legitimate users of the system in question. Over time, concealment of illicit activities has evolved from the manual editing of log files, to the development of simple tools for this and similar purposes, culminating in the development of rootkits ranging from the simple to the Byzantine.

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 [1]. The first rootkits were detected on SunOS machines in the early 1990s. Since then, a “projectile/armor” race has erupted between those trying to develop/detect rootkits [2;3]. 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 [3].

In many current techniques for detecting Linux rootkits, substantial a priori knowledge about the specific system under observation is required. Either (a) some application must be installed when the system is deployed, as is typical with host based intrusion detection, or (b) some system metrics must be saved to a secure location when the system is deployed.

The purpose of this research is to detect rootkits using a more mathematically and statistically rigorous method, while requiring less specific a priori knowledge of any given system. However, it should be noted that it will still be necessary to have some a priori knowledge of general systems of the same type under observation. In particular, information about the distribution of system calls is needed. In most operating systems this does not appear to be normally distributed, which focused most initial work in this research on general distribution models [4;5]. However, in certain special cases, a normality assumption is justified.

The experimental results presented in this work were obtained from the SUSE Linux distribution kernel version 2.6.8. It is the feeling of the authors that the statistical properties of the Linux operating system vary according to kernel version, and not distribution, but this will be investigated more thoroughly in the future with additional experimentation.

2. BACKGROUND
There are three known categories of rootkits. The first and simplest type are binary rootkits, composed of modified, malicious copies of 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 modified and malicious copy of a system library is placed on the host system. These first two categories of rootkit are relatively easy to detect.

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) [6]. Kernel-level rootkits attack the system call table by three known mechanisms [7].
System Call Table Modification. The attacker modifies the addresses stored in the system call table. The attacker, having written custom system calls [8] 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 [7]. Upon further investigation, it appears that the system call table redirection attack is simply a special case of the system call target modification attack [9]. The attacker simply modifies the system call function, modifying the address of the system call table therein, which handles individual system calls.

Previous research in this area [4;5] focused on detecting Linux kernel rootkits that operate via the system call table modification attack. The SUSE Linux operating system shares certain statistical similarities even across kernel versions, and this knowledge was leveraged in order to detect rootkits using outlier analysis techniques. Later, rootkits were successfully detected by relying on an assumption of normality, again using outlier analysis techniques. Finally, Windows rootkits utilizing a method similar to the system call table modification attack were detected by utilizing outlier analysis techniques. Fortunately, the System Service Descriptor Table (SSDT) of the Windows operating system used in the study was very normally distributed, simplifying detection. The SSDT is very similar to the Linux system call table.

3. PROPOSED DETECTION TECHNIQUE

This section will address the use of a modified ‘assumption of normality’ technique to detect a Linux kernel rootkit utilizing the system call redirection attack, which is a special case of the system call target modification attack. This technique focuses on detecting rootkits of the two latter types – system call redirection, and system call target modification. The system call table redirection attack is a special case of the system call target modification attack, in that the system call table event handler itself becomes the victim of system call target modification, and may be detected by the same methods. Several key concepts relevant to this detection model include:

- The operands of the jump instructions – memory addresses – are collected by disassembling the kernel.

3.1 Formal Model

In order for this analysis to occur, the running kernel must be disassembled, and all conditional and unconditional jump instructions (including push instructions) must be collected. For reasons which will become clear later, the order of appearance in the disassembled code of these instructions is important and this information will also be collected. These instructions are further analyzed, and their operands – memory addresses – are extracted for analysis. After these memory addresses are collected, they are converted from hexadecimal to decimal addresses. Z-scores are then calculated for these addresses, and those with Z-scores greater than or equal to three are considered outliers and as such are reserved for further analysis. Unfortunately, even an uninfected kernel contains memory addresses of this kind which are natural outliers, that is, are outliers but have not been modified by kernel rootkit infection. Even though the dataset contains approximately 70,000 data items, only a few (typically one dozen or less) have Z-scores greater than or equal to three.

This data, the memory addresses, is univariate, and it also contains several natural outliers making a conventional, normality based approach for outliers analysis impractical. The order of appearance of these memory addresses appears to be a factor. It will prove helpful to add a second dimension to the data, a dimension that takes into account the order of appearance of each individual as well as the individual’s Z-score. A second dimension will be added, a composite value constituted by the line number (order of appearance) multiplied by the individual’s Z-score. This value will be called ‘L*Z’, or the ‘LZ’ value.

This data will then be sorted by the ‘LZ’ score descending. In an uninfected system, the ratio between ‘LZ’ score of the individual with the highest ‘LZ’ score and the ‘LZ’ score of the individual with the lowest ‘LZ’ score is less than ten. The ratio between the ‘LZ’ score of the individual with the highest ‘LZ’ score and the remaining individuals will, of course, be much less than ten.

In a system infected by a runtime kernel patching kernel rootkit, the ratio between the ‘LZ’ score of the uninfected individual with the highest ‘LZ’ score and the ‘LZ’ score of the uninfected individual with the lowest ‘LZ’ score should again be less than 10. However, the ratios between the ‘LZ’ score of the uninfected individual with the highest ‘LZ’ score and the ‘LZ’ scores of the infected individuals are expected to be much greater than 10, and probably greater than 100.

3.2 Order of Appearance

The fundamental purpose of this model is to identify those individuals that have (a) Z-scores in excess of 3, and (b) also have an early order of appearance in the code of the disassembled kernel instructions. Recall that the order of appearance of the disassembled instructions appear to be a factor in determining the likelihood that any given individual will have been infected by a kernel rootkit. System calls can be thought of as either higher-level function calls (such as sys_read or sys_write) and lower level functions (such as the system calls constituting VFS, the virtual file system). Clearly, an attacker would prefer to only rewrite sys_read instead of an entire library.
of lower level system calls such as those comprising VFS. Conceptually, the system calls constituting the Linux kernel may be imagined as a pyramid, with sys_read, sys_write, and the other high level functions at the apex of the pyramid, and the lower level kernel functions such as VFS system calls, near the bottom of the pyramid. Interestingly, at boot time this `pyramid’ of system calls appears to be loaded into memory in an inverted manner. That is, the higher level functions such as sys_read and sys_write appear to be loaded into memory first, at the lower memory addresses. The lower level system calls tend to be loaded later, much higher in memory. Attackers wish to gain the upper hand with the minimum effort necessary, and many rootkits attack only high level system calls, leaving the low level calls untouched. Relying on this assumption, it can be shown system call memory addresses with Z-scores greater than or equal to three and very low orders of appearance are highly suspect.

Having shown which system call memory addresses are more likely to be infected, it should be noted that this data contains natural outliers with Z-scores greater than or equal to three, even in an uninfected system. However, the system calls having these outliers can be shown to be low level system calls, unlikely to be modified by an attacker. Specifically, these system calls containing natural outliers are:

- `aio_put_req` (Asynchronous I/O) – AIO Ring is a memory buffer in the address space of the user mode process that is also accessible by all processes in kernel mode.
- `mpage_writepage` (Memory mapping) – Loading files for execution into memory, sharing memory between processes.
- `move_to_swap_cache` (Swap cache) – Page Frame Reclamation Algorithm. Calls `add_to_swap_cache`.
- `page_put_link` - Ext2 filesystem.

These functions provide functionality for asynchronous I/O, memory mapping, and memory management. These low level functions have historically been of little interest to attackers. Having established this, it can now be shown that individuals with high standard deviation and high order of appearance are actually low level system calls containing natural outliers, and are unlikely to be selected as targets by an attacker. Amongst the individuals just described, what would identify other individuals as rootkit infected outliers? First, these individuals would have a high Z-score. Second, suspect individuals would be high level system calls – that is, system calls with a very low order of appearance. In an uninfected system the ratio between ‘LZ’ scores of the two individuals with the largest and smallest ‘LZ’ scores, and having Z-scores greater than or equal to three, is typically less than 10.

In an infected system, those individuals having a ‘LZ’ score ratio of one hundred or more (as compared to the individual with the largest ‘LZ’ score and having a Z-score less than three) have been infected by a kernel rootkit. Why? Because they have (a) been identified as outliers having Z-scores greater than or equal to three, and (b) they have been identified as high level system calls having a very early order of appearance.

Using the ‘assumption of normality’ model, it is assumed that the system call addresses in the disassembled kernel system calls are somewhat normally distributed. If this may be assumed, this simplifies the task of rootkit detection.

As discussed in previous research [4;5] a kernel rootkit is defined as some program \( p_2 \), which imitates a subset of operating system functionality \( p_1 \). Therefore, \( p_1 \) is a subset of \( p_2 \). The functionality that exists in \( p_2 \), but not \( p_1 \), is the additional functionality provided by the kernel rootkit in order to maintain control of compromised systems, attack other systems, destroy evidence, and decrease the chance of the attacker being detected by the authorities. More formally, the kernel rootkit functionality can be expressed as \( p_2 \setminus p_1 = p' \) [7].

The fundamental differences in this approach is the absence of the necessity to have statistical information about the properties of an uninfected system, and the need to analyze memory addresses in the entire kernel instead of only in the system call table. Therefore, the elements of the formal model that define the properties of an uninfected system become unnecessary and may be discarded [4]. The elements to be discarded are:

- \( s_i = M_1(p_1) \) – system call addresses in uninfected kernel
- \( s_2 = M_2(p_2) \) – system call addresses in clean system call table
- \( s_2' = M_2(p_1) \) – system call addresses in infected system call table
- \( D_1 = t(s_1) \) – Discordancy score of system call addresses in clean kernel
- \( D_2 = t(s_2) \) – Discordancy score of addresses in clean system call table
- \( D_2' = t(s_2') \) – Discordancy score of addresses in infected system call table

Having discarded half the elements from the original model [4], the new model is smaller, more elegant, and requires significantly less a priori knowledge about the system under observation. The only elements remaining in the new formal model are:

- \( s_i' = M_1(p_1) \) – system call addresses in infected kernel
- \( D_1' = t(s_i') \) – Discordancy score of system calls in infected kernel
- \( LZ = Order of appearance * Z-score \)
- \( D_2' = r(s_i') \) – Second discordancy score of addresses in infected kernel

The discordancy test \( t \) is simply the Z-score. The Z-score represents the number of standard deviations away from the mean for a particular individual \( x \), and is represented by

\[
z = \frac{(x - \bar{X})}{\sigma} [10].
\]

The ‘LZ’ score is represented by the line number in the disassembled code in which the value occurs multiplied by the individual’s Z-score. The second discordancy test \( r \) is represented by the ratio between the ‘LZ’ score of the individual with Z-score \( \geq 3 \) and the largest ‘LZ’ score and the ‘LZ’ score of any given individual. For an individual to be considered an outlier and infected by a kernel rootkit, it must satisfy
\[ D^*_1 \geq 3 \text{ and } D^*_2 > 100 \] (3.11)

### 3.3 Normality of Data

Table 1 shows the goodness of fit scores for the 32 bit Intel architecture kernel 2.6.8 disassembled system call memory addresses that are uninfected. Table 2 shows the goodness of fit scores for the 32 bit Intel architecture kernel version 2.6.8 infected with the enyelkm v1.1 rootkit.

These tables shows that, with enyelkm v1.1 rootkit infection, there are measurable but subtle changes in the best and worst fitting distributions. All that is necessary in this case is that the data be normal enough to allow application of the normality based tests in the formal model. Clearly, the normal distribution is one of the better fitting distributions in both uninfected and infected kernels.

In this work, normality is measured using the Anderson-Darling goodness of fit test. The Anderson-Darling metric tests if a sample comes from a particular distribution. It is a modification of the Kolmogorov-Smirnov (K-S) test that gives more weight to the tails of the distribution than the K-S test. The K-S test is distribution free in the sense that the critical values do not depend on the specific distribution being tested [11].

The Anderson-Darling test utilizes the specific distribution when calculating critical values. This approach has the advantage of producing a more sensitive test and the disadvantage that critical values must be calculated for each distribution. Tables of critical values are not usually supplied, since the test itself is applied with a statistical software program that produces the critical values [11].

The Anderson-Darling test determines whether data comes from a specific distribution. The formula for the test statistic \( A \) to assess if data \( \{Y_1 < \ldots < Y_N\} \) (this data must be ordered) comes from a distribution with cumulative distribution function \( F \) is

\[
A^2 = N - S
\]

Where:

\[
S = \sum_{k=1}^{N} 2k - 1 / N \left[ \ln F(Y_k) + \ln(1 - F(Y_{N+1-k})) \right]
\]

H\(_0\) = The data fits the specified distribution.

H\(_1\) = The data does not fit the specified distribution.

\( \alpha \) = Significance level.

The critical values for the Anderson-Darling goodness-of-fit test are dependent on the specific distribution that is being tested. Values and formulas have been published for a few particular distributions. The Anderson-Darling goodness-of-fit test is a one-sided test and the hypothesis that the distribution fits a specific form is rejected if the test statistic, \( A \), is larger than the critical value [11].

For a given distribution, the Anderson-Darling goodness-of-fit test may be multiplied by a constant - which typically depends on the sample size. This is known as the “adjusted Anderson-Darling” statistic. This is the metric that should be compared against the critical values. Different constants (and therefore different critical values) have been published. It is important to be aware of what constant was used for a given set of critical values. The necessary constant is typically given with the critical values [11]. A smaller Anderson–Darling score indicates that the distribution fits the data better.

#### Table 1. Anderson-Darling scores for uninfected 2.6.8 disassembled kernel

<table>
<thead>
<tr>
<th>Distribution</th>
<th>AD-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loglogistic</td>
<td>15883.557</td>
</tr>
<tr>
<td>Logistic</td>
<td>16875.174</td>
</tr>
<tr>
<td>3-Parameter Loglogistic</td>
<td>17575.839</td>
</tr>
<tr>
<td>Normal</td>
<td>25497.186</td>
</tr>
<tr>
<td>3-Parameter Lognormal</td>
<td>25554.644</td>
</tr>
<tr>
<td>Lognormal</td>
<td>26270.259</td>
</tr>
<tr>
<td>Weibull</td>
<td>28804.239</td>
</tr>
<tr>
<td>Smallest Extreme Value</td>
<td>28845.74</td>
</tr>
<tr>
<td>2-Parameter Exponential</td>
<td>32412.629</td>
</tr>
<tr>
<td>Exponential</td>
<td>32419.241</td>
</tr>
<tr>
<td>3-Parameter Weibull</td>
<td>3492063.179</td>
</tr>
</tbody>
</table>

#### Table 2. AD-scores for disassembled 2.6.8 kernel infected with enyelkm v1.1

<table>
<thead>
<tr>
<th>Distribution</th>
<th>AD-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic</td>
<td>17576.775</td>
</tr>
<tr>
<td>3-Parameter Loglogistic</td>
<td>18288.079</td>
</tr>
<tr>
<td>Loglogistic</td>
<td>21471.281</td>
</tr>
<tr>
<td>3-Parameter Lognormal</td>
<td>27036.452</td>
</tr>
<tr>
<td>Normal</td>
<td>27043.837</td>
</tr>
<tr>
<td>Lognormal</td>
<td>27335.94</td>
</tr>
<tr>
<td>3-Parameter Weibull</td>
<td>29131.015</td>
</tr>
<tr>
<td>Weibull</td>
<td>29142.046</td>
</tr>
<tr>
<td>Smallest Extreme Value</td>
<td>29142.187</td>
</tr>
<tr>
<td>2-Parameter Exponential</td>
<td>32830.144</td>
</tr>
<tr>
<td>Exponential</td>
<td>32839.034</td>
</tr>
</tbody>
</table>

### 3.4 Experimental Results

Having established that the dataset is normal enough to facilitate the use of normality based tests, one may proceed by applying the model to a 32-bit Intel architecture 2.6.8 kernel by disassembling the kernel and collecting the memory addresses of the conditional and unconditional jump instructions from the disassembled code yields approximately 71,000 data items.

In the dataset, exactly eleven of these individuals pass the first discordancy test, having Z-scores greater than or equal to three. These individuals are presented in Table 3, below.

#### Table 3. Individuals passing first discordancy test

<table>
<thead>
<tr>
<th>Dec</th>
<th>Line</th>
<th>Z-score</th>
<th>L*Z</th>
<th>Trojaned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. CONCLUSIONS AND FUTURE WORK

Using the ‘assumption of normality’ model to detect the system call target modification attack is complicated by the discovery of natural outliers, or outliers that occur even in an uninfected system of this kind, within the data. These outliers occur in the disassembled jump instructions of specific low level system calls, specifically the following:

- `aio_put_req` (Asynchronous I/O) – The AIO Ring is a memory buffer in the address space of the user mode process that is also accessible by all processes in kernel mode.
- `mpage_writepage` (Memory mapping) – Used for loading files for execution into memory, and sharing memory between processes.
- `move_to_swap_cache` (Swap cache) – This is part of the page frame reclamation functionality. Calls `add_to_swap_cache`.
- `page_put_link` – This is part of the API for the ext2 filesystem.

These system calls typically concern sharing information between processes running in user space and processes running in kernel space. For this reason, outliers are generated – these system calls must move between kernel space (typically lower in memory), and user space (typically higher in memory). The formal model takes into account the following facts:

- Outliers exist even in an uninfected system;
- Higher level system calls tend to be loaded into memory first (receiving lower memory addresses), and lower level system calls tend to be loaded into memory later (receiving higher memory addresses);
- Outliers that are a product of low level system calls appearing higher in memory are unlikely to be selected for modification by an attacker;
- Outliers that are a product of high level system calls appearing lower in memory are likely candidates for modification by an attacker.

The detection method presented in this chapter depends on the presence of natural outliers to illuminate the presence of those outliers that stem from the activities of a rootkit. What happens in the scenario where the Linux kernel under analysis contains no natural outliers? Assuming the statistical properties of the kernel presented in this chapter do not change, absence the presence of natural outliers, all memory addresses with Z-scores in excess of 3 would be outliers generated by rootkit activity.

Future work will include acquiring and testing additional rootkits that utilize either the system call target modification attack or the system call target redirection attack. These rootkits, once acquired, will be tested on as many operating system/architecture variants as possible in order to insure that the assumptions relied upon in this paper are valid.
5. REFERENCES


