A Methodology to Characterize Kernel Level Rootkit Exploits that Overwrite the System Call Table

John G. Levine, Julian B. Grizzard, Phillip W. Hutto* , Henry L. Owen
School of Electrical and Computer Engineering
* College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0250

Abstract

A cracker who gains access to a computer system will normally install some method, for use at a later time that allows the cracker to come back onto the system with root privilege. One method that a cracker may use is the installation of a rootkit on the compromised system. A kernel level rootkit will modify the underlying kernel of the installed operating system. The kernel controls everything that happens on a computer. We are developing a standardized methodology to characterize rootkits. The ability to characterize rootkits will provide system administrators, researchers, and security personnel with the information necessary in order to take the best possible recovery actions. This may also help to detect and fingerprint additional instances and prevent further security instances involving rootkits. We propose new methods for characterizing kernel level rootkits. These methods may also be used in the detection of kernel rootkits.

Index Terms— **Computer crime, cracking, hacking, information assurance, rootkits, system compromise, trojan.**

1. INTRODUCTION

1.1 Definition of a Rootkit

A rootkit can be considered as a "Trojan Horse" introduced into a computer operating system. According to Thimbleby, Anderson, and Cairns, there are four categories of trojans. They are: *direct masquerades*, i.e. pretending to be normal programs; *simple masquerades*, i.e. not masquerading as existing programs but masquerading as possible programs that are other than what they really are; *slip masquerades*, i.e. programs with names approximating existing names; and *environmental masquerades*, i.e. already running programs not easily identified by the user [1]. We are primarily interested in masquerades as well as environmental masquerades. A kernel level rootkit may have characteristics of direct masquerades in that it will consist of malicious system calls pretending to be normal system calls. It may also have characteristics of simple masquerades in that it can masquerade as system calls other than what they really are. Kernel level rootkits may be considered environmental masquerades in that they are already running and cannot be easily identified by computer users.

A cracker (we refer to anyone who attempts to compromise a computer system as a cracker as opposed to a hacker) must already have root level access on a computer system before they can install a rootkit. Rootkits do not allow a cracker to gain access to a system. Instead, they enable the cracker to get back into the system with root level permissions [2]. Once a cracker has gained root level access on a system, a trojan program that can masquerade as an existing system function or capability can then be installed on the compromised computer system.

Rootkits are a phenomenon that has recently drawn attention. Prior to rootkits, system utilities could be trusted to provide a system administrator with accurate information. Modern crackers have developed methods to conceal their activities and programs to assist in this concealment [3]. Rootkits are a serious threat to the security of a networked computer system.

The vulnerabilities that exist in modern operating systems as well the proliferation of exploits that allow crackers to gain root access on networked computer systems provide crackers with the ability to install rootkits. System administrators need to be aware of the threats that their computers face from rootkits as well as the ability to recognize if a particular rootkit has been installed on their computer system.

1.2 Kernel Level Rootkits

Kernel level rootkits are one of the most recent developments in the area of computer system exploitation by the cracker community [4]. The kernel is recognized as the most fundamental part of most modern operating systems. The kernel can be considered the lowest level in the operating system. The file system, scheduling of the CPU, management of memory, and system call related operating system functions are all provided by the kernel [5]. Unlike a traditional binary rootkit that modifies critical system level programs, a kernel level rootkit will replace or modify the kernel itself. This allows the cracker to control the system without others being aware of this. Kernel level rootkits usually cannot be detected by traditional means available to a system administrator.

The kernel controls any application that is running on the computer. If the application wants to access some system resource, such as reading to or writing from the disk, then the application must request this service from the kernel. This is accomplished through the use of a system call, or sys_call. The application performs a sys_call passing control to the kernel which performs the requested work and provides the output to the requesting application. A kernel level rootkit modifies these system calls to perform some type of malicious activity. A kernel level rootkit can use the capability of loadable kernel modules (LKMs). LKMs are a feature that is available in Linux. A kernel rootkit can use a custom kernel module to modify a sys_call to hide files and processes as well as provide backdoors for a cracker to return to the system. These types of rootkits usually modify the sys_call_table. They replace the addresses of the legitimate sys_calls with the addresses of the sys_calls that are to be installed by the cracker's LKM. The requested sys_call can then be redirected away from the legitimate sys_call to the kernel level rootkit's replacement sys_call. Loadable Kernel Modules are available in various UNIX based operating systems. Figure 1 show the results of a kernel rootkit like knark on the system call table within kernel memory.

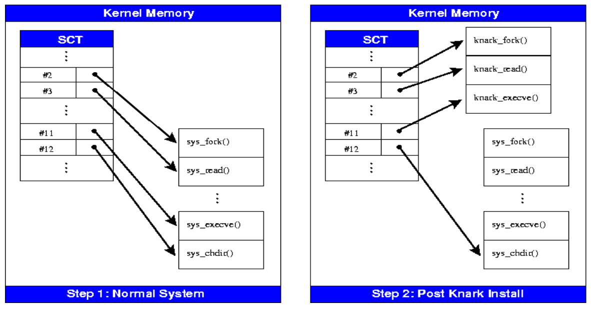


Figure 1 - Redirected System Call Table

2 EXISTING METHODOLOGIES TO DETECT ROOTKITS

Current tools and methodologies exist to detect systems compromised by rootkits. These methodologies include both manual and automated measures. Examples include comparison of cryptographic checksums against known good system installations, checking of various system directories, and examination of files for suspicious signatures. We will briefly examine some of these methodologies and identify their shortcomings with respect to detecting kernel level rootkits. These methods are limited in that they only detect the presence of a rootkit in most cases without characterizing the specific rootkit that is installed on the target system. In other words, these methods can tell you that you are infected with a rootkit without identifying the specific rootkit. We believe that it will benefit network security personnel to know the specific rootkit that they are dealing with on the infected system.

2.1 Tools Available to Detect Rootkit Exploits

There are tools available for System administrators to detect if a system has been compromised. The two primary means of detecting a compromised system are to conduct signature analysis on the system or to compare cryptographic checksums of system files with known good cryptographic checksums. While both of these methods are able to detect exploitation by traditional or binary level rootkits they may not work in the detection of kernel level rootkits.

There is a free program that checks a system for rootkits. This program is known as chkrootkit [6]. This program runs a shell script that checks specific system binaries to determine if a rootkit has been installed on the system. This program also checks to see if the network interfaces on the computer have been set to the promiscuous mode, which is a common ploy used by crackers in order to capture network traffic. In addition, chkrootkit looks for the creation of certain directories as a result of a system being infected with a rootkit exploit. For example, The knark kernel rootkit creates a directory called knark in the /proc/ directory. A system infected with the knark kernel level rootkit can be detected by chkrootkit because of the presence of this directory. We speculate that the developer of knark specifically chose the /proc/ directory for the sub-directory that is to be created. The /proc/ directory is one that is constantly changing during the operation of the computer. Any new process that is started will have a separate directory created here. Because of this, the /proc/ directory is normally not chosen as a directory that will be checked with a cryptographic signature. This hidden directory is created by knark and the chkrootkit program looks for this specific directory. chkrootkit also checks the system logs. chkrootkit is signature based. Therefore the signature must be known in order to detect if a rootkit has been installed on a system. The code in the chkrootkit script file that is used to detect an infection by knark is shown in Figure 2.

```
## knark LKM
if [ -d /proc/knark]; then
 echo "Warning:Knark LKM installed"
fi
```

Figure 2 - chkrootkit code to detect knark lkm

We find it unusual that the knark rootkit does not use its directory hiding capability to hide this directory. The chkrootkit check can be defeated if this directory is renamed to something besides /proc/knark.

Programs such as chkrootkit may not detect new rootkits or modifications to existing rootkits.

2.2 Running a cryptographic checksum/file integrity checker program

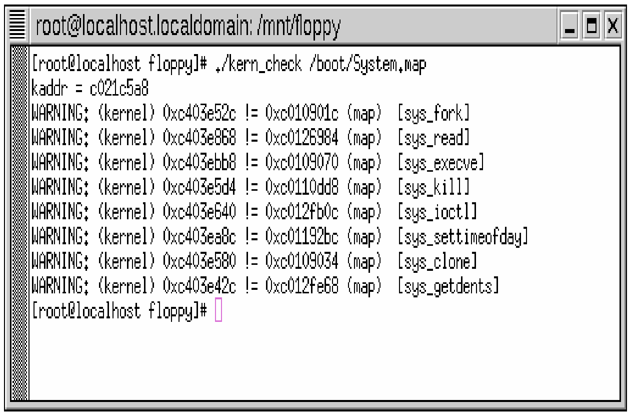
A cryptographic checksum program, otherwise known as a file integrity checker program, can be run on the computer system in question. There are several host based IDS tools that look at changes to the system files. These programs take a snapshot of the trusted file system state and use this snapshot as a basis for future scans. This snapshot is normally based on calculating a cryptographic checksum of the files that are to be analyzed in the future.

These types of programs may not be able to detect Kernel Level Rootkits at the present time. Kernel level rootkits operate at a lower level than binary or application level rootkits. Kernel level rootkits do not need to change any system files on the target computer. Instead, they can modify the kernel code in system memory. As a result, the cryptographic checksums of the system files will not change, even after a system has been infected with a kernel level rootkit. The kernel level rootkit is able to intercept system calls at the kernel level and compromise the operations of the target computer without changing any system files [7]. Because of this occurring at the kernel level, a file integrity checker program will not detect that the system has been infected even after a kernel level rootkit has been installed on the system. It has been proposed by Dino Dai Zovi of Sandia labs that all kernel modules be cryptographically signed to ensure that only trusted code will run at the kernel level [8].

Other researchers are also looking at using cryptographic checksums to maintain kernel integrity. The StMichael Project has developed a kernel module that produces a cryptographic checksum of the kernel and uses this checksum in order to detect if the kernel is compromised [9]. Cryptographically signed kernels and checksums are an area of research that warrants further investigation from a security aspect. However, these methodologies only detect that the kernel may have been compromised without identifying the type or nature of the compromise.

2.3 The kern_check program

Samhain Labs [10] has developed a small command-line utility to detect the presence of a kernel level rootkit. It may be possible to detect the presence of a kernel level rootkit by comparing the sys_call addresses in the current sys_call_table with the original map of kernel symbols that is created by Linux when the system is compiled. A difference between these two tables will indicate that something, most likely a kernel level rootkit, has modified the sys_call_table [11]. It must be noted that each new installation of the kernel as well as the loading of a kernel module will result in a new map of kernel symbols. Figure 3 shows the output of running the kern_check program on a system infected with a kernel level rootkit. This program compares the sys_call with the one stored in a system file (/boot/System.map).



```
[root@localhost floppy]# ./kern_check /boot/System.map
kaddr = c021c5a8
WARNING: (kernel) 0xc403e52c != 0xc010901c (map)  [sys_fork]
WARNING: (kernel) 0xc403e868 != 0xc0126984 (map)  [sys_read]
WARNING: (kernel) 0xc403ebb8 != 0xc0109070 (map)  [sys_execve]
WARNING: (kernel) 0xc403e5d4 != 0xc0110dd8 (map)  [sys_kill]
WARNING: (kernel) 0xc403e640 != 0xc012fb0c (map)  [sys_ioctl]
WARNING: (kernel) 0xc403ea8c != 0xc01192bc (map)  [sys_settimeofday]
WARNING: (kernel) 0xc403e580 != 0xc0109034 (map)  [sys_clone]
WARNING: (kernel) 0xc403e42c != 0xc012fe68 (map)  [sys_getdents]
[root@localhost floppy]#
```

Figure 3 - kern_check output of knarked system

The output indicates that the addresses of eight sys_calls currently listed in the sys_call_table do not match the addresses for those sys_calls in the original map of the kernel symbols. This map of kernel systems is available on the system as /boot/System.map. We have conducted an in-depth analysis of how the knark kernel level rootkit modifies these system calls as part of our research, this analysis is available at [15]. It is theoretically possible for another kernel level rootkit to change the same eight system calls. At present, there is no method of determining which kernel level rootkit has changed these system calls.

3 CHARACTERIZING KERNEL LEVEL ROOTKITS

We have looked at various programs that currently exist to detect rootkits. These programs may indicate that some type of rootkit is installed on the target system but in most cases they fail to indicate the particular rootkit that is installed. We have developed a methodology that will help to characterize a kernel level rootkits that overwrites the System Call Table and present this methodology. This methodology will also work to detect the presence of Kernel Level Rootkits that modify specific system call code without changing the System Call Table.

3.1 Archiving the compromised System Call code from kernel space.

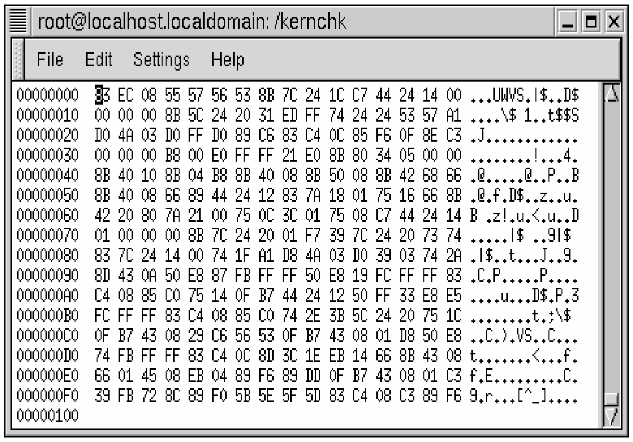
Our methodology depends on the ability to archive a copy of the system call code that currently exists in kernel memory for characterization analysis. It is this archived code that we propose to use to be able to characterize kernel rootkit exploits. We have developed a C program that can copy the system call code that is referenced by a start and end address and write the executable object code to a file for future reference. We feel that this is significant because it allows the analyst to be able to retrieve of the code that is currently running in the system kernel. Further, some types of kernel level rootkits such as knark do not remain resident in memory after the system is rebooted. Our program allows for a copy of any suspicious system calls to be copied offline for follow on analysis prior to rebooting the system.

This program that we have developed, called *ktext*, is listed in the appendix and is available at the website that we have previously referenced [15]. We have tested this program in the following manner. We installed several kernel rootkits on several target systems. We then ran the kern_check program on these systems. With the knark kernel rootkit, the kern_check program informed us that eight system calls were being redirected. The kern_check program also indicated the address within kernel space of these new redirected system calls (see figure 3). We then used our program to make a copy of the system calls that were indicated as having been redirected. Our analysis of the source code used to create this rootkit indicated that the new redirected system calls were being written sequentially into kernel memory. This may not always be the case and it may be necessary to conduct an analysis of the object code to identify start and end address of the individual system calls.

We then rebooted this system and checked the system with kern_check. The kern_check program indicated that no system calls were being redirected on the target system. We then reinstalled the knark kernel rootkit via its loadable kernel module. A subsequent validation by the kern_check program indicates that the system is once again compromised by the knark program. The new knark system calls are now located at different addresses within the kernel memory, as indicated by the output of the kern_check program. The new instances of the eight modified system calls appear to be the same size as the system calls from the previous knark installation. We then used these new addresses to make a subsequent copy of the system calls for comparison against our previously archived version of this compromised system call. A comparison indicated that the files that we extracted are 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 have observed similar results for the other rootkits that we analyzed.

Other tools make copies of kernel code in order to detect if the kernel had been compromised. The *Samhain* tool, developed by Samhain Labs, has the ability to make a copy of the first eight bytes of each system call in order to detect if a jump has been inserted into the start of the original system call code [11]. The *Samhain* tool is a "whitehat" kernel level rootkit that is installed by the system administrator. This tool will only tell you that your kernel has been compromised but it will not identify the compromising rootkit. The previously mentioned StMichael project will also modify the underlying kernel of the system that it is installed on. Our methodology will not modify the underlying kernel and will make a copy of the entire system call for reference and analysis. Our method will provide for a baseline to compare against subsequent rootkits in order to characterize the rootkit.

The archived files can be examined with a tool such as bvi (binary visual editor) which is available on the Internet [12]. The output of bvi is the addresses of the data relative to the beginning of the file (far left), the actual data in hexadecimal notation (center), and the data in ASCII format (far right) as indicated in Figure 4. One can search within the file hexadecimal notation for the start and end of each system call by looking for the individual opcodes for pushing and popping the registers (each system call is a separate C code routine that will push and pop values on to the stack. You can also identify the end of each system call routine by looking for the one byte return opcode (ret – C3 in the Intel x86 architecture [13]).



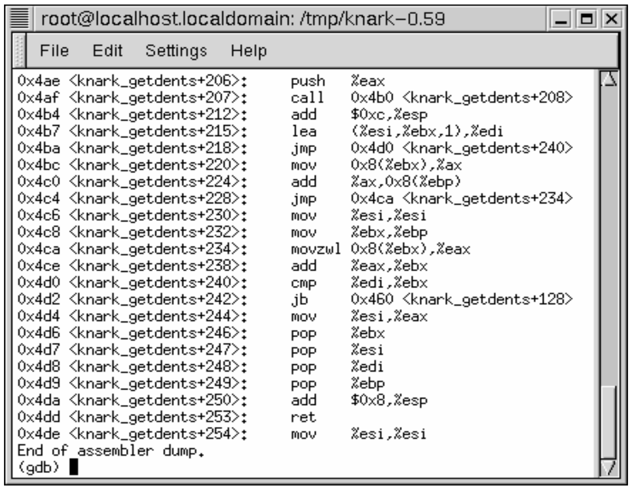
```
00000000  83 EC 08 55 57 56 53 8B 7C 24 1C C7 44 24 14 00 ...UWVS.|$..D$
00000010  00 00 00 8B 5C 24 20 31 ED FF 74 24 24 53 57 A1 ....\$ 1..t$$S
00000020  D0 4A 03 D0 FF D0 89 C6 83 C4 0C 85 F6 0F 8E C3 .J............
00000030  00 00 00 B8 00 E0 FF FF 21 E0 8B 80 34 05 00 00 ........!...4.
00000040  8B 40 10 8B 04 B8 8B 40 08 8B 50 08 8B 42 68 66 .@.....@..P..B
00000050  8B 40 08 66 89 44 24 12 83 7A 18 01 75 16 66 8B .@.f.D$..z..u.
00000060  42 20 80 7A 21 00 75 0C 3C 01 75 08 C7 44 24 14 B .z!.u.<.u..D
00000070  01 00 00 00 8B 7C 24 20 01 F7 39 7C 24 20 73 74 .....|$ ..9|$
00000080  83 7C 24 14 00 74 1F A1 D8 4A 03 D0 39 03 74 2A .|$..t...J..9.
00000090  8D 43 0A 50 E8 87 FB FF FF 50 E8 19 FC FF FF 83 .C.P.....P....
000000A0  C4 08 85 C0 75 14 0F B7 44 24 12 50 FF 33 E8 E5 ....u...D$.P.3
000000B0  FC FF FF 83 C4 08 85 C0 74 2E 3B 5C 24 20 75 1C ........t.;\$
000000C0  0F B7 43 08 29 C6 56 53 0F B7 43 08 01 D8 50 E8 ..C.).VS..C...
000000D0  74 FB FF FF 83 C4 0C 8D 3C 1E EB 14 66 8B 43 08 t.......<...f.
000000E0  66 01 45 08 EB 04 89 F6 89 DD 0F B7 43 08 01 C3 f.E.........C.
000000F0  39 FB 72 8C 89 F0 5B 5E 5F 5D 83 C4 08 C3 89 F6 9.r...[^_]....
00000100
```

Figure 4 - bvi analysis of getdents system call

Figure 4 showed us the bvi output for the knark_getdents system call that replaces the original sys_getdents system call. This system call is used by the kernel to output the contents of a directory. Kernel level rootkits will compromise this system call in order to hide files and directories on the target system.

Analysis can be greatly simplified if the LKM code that is used to install the kernel rootkit is still available on the target system. This LKM code will most likely still exist as an object file (.o extension). It this file is available, it can be loaded into a program such as gdb (The GNU Debugger, available with most Linux and Unix distributions) in order to be disassembled. Each system call can be disassembled using the disass <sys_call name> command. What is significant to note about the output of this command is that there is a mapping to the output of the bvi program.

There are 256 bytes of output displayed as hexadecimal opcode in the bvi screen. The second to last byte is 89, which is the opcode for the move instruction (MOV) [13]. This matches up with the last instruction displayed in Figure 5, which is the output from the gdb program. The third to last symbol that is displayed by the bvi output is the C3 opcode. The C3 opcode is the return (RET) command [13]. Each symbol call should have this command near the end of its associated opcode. Even if the LKM opcode is not available, the approximate end of each system call can be found by locating the C3 opcode. A system call should only contain one return statement. If the LKM for the rootkit is available, then it is possible to do a side by side comparison of the bvi output to the gdb output to analyze the system call. In any case, we believe that each particular rootkit should have a consistent implementation of its replacement system calls which can be used to classify that particular rootkit. Our research thus far has proved this to be true and we will continue to research this area.



```
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 5 - gdb output of getdents system call

To summarize the steps to our methodology:

1. Identify system calls that have changed.
2. Determine the size to copy of each system call
3. Conduct a byte by byte analysis to disassembly this copied code
4. Archive actual compromised system call code/checksum for future comparison

3.2 Using Archived System Calls to detect a new class of system compromise

At present, kernel level rootkits compromise a system in one of two methods. The first method is to overwrite addresses in the system call table with the address of the malicious system call. The second method is to create an entirely new system call table within kernel space and redirect all system calls to this new table which contains the malicious system call addresses. Currently, there are no kernel level rootkits that attempt to compromise a system by overwriting the system call instructions with malicious code [14]. This would be very difficult to accomplish do to the fact that race conditions might occur while trying to overwrite the system calls. However it may be possible.

We propose that an archived copy of each system call can be produced based on the methodology that we have previously presented. The addresses of the system calls are already available within the /boot/System.map file (available on the Linux Operating System). A cryptographic checksum of these system calls can then be produced for future comparison against the system to ensure that the system calls have not been compromised. These archived system calls could also be used as baseline for further investigation.

4 CONCLUSION

We have presented a methodology to classify kernel level rootkits exploits that overwrite the system call table within this paper. Two rootkits that have the same implementation of a compromised system call are the same rootkits. A rootkit that has elements of some previously characterized rootkit is a modification and a rootkit that has entirely new characteristics is new.

We demonstrated the application of this methodology against a particular class of kernel rootkit exploits. This can help to generate rootkit signatures to aid in the detection of these types of exploits. This methodology will allow system administrators and the security community to better understand kernel level rootkits in order to plan and react accordingly.

The adore rootkit is an example of this particular class of kernel level rootkit. In 2001 there were 1798 reported incidents of the adore rootkit in the United States [16]. A new version of this rootkit was recently released [17].

APPENDIX

The ktext program:

```
/* compile & usage statement */

/* GPL License */

#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 sj_s_text, sj_e_text;
ulong size;
int error = 0;
file = argv[1];

sj_s_text = strtoul(argv[2], NULL, 0);
sj_e_text = strtoul(argv[3], NULL, 0);

size = sj_e_text - sj_s_text;
printf("sj_s_text: %x sj_e_text: %x
size: %x\n", sj_s_text, sj_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, sj_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);

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

REFERENCES

[1] H. Thimbleby, S. Anderson, P. Cairns, "A Framework for Modeling Trojans and Computer Virus Infections," *The Computer Journal,* vol. 41, no.7 pp. 444-458, 1998.

[2] E. Cole, *Hackers Beware,* Indianapolis, In: New Riders, 2002, pp. 548-553.

[3] D. Dettrich, (2002, 5 JAN) "*Root Kits" and hiding files/directories/processes after a break-in,* [Online]. Available: http://staff. washington. edu/dittrich/misc/faqs/rootkits.faq

[4] E. Skoudis, *Counter Hack,* Upper Saddle River, NJ: Prentice Hall PTR: 2002.

[5] A. Silberschatz, P. Galvin, G. Gagne, *Applied Operating System Concepts,* New York, NY: John Wiley & Sons: 2003, p. 626.

[6] N. Murilo, K. Steding-Jones, “chkrootkit V. 0.36” www.chkrootkit .org.

[7] E. Cole, *Hackers Beware, p. 550*

[8] http://www.sans.org/rr/papers/index.php?id=449, NOV 2003

[9] http://sourceforge.net/projects/stjude, AUG 2003

[10] Samhain Labs, *Loadable Kernel Module Rootkits,* http://la-samha.de/library/lkm.html, July 2002.

[11] Samhain Labs http://la-samha.de/samhain, NOV 2003.

[12] http://bvi.sourceforge.net/, NOV 2002.

[13] http://www.intel.com/design/pentium4/manuals/245471.htm, NOV 2003.

[14] http://la-samhna.de/library/rootkits/basics.html NOV 2003.

[15] http://users.ece.gatech.edu/~owen/Researh /Rootkit/Rootkit.htm

[16] http://archives.neohapsis.com/archives/incidents/2001-04/0056.html, JAN 2004.

[17] http://archives.neohapsis.com/archives/incidents/2001-04/0056.html, JAN 2004.