

Detecting Rootkits And Kernel-level Compromises In Linux

Mariusz Burdach 2004-11-18

Editor's note: a Korean translation of this article, courtesy of Netsecure Technology, is [available here](#) as a PDF document. Other requests for translation can be sent to the [editors](#).

This article is intended to outline useful ways of detecting hidden modifications to a Linux kernel. Often known as a rootkit, this stealthy type of malware gets installed in the kernel of an operating system and requires special techniques by Incident handlers and Linux system administrators to be detected.

In this article we will make use of just one tool, `gdb`, the GNU debugger, to detect whether a Linux operating system has been compromised. The package that includes this tool can be found in almost every Linux distribution by default. The second goal of this paper is a presentation of an intruder's popular methods of "patching" the kernel of a Linux operating system. By understanding the attack vector, we can easily detect that our machine has been compromised or select the right tools to monitor our critical machines.

This focus on detecting kernel modifications is important because it is the most stealthy of all methods for an intruder to install malicious code in an operating system. Once this malicious code is in place, intruders can defeat most commercial and free host intrusion detection systems (IDSs) which monitor the integrity of the operating system's files.

Introducing the threat of rootkits

More and more user-mode malicious programs, such as Trojan horses, backdoors or rootkits, modify existing operating system software. To install one of these tools on a victim's machine, an attacker must replace or modify the normal programs that are associated with the operating system. For example, let's consider a replacement of the ubiquitous `ls` command. Normal users and administrators use the `ls` command to list the contents of a directory, but a modified version of `ls` will hide the attacker's files. Tools which can detect this kind of modification are called file *integrity checkers*.

Let's suppose that an attacker doesn't replace or modify any existing programs, such as `ls`, on the file system. Instead, suppose the attacker replaces or modifies various components of the kernel. We know that many user-mode programs, such as `ps`, `ls` or `lsOf`, use the kernel to perform some of their tasks. For example, when an administrator runs the `ls` command to list the contents of a directory, the `sys_getdents` kernel-level system call is invoked. An attacker can therefore modify this kernel component to hide some files or processes.

Now, let's consider the another example where an attacker modifies both the `sys_open` and the `sys_read` system calls to block access to a set of selected files. These same system calls are usually

used by file integrity checkers to verify the integrity of important system files, like the kernel image or loadable kernel modules. When these tools try to compare the hashes of files to their previous values, they will remain the same even when these files have, in fact, been modified. In other words, critical files could be shown to be intact by file integrity checkers when they are not, just by hooking two system calls. It should be quite obvious, then, that when kernel components are modified by an attacker, users and administrators cannot trust any of the results received from the kernel or from any security related tools that run as a user.

Linux kernel-mode rootkits, or other kinds of malicious code, are installed directly in a memory area reserved for the kernel code and they are really powerful. They modify kernel structures in order to filter data which will be hidden from system administrators. To filter this data it is necessary to take control over some kernel components like system calls, interrupt handlers, internal functions of the netfilter, and more. It is easy to imagine several places in the kernel of an operating system where such control can be manipulated. At this point, it is now important to understand some of the most popular attack vectors.

Understanding the attack vector

The most common targets of a compromise are system calls. This vector is chosen by intruders for two reasons: because it is the easiest way to take control over a compromised machine, and also because system calls are very powerful. System calls are basic functions used by an operating system. For example, they are used to read and write data to and from files, they are used to get an access to various devices, to run executables, and so on.

There are about 230 system calls in the current stable version of the Linux kernel, 2.4.27, and about 290 system calls in the Linux kernel 2.6.9. Note that the number of system calls changes depending on the version of the kernel. The full list of system calls in your kernel is always available in the file `/usr/include/asm/unistd.h`. It should be also noted that not all system functions are normally modified by intruders -- however, there are a few very popular ones. These system calls are presented below, in the Table 1. They should be closely monitored by administrators and, of course, by host intrusion detection systems. The full description of each system call can be found in the system manual (2) -- "Linux Programmer's Manual" for those who are interested in further details.

System call name	Short description	ID
<i>sys_read</i>	This system call is used for reading from files	3
<i>sys_write</i>	This system call is used for writing to files	4

<i>sys_open</i>	This system call is used to create or open files	5
<i>sys_getdents/</i> <i>sys_getdents64</i>	This system call is used to list a content of directories (also /proc)	141/220
<i>sys_socketcall</i>	This system call is used for managing sockets	102
<i>sys_query_module</i>	This system call is used for querying loaded modules	167
<i>sys_setuid/sys_getuid</i>	This system calls are used for managing UIDs	23/24
<i>sys_execve</i>	This system call is used for executing binary files	11
<i>sys_chdir</i>	This system call is used to change the directory	12
<i>sys_fork/sys_clone</i>	This system calls are used to create a child process	2/120
<i>sys_ioctl</i>	This system call is used to control devices	54
<i>sys_kill</i>	This system call is used to send signal to processes	37

Table 1. Important Linux system calls, their description, and their system call IDs.

For the above table, note that the ID is the number of an entry in the system call table. For the purposes of this article, the IDs are used for Linux kernel 2.4.18-3.

While all of the examples presented in this article were tested on the Red Hat 7.3 with kernel 2.4.18-3, similar steps can be done during an investigation of other versions, including the latest Linux kernel 2.6.x. Some differences can appear in internal structures of the Linux kernel 2.6.x, however. For example, the address of system call table is kept inside of the function *syscall_call* instead of system call handler named *system_call*.

Modifying the system call table

Current addresses of system calls are kept in the system call table, in a memory area reserved for the

kernel of an operating system. Addresses there are kept in the same order as their functions, and are presented in the `/usr/include/asm/unistd.h` file. System calls are identified by the number of the entry (ID) in the system call table, as we saw in the table above.

Let's start with an example. When the `sys_write` system call is invoked, its ID of 4 is placed into the `eax` register and a software interrupt is generated (int 0x80). There is a special interrupt handler which keeps this address in its interrupt descriptor table and is responsible for handling the interrupt (again, int 0x80). Next, the system call handler `system_call` is invoked. This handler can locate the direct address of the requested system call by knowing the address of the system call table and the ID of the system call (which is kept by the `eax` register). There is also a longer way to invoke this system call handler, but I have omitted some details to simplify this article.

The first method an intruder uses when taking control over a desired system call is to overwrite the address of the original system call in the system call table. When the system call is requested, the handler calls a replacement function. We can easily watch these addresses, kept in the system call table, by using the `gdb` tool. That is why `gdb` is so useful for detecting this kind of malware.

Of course, another problem arises. We have to be sure that the current addresses in the system call table are not modified -- that we are not already compromised. How can we verify this? The addresses of system calls are always permanent and do not change after a reboot of the operating system. These addresses are set during kernel compilation, so knowing the original addresses we can compare them to addresses currently placed in the system call table. This information on the original addresses is written into two files on the file system during compilation time. The first one is the `System.map` file. This file contains the names of symbols and their corresponding addresses. The second file is the kernel image which is loaded into a kernel memory during system initialization. An uncompressed version of the kernel image is presented as the `vmlinux-2.4.x` file, and is usually placed in the `/boot` directory, or in a build directory defined before the kernel compilation.

Sometimes only a compressed version of the kernel may be available (named `vmlinuz-2.4.x`). In this case, before starting our investigation we have to uncompress that kernel image. If an intruder hasn't modified these files (the compressed/uncompressed kernel image and the `System.map`), or if we had a trusted copy of these files, we can compare the original addresses to addresses which are currently kept by the system call table. It should need not be mentioned that we should copy these critical files or at least create a hash value from each of them immediately after a kernel compilation.

We could also use a simple loadable kernel module, presented in the references section of this article, to print virtual addresses of each system call. To do this we compile the source code as follows: `gcc -c scprint.c -I/usr/src/linux/include/`. After the loading of the compiled module (the `scprint.o` object), addresses of each system call are written automatically to the `syslog` file. From time to time, we

should run this module to compare original addresses to the current state of the kernel.

In most cases, the kernel is modified by rootkits after a system initialization. It is done by loading a malicious kernel module or by injecting some malicious code directly into the `/dev/kmem` object. Rootkits usually do not modify the kernel image or the `System.map` file. Therefore, to detect any modification of entries in the system call table we have to print all addresses, currently stored by the system call table, and then compare the result to the addresses kept in the kernel image (in our case, `vmlinux-2.4.x`). The memory of the operating system is represented by the object `kcore` which can be found in the `/proc` virtual file system.

The first step is to find an address of the system call table. This should be an easy task, as shown below, because the symbol `sys_call_table` is presented in the `System.map` file:

```
[root@rh8 boot]# cat System.map-2.4.18-13 | grep sys_call_table c0302c30 D
sys_call_table
```

Now, using the `nm` command, we can find an address of the system call table. This command allows us to print all symbols from the kernel image that has not been stripped:

```
[root@rh8 boot]# nm vmlinux-2.4.18-13 | grep sys_call_table
c0302c30 D sys_call_table
```

Using the `gdb` tool, we can print the entire contents of the system call table from the kernel image, as shown in the listing below. Printed addresses correspond to system calls as defined in the `entry.S` file in the source files of the kernel. For example, the entry 0 (`0xc01261a0`) is the `sys_ni_syscall` system call, entry 1 (`0xc011e1d0`) is the `sys_exit` system call, entry 2 (`0xc01078a0`) is the `sys_fork` system call, and so on.

```
#gdb /boot/vmlinux-2.4.*
(gdb) x/255 0xc0302c30
0xc0302c30 :      0xc01261a0 0xc011e1d0 0xc01078a0 0xc013fb70
0xc0302c40 : 0xc013fcb0 0xc013f0e0 0xc013f230 0xc011e5b0
0xc0302c50 : 0xc013f180 0xc014cb10 0xc014c670 0xc0107940
0xc0302c60 : 0xc013e620 0xc011f020 0xc014bcd0 0xc013e9a0
...
```

We can also print the address of each system call by providing its name, as shown below:

```
(gdb) x/x sys_ni_syscall
0xc01261a0 :      0xffffdab8
((gdb) x/x sys_fork
0xc01078a0 :      0x8b10ec83
```

Now, by using the `gdb` tool (or the module we compiled, `scprint.o`), we must dump the current entries from the system call table. Finally, we compare the result to values received from the kernel image or the values received after kernel compilation.

To print out the current state of the kernel, we must run the `gdb` tool with two parameters. The first one is the image of the kernel (`vmlinux-2.4.x`), and the second is the object `/proc/kcore`. Then we use the address of the system call table, received from the `System.map` file, to print the entries of the system call table.

```
#gdb /boot/vmlinux-2.4.* /proc/kcore
(gdb) x/255x 0xc0302c30
0xc0302c30 :      0xc01261a0      0xc011e1d0      0xc01078a0      0xc88ab11a
0xc0302c40 : 0xc013fcb0      0xc013f0e0      0xc013f230      0xc011e5b0
0xc0302c50 : 0xc013f180      0xc014cb10      0xc014c670      0xc0107940
0xc0302c60 : 0xc013e620      0xc011f020      0xc014bcd0      0xc013e9a0
...
```

As we can see from the above output, one of the addresses of the system call has indeed been changed. This is entry number 3 in the system call table (counting entries from 0), and is bolded in the above output for clarity. In the `/usr/include/asm/unistd.h` file we can find the name of this suspicious system call, which we determine to be `sys_read`.

The another sign of a system compromise is that the new virtual address of this function (`sys_read`) is above `0xc8xxxxx`. It is, by nature, quite suspicious. A Linux operating system by default can address up to 4 GB. Virtual addresses range from `0x00000000` to `0xffffffff` in hexadecimal notation. An upper part of this virtual memory area is reserved for kernel code (their values range from `0xc0000000` to `0xffffffff`). When a new loadable kernel module is loaded, the `vmalloc` function allocates a part of this memory for the code of the module. It allocates a memory region -- usually starting from `0xc8800000`. So, whenever the reference to the address of the system call is above this address, as we saw in this example, it indicates that our kernel could be compromised. At this point, it becomes necessary to look

closer at this system call.

System call hooking

Now we will examine a method of detecting system call hooking. None of entries in the system call table are modified with this method. Instead, the first few instructions of the original function are overwritten with a jump to a replacement function (called a detour function). Let's imagine that an intruder wants to hook the `sys_read` system call. He must first load the replacement function into memory and then place the address of this function in the first few bites of the original function. In doing so, the intruder must redirect an execution flow of the original function to the replacement one. Assembly instructions, like `call` or `jmp`, are usually used.

To detect whether any system call has been hooked, we must print out all the instructions of the target function. We start by running the `gdb` tool with two parameters (the kernel image and the `/proc/kcore` object). Next, we must disassemble the original function by using the `disass` command inside the `gdb` tool, as shown below.

```
#gdb /boot/vmlinux-2.4.* /proc/kcore
(gdb) disass sys_read
Dump of assembler code for function sys_read:
0xc013fb70 :          mov     $0xc88ab0a6,%ecx
0xc013fb73 :          jmp     *%ecx
0xc013fb77 :          mov     %esi,0x1c(%esp,1)
0xc013fb7b :          mov     %edi,0x20(%esp,1)
0xc013fb7f :          mov     $0xffffffff7,%edi
...
```

From the above output, we can see that the first instruction moves the value (the address of the replacement function) to the `ecx` register. The second instruction does an indirect jump to this virtual address - `0xc88ab0a6`.

To make sure that the `sys_read` system call was hooked, we must disassemble the original function. The original function is presented in the kernel image `vmlinux-2.4.x`.

```
#gdb /boot/vmlinuz-2.4.*
(gdb) disass sys_read
Dump of assembler code for function sys_read:
0xc013fb70 :          sub     $0x28,%esp
0xc013fb73 :          mov     0x2c(%esp,1),%eax
0xc013fb77 :          mov     %esi,0x1c(%esp,1)
0xc013fb7b :          mov     %edi,0x20(%esp,1)
0xc013fb7f :          mov     $0xffffffff7,%edi
...
```

The output confirms that the `sys_read` system call has indeed been modified. To investigate what this new function really does, we can disassemble the function by using the `gdb` tool.

A modification of the system call handler

Using the method as described above, we can also disassemble other critical functions in the kernel memory. One of them is the system call handler, named `system_call`, that is used to locate and call requested system calls. The handler uses the system call table to find an address of the requested system call. By disassembling this handler, we can check if the right address of the system call table is used or if the handler is hooked. In an attack scenario, an intruder can create his own system call table using replacement system calls. Then, he can place a new address for the system call table in the system call handler.

```
(gdb) disass system_call
Dump of assembler code for function system_call:
0xc01090dc :          push   %eax
0xc01090dd :          cld
0xc01090de :          push   %es
0xc01090df :          push   %ds
0xc01090e0 :          push   %eax
0xc01090e1 :          push   %ebp
0xc01090e2 :          push   %edi
0xc01090e3 :          push   %esi
0xc01090e4 :          push   %edx
0xc01090e5 :          push   %ecx
0xc01090e6 :          push   %ebx
0xc01090e7 :          mov     $0x18,%edx
```

```
0xc01090ec :    mov    %edx,%ds
0xc01090ee :    mov    %edx,%es
0xc01090f0 :    mov    $0xffffe000,%ebx
0xc01090f5 :    and    %esp,%ebx
0xc01090f7 :    testb $0x2,0x18(%ebx)
0xc01090fb :    jne    0xc010915c
0xc01090fd :    cmp    $0x100,%eax
0xc0109102 :    jae    0xc0109189
0xc0109108 :    call   *0xc0302c30(,%eax,4)
0xc010910f :    mov    %eax,0x18(%esp,1)
0xc0109113 :    nop
End of assembler dump.
```

Please note that this disassembled handler contains the address of the original system call table.

Useful tools

Of course, it would be even better to automate all tasks described in this article. One of the ways to use this is with host based intrusion detection systems which monitor critical kernel structures in real time. For example, the Samhain tool can be used. This tool was briefly described in a previous Infocus article, "[Host Integrity Monitoring: Best Practices for Deployment](#)". The Samhain tool is able to monitor the system call table, the first few instructions of every system call including some handlers, the interrupt description table, and much more.

When considering an IDS to monitor the integrity of our kernel, we must remember one basic rule -- every kind of monitoring tool such as this must be installed on a clean install of the operating system, as this is the only way to know that we have not already been compromised. If we don't install any host based IDSes we should at least create a hash sum of the kernel image.

Summary

As we have seen, the gdb tool can be very useful for detecting a kernel-level operating system compromise.

The detection of a kernel level compromise can be very complicated without confirmation that at least one source of knowledge is trusted. In examples used in this article, the

kernel image is the trusted source of knowledge.

References

- [1] Daniel P. Bovet, Marco Cesati. "Understanding the Linux Kernel", 2nd Edition. O'Reilly; 2002.
- [2] "Host Integrity Monitoring: Best Practices for Deployment", <http://www.securityfocus.com/infocus/1771>.
- [3] "Linux on-the-fly kernel patching without LKM", <http://www.phrack.org/phrack/58/p58-0x07>
- [4] System manual (2), "Linux Programmer's Manual".
- [5] The GNU Project Debugger, <http://www.gnu.org/software/gdb/gdb.html> .
- [6] `scprint.c` is the loadable kernel module that allows one to print system calls from the kernel memory. This is available for [download](#) from SecurityFocus.
- [7] The samhain file integrity / intrusion detection system, <http://la-samhna.de/samhain/>

Author Credit

View [more articles](#) by Mariusz Burdach on SecurityFocus.

[Privacy Statement](#)

Copyright 2006, SecurityFocus