Detecting Kernel-Mode Rootkits

André Almeida

Instituto Superior Técnico, Av. Rovisco Pais,
1049-001 Lisboa
{andre.dalmeida}@ist.utl.pt

Abstract

“Hooking” techniques are a powerful method not only for those who attack but also for those who defend. Hooking is a technique that intercepts the execution flow of an application, or even Kernel, and allows the attacker to obtain important data or modify information.

It is interesting to notice the multiple places of an operating system sensible to “Hooking”. Either in User-mode or in privileged mode, there are crucial zones sensible to slight changes, creating a different global behavior of the system, helping the attacker covering his tracks. This way, the system will provide false information for an application that requests some state of the operating system.

Fortunately, knowing the inner-workings of these subversive programs, we can detect their presence if they are installed. Typically, it is always possible to detect traces of intrusion, but depending on the technological advance of the Rootkit, this can be an extremely difficult task.

1. Introduction

The term rootkit originated in the mid-nineties. The idea was based on a set of tools (kit) whose purpose would be to subvert the system in order to change some of its original behavior. Usually these tools would maintain covert root access to a system and hide the intruder's presence. This is why the resulting term is rootkit.

A rootkit tries to remain undetected by using techniques to hide parts of the system to a User-land program. It may hide files on the hard drive, running processes, active network connections or any other element that may reveal unauthorized activity.

The complexity of a rootkit depends on the technique it uses to achieve its stealth to avoid detection. The application of these techniques requires deep knowledge of the operating system and advanced programming skills. Depending on how advanced the rootkit is, it is important to understand memory layout, Kernel objects, file and network drivers, privilege levels; some of them are not documented.

This article presents a proactive detection mechanism which scans many sensitive places of the operating system during its operation. (It is not infallible, of course).

2. Attacking the Kernel

If the intruder gets into Kernel-mode, comparing to a User-mode attack, challenges are higher for those who seek defenses and it gets harder to ensure that the system has not been compromised.

There is no doubt that a good integrity checker for the physical memory is extremely effective against rootkits that replace simple system utilities such as *ls* or *netstat*. But when the attack is performed at Kernel-mode, integrity scanning must be made at another level, concerning dynamic and non-dynamic Kernel memory. For example the *ls* command, which returns the list of the contents of a directory, delivers control to Kernel invoking the *sys_getdents*. The rootkit can modify this system call in order to hide files or even processes (of the directory listed is */proc*).

Kernel rootkits are installed directly into Kernel memory area modifying important Kernel structures, function pointer tables, or patching the Kernel code, in order to filter the data that the attacker pretends to conceal from the administrator.

The current section covers the attack vectors used nowadays for intercepting system calls and other subverting approaches. Also describes the implementation of one application intended to detect the threats.
This section explains several methods that can be used to intercept the execution flow of a system call. The figure 1 shows the path of an invocation to a system call from the moment that a user land application calls, until it reaches the depths of the Kernel. The system call in the figure is sys_getdents (get directory entries), and by taking a first glance at the image we can see that there are many places where the interception can be installed in order to provide what the attacker wants: Control of the call.

Figure 1 – Linux Kernel Attacks – The Big Picture

Next follows the explanation of each technique that can be used to redirect the system call table.

2.1. Simple system call redirection

Once the system call table is found, an entry in the table can be replaced with the address of another function. Before Kernel Version 2.6, when the sys_call_table variable was exported, almost every rootkit used this method to hide its presence.

```c
saved_syscall = sys_call_table[SYSCALL_NUMBER];
sys_call_table[SYSCALL_NUMBER] = new_syscall;
```

Today, Kernel developers hardened this task because the table is not directly available to the Kernel module, so the only challenge is to find where it is in Kernel memory. After that, it is a matter of pointer replacement. This is the simplest way of hooking a system call, and also the easiest method to detect. Nevertheless, today, rootkits tend to use other methods.

2.2. Inline hooking Kernel functions

Inline hooking works by replacing the first bytes of a function with an unconditional jump, forcing the instruction pointer to jump to the hijacking function.

Figure 2 shows the bytes before the replacement in red. The HijackFunction can control the input and/or output data flow of the function which is being intercepted. When the Function is invoked, the new bytes force the instruction pointer to jump to the HijackFunction. Inside this function, the attacker has control over the arguments and is able to modify them. It's important not to forget to call the Function using the trampoline, which is a function that executes the old bytes that were replaced and then jumps to the following instruction, keeping the call to the function consistent. The returning value can also be modified.

Note that this hook implies the modification of read-only segments of memory, more precisely, .text sections. Before writing to memory, it's important to change the allowable accesses of the region using the function mprotect in order to avoid a segmentation fault for writing to non-writable position in memory. The memory segment should be made writable and after the byte code modification, the permissions of the segment should be restored.
Since inline hooking relies in code section modification and because these sections are read-only, a memory integrity check (with a trusted baseline) should be enough to detect that something might be wrong.

2.3. Hooking interrupt 0x80 / Sysenter instruction

First, it is important to understand the role of the interrupt descriptor table (IDT). This table is a data structure designed to implement an interrupt vector table in the x86 architecture. There are 256 interrupts supported which can be triggered with one of three types of interrupts: Software interrupts, hardware interrupts and processor exceptions. The first 32 entries are reserved for processor exceptions, and any 16 of the remaining entries can be used for hardware interrupts. The rest are available for software interrupts.

Each entry in this table holds, between other security related information, the address of the interrupt handler (or Interrupt Service Routine - ISR) that processes the interruption. When an interrupt is triggered, the processor multiplies the interrupt number by the length of each entry in the table (8 bytes), and then adds the offset of the beginning of the table, finding the correct address of the interrupt handler (The i386 Interrupt Descriptor Table, 2007).

The interrupt number 0x80 is a software interrupt designed to dispatch the intended system call to be executed for the User-mode application, this way, calling the system_call function in Kernel code. The system_call function inspects the contents of the eax register and invokes the respective system call routine indexed by the eax value (the number of the system call) in the system call table.

The location of the IDT is maintained by a 48 bits special register named IDTR which can be stored and loaded using the sidt (store interrupt descriptor table) and lidt (load interrupt descriptor table) instructions, respectively. An attacker can replace the interrupt service routine pointer for interrupt 0x80 (in the table) with another interrupt handler that fits the needs of the attacker, hiding information for certain system calls. A rootkit can also make a copy of the table and change the "official" base address of the IDT, to point to a new table, by modifying the IDTR register. This can be stealth because the attacker can hide modifications made to the interrupt table if the anti-rootkit scanner scans only for modifications of the original table.

Another alternative is to patch the system_call function, which is located in the Kernel code, via inline hooking.

Although, newer platforms such as Windows XP, 2003, Vista and recent versions of Linux Kernel 2.6 use another method to call the system services. The occurrence of an interruption causes the CPU to load one interrupt gate and one segment descriptor from memory to know what interrupt handler to call. This incurs in a con-
siderable overhead that is aggravated with high frequency of system service calls made by User-mode applications.

Because of performance reasons, INTEL and AMD simultaneously and independently developed their versions of a fast system call in alternative to the interrupt 0x80 mechanism. INTEL came up with the SYSENTER instruction (SYSENTER, 2006) and AMD with SYSCALL. Both are very similar in what concerns to its functionality, but in this work I will focus on the SYSENTER fast system call.

When the SYSENTER instruction is invoked, processor passes control to the "IA32_SYSENTER_EIP" which is a register in one of the Model-Specific Registers (MSRs). In Linux Kernel, IA32_SYSENTER_EIP contains the address of the function sysenter_entry, which will process the system call as explained earlier in this subsection.

If we want to control the execution flow triggered by the SYSENTER instruction, we need to manipulate the IA32_SYSENTER_EIP register, but, since this register is privileged, we are only able to read and write to it from Ring Zero, using the rdmsr and wrmsr instructions.

The following assembly code illustrates how this hook can be installed:

```
movl $0x176, ecx // location of the IA32_SYSENTER_EIP register
rdmsr // read register value
movl eax, orig_sysenter // store in orig_sysenter
movl new_sysenter, eax // store into eax register the new SYSENTER routine
wrmsr // save it to IA32_SYSENTER_EIP
```

After the execution of this code, if SYSENTER was not already hijacked, orig_sysenter should be pointing to sysenter_entry Kernel function.

It is important not to forget to restore the register when unloading the rootkit or the system will crash because every system call uses the SYSENTER instruction and it is pointing to an invalid address. When inside the new_sysenter function, one is able to check and modify the arguments in the stack which have all the information about the invoked system call from user land. After the payload, the new_sysenter function must jump to the orig_sysenter, this way, keeping the original system behavior.

Yet, I never saw a Linux rootkit using this technique.

Detecting a SYSENTER hook is extremely easy: Just read the value of IA32_SYSENTER_EIP and check if it has the address of sysenter_entry function. If not, some malware should be controlling the execution flow.

### 2.4. Runtime Kernel patching

Runtime Kernel patching is similar to inline hooking in the sense that it is based in Kernel memory modification. In 1998, Silvio Cesare presented several methods (Runtime Kernel patching, 1998) for extracting and modifying sensitive information from the Kernel through the /dev/kmem device. Three years later, in 2001, sd and devik published an article in the phrack magazine showing how to subvert the Kernel into using another system call table (Linux on-the-fly Kernel patching without LKM, 2001), instead of the original one.

As explained in the second chapter, nowadays, this is not much of a concern because the /dev/kmem device is read-only in the recent versions of the Kernel. However, if the Linux Kernel is compiled with LKM support, it is still possible to use the same method to force the Kernel to use another system call table created by the rootkit, possibly eluding a weak anti-rootkit if it only searches for modifications in the original table.

Because Kernel no longer exports the sys_call_table symbol, this method seems to be the best way to find its address. The system_call function takes the system call number and uses it to index into the sys_call_table vector to find the specific system call needed by the user land application. Somewhere after the beginning of that function, resides the call to the actual table. We need to search for the following code:

```
call *sys_call_table(,%eax,4)
```
Which correspond to bytes "ff 14 85 XX XX XX" where XX bytes belong to the actual sys_call_table address, the one we are searching. The following steps resume the technique:

♦ Find IDT base address with sidt instruction.
♦ Seek to the entry corresponding to int 0x80 in the table.
♦ Get the interrupt handler address of system_call from the table entry.
♦ Search for the three consecutive bytes: ff 14 85.

Now, the attacker can change the four bytes after the pattern, placing the address of a new system call table, which he can happily modify. Since this method relies in byte code searching, it is not guaranteed that will work between Kernel versions, but I have tested it with many Linux 2.6 Kernels and it seems to be quite portable. Probably because it's unlikely that the system call mechanics are changed.

3. Defending the Kernel

Rootkits can be hard to detect, especially when they operate in Kernel-land because at this level they are able to alter functions used by all running applications, including anti-rootkit software.

The current section describes what actions can be taken in order to protect the Kernel from the attacks shown in the previous section.

3.1. Kernel Integrity Checking

Generally, integrity checkers are more concerned about finding suspicious activities by searching in the physical storage. Physical storage is, indeed, important to analyze and in an infected system it may contain evidences of an intrusion, however, malware only needs to survive in memory. If the attacker concerns enough about stealthiness, he should avoid the file system altogether. This means that the consistency of the running Kernel memory also should receive attention and this is what this subsection is all about.

3.1.1. Fingerprinting sensitive places

Fingerprinting works like a Kernel memory "photograph" of sensitive places, taken at the time that the system was clean. Later in time, the administrator can compare the current Kernel memory image with the fingerprint taken previously, and check for system modifications.

This method can lead to false positives if the administrator does not make a fingerprint after every relevant system change. Note that Kernel fingerprint detection is only sensitive to system changes, which means that if the fingerprint is taken when a rootkit is active, it will not detect any system change (unless the rootkit is removed, of course).

The fingerprint should be saved in a safe place - preferably encrypted and/or signed - where the administrator is sure that the attacker cannot reach, in order to prevent him from taking another fingerprint of the subverted Kernel and replace the original one.

A good fingerprint should be based in the following: System call table, Interrupt Descriptor Table, IA32_SYSENTER_EIP address, IDTR register, Kernel Symbols (first bytes of the functions) and other important function pointers such as the /proc virtual file system lookup function (which will detect the adore-ng rootkit, for example).

Zeppoo anti-rootkit (Zeppoo: Decent Rootkit Detection for Linux, 2006) implements this detection method pretty well.
3.1.2. Verifying the integrity of important dynamic regions in Kernel memory

In what concerns to dynamic Kernel memory, this is somehow similar to fingerprinting but with the difference that the last one blindly saves the information without checking its validity.

Here, a good anti-rootkit should inspect the correctness of all regions where important information is stored in the system, in other words, looking for kooks.
These are the type of integrity checks that are used in the next subsection with the "Proactive Detection Approach". Continue reading for deeper details.

3.2. A Proactive Detection Approach

In this subsection, I present a proactive detection approach based in the analysis of certain parts of the Kernel before the invocation of a system call. Note that this is different from a Proactive Defense system - as the Kaspersky Anti-Rootkit Software for Windows - because this method does not prevent the malware from being installed, but it detects that it is in Kernel memory.

Because the system call table is usually targeted by rootkits and depending on each system call that is being called, the malware may be subverting different parts of the Kernel, it should be a good idea to check specific locations of the system for consistency, when each system call is invoked. Being able of executing code after a user land request to the Kernel and before the execution of a system call, gives us the possibility to scan certain parts of the system when it is more convenient, hence the word "proactive".

The basic idea behind this method is to clone the system call table, and replace every function pointer to point to stub functions, where some verifications to the system can be performed. When the checks are done, the actual function call is invoked with the help of the cloned table, which is still the same as the original table before the installation of the defense system.

By looking at the figure 2, the following pertinent question might outcome: "Why not redirect every entry in the system call table to the same Inspection Stub?". The reason for this relies in the fact that with a unique Inspection Stub, we would only have one way to identify the destination system call - reading the value of EAX register - which is completely unreliable if a rootkit intercepts the system call and changes it.

Since system calls are invoked so many times during system operation, it is important to reduce - as much as possible - the overhead created by the checks of this detection system.

Relevant checks to be made in the "System Inspection Routine" are described:

♦ **Check the system call table entry** - When a certain system call is invoked, we can check if the corresponding entry in the original system call table is correct, meaning that it should contain the address of the respective Inspection Stub.
Check Call Stack - Note that when the Inspection Stub is executed, the call stack should contain always the same functions. This leaves us with a method for detecting another possible incongruity: Check if the call stack is something similar to the following:

\[
\text{[<e0a8b2cd>] Inspection\_Stub+0xd/0x20 [Inside module code]} \\
\text{[<c01040f2>] sysenter\_past\_esp+0x6b/0xa9 [Inside Kernel code]}
\]

If the call stack shows more than these two functions, probably there is rootkit code being executed. Nevertheless, for performance reasons, the default Kernel configuration comes without frame pointers which means that we won’t be able to have good stack traces having the EBP register as a general purpose register. Selecting the “Compile the Kernel with frame pointers” option (if present) in the Kernel configuration menu, sets the CONFIG_FRAME_POINTER flag enabling the EBP register as the frame pointer. Actually, the frame pointer has no other use unless to provide a correct stack trace, so developers decided to remove the use of it in order to have a smaller and slightly faster Linux Kernel image for default installations. So, for this check to work, one has to enable this option before compiling the Kernel.

- **Check first bytes of the system call function and the system call table dispatcher function** - Here, one of two alternatives can be taken: (1) Check for an unconditional jump at the beginning of each function to detect a possible inline hook. (2) Test the integrity of the first bytes with a trusted database. This database should have previously gathered this information.

- **Check IA32_SYSENTER_EIP** – if the system uses the SYSENTER fast call, it is important to check if the IA32_SYSENTER_EIP register has the address of the correct Kernel function, which is sysenter_past_eip.

- **Check “int 0x80”** - In case the system is using the old method (with the Interrupt Descriptor Table), test if entry 0x80 in the IDT has the address of the correct interrupt service routine, which is the system_call function.

- **Check for patched Kernel code** - As stated above in 2.4, we can check if the correct system call table is being used by inspecting the byte codes in the system_call function.

- **Check Virtual file system** - If the system call is related to disk input/output, the defense system may check some relevant function pointers of the virtual file system (The Linux Virtual File-system Layer).

When a check in this system detects that something is wrong, it cannot stop the system call from being executed because it will get the system unstable. Instead, it should alert the administrator and give as much details as possible so he or her can understand what is wrong and where. With this proactive detection system running, we can defend the system against most of the attacks stated in section 2.
Table 1 – Correspondence between each system check and Kernel attack.

<table>
<thead>
<tr>
<th>Check to perform</th>
<th>Attack to detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Call Table Pointers</td>
<td>Simple Redirection</td>
</tr>
<tr>
<td>Verify CallStack</td>
<td>IDT/Sysenter/Simple Redirection</td>
</tr>
<tr>
<td>Byte Code</td>
<td>Inline Hooking</td>
</tr>
<tr>
<td>Sysenter</td>
<td>Sysenter Hook</td>
</tr>
<tr>
<td>int 0x80</td>
<td>IDT Hook</td>
</tr>
<tr>
<td>Patched Code</td>
<td>Runtime Patching</td>
</tr>
<tr>
<td>VFS</td>
<td>Other Lower level hooks</td>
</tr>
</tbody>
</table>

The table 1 relates each check described above with the respective attack that it intends to detect.

### 3.2.1. Performance Tests

Each system check, along with the redirection of the system call itself contributes to an overall delay of the total time that the system call takes. The present subsection studies the total overhead produced by the proposed defense system.

<table>
<thead>
<tr>
<th></th>
<th>Average time (ms)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total system call time (no IDS)</td>
<td>520</td>
<td>0,0</td>
</tr>
<tr>
<td>No checks</td>
<td>535</td>
<td>2,8</td>
</tr>
<tr>
<td>All checks</td>
<td>1126</td>
<td>116,5</td>
</tr>
<tr>
<td>SCT</td>
<td>558</td>
<td>07,3</td>
</tr>
<tr>
<td>Call Stack</td>
<td>619</td>
<td>19,0</td>
</tr>
<tr>
<td>Sysenter</td>
<td>804</td>
<td>54,6</td>
</tr>
<tr>
<td>Int 80</td>
<td>606</td>
<td>16,5</td>
</tr>
<tr>
<td>System_call jmp code</td>
<td>578</td>
<td>11,2</td>
</tr>
<tr>
<td>Syscall Inline check</td>
<td>565</td>
<td>3,6</td>
</tr>
<tr>
<td>All checks except sysenter</td>
<td>752</td>
<td>44,6</td>
</tr>
</tbody>
</table>

Table 2 illustrates the estimated overhead in percentage relatively to the original system call time.

It is crucial to analyze each check to understand which are consuming more time. The results in the previous table were grabbed from the following test conditions: Measure the time taken during five million invocations to the system call `sys_time`. The “Average time (ms)” column is the average time taken in twenty of these tests. “Percentage” column helps to understand the overhead produced by these checks towards a system with no IDS installed. The tests were performed in an Intel Pentium M 760 Processor (2MB L2 Cache, 2.0 Ghz) with 1.3GB of RAM.

We can clearly see that the `sysenter` check is consuming considerable time. The reason for this is in the `rdmsr` instruction.

Because the remaining verifications are based on simple byte comparison in memory (except for the call stack check which is slightly slower), they are faster and do not represent a significant overhead in a system call.

We can minimize this overhead by taking the following into consideration: Unlike the `sysenter` and `int 0x80` checks, the remaining verifications are unique for each system call, which means that we can execute the first ones with less frequency, for instance, after each ten system calls.
4. Conclusions

Detection techniques do not prevent the system from being attacked in the first place. Generally, they only indicate what is wrong with the system and where is the incongruity. Some of them only alert when the detection tool is executed (because of a scheduled scan or on administrator’s demand). The approach presented in 3.2 is permanently running in background as a Linux Kernel module, and logs suspicious system changes when detected. Although, it still does not prevent the attacks from taking place because the invocations to the system calls cannot be stopped. If they are, the system will be left in an inconsistent state. Generally, the smartest thing to do when the administrator receives the alert of an intrusion, is to reinstall the entire system. The administrator can trust his or her personal anti-rootkit tools for removing the threat, but can never be sure that the system is clean simply because rootkits are a constantly growing threat. There are always new and different ways of hiding resources in the system and some of them have never been disclosed to the public.

5. Bibliography


