Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection

Existing techniques to detect kernel-level rootkits expose some infections, but don't identify specific attacks. This rootkit categorization approach helps system administrators identify the extent of specific infections, aiding in optimal recovery and faster reactions to future attacks.

JOHN LEVINE, JULIAN GRIZZARD, AND HENRY OWEN
Georgia Institute of Technology

On today's Internet, computers are vulnerable to a variety of exploits aimed at compromising their intended operations. Denial-of-service attacks can prevent a target from providing service to legitimate clients (such as a Web server) or prevent the targeted system itself from connecting to other computers. Some DoS attacks can cause systems to temporarily cease all operations. In other attacks, attackers attempt to gain root-level access and control a system as if they were the system administrators. Attackers can retain this access through various tools, including *rootkits*.

Rootkits are toolsets used by an attacker to retain root-level access to a system in a covert manner. To determine whether an attacker has installed a rootkit—and the extent to which it's compromised the system—administrators need trusted host-based techniques. Several existing network-based techniques let system administrators monitor their system's status. Network-based intrusion detection systems, for example, detect malicious activity at various network levels. Other host-based programs check file integrity at the system or host level. Such existing methods, however, might not detect the presence of a kernel-level rootkit or categorize its functionality.

Here, we present a framework to detect and classify rootkits and discuss a methodology for determining if a system has been infected by a kernel-level rootkit. Once infection is established, administrators can create new signatures for kernel-level rootkits to detect them. We conducted our research on a Red Hat Linux-based system, but our methodology is applicable to other Linux distributions based on the standard Linux kernel. We also believe the method can apply to other Unix-based systems and Windows-based systems.

Rootkit overview

Before widespread use of rootkits, system administrators generally trusted their system utilities to provide accurate information. The recent widespread use of rootkits means that attackers can now easily conceal their activities.[1] System administrators must therefore be constantly aware that seemingly trusted system utilities might be reporting false information.

Rootkit installation

A rootkit is like a Trojan horse in a computer operating system (OS), except that the attacker installs the rootkit. To install a rootkit, attackers must first have root-level access on a computer system. Once they have this access, they can install a Trojan-like program that masquerades as a new or existing system program. We use the term "program" to mean a sequence of instructions, and thus consider code in the kernel a program. This rootkit lets them subsequently reenter a system with root-level permissions.[2]

According to Harold Thimbleby and his colleagues, there are four categories of Trojans[3]:

- *direct masquerades* pretend to be normal programs;
- *simple masquerades* masquerade not as existing programs, but as new programs that appear to be different than they are;
- *slip masquerades* are programs with names approximating existing names; and
- *environmental masquerades* are OS programs that the user can't easily identify.

Here, we're primarily interested in direct masquerades

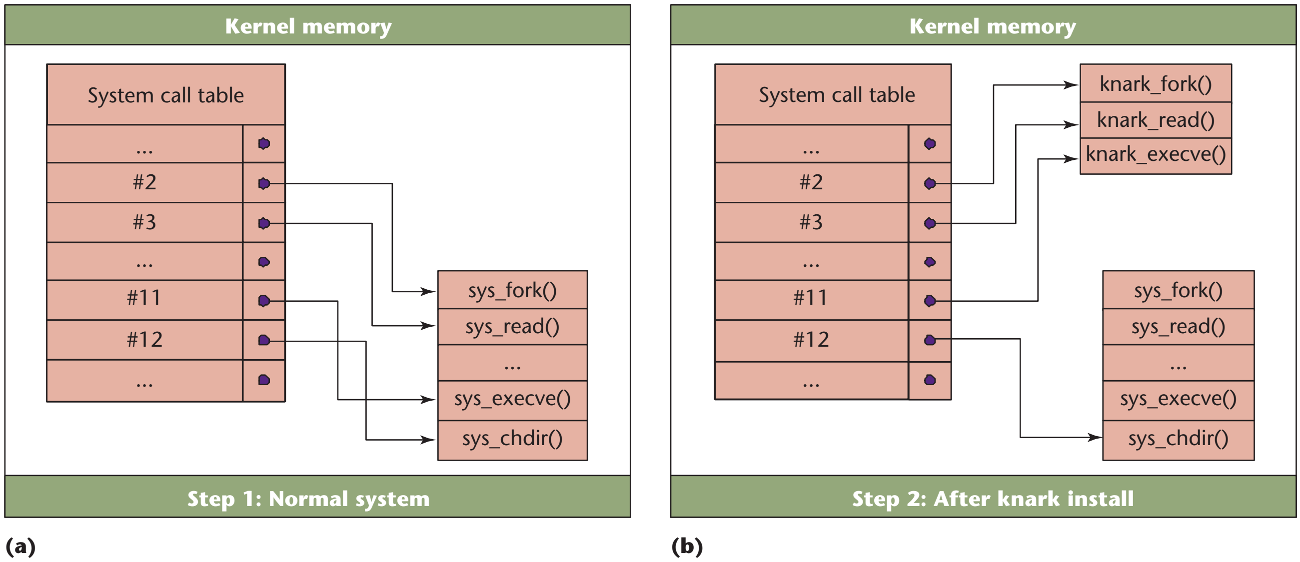


Figure 1. System call table modification. (a) Normal system operation. (b) Following installation, the knark rootkit redirects `sys_calls`.

and environmental masquerades. Although we address rootkits targeted at Linux or Linux-type OS kernels, our techniques apply to all rootkit types.

Kernel-level rootkits

Kernel-level rootkits are one of the most recent developments in the attacker community's arsenal.[4] The kernel is generally considered the core; that is, the lowest level of most modern OSs. The kernel provides the file system, CPU scheduling, memory management, and system-call-related OS functions.[5] Programs operating at user-level interface to the kernel through a system call. When an application performs a `sys_call`, it passes control to the kernel, which then performs the requested work and returns output to the requesting application. Therefore, system calls are one of the primary targets for kernel-level rootkit developers, but many of the kernel's data structures and code sections can be targeted. The `sys_call` addresses are maintained in the kernel memory's system call table data structure. Unlike a traditional user-level rootkit, which modifies critical system-level programs on disk, a kernel-level rootkit can replace or modify the system call table and other data structures within the kernel itself. This allows the attacker to covertly control the system. We focus on kernel-level rootkits that modify the system call table, although other kernel targets can be modified as well, including the virtual file system data structures.

There are three types of kernel-level rootkits that change system calls: those that modify the system call table, those that modify system call table targets, and those that redirect the system call table.

Table modification. With table modification, the attacker modifies selected `sys_call` addresses stored in the system call table. The kernel-level rootkit penetrates kernel memory using one of two features: a loadable kernel module (LKM), which can be built for Linux and some Unix-based OSs;[6] or system calls that read and write on the `kmem` device file.

Attackers can develop LKMs that create malicious `sys_calls` to hide files and processes, as well as to provide backdoors for return visits to the system. These LKM's also modify the `sys_call` address table by replacing legitimate `sys_call` addresses with the malicious ones.[7]

With a kernel-level rootkit, attackers can redirect a `sys_call` away from the legitimate `sys_call` to the kernel-level rootkit's replacement `sys_call`. An example table-modification rootkit is Creed's knark rootkit, introduced in 2001. Figure 1 shows how the rootkit redirects `sys_calls`.

Table target. With the table target kernel-level rootkit, the attacker overwrites the legitimate `sys_call` targets in the system call table with malicious code. The system call table does not need to be changed. Kernel-level rootkits can overwrite the first few instructions of the `sys_call` with a `jmp` instruction that redirects execution to the malicious code. We can detect this target-overwrite approach by comparing the current opcode bytes for each `sys_call` with their expected value, which must be previously stored offline.

Table redirect. In this type of kernel-level rootkit, the attacker redirects references to the entire system call table to a new system call table in a new kernel memory location. This new call table might contain the addresses of malicious `sys_call` functions, as well as original addresses for any unmodified `sys_call` functions.

One way attackers accomplish this redirection on a Linux system is by writing to `/dev/kmem`. The `/dev/kmem` device provides access to the running kernel's memory region in Linux kernels up to version 2.6.13. If attackers can find the proper memory location, they can overwrite portions of the kernel memory at runtime. Kernel-level rootkits redirect the system call table by overwriting the pointer to the original system call table with the address of a new system call table the attacker creates within the kernel memory.[6] Unlike the table modification method, this approach doesn't modify the original system call table. It can therefore pass many currently used consistency checks.

A rootkit classification framework

To develop a framework for classifying rootkits, we borrow some ideas from an existing framework for modeling Trojans and computer virus infections.[3] Our work's focus is more specific, however, in that our goal is to develop a method to classify rootkits masquerading as existing programs—either as new rootkits or as modifications of existing rootkits.

Fred Cohen defines a computer virus as a computer program that can replicate all or part of itself and attach this replication to another program.[8] The types of rootkits we target don't typically have this capability. An ideal rootkit program that aims to replace an existing program on the target system must appear to have the original program's functionality, while also having some additional malicious functionality. This added functionality can allow backdoor root-level access; it might also enable the program to hide specified files, processes, and network connections on the target system.

We use a rootkit's added functionality and associated elements to detect and classify rootkits. Various methods exist to compare the original and rootkit programs and identify the difference—or delta (Δ)—in functionality between them. This Δ can serve as a potential signature for identifying the rootkit in the case of nonpolymorphic rootkits.

Although evaluating a program file by its cyclical redundancy check (CRC) checksum is faster and requires less memory than comparing file contents,[9] this comparison tells us only that a current program file differs from its original program file. Using this check to detect rootkits won't tell us if the rootkit is a new rootkit or a modification of an existing rootkit. Our work builds on research to detect Trojan horse programs by comparing them to the original program that they're intended to replace.[9]

Framework overview

Our framework assumes that we have two programs:

- p_1, the original program, and
- p_2, a malicious version of program p_1 that provides rootkit capabilities on the target system.

Because they'll produce similar results for most inputs, we assume that an ideal rootkit and the program it intends to replace are indistinguishable in execution. Therefore, while not equal,[3] the two programs are similar enough to make it difficult to tell them apart by simply supplying different inputs to the programs.

Our framework uses three quantifiers (as defined in Thimbleby[3]) and one additional quantifier:

- similarity ($\sim$): a poly log computable relation on all possible computer representations (R), including the machine's full state (memory, screens, registers, inputs, and so on). A single representation of R is r. *Poly log computable* is a function that can be computed in less than linear time (and we can therefore evaluate a representation without examining the entire computer representation).
- indistinguishable ($\approx$): two programs that produce similar results for most inputs.
- a program's meaning ($[[\bullet]]$): what a program does when it's run.
- functionality ($\{((\bullet))\}$): set of possible functionalities for a program.

If p_2 is part of an ideal rootkit, then p_1 and p_2 are indistinguishable and will produce similar outputs for most inputs. We can therefore state that p_1 is indistinguishable from p_2 if and only if

$$\text{for most } r \in R : [[p_1]]r \sim [[p_2]]r \Rightarrow p_1 \approx p_2.^{3}$$

This means that, for most machine representations out of all possible representations, the results of program p_1 are similar to the results of program p_2, which implies that p_1 is indistinguishable from p_2.[3] Therefore, by comparing only program behavior for a set of random inputs, it is difficult to determine that p_2 is malicious.

Categorizing rootkits

We apply set theory to categorize rootkits as follows. If p_2 is an ideal rootkit of p_1, then most elements of $\{((p_1))\}$ exist in $\{((p_2))\}$. We can approximate that $\{((p_1))\}$ is a subset of $\{((p_2))\}$, because most $\{((p_1))\}$ elements exist in $\{((p_2))\}$, but $\{((p_1))\}$ is not equal to $\{((p_2))\}$. We write this as:

$\{((p_1))\} \subset \{((p_2))\}$ and $\{((p_1))\} \neq \{((p_2))\}$, meaning $\{((p_2))\}$ has at least one element that does not belong to $\{((p_1))\}$.

We identify the difference—or Δ—between p_1 and p_2 as follows:

$$\{((p_2))\} \setminus \{((p_1))\} = \{((p'))\}$$

is the difference between p_2 and p_1, containing only those elements belonging to $\{((p_2))\}$ that are not in $\{((p_1))\}$.

Next, assume we've identified p_3, another rootkit of p_1. We can identify this collection of programs as a type p_2 rootkit as follows. If $\{((p_3))\} - (\{((p'))\} \cap \{((p_3))\}) = \{((p_1))\}$, then p_3 has the same elements as program p_2 and is the same rootkit. If the preceding statement is not true, but some elements of $\{((p'))\}$ are contained in $\{((p_3))\}$ (there exist some $x \in \{((p'))\}$ such that $x \in \{((p_3))\}$), then we can assume that p_3 might be a modification of p_2. If there are no elements of $\{((p'))\}$ in $\{((p_3))\}$ (for all $x \in \{((p'))\}$, $x \notin \{((p_3))\}$), then we can assume that p_3 is an entirely new rootkit.

Although we present only a few examples here, we've examined numerous rootkits using our methodology. (The "Known rootkits" sidebar offers a list of currently known rootkits.)

Rootkits that modify the system call table

Several tools exist that can detect kernel-level rootkits on Linux based systems. One such tool is kern_check (http://la-samhna.de/library/kern_check.c). The kern_check program detects whether a kernel-level rootkit exists on a system, but it fails to indicate the rootkit's type. Our methodology helps categorize specific classes of kernel-level rootkits and can be applied to rootkits at other levels as well.

Our method categorizes a rootkit using an archived copy of all system call instructions from kernel memory. To accomplish this, we developed *ktext*, a C program that copies system call code, referenced by a start and end address, and then writes the executable object code to a file for future reference (see Figure 2). This lets analysts retrieve code that is currently running in the system kernel. Further, some types of kernel-level rootkits, such as knark, don't remain resident in memory after the system is rebooted. With our program, analysts can copy suspicious system calls offline for follow-on analysis prior to rebooting the system. A more robust approach for examining the kernel code requires a more secure trusted computing base, such as a virtual machine monitor.

Application results

To test ktext, we installed several kernel-level rootkits on several target systems and then ran the kern_check program, which compares the current system call table's `sys_call` addresses with the original kernel symbol map. The symbol map is created at kernel compile time and is stored in a file call `System.map`. This file can be stored offline. Any difference between the two tables indicates a system call table modification (see "The kern_check utility" sidebar).

With the knark kernel-level rootkit, the kern_check program identified eight redirected system calls and their addresses within kernel space (sidebar Figure A). We used ktext to copy the redirected system calls. Our analysis of the source code used to create the rootkit indicated that the redirected system calls were being written sequentially into kernel memory. (This might not always be the case; at times, it might be necessary to analyze the object code to identify individual system calls' start and end addresses.)

Next, we rebooted the system and again ran kern_check, which indicated that no system calls were being redirected. To test repeatability, we reinstalled the knark kernel-level rootkit via its loadable kernel module, and the kern_check program subsequently validated that the knark

Known rootkits

The Chkrootkit tool (www.chkrootkit.org) checks for signs of a rootkit installation and can detect many user-level and kernel-level rootkits. The tool's website includes the following list of rootkits, worms, and viruses that it can detect, beginning with the first detected.

1. lrk3, lrk4, lrk5, lrk6 (and variants)
2. Solaris rootkit
3. FreeBSD rootkit
4. t0rn (and variants)
5. Ambient's Rootkit (ARK)
6. Ramen Worm
7. rh[67]-shaper
8. RSHA
9. Romanian rootkit
10. RK17
11. Lion Worm
12. Adore Worm
13. LPD Worm
14. kenny-rk
15. Adore LKM
16. ShitC Worm
17. Omega Worm
18. Wormkit Worm
19. Maniac-RK
20. dsc-rootkit
21. Ducoci rootkit
22. x.c Worm
23. RST.b trojan
24. duarawkz
25. knark LKM
26. Monkit
27. Hidrootkit
28. Bobkit
29. Pizdakit
30. t0rn v8.0
31. Showtee
32. Optickit
33. T.R.K
34. MithRa's Rootkit
35. George
36. SucKIT
37. Scalper
38. Slapper A, B, C and D
39. OpenBSD rk v1
40. Illogic rootkit
41. SK rootkit
42. sebek LKM
43. Romanian rootkit
44. LOC rootkit
45. shv4 rootkit
46. Aquatica rootkit
47. ZK rootkit
48. 55808.A Worm
49. TC2 Worm
50. Volc rootkit
51. Gold2 rootkit;
52. Anonoying rootkit
53. Shkit rootkit
54. AjaKit rootkit
55. zaRwT rootkit
56. Madalin rootkit

```
#include <stdio.h>
#include <sys/mman.h>
#include <syscall.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

extern int errno;

int main(int argc, char **argv)
{
  char * filename;
  char * file;

  /* usage first argument: output filename,
   * second argument: start address, third
   * argument: end address
   * of data to copy from /dev/kmem
   */

  char * ktext;
  int fp, fp_out;
  long int s_text, e_text;
  ulong size;
  int error = 0;
  file = argv[1];

  s_text = strtoul(argv[2], NULL, 0);
  e_text = strtoul(argv[3], NULL, 0);

  size = e_text - s_text;
  printf("s_text: %x e_text: %x size:
     %x\n", s_text, e_text, size);

  fp = open("/dev/kmem", O_RDWR, 0);
  printf("fp - open /dev/kmem: %d\n",fp);

  ktext = malloc(size);
  printf("ktext - malloc: %d\n", ktext);

  error = lseek(fp, s_text, SEEK_SET);
  printf("error.1 - lseek: %d\n", error);
  perror("lseek");

  error = read(fp, ktext, size);
  printf("error1 -fread ktext : %d\n", error);

  fp_out = creat(file, O_RDWR);
  printf("fp_out - fopen output: %d\n", fp_out);

  error = write(fp_out, ktext, size);
  printf("error - fwrite ktext: %d\n", error);

  free(ktext);
  close(fp_out);
  close(fp);
}
```

Figure 2. The ktext program. Ktext copies the system call code, then writes the executable object code to a file for future reference.

program had again compromised the system. Kern_check indicated that the new knark system calls were located at different addresses within the kernel memory as expected.

The new instances of the eight modified system calls were the same size as those from the previous knark installation. Using these new addresses, we made a copy of the system calls to compare against our previously archived version of compromised system calls. A comparison indicated that the extracted files were identical. This check was only a proof of concept test to determine if we could extract the system call code from kernel memory for comparison. We've achieved similar results for other rootkits that we've analyzed.

To get a quick overview of the archived files, we use the binary visual (bvi) editor tool (http://bvi.sourceforge.net). As Figure 3 shows, bvi outputs:

- the addresses of the data relative to the file's beginning (far left),
- the actual data in hexadecimal notation (center), and
- the data in ASCII format (far right).

We can search within the file hexadecimal notation for each system call's start and end by looking for the individual opcodes for pushing and popping the registers (each system call is a separate C code routine that pushes and pops values onto the stack). We can also identify each system call routine's end by locating the one-byte return opcode (`ret - C3` in the Intel x86 architecture). These search methods are not robust enough for all types of attacks, but can give a quick overview in many cases. Given the complex nature of x86 opcodes, we suggest using a modified `objdump` or more advanced disassemblers for complete and accurate analysis.

Figure 3 shows the bvi output for the `knark_getdents` system call, which replaced the original `sys_getdents` system call. The kernel uses this system call to output a directory's contents. By compromising

The kern_check utility

Samhain Labs has developed kern_check,[1] a small command-line utility that detects kernel-level rootkit presence by comparing a system call table's current `sys_call` addresses with the original kernel symbols map generated when the Linux kernel is compiled. Any difference between the two tables indicates a system call table modification. However, recompiling the kernel with different options or recompiling a new kernel version will most likely result in a new kernel symbols map. It is important to ensure that the `System.map` file used by kern_check is accurate and up to date.

Figure A shows the output of the kern_check program running on a system infected with the knark kernel-level rootkit. As the output indicates, the addresses of eight system calls in the system's current call table stored in kernel memory (`/dev/kmem`) don't match the calls' addresses in the original kernel symbols map (available in `/boot/System.map` in Red Hat Linux-based systems). As the figure shows, the system call table has most likely been modified by a kernel-level rootkit.

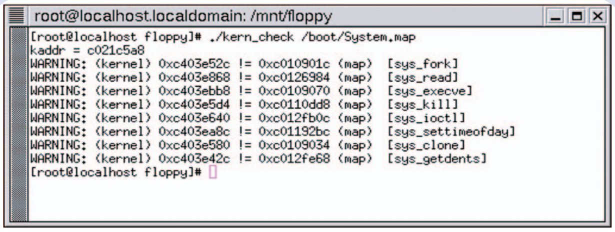


Figure A. Output of the kern_check program running on a knark-infected system.

When we began our research, the kern_check program couldn't detect kernel-level rootkits that redirected the system call table. The reason was that kern_check used the `query_module` command to retrieve the kernel's system call table address, but Linux 2.6 Kernel doesn't export the system call table address. We therefore modified kern_check (which is released under the GPL license) to work even if the `query_module` capability is disabled so it could detect kernel-level rootkits that redirect the system call table. (We sent Samhain Labs our proposed modifications to kern_check and the company subsequently released a new version that can detect kernel-level rootkits that redirect the system call table. The new `kern_check` program incorporates many of the methods we identified through our rootkit examination method.)

Reference

1. Samhain Labs, *Detecting Kernel Rootkits*, July 2003, http://la-samha.de/library/rootkits/detect.html.

this system call, kernel-level rootkits can hide files and directories on the target system.

Applying the method without LKM objects

Our analysis is greatly simplified if the LKM object used to install the kernel-level rootkit is still available on the target system. This LKM object can still exist as an object file (.ko or .o extension). If this file is available, we can disassemble it using a program such as the GNU Debugger (gdb) or `objdump`, which are available with most Linux and Unix distributions. The gdb tool can disassemble each system call using the `disass <sys_call name>` command, which shows the instruction sequence for the functions that map to the bvi program output.

For example, in the bvi screen, there are 256 bytes of output displayed as hexadecimal opcode. Bytes 251–253 are 83 C4 08, which is the opcode for an `add` instruction. This matches the last instruction displayed in Figure 4 before the return, which is the gdb program's output. The third to last symbol displayed by the bvi output (byte 254) is the C3 opcode, which is the return (`ret`) command. Each system call should have this command at the end of its executable path. Even if the LKM opcode is unavailable, we can find each system call's end by locating the final C3 opcode. A system call can contain more than one return statement. If the rootkit's LKM object file is available, it's possible to do a side-by-side comparison of the bvi and gdb output to analyze the system call. In any case, each nonpolymorphic rootkit should have a consistent implementation of its replacement system calls, which can be used to classify that particular rootkit. Our research thus far validates this.

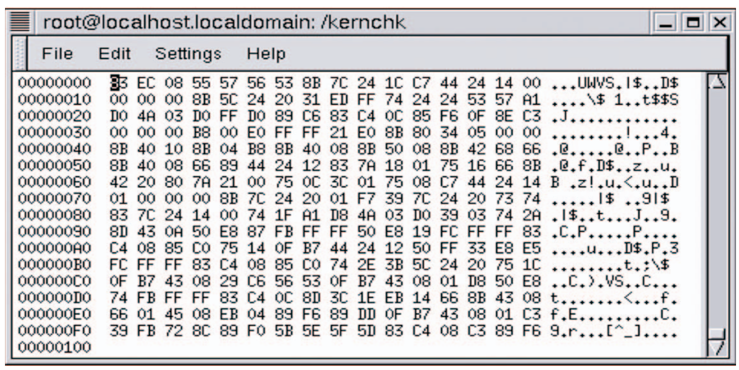
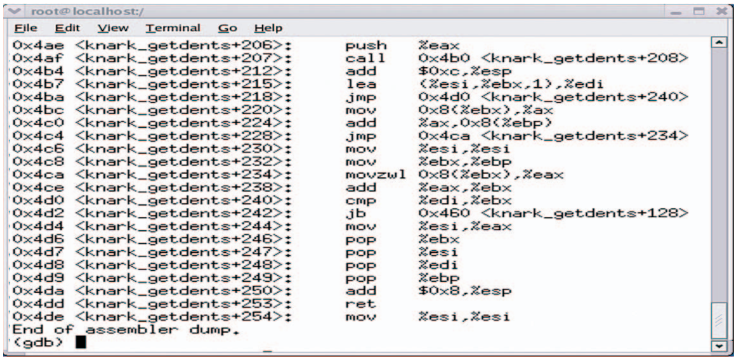


Figure 3. A binary visual editor analysis of `getdents` system call, which outputs directory contents. In this case, the original `sys_getdents` system call was replaced by the `knark_getdents` system call.



```
root@localhost:/
File  Edit  View  Terminal  Go  Help
0x4ae <knark_getdents+206>:     push    %eax
0x4af <knark_getdents+207>:     call    0x4b0 <knark_getdents+208>
0x4b4 <knark_getdents+212>:     add     $0xc,%esp
0x4b7 <knark_getdents+215>:     lea     (%esi,%ebx,1),%edi
0x4ba <knark_getdents+218>:     jmp     0x4d0 <knark_getdents+240>
0x4bc <knark_getdents+220>:     mov     0x8(%ebx),%ax
0x4c0 <knark_getdents+224>:     add     %ax,0x8(%ebp)
0x4c4 <knark_getdents+228>:     jmp     0x4ca <knark_getdents+234>
0x4c6 <knark_getdents+230>:     mov     %esi,%esi
0x4c8 <knark_getdents+232>:     mov     %ebx,%ebp
0x4ca <knark_getdents+234>:     movzwl  0x8(%ebx),%eax
0x4ce <knark_getdents+238>:     add     %eax,%ebx
0x4d0 <knark_getdents+240>:     cmp     %edi,%ebx
0x4d2 <knark_getdents+242>:     jb      0x460 <knark_getdents+128>
0x4d4 <knark_getdents+244>:     mov     %esi,%eax
0x4d6 <knark_getdents+246>:     pop     %ebx
0x4d7 <knark_getdents+247>:     pop     %esi
0x4d8 <knark_getdents+248>:     pop     %edi
0x4d9 <knark_getdents+249>:     pop     %ebp
0x4da <knark_getdents+250>:     add     $0x8,%esp
0x4dd <knark_getdents+253>:     ret
0x4de <knark_getdents+254>:     mov     %esi,%esi
End of assembler dump.
(gdb)
```

Figure 4. The gdb output of *`getdents`* system call. The 256 bytes of output are displayed in disassembled form; bytes 251-253 are 83 C4 08 (fifth row from bottom of terminal),which is disassembled as an `add` instruction. The last `ret` instruction, C3 (fourth row from bottom), indicates the end of the disassembled function. An instruction-by-instruction analysis can show what the function does.

Rootkits that redirect the system call table

To show how our method detects kernel-level rootkits that redirect the system call table, we'll use an example of its application against the SucKIT kernel-level rootkit.

Some techniques, including Samhain's kern_check program, check the system call table in kernel memory against the `/boot/System.map` file to detect kernel-level rootkits. However, the original kern_check program failed to detect rootkits of the SucKIT variety as well as any type of rootkits on more recent Linux kernel versions.

In examining the SucKIT rootkit, we found the first Δ in functionality between SucKIT and the program it replaced: SucKIT overwrote a kernel memory location containing the system call table's address. It accomplished this by querying a specific register within the processor and used the resulting information to find the system call table's reference address within the kernel. SucKIT then overwrote this address with that of a new system call table containing its own malicious system call addresses.

So, our Δ consists of a redirected system call table address, a new system call table, and some new malicious system calls. Given this, we can use SucKIT's own method to query the processor to retrieve the system call table's address and then see whether a rootkit has changed this address. The original address—stored in the `System.map` file in earlier kernels—is available when the kernel is first compiled. If the addresses differ, we can make a more detailed check of the kernel memory's current system call table. In doing so, we create a Δ between the system call addresses in the kernel memory's system call table and the system calls addresses in the `System.map` file.

If the `System.map` file is current, then differences between it and the kernel memory's system call table can indicate that system call redirection is occurring and that a rootkit has infected the system. We can establish a preliminary signature based on the number of system calls that are being redirected. If kernel-level rootkits change a different number of system calls, we can assume we have at least two different kernel-level rootkits. If two rootkits change the same system calls, we can conduct a more detailed analysis of each infected system to see if they are in fact unique.

If we don't have the rootkit source code, we can still look for differences using the kernel debugger (kdb) program or using `/dev/kmem` to copy segments of kernel memory and examine the data offline. With kdb, we can examine the malicious calls' actual machine code because we'll have their start addresses within kernel memory. We can also try to disassemble these malicious system calls, either manually or through the kdb program that can be installed on a forensics system.

In any case, we can now detect system call table redirection on the target system. Although an attacker could develop a kernel-level rootkit that provides false information about the system call table's entry point, we're unaware of any current kernel-level rootkit that can do this. Again, in this case, a more robust trusted computing base is needed.

Application results

As we describe in the sidebar "The kern_check utility," we modified kern_check to better detect kernel-level rootkits. Figure 5 shows the results of running this modified kern_check program on a system infected with the SucKIT rootkit.

Our results are exactly what we'd expect given our analysis of the SucKIT source code. SucKIT created 25 new malicious system calls that subverted the original system calls. SucKIT also redirected the system call table reference to a new system call table, which it created in kernel memory (the modified kern_check program output's first line is this new system call table's address: kaddr = 0xcc1e8000). This address differs from the system call table's address stored in the `System.map` file, which is the address of the original system call table on the target system. We retrieved this address using the `grep` command to search the `System.map` file (Figure 5, last two lines). Typically, this file should be stored offline. If we run the modified kern_check program against this address, we'd detect no system call redirection. However, because we run the modified kern_check with the address retrieved when we query the processor, we can detect system call redirections.

Applying the method without source code

Even without the SucKIT source code, we can still use this methodology to detect a kernel-level rootkit targeting system calls. If the address retrieved from the modified kern_check program matches the address from the `System.map` file, but specific system call addresses differ, then we know that a kernel-level rootkit that modifies the system call table is likely installed on the system. If the address retrieved by the modified kern_check program does not match the `System.map` address, then a kernel-level rootkit that redirects the system call table is likely installed on the target system.

The `System.map` file is created when a Linux kernel is compiled. It should remain consistent for all installations of that kernel. If this file is unavailable, the system will still work, but debugging will be difficult. It should be possible to retrieve a copy of the `System.map` file for a standard Linux installation on a particular architecture. For custom installations (such as those with kernel patches), analysts can copy the `System.map` on any critical system when it's first compiled for future reference. It's important to note that attackers can easily modify the `System.map` file if it exists on the system (at /boot/System.map, for example) so analysts should always use a known good copy.

```
root@localhost:/check/kern_check
File  Edit  View  Terminal  Go  Help
[root@localhost kern_check]# ./kern_check /boot/System.map
kaddr = cc1e8000
WARNING: (kernel) 0xcc1e9308 != 0xc0105ad4 (map)  [sys_fork]
WARNING: (kernel) 0xcc1e96bf != 0xc013148c (map)  [sys_read]
WARNING: (kernel) 0xcc1e9605 != 0xc013158c (map)  [sys_write]
WARNING: (kernel) 0xcc1e99ff != 0xc0130b7c (map)  [sys_open]
WARNING: (kernel) 0xcc1e9a97 != 0xc0130ca0 (map)  [sys_close]
WARNING: (kernel) 0xcc1e9ef1 != 0xc0130c2c (map)  [sys_creat]
WARNING: (kernel) 0xcc1e9f49 != 0xc013bdb4 (map)  [sys_unlink]
WARNING: (kernel) 0xcc1e99de != 0xc0105b24 (map)  [sys_execve]
WARNING: (kernel) 0xcc1e9bb2 != 0xc0137894 (map)  [sys_stat]
WARNING: (kernel) 0xcc1e9c65 != 0xc0137a44 (map)  [sys_fstat]
WARNING: (kernel) 0xcc1e9b5a != 0xc012ffa8 (map)  [sys_utime]
WARNING: (kernel) 0xcc1e9aec != 0xc011e138 (map)  [sys_kill]
WARNING: (kernel) 0xcc1e9158 != 0xc010b834 (map)  [sys_olduname]
WARNING: (kernel) 0xcc1e9c0a != 0xc013796c (map)  [sys_lstat]
WARNING: (kernel) 0xcc1e9fa0 != 0xc0137b14 (map)  [sys_readlink]
WARNING: (kernel) 0xcc1e9cbd != 0xc0137900 (map)  [sys_newstat]
WARNING: (kernel) 0xcc1e9d1b != 0xc01379d8 (map)  [sys_newlstat]
WARNING: (kernel) 0xcc1e9d79 != 0xc0137aac (map)  [sys_newfstat]
WARNING: (kernel) 0xcc1e9299 != 0xc0105aec (map)  [sys_clone]
WARNING: (kernel) 0xcc1e93e3 != 0xc013e5b0 (map)  [sys_getdents]
WARNING: (kernel) 0xcc1e9374 != 0xc0105b0c (map)  [sys_vfork]
WARNING: (kernel) 0xcc1e9dd7 != 0xc0137c90 (map)  [sys_stat64]
WARNING: (kernel) 0xcc1e9e35 != 0xc0137cfc (map)  [sys_lstat64]
WARNING: (kernel) 0xcc1e9e93 != 0xc0137d68 (map)  [sys_fstat64]
WARNING: (kernel) 0xcc1e94f4 != 0xc013e75c (map)  [sys_getdents64]
[root@localhost kern_check]# grep "D sys_call_table" /boot/System.map
c02d1890 D sys_call_table
```

Figure 5. Modified kern_check results on a SucKIT-infected system. The system call table has been redirected, as a comparison of the `kaddr` value and the `grep D sys_call_table` output indicate. In all, 25 system calls were redirected.

Other kernel-level rootkits

Although we've focused this work on kernel-level rootkits that target the system call table, there are many other targets in the Linux kernel that attackers might find of interest. For example, as we describe below, the adore-ng rootkit targets the virtual file system data structures. More advanced rootkits can target core kernel data structures such as the page tables.

Targeting the virtual file system

Released in January 2004, the adore-ng kernel-level rootkit targets the virtual file system rather than the system call table. In targeting the VFS, adore-ng can compromise the kernel and hide the attacker's presence. The VFS is a software layer in the Linux kernel that handles all system calls related to the standard Unix file system. VFS can handle several different types of file systems.[10] The adore-ng kernel-level rootkit replaces existing handler routines—which provide directory listings to the `/proc` and `/file` systems—with its own routines. This lets the attacker hide specified files and processes from user mode programs.[6]

Adore-ng redirects reference to the `proc_root_lookup` function call to a malicious lookup function call that it creates. This redirection occurs outside of the kernel code's static text section in the kernel's dynamic data section. We examined the source code of adore-ng to determine how the system was compromised. This showed us where the `proc_root_lookup` redirection was occurring in kernel space and also gave us the address of the malicious replacement lookup function for follow-on analysis. Given a copy of a kernel-level rootkit, we can analyze its infection vector to categorize it and aid in its subsequent detection.

Anticipating new kernel-level targets

Currently, kernel-level rootkits target the system call table, virtual file system structures, page tables, and a handful of other structures. More generally, anything in the kernel can be a target. Our methodology can be applied to the entire kernel, but it is important to emphasize that a more robust access to kernel memory is needed for inspection tools. Future kernel-level rootkits might target subsystems such as the scheduler, network stack, hardware drivers, and so on. Administrators can create a copy and cryptographic checksums of these subsystems to ensure the integrity of their kernels remains intact.

Other rootkit types

In addition to Linux kernel-level rootkits, there are many other types of rootkits and combinations of techniques. Some of the early rootkits developed were Unix user-level rootkits. These days, many worms, viruses, and bots are beginning to include rootkits in their payloads to hide themselves.

"Blended" rootkits

We applied our methodology against a new type of rootkit retrieved from a compromised Linux system. The new rootkit, named the *zk rootkit*, is a modification of SucKIT. Our analysis revealed it to be a "blended" rootkit containing elements of both user-level and kernel-level rootkits.

Applying our method let us identify specific Δ characteristics that analysts can use to detect and categorize this

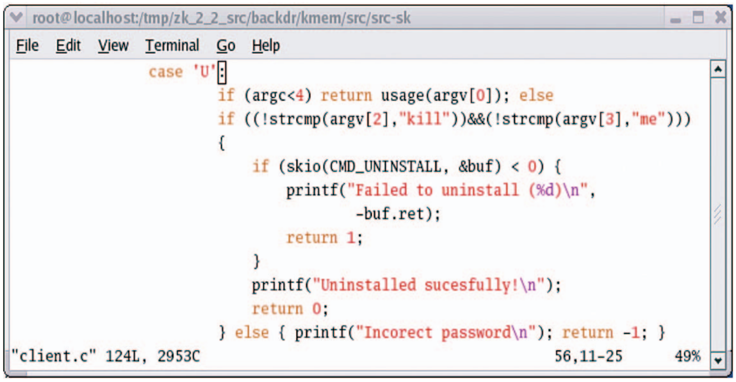


Figure 6. Locating the uninstall password for zk rootkit.

rootkit. Our method also let us identify the uninstall password for the zk rootkit. The usage statement indicated that a password was required to uninstall the zk rootkit. The rootkit documentation, however, contained no reference to this uninstall password, nor was there any indication of how to set the password. We used the zk usage statement to try and identify a Δ.

First, we conducted a `grep` search for the term "password" within the zk rootkit's source code directory. The results indicated that the term "password" appeared in the zk rootkit's `client.c` source code file. The SucKIT rootkit had a file with the same name. Comparing the files using the resident `diff` command indicated that the two files in fact differed. Then, as Figure 6 shows, we ran a more complete search on the zk `client.c` file and identifed a password ("kill me").

Given this, we successfully uninstalled the zk rootkit using the command `#./zk u kill me`. Finally, we ran the modified kern_check program on the system, which indicated that the system was no longer infected.[11]

Microsoft rootkits

Although our research has been primarily focused on Linux-based systems, our methodology produced information that let us detect and classify rootkits on other systems as well. For example, we analyzed a compromised Microsoft 2000 honeypot to locate a specific Δ to identify a possible rootkit on the system. The Δ characteristics we found were three new directories; we also identified registry changes. Given this Δ, we were able to categorize the Microsoft rootkit.[11] The rootkit can be classified as a user-level rootkit because the new directories created were hidden without modifying the kernel.

Applying our methodology generates rootkit signatures that can help both system administrators and the security community at large locate known kernel-level rootkits and react faster to new types of attacks. We believe that understanding the extent of a rootkit installation can lead to a sound method of uninstalling rootkits. One of the biggest challenges is establishing a better trusted computing base that is more resistant to rootkit attacks. ▢

Acknowledgments

We thank the anonymous reviewers for their comments. We also thank Thorsten Holz for his comments and suggestions.

References

1. D. Dittrich, "Root Kits" and "Hiding Files/Directories/Processes after a Break-In," http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq.
2. E. Cole, *Hackers Beware*, New Riders, 2002, pp. 548–553.
3. H. Thimbleby, S. Anderson, and P. Cairns, "A Framework for Modeling Trojans and Computer Virus Infections," *The Computer J*, vol. 41, no. 7, 1998, pp. 444–458.
4. E. Skoudis, *Counter Hack*, Prentice Hall, 2002, p. 434.
5. A. Silberschatz, P. Galvin, and G. Gagne, *Applied Operating System Concepts*, John Wiley & Sons, 2003, p. 626.
6. Samhain Labs, *The Basics–Subverting the Kernel*, Mar. 2004; http://la-samha.de/library/rootkits /basics.html.
7. Samhain Labs, *Detecting Kernel Rootkits*, July 2003, http://la-samha.de/library/rootkits/detect.html.
8. F. Cohen, "Computer Viruses," *Computers & Security,* vol. 6, no. 1, 1987, pp. 22–35.
9. V. Bontchev, "Analysis and Maintenance of a Clean Virus Library," VX Heavens, Aug. 2003; http://vx.netlux.org/lib/static/vdat/epvirlib.htm.
10. D. Bovet and M. Cesati, *Understanding the Linux Kernel*, O'Reilly & Associates, 2003, p. 372.
11. J. Levine, *A Methodology for Detecting and Classifying Rootkit Exploits*, doctoral dissertation, School of Electrical and Computer Engineering, Georgia Institute of Technology, 2004.

***John G. Levine** is an active duty army officer currently serving as an assistant professor in the Department of Electrical Engineering and Computer Science at the United States Military Academy at West Point, New York. His research interests include information assurance and secure computing architectures. He has a PhD in electrical and computer engineering from the Georgia Institute of Technology. He is a member of the IEEE. Contact him at John.Levine@usma.edu.*

***Julian B. Grizzard** is a PhD candidate in the School of Electrical and Computer Engineering at the Georgia Institute of Technology. His research interests include operating systems, networking, and security. Grizzard has an MS in electrical and computer engineering from Georgia Institute of Technology in 2004. He is a member of the IEEE. Contact him at grizzard@ece.gatech.edu.*

***Henry L. Owen** is a professor in the School of Electrical and Computer Engineering. His research interests include Internet security and internetworking. Owen has a PhD in electrical engineering from the Georgia Institute of Technology. Contact him at henry.owen@ece.gatech.edu.*