Linux Rootkit Detection With OSSEC

GIAC (GCIA) Gold Certification

Author: Sally Vandeven, sallyvdv@gmail.com
Advisor: Dominicus Adriyanto Hindarto

Accepted:
March 26, 2014

Abstract

Rootkits are one the most insidious forms of malware because they are designed to hide their existence on a system making them very difficult to detect. Yet there are utilities that claim to be effective at rootkit detection. OSSEC is one such utility. It is an open source host based IDS/IPS that also includes rootkit detection for Linux systems. This paper will examine and measure OSSEC's ability to detect and identify several different Linux rootkits including both user mode and kernel mode variants.

1. Introduction

Most malware consists of a malicious application that gets installed on a victim's computer. That application must somehow get executed, often by tricking the user into clicking on something that will cause the application to launch, and then the software can carry out its evil deed. A rootkit is a special type of malware that actually replaces or makes changes to existing operating system components in order to alter the behavior of the system, usually with the intent of hiding itself and other processes, files, network connections or other operations that might expose the attacker's activities (Skoudis & Zeltser, 2003, p. 303).

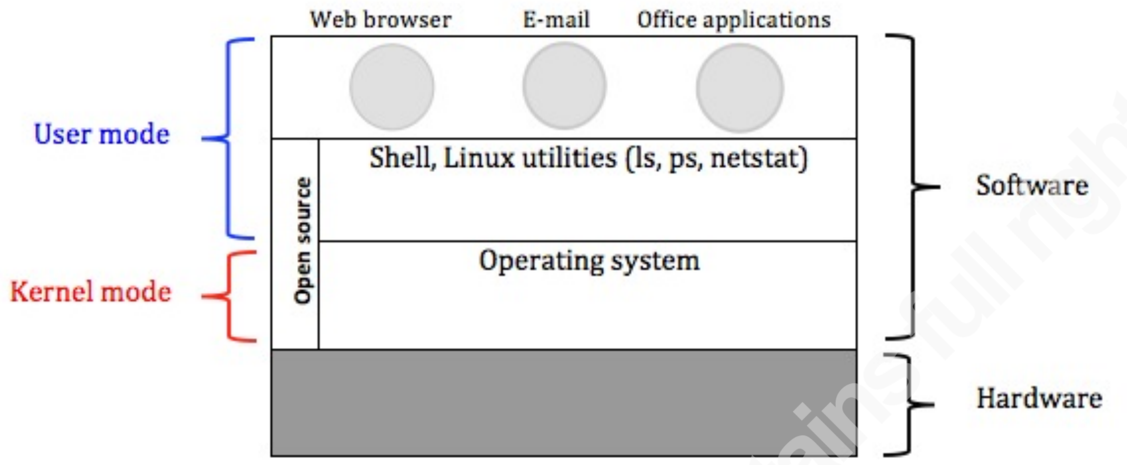
OSSEC is a free, open source host based intrusion detection system that attempts to detect the presence of such rootkits on Linux systems (http://www.ossec.net/doc) using a combination of file integrity checking, signature based detection and anomaly detection (Cid, 2008, p.161).

This paper will discuss in detail the methods OSSEC employs for rootkit detection and then summarize the results of live tests of several types of rootkits and OSSEC's success at detecting them.

2. What is a Rootkit?

Rootkits are generally grouped into one of two categories: user mode rootkits and kernel mode rootkits. The first type, user mode, modifies or replaces operating system components or system binaries that are used directly by the user and run in what is called "user mode" and the second modifies the kernel itself. Rootkits have taken on these names most likely because they accurately reflect the level at which they run on the computer.

Figure 1

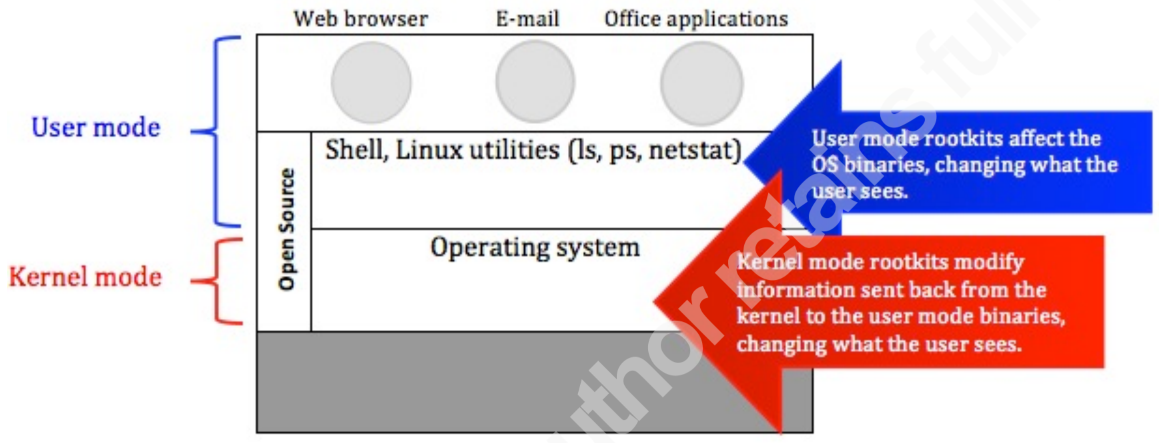


Kernel mode vs user mode in Linux

Andrew Tanenbaum describes kernel mode and user mode in his book *Modern Operating Systems* (Tanenbaum, 2008, p.2). He writes that the operating system runs in kernel mode and has full access to all hardware and resources in the computer. All other software runs in user mode and must get access to hardware resources via the kernel or operating system. Figure 1 shows Tanenbaum's depiction of the relationship between user mode and kernel mode software. If a user would like information about files, for example, the user would issue the *ls* command to show a listing of files. *Ls* is a Linux utility that runs in user mode. The *ls* command will query the kernel or operating system because it controls the access to the file system. The kernel will pass back the requested information to the *ls* command, which in turn gets passed back to the user.

Figure 2 shows where the two types of rootkits operate in this layered model of the computer's organization. User mode rootkits run at the same level as other user mode software often by replacing Linux utilities with modified, malicious versions. Kernel mode rootkits operate in the protected and privileged space of the operating system.

Figure 2



Web browser
E-mail
Office applications
User mode
Kernel mode
Open Source
Shell, Linux utilities (ls, ps, netstat)
Operating system
User mode rootkits affect the OS binaries, changing what the user sees.
Kernel mode rootkits modify information sent back from the kernel to the user mode binaries, changing what the user sees.

Linux user mode and kernel mode rootkits

Installing a rootkit on a system requires root level access for either type of rootkit. How an attacker gains root level access to a system is beyond the scope of this paper but may be accomplished, for example, by stealing credentials or launching an exploit of some sort. Once the attacker has root level access, the rootkit can be injected into the system.

2.1. User mode rootkits

A user mode rootkit modifies operating system executable files or libraries that interact with the kernel on the user's behalf. Examples of executable files that a rootkit might want to target include the system binaries *ls*, *ps*, *netstat* and *sshd*. These allow the user to view files, processes, network connections and perform remote logins respectively. Because Linux is open source software, an attacker can download the source to these programs and make any modifications she desires. This allows the attacker to build in functionality that effectively controls which files, processes, network connections and remote logons a user or system administrator ultimately sees, resulting in "hidden" information of the attacker's choosing. For a user mode rootkit to be successful, the system administrator must also use these trojaned binary files to administer and monitor the system. A system administrator who does not check the integrity of the executable files she is utilizing may very well be fooled by such a rootkit. This stealth may help the attacker maintain access to a system for a long time.

2.1.1. How a user mode rootkit gets injected into a system

Since the goal of a rootkit is to provide an attacker with a stealthy environment in which to carry out his activities, the attacker does not want to raise any suspicion. In other words, it is paramount that the system appear to be functioning normally. Therefore, the executables that the attacker replaces must be placed where the users expect to find them. On most Linux systems these files are located in either /bin, /usr/bin, /sbin or /usr/sbin (Nguyen, 2004). Typically, these directories are owned by root and other users are only allowed to execute the programs. For this reason, the attacker must have root level access in order to place his malicious files on the system. Once the attacker gains root level access, perhaps by guessing or cracking the password, he can insert his own malicious executable files or libraries, for example, by using the *cp* (copy) command to overwrite the existing version of a binary like *netstat* with his version of *netstat*.

2.1.2. How the user mode rootkit functions

User mode rootkits provide a stealthy environment to the attacker by hiding the attackers activities. This is usually accomplished by adding filtering capability to an executable file so that users, including system administrators, receive only the output that the attacker wants them to receive. For example, if the attacker wants to open up a port that he will use as a backdoor into the system, he will add functionality to programs like *netstat* or *lsof* that report information about open ports. The added functionality will filter the output that is returned to the user, showing the state of all ports except the attacker's chosen backdoor port. The attacker adds this functionality by modifying the source code for those programs, compiling and then installing on the target computer.

If the attacker wants to launch a process that will exfiltrate data from a system, he will hide his activities by modifying and installing malicious executables that report information about both the processes running on the system and the current network connections. These malicious executables will then "lie" to the user by omitting the attackers malicious process and port information.

2.1.3. Detecting user mode rootkits

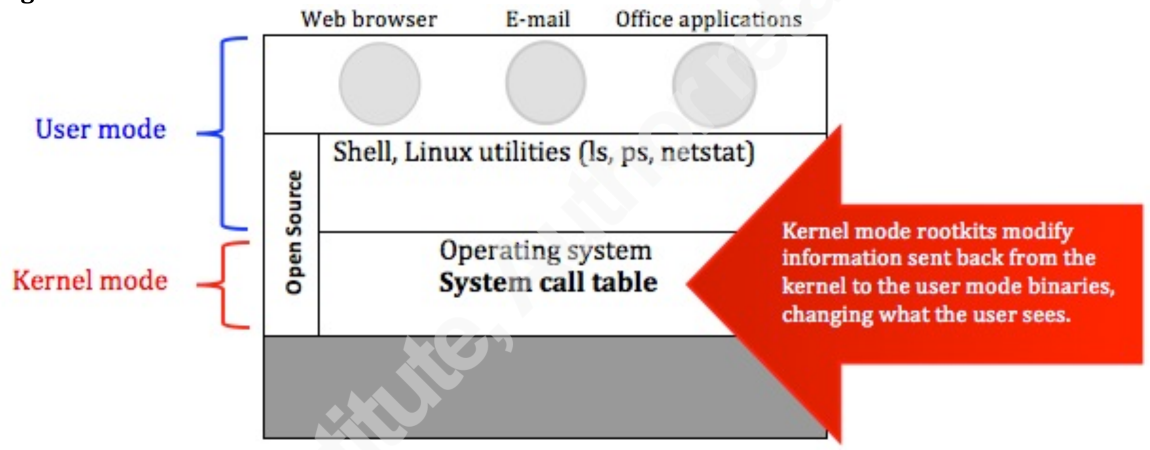
When operating systems components are regularly checked using file integrity checking tools like OSSEC, detection of user mode rootkits is quite successful. When an attacker replaces or modifies a binary file in system directories its cryptographic hash will change. If a system administrator watches for changes to these cryptographic hashes on a regular basis, comparing the hashes with a table of known good hashes, the rootkit can be easily discovered. Because detection is so straightforward, user mode rootkits are no longer common (Mcclure et al., 2012, p. 303).

2.2. Kernel mode rootkits

A kernel mode rootkit makes modifications to the kernel itself. As discussed above, the kernel is the interface between users and the hardware. Programs that a user or system administrator runs on a Linux system run in user mode and query the kernel for information about files, processes, network connections etc. as shown in Figure 1. Considering the example above, when the user wants a listing of files in the current directory the user would issue the *ls* command, which is a user mode utility. The *ls* command would then query the kernel, the kernel would return the listing of files to the *ls* utility and the *ls* utility would return that information to the user. A kernel mode rootkit would manipulate the information passed back from the kernel to the user mode utility. After performing a file integrity check of the *ls* binary confirming that it has not been tampered with, the system administrator trusts the *ls* command and so trusts the information it provides. In other words, the system administrator is not aware that he is not being told "the truth" about the directory listing he received.

Kernel mode rootkits provide a stealthier environment in which an attacker can operate because they are harder to detect and they are more thorough. It is trivial to determine that a user mode binary file has been tampered with. This is done by regularly comparing the cryptographic hashes on the binaries against a known good database, as noted above. However, detecting that the kernel has been tampered with is more difficult because so many of the tools used for detection are user mode tools, which can be fed inaccurate information by an undermined kernel. Kernel mode rootkits manipulate the information sent back from the kernel to user mode programs by interfering with the system call table (Skoudis, Zeltser, 2003, Chapter 8). System calls are the "fundamental interface between and application and the Linux kernel" according to the Linux manual (Kerrisk, 2013) and the system call table is a kernel data structure that maintains pointers to the locations in kernel memory where these system calls reside. Figure 3 shows the logical arrangement of the System call table in kernel mode space.

Figure 3


Web browser
E-mail
Office applications
User mode
Kernel mode
Open Source
Shell, Linux utilities (ls, ps, netstat)
Operating system
System call table
Kernel mode rootkits modify information sent back from the kernel to the user mode binaries, changing what the user sees.

System call table is kernel mode data structure

A kernel mode rootkit can consistently "lie" to any user mode process that issues those system calls. For example, in order for a user mode rootkit to hide a file it would have to alter and install malicious copies of all binaries that show files on the system. Since most of those binaries are using the same system call to query the file system, the kernel mode rootkit can fool them all without even knowing who is asking. Although the book was written over 10 years ago, *Malware: Fighting Malicious Code* gives an excellent fundamental overview of the Linux kernel, system calls and kernel mode rootkits (Skoudis, Zeltser, 2003, Chapter 8).

A rootkit can be injected into the system in one of several ways. Once injected into a system, the rootkit may use a variety of techniques to interfere with the system call table. Each of these will be discussed in a section below.

2.2.1. How a kernel mode rootkit gets injected into the system

There are currently three known methods for injecting a kernel mode rootkit into a system. These are summarized below based on the detailed description in McClure's book, *Hacking Exposed: Network Security Secrets & Solutions* (McClure et al., 2012, Chapter 5).

The first and still the most common method is by installing a loadable kernel module (LKM). LKMs were introduced to Linux around 1995 (Henderson, 2001). LKMs provide flexibility for both administrators and developers by allowing kernel level code to be added to or removed from a running kernel. In other words, this feature allows privileged users to alter the functionality of the kernel without recompiling it and often without requiring a reboot. LKMs were designed in 1995 to be used for device drivers, system calls, network drivers and some file system drivers (Henderson, 2001, Section 2.5). But shortly thereafter, attackers began using LKMs to manipulate the kernel in order to hide their malicious activities. The attacker must have root privileges but once she does, she can insert a malicious module into the kernel using, for example, the *insmod* or *modprobe* command. *Insmod* is a simple utility that inserts a module from any path and *modprobe* will load a module and any dependencies but the module must exist in the /lib/modules directory (Corbet & Rubini, 2005, Section 2.4.2). Placing a module in the /lib/modules directory may get noticed so attackers will often use the simpler *insmod* method. The Linux utility *lsmod* will provide the user with a listing of the currently inserted kernel modules. *Lsmod* gets the information by querying the /proc directory, specifically /proc/modules. The Linux operating system provides a view into the memory of a running system through the /proc directory. There are no files on a disk associated with the entries in this directory, instead it is an interface for viewing various parts of memory (Skoudis & Zeltser, 2003, p. 388). The /proc directory will be examined in more detail in section "Kernel Mode Rootkit Examples".

The second method used by attackers to inject a kernel mode rootkit is by using the special character device /dev/kmem. This was first demonstrated with the release of Phrack magazine #58 in 2001 with a rootkit named SucKIT. The authors state that the name stands for "stupid 'super user control kit'" (devik & sd, 2001).

The special device /dev/kmem points to an image of the running kernel's memory space. Using this technique the attacker modifies the running kernel in memory interfering in the same way that a LKM might with the system call table (Skoudis & Zeltser, 2003). Since these are changes made to the running kernel in memory, a system reboot would cripple the rootkit. Because it was more often used by attackers than by kernel developers, many current Linux distributions have disabled support for the /dev/kmem device including Fedora distributions starting in 2005 with the release of RHEL4 (Fedora wiki, 2014) and Ubuntu distributions in 2009 (Ubuntu wiki, 2014).

The third method used to inject a kernel mode rootkit is by using a special device similar to /dev/kmem called /dev/mem. This points to an image of physical memory, that is, not just kernel memory but the entire physical memory image. This was first introduced at Blackhat in 2009 (Lineberry, 2009). Based on my research, this method does not seem to have been widely used although there is a proof-of-concept rootkit from 2005 called phalanx (McClure et el. 2012, p. 304).

Because most Linux kernels still continue to support LKMs this is the method most often chosen by attackers to inject a rootkit (Corbet & Rubini, 2005, p. 3). Section 5 discusses specific examples of kernel mode rootkits and gives details about how each was injected.

2.2.2. How system calls are intercepted

After a kernel mode rootkit is injected it typically manipulates the system call table in order to create a stealthy environment for itself. Remember that the system call table is a kernel data structure that maps the memory locations for the system call functions used by user mode processes. The rootkit may swap out addresses in the system call table so that the address points not to the original function but instead to the attacker's malicious function. Or the attacker can modify the base location of the system call table itself, basically replacing the entire legitimate call table with the attacker's table. Other methods, though not widely used, include listening in on and intercepting system interrupts and intercepting communications with the virtual file system (McClure et al., 2012, p. 305).

However the system calls are interfered with, the goal of the rootkit is to modify information seen by users in order to hide its own processes and files and activities.

3. OSSEC

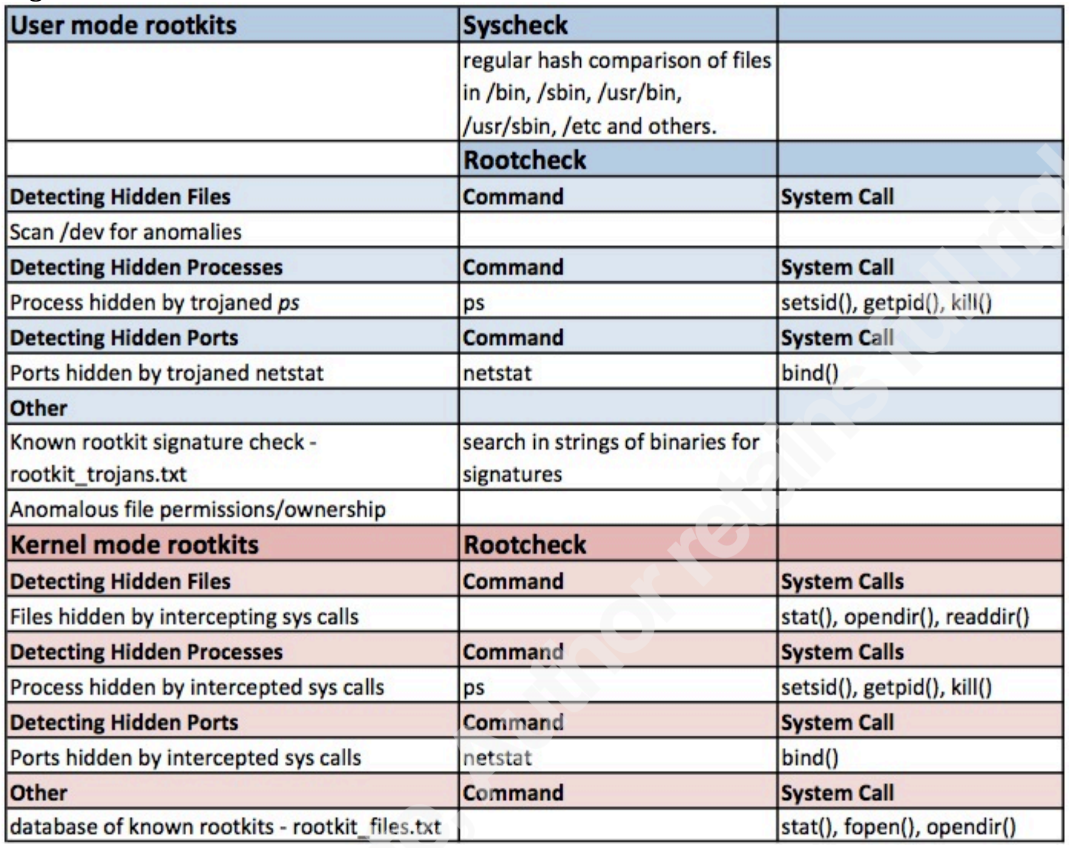
3.1. What is OSSEC?

OSSEC is a full-featured, multi-platform, host based intrusion detection system that performs many functions with the goal of monitoring and protecting the host. Although OSSEC can be installed on a single system and run as a standalone HIDS, this test system was set up using the server/agent model. The central server receives and analyzes data from each agent. Configurations, rules and alert log files are stored on the server. The most up to date and complete documentation can be found at http://www.ossec.net/doc/. Among OSSEC's features are file integrity monitoring, log integration and monitoring and rootkit detection for Windows and Linux systems. This paper will focus specifically on OSSEC's rootkit detection capabilities for Linux.

3.2. How OSSEC detects rootkits

OSSEC uses a variety of methods to detect rootkits. OSSEC can be configured to receive all syslog messages from the OSSEC agents. The OSSEC server will analyze the incoming messages by applying a set of configurable rules, similar to Snort. Some of these rules may aid an administrator in detecting rootkits, such as, rules that indicate privilege escalation for example. There are also two separate OSSEC modules that run on every agent, *Syscheck* and *Rootcheck*. These modules perform various tests that aid in detecting both user mode and kernel mode rootkits. Figure 4 organizes the various methods of detection that OSSEC uses for each type of rootkit. While *Rootcheck* performs most of the checks, *syscheck* is key when looking for user mode malicious binary files. Detection for each type of rootkit is discussed separately in the following sections.

Figure 4



User mode rootkits	**Syscheck**	
	regular hash comparison of files in /bin, /sbin, /usr/bin, /usr/sbin, /etc and others.	
	Rootcheck	
Detecting Hidden Files	**Command**	**System Call**
Scan /dev for anomalies		
Detecting Hidden Processes	**Command**	**System Call**
Process hidden by trojaned *ps*	ps	setsid(), getpid(), kill()
Detecting Hidden Ports	**Command**	**System Call**
Ports hidden by trojaned netstat	netstat	bind()
Other		
Known rootkit signature check - rootkit_trojans.txt	search in strings of binaries for signatures	
Anomalous file permissions/ownership		
Kernel mode rootkits	**Rootcheck**	
Detecting Hidden Files	**Command**	**System Calls**
Files hidden by intercepting sys calls		stat(), opendir(), readdir()
Detecting Hidden Processes	**Command**	**System Calls**
Process hidden by intercepted sys calls	ps	setsid(), getpid(), kill()
Detecting Hidden Ports	**Command**	**System Call**
Ports hidden by intercepted sys calls	netstat	bind()
Other	**Command**	**System Call**
database of known rootkits - rootkit_files.txt		stat(), fopen(), opendir()

OSSEC's detection methods for user mode and kernel mode rootkits

3.2.1. User mode rootkit detection

As described in section 2.1, user mode rootkits typically modify or replace existing operating system executable files and libraries. The OSSEC module *Syscheck* performs regular file integrity checks on these files by comparing the current cryptographic hash with a known good hash for the file. If the hashes do not match then OSSEC reports a change to the file, indicating a possible rootkit. The database of known good hashes is initially created the first time *syscheck* runs after an OSSEC agent is installed. The agent sends an encrypted hash of every file in the directories defined in the OSSEC configuration file, /var/ossec/etc/ossec.conf in the *syscheck* section,. Directories that are hashed by default include */bin*, */usr/bin*, */sbin*, */usr/sbin* and */etc*. The default interval after the initial build of the *syscheck* database is 21600 seconds (6 hours) according to the OSSEC online documentation at http://www.ossec.net/doc/manual/syscheck/index.html and 79200 seconds (22 hours) in the most recent version available for download at http://www.ossec.net/files/ossec-hids-2.7.1.tar.gz. The interval that *Syscheck* runs as well as the directories that will be hashed are easily configurable and require that the agent be restarted on the client machine before any changes would take effect.

OSSEC also maintains two files used in rootkit detection. The first is rootkit_files.txt and contains a list of file names known to be user mode rootkits. The files are searched based on file name only and if OSSEC can open the file for reading with the fopen() system call then an alert is logged for a possible user mode rootkit. The second file maintained for rootkit detection is rootkit_trojans.txt. This file contains signatures that known rootkits have embedded in the binary file. By default, the binaries in the directories /bin, /sbin, /usr/bin and /usr/sbin are searched. *Rootcheck* extracts the embedded strings from each binary file and uses a regular expression to identify a match. OSSEC refers to this as signature detection because many rootkits contain unique strings in trojaned versions of common utilities such as *login* or *ps*. Additional signatures may be added to the /var/ossec/etc/shared/rootkit_trojans.txt file.

OSSEC's *rootcheck* module runs at regular intervals and among other things attempts to find both user mode and kernel mode rootkits by querying the system for information in multiple ways and comparing the results. When the *rootcheck* module finds discrepancies in information about a file, a process, port or network interface it will raise an alert for a suspected rootkit.

The Linux utility *stat* will show the status of a file or directory. It provides extensive information about a file or directory but of interest here the link count. When *stat* is run on a directory the link count will show how many files the directory contains. The *rootcheck* module will also use the readdir() system call to determine the link count of a directory. If there is a discrepancy it is flagged as a possible rootkit. (Cid, 2011, module check_rc_sys.c)

The Linux utility *ps* can show running processes on a system. Rootkits commonly attempt to hide the attacker's processes to prevent detection. A user mode rootkit could do this by modifying the *ps* binary file so that it omits the attackers processes from process listings. *Rootcheck* attempts to detect such behavior by issuing multiple system calls and comparing the results to the output from *ps*. *Rootcheck* calls getsid(), getpgid() and kill() for all process IDs and if any of them finds a process that is not listed by *ps* then *rootcheck* logs an alert about a possible user mode rootkit or "trojaned" version of the *ps* utility (Hay & Cid, 2008, p. 162). These system calls are discussed in more detail in section 3.2.2.

Another common binary that a user mode rootkit might replace is the *netstat* utility (Hay & Cid, 2008, p. 164). *Netstat* provides information about network connections and open ports. Since malware will often attempt to connect back to an attacker's machine in order to transfer data between the attacker and victim, rootkits will commonly attempt to hide those connections to help maintain stealth. *Rootcheck* uses the *netstat* utility to show the open ports on the system. Then it uses the bind() system call to attempt to connect to every TCP and UDP port. If it is unable to bind to a port then that port must be in use and should also be found in the *netstat* output. If *netstat* does not show the port, it is flagged as a possible rootkit because the *netstat* utility may have been modified to hide it. In other words, it may indicate a trojaned version of *netstat* (Cid, 2011, module check_rc_ports.c)

Sniffing network traffic, especially in promiscuous mode, allows an attacker to capture data transfers that may contain sensitive data. Attackers may attempt to conceal the sniffer by installing a modified version of the *ifconfig* utility, which reports on the state of network interfaces. Like other hardware on a computer system, every network interface is managed by the kernel. The kernel uses a device driver as a "liaison" to the interface. When a privileged user requests that an interface be put in to promiscuous mode so that it is able to sniff all traffic on the network, the kernel asks the device driver to set the *IFF_PROMISC* flag on that device (Corbet, Rubini, 2005, chap 14). One way to detect a sniffer on the network is by looking for network interfaces that are operating in promiscuous mode. *Rootcheck* uses two different methods to query the *IFF_PROMISC* flag on network interfaces and then compares the results looking for a discrepancy. First, it asks the kernel directly for the value in *IFF_PROMISC* and then it runs the user mode utility *ifconfig* and examines the output to see what it thinks the value of IFF_PROMISC is. If there is a discrepancy, it reports a possible user mode rootkit. In other words, perhaps the *ifconfig* utility has been modified to hide the true status of a network interface.

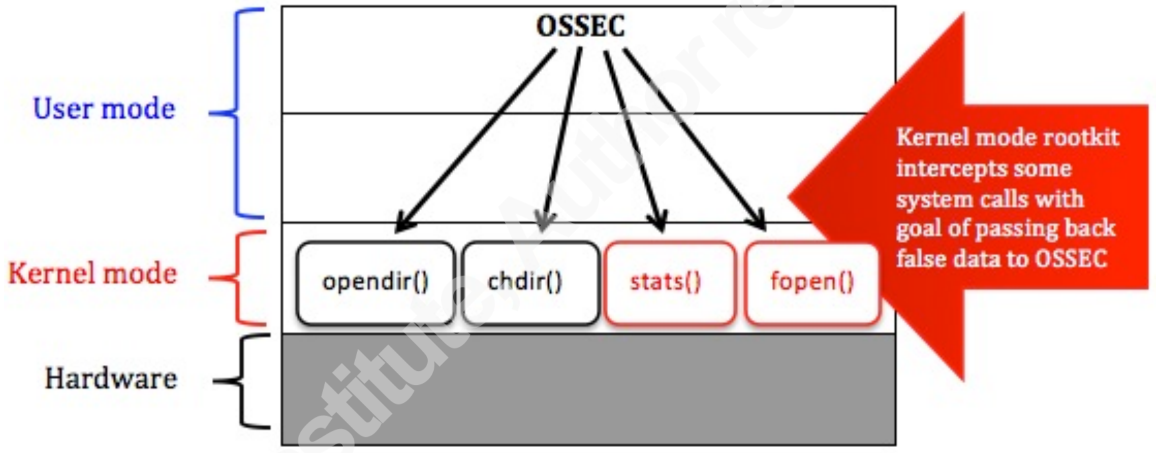
Additionally, the *rootcheck* module examines anomalous behavior on the system looking for user mode rootkits as well. First, it will check files in the /dev directory. The /dev directory should contain device files or directories containing device files. This directory is carefully checked for anomalous files because rootkits commonly use this location to hide their files. Next, *rootcheck* will look through the entire file system for files that have characteristics commonly used by rootkits. For example, it will look for and log all files that are owned by root but can be written to and/or executed by other users. It will record files and directories that are hidden using "." as the first character of the file name. Additionally, it will record any binaries that have the SUID bit set. This means that the file, when executed, inherits the permissions of the owner of the file instead of as the user executing the file. This could allow a non-privileged user to execute a program as a privileged user and OSSEC regards this as suspicious. (Hay & Cid, 2008, p 162).

3.2.2. Kernel mode rootkit detection

Since a kernel mode rootkit makes changes to the kernel with the goal of intercepting system calls as described in section 2.2, it can manipulate information sent to and from user mode tools. OSSEC is a user mode application so it also relies on data passed to it from the kernel. The *rootcheck* module looks for possible intercepted system calls that may be hiding files and processes.

Figure 5 shows the system calls used by OSSEC's *rootcheck* module to look for the existence of hidden files used by known rootkits.

Figure 5


OSSEC
User mode
Kernel mode
Hardware
opendir()
chdir()
stats()
fopen()
Kernel mode rootkit intercepts some system calls with goal of passing back false data to OSSEC

Rootcheck module makes multiple system calls to check for known rootkit related files

The list of files from known rootkits that *rootcheck* attempts to locate can be found on the OSSEC server in */var/ossec/etc/shared/rootkit_files.txt*. *Rootcheck* tries to open each file using the system calls opendir(), chdir(), stats() and fopen(). Figure 5 illustrates an example where *rootcheck* is able to open a file with the opendir() and chdir() system calls but not with stats() or fopen(). This may be an indication that a rootkit has intercepted the stats() and fopen() system calls but not the opendir() or chdir() calls. In this example, *rootcheck* would report a possible kernel level rootkit based on that discrepancy. This detection method will have limited success since the file names must be present in rootkit_files.txt or *rootcheck* will not attempt detection. In other words, a file hidden by a rootkit can only be detected if the name of the file is known. A custom rootkit or simple modification of the source code for a known rootkit could thwart this method of detection.

In section 3.2.1 it was noted that rootkits often attempt to hide running processes to maintain stealth. A user mode rootkit would hide a process by installing a modified version of the *ps* utility. This may be detected during the file integrity checks of the *syscheck* module or it may be detected by the *rootcheck* module when comparing the output from the *ps* utility to the output from three different system calls that query running processes. Similarly, *rootcheck* will look for processes hidden by a kernel mode rootkit by querying the system using the getsid(), getpgid() and kill() system calls and comparing their outputs. The Linux man pages for getsid() and getpid() explain that every process has both a session ID and a process ID. The Linux man page for kill() explains that sending a zero as the signal to a process ID will not kill a process but instead will return a 0 if the process exists. If the process does not exist the return value from kill() will be -1. *Rootcheck* cycles through all possible process IDs, issuing all three system calls and comparing the results. If getsid() finds the process but getpgid() or kill() does not it may indicate a kernel level rootkit that has intercepted a system call. In this case, OSSEC will log an alert identifying the suspect process.

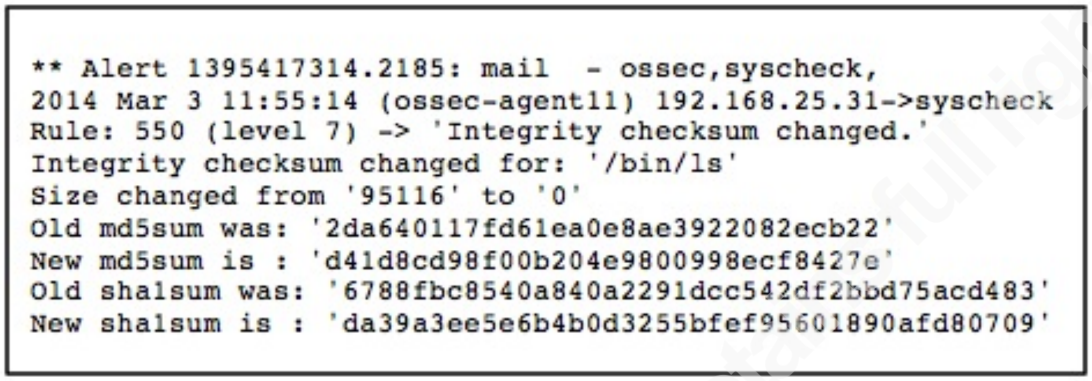
3.3. What OSSEC does not do

As noted in section 2.2, the most common method used to inject a kernel mode rootkit is to insert a loadable kernel module. OSSEC does not detect the kernel module loading, nor does it detect changes that a rootkit makes to the system call table.

4. User Mode Rootkit Examples

As described in section 2.1, user mode rootkits modify operating system executable files or libraries that interact with the kernel on the user's behalf. Rootkits that replace Linux binary files are trivial to detect with OSSEC's *syscheck* module. Successful detection depends on proper tuning of the OSSEC configuration file. *Syscheck* will only perform integrity checking on the files and directories specified in the ossec.conf file. When *syscheck* discovers that the cryptographic hash for a file has changed, an alert is logged that would look similar to Figure 6.

Figure 6



```
** Alert 1395417314.2185: mail  - ossec,syscheck,
2014 Mar 3 11:55:14 (ossec-agent11) 192.168.25.31->syscheck
Rule: 550 (level 7) -> 'Integrity checksum changed.'
Integrity checksum changed for: '/bin/ls'
Size changed from '95116' to '0'
Old md5sum was: '2da640117fd61ea0e8ae3922082ecb22'
New md5sum is : 'd41d8cd98f00b204e9800998ecf8427e'
Old sha1sum was: '6788fbc8540a840a2291dcc542df2bbd75acd483'
New sha1sum is : 'da39a3ee5e6b4b0d3255bfef95601890afd80709'
```

Syscheck generated alert when a file's cryptographic hash has changed

User mode rootkits that modify run time library functions prove to be more difficult to detect because binary files on disk are not changed and so *syscheck* is not effective at detection. Instead, malicious libraries are inserted into the running process and exist only in memory. The OSSEC *rootcheck* module can be helpful in discovering this type of user mode rootkit and is described in detail in the next section.

4.1. Jynx2

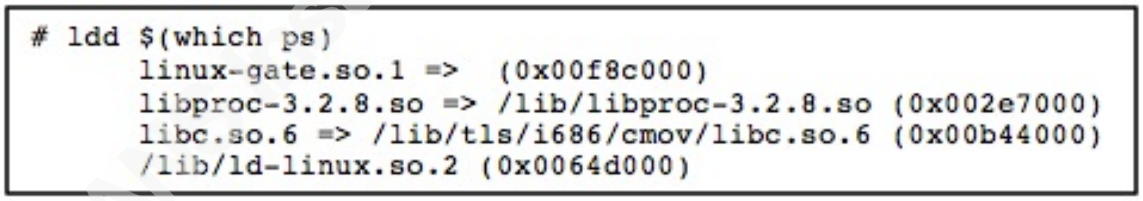
The jynx2 rootkit operates by loading malicious library functions into system binaries dynamically at runtime instead of replacing the actual binary on the file system with a modified version (ErrProne, 2012). This provides similarly stealthy results but has the added bonus that a statically calculated hash over system binaries would not reveal this rootkit.

4.1.1. Inserting the shared library

In Linux, when a program is executed, the system checks for any additional shared libraries that need to be loaded at run time. This is done by first consulting the files /etc/ld.so.conf and /etc/ld.so.preload. Any libraries found here will be loaded first and will take precedence over other libraries. In addition, the environment variable LD_PRELOAD may be used to point to a shared library that will be loaded and take precedence just like a library in /etc/ld.so.preload (Wheeler 2000). The jynx2 rootkit makes use of this feature by creating a shared library that will cause files, processes and ports to be hidden when desired. Jynx2 forces the use of this library by adding it to the /etc/ld.so.preload file. The /etc/ld.so.preload file is then hidden by the rootkit. Once the rootkit has been injected, utilities like *ls*, *ps* and *netstat* will load the jynx2 shared library causing it to insert functionality into the running process that will hide evidence of files, processes and connections used by the malware. In addition, the rootkit provides a shared library called reality.so that can be loaded when the attacker would like to view all files, processes and network connections including the hidden ones. In other words, it provides a way for the attacker to view the actual state of the compromised system.

This rootkit was tested on a Ubuntu distribution running Linux kernel version 2.6.32-21. The Linux command *ldd* will show shared libraries used by a process. Figure 7 shows the shared libraries used by the *ps* utility before the jynx2 rootkit was injected and Figure 8 shows the shared libraries after the malware injection. Once the rootkit is injected by adding an entry for it in /etc/ld.so.preload, the *ps* utility will load the jynx2.so shared library every time it is run.

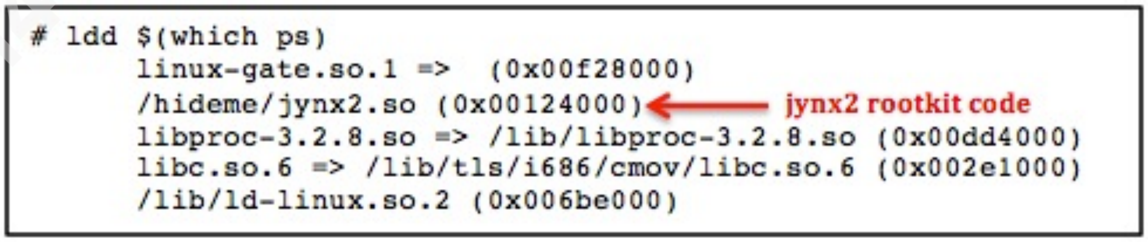
Figure 7



```
# ldd $(which ps)
        linux-gate.so.1 =>  (0x00f8c000)
        libproc-3.2.8.so => /lib/libproc-3.2.8.so (0x002e7000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x00b44000)
        /lib/ld-linux.so.2 (0x0064d000)
```

Shared library dependencies for the ps utility before rootkit injection

Figure 8



```
# ldd $(which ps)
        linux-gate.so.1 =>  (0x00f28000)
        /hideme/jynx2.so (0x00124000)
        libproc-3.2.8.so => /lib/libproc-3.2.8.so (0x00dd4000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x002e1000)
        /lib/ld-linux.so.2 (0x006be000)
```

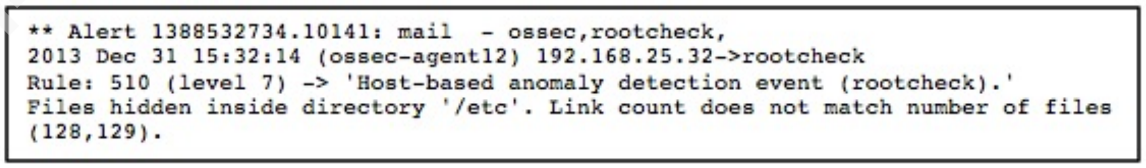
Shared library dependencies for the ps utility after rootkit injection

Running the *ldd* command on other utilities such as *ls* and *netstat* would show the jynx2.so dependency as well.

OSSEC may log an alert when the user initially injects the rootkit because, as noted above, the initial injection requires root privileges. If the elevation of privileges is achieved by knowing or guessing a root level password, the su operation is logged by default in OSSEC. Once the attacker has root privileges, if the subsequent actions taken by were logged to syslog then OSSEC could be configured to alert on those actions. In some environments this type of auditing may take place and would help to detect the kernel module injection into the system. If the elevation of privilege occurs through some exploit then it may go undetected. For this test, the initial elevation of privilege to root was accomplished by logging in as the root user using the "su -" command, which was detected by OSSEC.

Once injected, OSSEC will not detect the change with *syscheck*, because the only file that has been altered by the rootkit is /etc/ld.so.preload but that is hidden by the rootkit so *syscheck* does not calculate a hash of the file. Rootcheck, however, will notice that something is wrong. As noted in section 3.2.1, the *rootcheck* module will attempt to query the system in different ways in order to detect hidden objects. *Rootcheck* runs the *stat* utility on any directories specified in the ossec.conf file. *Stat* will return a count of the number of files that it sees in the directory and calls it "link count". *Rootcheck* then does a cross check on that link count using the system call readdir(). If readdir() finds more files than *stat* showed it is flagged as a possible kernel level rootkit. This could be considered a user mode rootkit, however, because jynx2 is not actually changing the kernel or modifying the system call table. Instead it is inserting a shared library into the *stat* process that provides similar functionality but is modified to hide certain files.

Figure 9



```
** Alert 1388532734.10141: mail  - ossec,rootcheck,
2013 Dec 31 15:32:14 (ossec-agent12) 192.168.25.32->rootcheck
Rule: 510 (level 7) -> 'Host-based anomaly detection event (rootcheck).'
Files hidden inside directory '/etc'. Link count does not match number of files
(128,129).
```

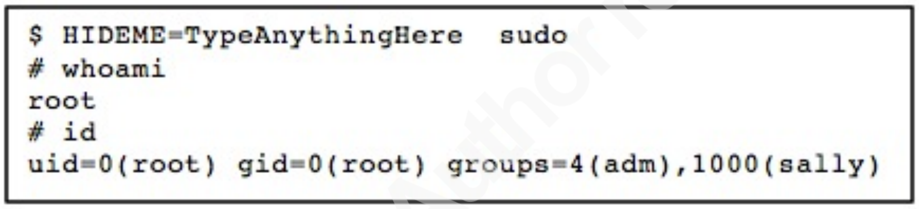
Rootcheck logs an alert when file count returned from stat and readdir() differ.

Figure 9 shows the alert generated when OSSEC runs the *rootcheck* module on my test system infected with the jynx2 rootkit. Because the /etc/ld.so.preload file has been hidden by the rootkit from one but not both of the system calls used by *rootcheck*, stat() and readdir(), it logs an alert about that finding.

4.1.2. Jynx2 privilege escalation feature

The jynx2 rootkit also has as SUID shell or privilege escalation feature that is implemented by defining an environment variable in the config.h include file for jynx2. On the infected system at the command prompt, setting an environment variable that was previously defined in the config.h file to some value other than NULL and issuing the *sudo* command signals the rootkit to create a root shell. In this test the environment variable name was "HIDEME".

Figure 10



```
$ HIDEME=TypeAnythingHere  sudo
# whoami
root
# id
uid=0(root) gid=0(root) groups=4(adm),1000(sally)
```

jynx2 privilege escalation goes undetected by OSSEC

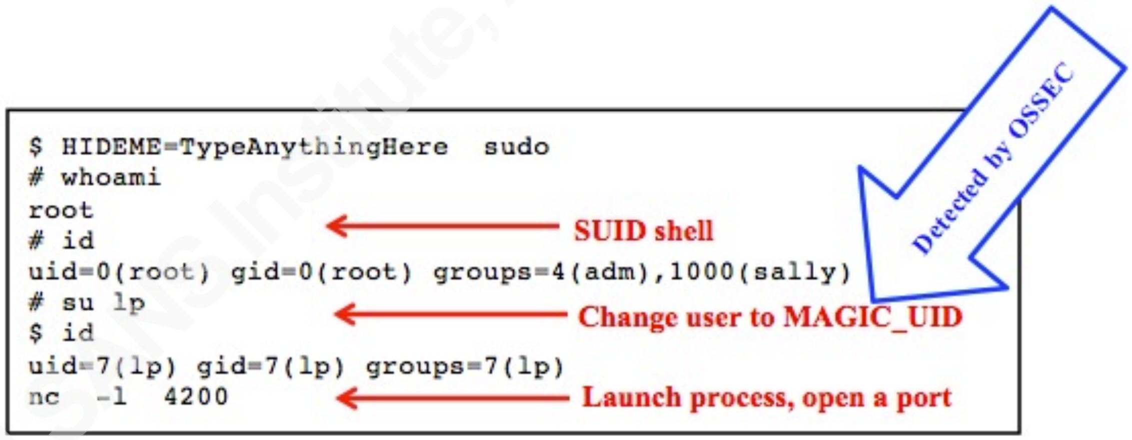
Figure 10 shows how the jynx2 rootkit can create a root shell. A process with setuid/setgid permissions is effectively running with the privileges of the owner of that executable file, in this case root, so it has the ability to change its own UID to any value (Bovet, Cesati, 2006, p. 810). The executable file *sudo* is owned by root and has the setuid bit set so that any user launching the process will have an effective UID of 0. The rootkit takes advantage of this setuid feature of Linux by using the effective UID 0 permissions of the running process to also change the process UID to 0 and then spawning a shell. The spawned shell will inherit the permissions of the parent process. In this example, the jynx2 library module is inserted dynamically into the running process *sudo*. The rootkit examines the value of the HIDEME variable and detects that it not NULL. That is a signal to the rootkit that the attacker is requesting a shell. The rootkit causes the *sudo* process to change its UID to 0 and then spawn a shell that inherits root permissions. By default, this action does not log to syslog so OSSEC is unaware of its occurrence. If a message were logged to syslog on the infected host, that message would also be received by the OSSEC server and would trigger an alert with a description like "User successfully changed UID to root." or "Successful sudo to ROOT executed" to warn of a privilege escalation.

The behavior of the SUID shell feature of jynx2 was determined by analyzing the source code for the rootkit (ErrProne, 2012, module jynx2.c)

4.1.3. Hiding ports and processes

Jynx2 will hide processes and ports as well. The SUID shell as described above makes this a very easy procedure. First, the attacker launches a SUID shell as outlined above. From that shell, the attacker uses *su* to change her account to the account associated with the MAGIC_UID variable also from the config.h file. Any process launched or listening ports opened at that point will be hidden from other users. Figure 11 shows the commands used to launch a *netcat* listener on port 4200 from a SUID shell.

Figure 11

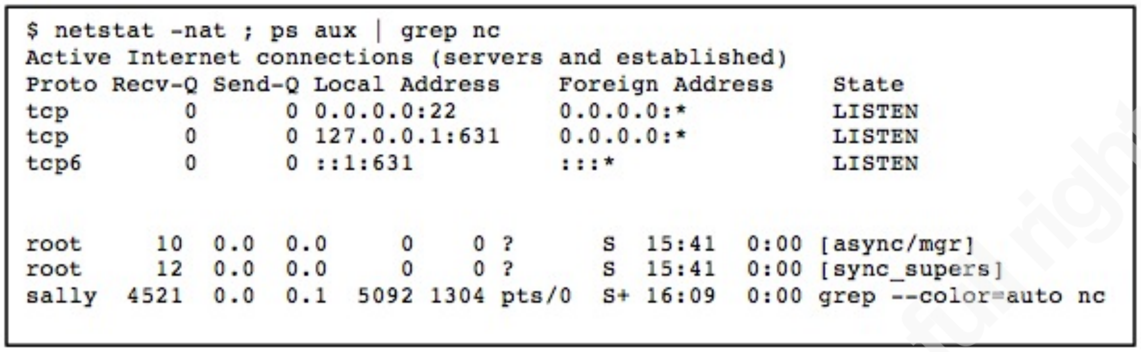


```
$ HIDEME=TypeAnythingHere  sudo
# whoami
root
# id
uid=0(root) gid=0(root) groups=4(adm),1000(sally)
# su lp
$ id
uid=7(lp) gid=7(lp) groups=7(lp)
nc  -l  4200
```

Starting a netcat listener using the "magic" UID. Only "su lp" gets noticed by OSSEC.

Figure 12 shows the status of ports and processes before the *netcat* listener was launched. The process information output was filtered on the string "nc" for readability.

Figure 12



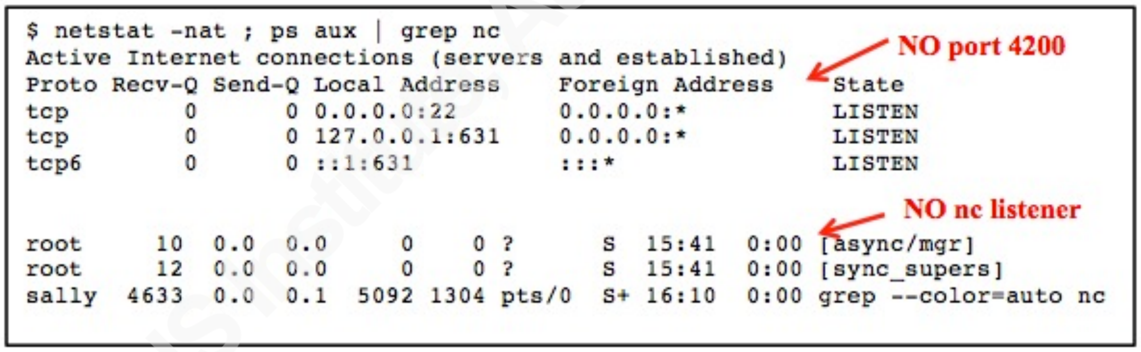
```
$ netstat -nat ; ps aux | grep nc
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address       Foreign Address       State
tcp        0      0 0.0.0.0:22          0.0.0.0:*             LISTEN
tcp        0      0 127.0.0.1:631       0.0.0.0:*             LISTEN
tcp6       0      0 ::1:631             :::*                  LISTEN


root      10  0.0  0.0      0    0 ?      S  15:41  0:00 [async/mgr]
root      12  0.0  0.0      0    0 ?      S  15:41  0:00 [sync_supers]
sally   4521  0.0  0.1   5092 1304 pts/0  S+ 16:09  0:00 grep --color=auto nc
```

Open ports and nc processes prior to launching nc

Figure 13 shows the state of open ports and processes as seen by any other user on the jynx2 infected system. Both port 4200 and the nc process are hidden illustrating the rootkit's successful hiding technique.

Figure 13



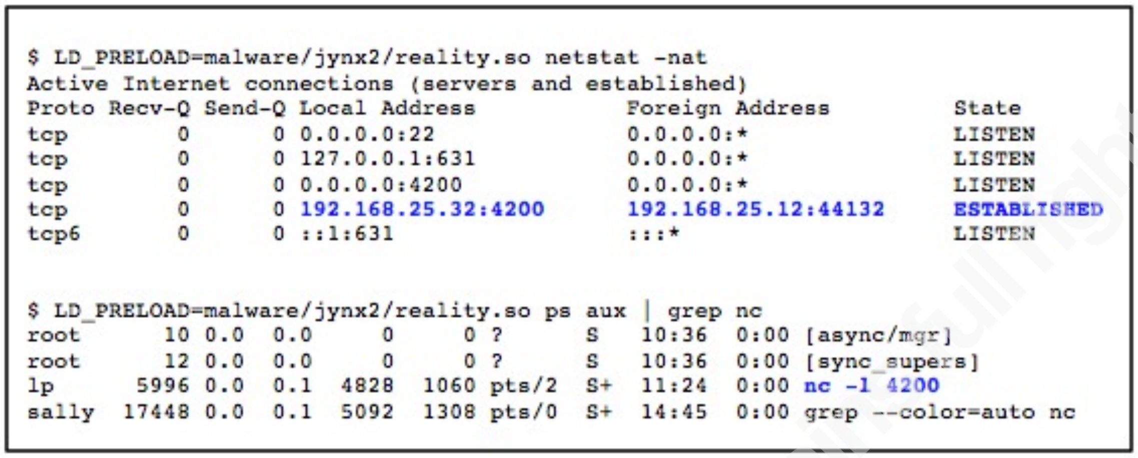
```
$ netstat -nat ; ps aux | grep nc
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address       Foreign Address       State
tcp        0      0 0.0.0.0:22          0.0.0.0:*             LISTEN
tcp        0      0 127.0.0.1:631       0.0.0.0:*             LISTEN
tcp6       0      0 ::1:631             :::*                  LISTEN


root      10  0.0  0.0      0    0 ?      S  15:41  0:00 [async/mgr]
root      12  0.0  0.0      0    0 ?      S  15:41  0:00 [sync_supers]
sally   4633  0.0  0.1   5092 1304 pts/0  S+ 16:10  0:00 grep --color=auto nc
```

Open ports and processes after launching nc

Figure 14 shows the "reality" state of the open ports and processes as seen by a user when the rootkit's reality.so shared library is loaded dynamically with the *netstat* and *ps* processes.

Figure 14



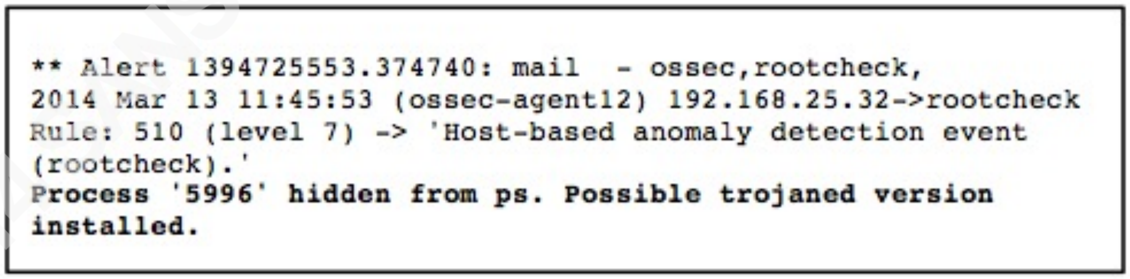
```
$ LD_PRELOAD=malware/jynx2/reality.so netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:4200            0.0.0.0:*               LISTEN
tcp        0      0 192.168.25.32:4200      192.168.25.12:44132     ESTABLISHED
tcp6       0      0 ::1:631                 :::*                    LISTEN

$ LD_PRELOAD=malware/jynx2/reality.so ps aux | grep nc
root        10 0.0  0.0     0     0 ?       S    10:36   0:00 [async/mgr]
root        12 0.0  0.0     0     0 ?       S    10:36   0:00 [sync_supers]
lp        5996 0.0  0.1  4828  1060 pts/2   S+   11:24   0:00 nc -l 4200
sally    17448 0.0  0.1  5092  1308 pts/0   S+   14:45   0:00 grep --color=auto nc
```

Jynx2 module reality.so loaded at runtime will show true state of system

Interestingly, OSSEC detects the hidden *nc* process although it does not detect the listening TCP port. The OSSEC module *rootcheck* calls getsid(), getpgid() and kill() for all possible process IDs and if any of them finds a process that is not listed by *ps* an alert is logged such as the one shown in Figure 15. This alert indicates that there is a process running that *ps* does not see. In this case, PID 5996 is hidden because the jynx2 rootkit has loaded the runtime library, jynx2.so, into *ps* to hide any processes started by the MAGIC_UID user.

Figure 15



```
** Alert 1394725553.374740: mail  - ossec,rootcheck,
2014 Mar 13 11:45:53 (ossec-agent12) 192.168.25.32->rootcheck
Rule: 510 (level 7) -> 'Host-based anomaly detection event
(rootcheck).'
Process '5996' hidden from ps. Possible trojaned version
installed.
```

OSSEC alerts when kill finds a process that ps did not

OSSEC does not log an alert for the hidden TCP port. The OSSEC documentation states that the *rootcheck* module will attempt to bind() to each port and compare the results with output from *netstat*. Ports that OSSEC cannot bind to should be in the *netstat* output, if not then the following alert would be logged:

`“Port 4200 hidden.  Kernel-level rootkit or trojaned version of netstat.”`. On this test machine, TCP port 4200 was hidden and tested in both a LISTENING and an ESTABLISHED state. In both cases the event was not detected by OSSEC.

4.1.4. OSSEC detection

The OSSEC server logged an alert for the initial privilege escalation in order to install the rootkit and also when the “su lp” command was issued from the SUID shell. OSSEC was able to detect hidden files and hidden processes, however, OSSEC was not able to detect the SUID shell or the hidden TCP port.

5. Kernel Mode Rootkit Examples

Kernel mode rootkits manipulate the information sent back from the kernel to user mode programs and accomplish this most often by interfering with the system call table. The most common method of injection of kernel mode rootkits is by using loadable kernel modules to inject malicious code into a running process. This section will detail the exploration of three different kernel rootkits. All three use the LKM injection technique but all three make use of the loaded module differently. OSSEC is able to alert on some aspects of these rootkit but not all.

5.1. The /proc directory

The /proc directory in Linux provides information to processes about kernel memory allocation. The directories found in /proc are sometimes called virtual directories because they do not actually exist on disk. The /proc files are organized representations of some of the information stored in memory regarding running processes (Terrehon & Bauer, 1999). Every time a process starts or finishes, a new directory is created under /proc. The PID for the process is used as the directory name. There are other directories under /proc like /proc/modules and /proc/net. /proc/modules contains information about loadable kernel modules and /proc/net contains information about network connections. Some of the user mode utilities that system administrators use like lsmod and netstat take the information stored in /proc and reformat it (Skoudis & Zeltser, 2003, Chap. 8). This implies that if rootkits can hide things in /proc they will also be hidden from the user mode tools. We will see examples of this in the test results that follow. OSSEC could be configured to monitor the /proc directory and alert on changes, however, since this directory is constantly changing it would likely result in too many false positives and would therefore be impractical.

5.2. Linux 2.6 rootkit backdoor

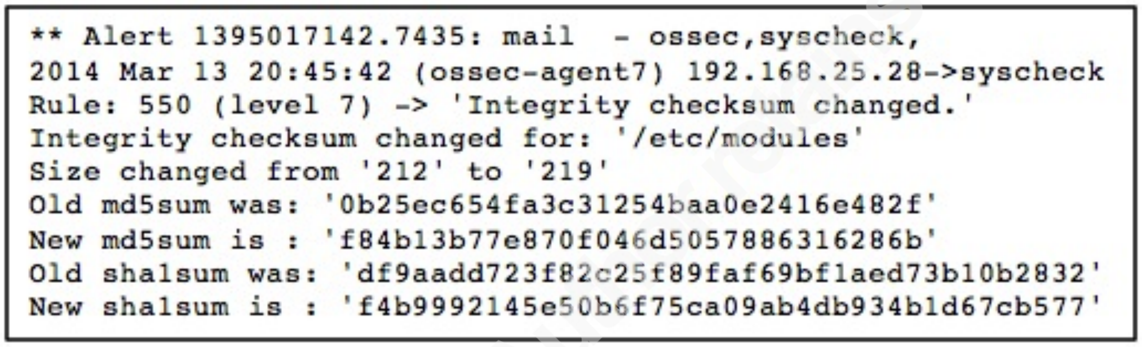
The Linux 2.6 Rootkit backdoor (LRK-BD) is a kernel mode rootkit that is injected via an loadable kernel module (Analiz, 2012). Its functionality is limited to privilege escalation. While it may seem counter intuitive to install a rootkit whose only function is to provide root access, especially since the installation of the rootkit requires root privileges, it provides the attacker with a stealthy method of privilege escalation after the initial installation.

The LRK-BD consists of a kernel module called *security.ko* and a control program called *kontrol.* The control program takes a password as input and launches a shell, invoking the loaded kernel module security.ko. The source code for the kernel module was not available, however, a search through the embedded strings provides some insight as to how the privilege escalation may have been achieved. Among the strings found in the kernel module are *prepare_creds* and *commit_creds*. As explained by Perla and Massimiliano, there are credential structures associated with every process in Linux kernel versions since 2.6.29 that contain the user and group IDs to help determine the level of access to various resources by the running process. This credential structure can be modified by creating a copy of the current credentials with prepare_creds(), setting the UID/GID fields to 0 and using commit_creds() to make the changes. This is a method often used by kernel mode rootkits to achieve privilege escalation (Perla & Massimiliano, 2010, Chap. 4).

5.2.1. Inserting the kernel module

This rootkit was tested on a Ubuntu distribution running Linux kernel version 3.2.0-32. The *insmod* command was used to insert the kernel module. Since this requires root level access, the initial privilege escalation was logged by OSSEC. In order for this rootkit to provide stealthy privilege escalation to the attacker the kernel module must persist across reboots. One way to accomplish this is to place the malicious module in the `/lib/modules` directory and to edit the `/etc/modules` file, adding an entry for the new kernel module (Loadable_modules, 2014). Since the OSSEC *syscheck* module checks for changes to the files in the `/etc` directory, the change to the `/etc/modules` file triggered the alert shown in Figure 16.

Figure 16



```
** Alert 1395017142.7435: mail  - ossec,syscheck,
2014 Mar 13 20:45:42 (ossec-agent7) 192.168.25.28->syscheck
Rule: 550 (level 7) -> 'Integrity checksum changed.'
Integrity checksum changed for: '/etc/modules'
Size changed from '212' to '219'
Old md5sum was: '0b25ec654fa3c31254baa0e2416e482f'
New md5sum is : 'f84b13b77e870f046d5057886316286b'
Old sha1sum was: 'df9aadd723f82c25f89faf69bf1aed73b10b2832'
New sha1sum is : 'f4b9992145e50b6f75ca09ab4db934b1d67cb577'
```

OSSEC alert due to change in hash for the file /etc/modules

5.2.2. The privilege escalation function

To launch the LRK-BD privilege escalation, The control program, *kontrol*, sends a command to the kernel module by sending a password to /proc/security. The file /proc/security was created when the malicious kernel module, security.ko, was inserted with the *insmod* command. The attacker can interact with the kernel module via this virtual file in /proc. A simplified but equally effective version of the command is shown below:

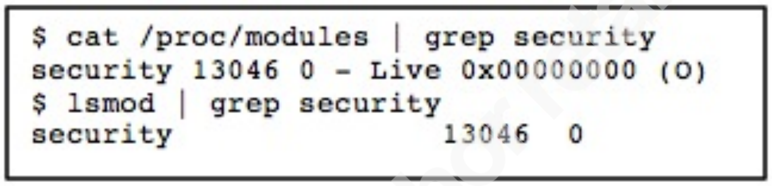
```
$ echo fabrika >> /proc/security
$ whoami
root
$ id
uid=0(root) gid=0(root) groups=0(root), 4(adm),  24(cdrom), 
1000(sally)
```

The string "fabrika" is a password that the module requires and was found by examining the kontrol.c source code. The elevation to UID 0 is not noticed by OSSEC but the root shell process is not hidden so the shell could be detected using the *ps* command:

```
$ ps aux | grep bash
sally   3630 0.0 0.3  7120 3528 pts/0  Ss   13:38  0:00 bash
root    4797 0.7 0.3  7120 3528 pts/3  Ss+  13:46  0:00 bash
sally   4902 0.0 0.0  4372  836 pts/0  S+   13:46  0:00 grep
--color=auto bash
```

The LRK-BD rootkit was detected by OSSEC twice during installation but once installed and operational, provides a stealthy privilege escalation exploit. The malicious kernel module inserted by the attacker is not detected at any time by OSSEC but it can be seen in /proc/modules and when using the lsmod command as shown in Figure 17.

Figure 17



```
$ cat /proc/modules | grep security
security 13046 0 - Live 0x00000000 (O)
$ lsmod | grep security
security               13046  0
```

Malicious kernel module, security, can be seen in /proc/modules and with lsmod

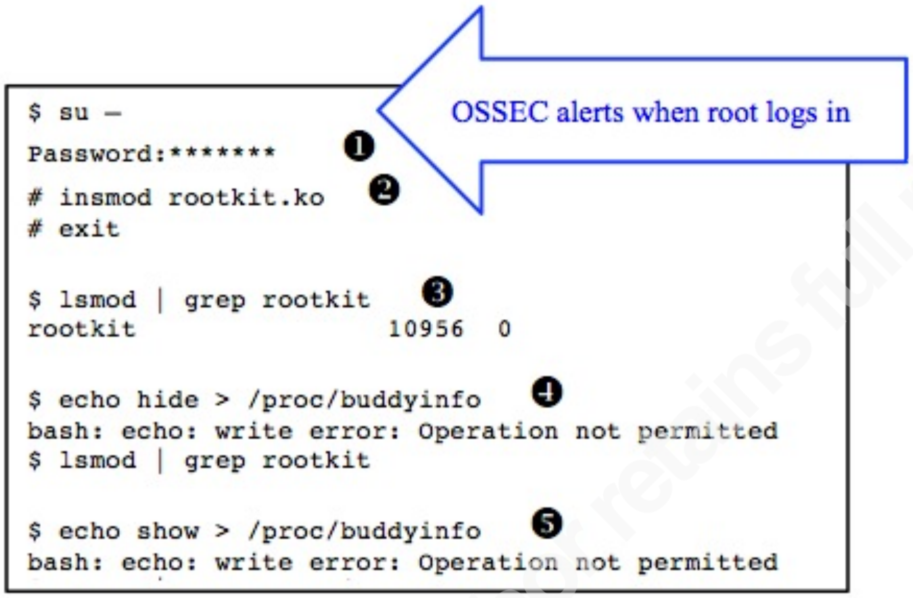
5.3. Average Coder rootkit

The Average Coder rootkit was written by Matias Fontanini and uses the LKM method of injection (Fontanini, 2011). This rootkit can hide the kernel module from user mode tools that print the contents of /proc/modules, like *lsmod*, and it can also hide ports and processes. It does not offer a file hiding function.

5.3.1. Inserting and hiding the kernel module

Figure 18 shows the kernel module being injected into the system, followed by a test of the hide/show module functionality.

Figure 18



```
$ su –
Password:*******
# insmod rootkit.ko
# exit

$ lsmod | grep rootkit
rootkit                 10956   0

$ echo hide > /proc/buddyinfo
bash: echo: write error: Operation not permitted
$ lsmod | grep rootkit

$ echo show > /proc/buddyinfo
bash: echo: write error: Operation not permitted
```

❶ In this test, privilege was escalated initially using *su*, which caused OSSEC to log an alert for privilege escalation.

❷ The *insmod* command is used to insert the kernel module and requires root privilege. OSSEC did not notice that a kernel module was inserted.

❸ The inserted module can be seen with the *lsmod* command and does not require root privileges.

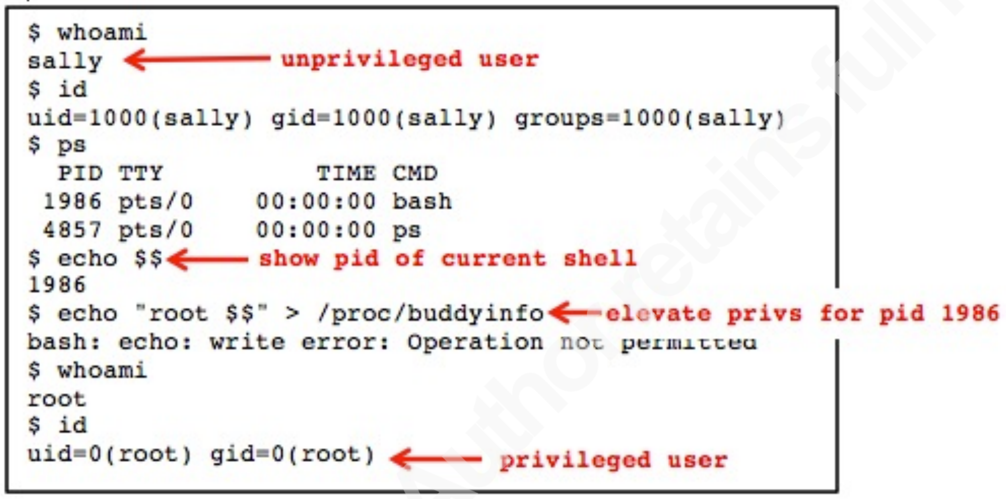
❹ Step 4 hides the kernel module. "buddyinfo" is a standard Linux file found in the /proc directory. It does not have write access for any users, so an attempt to write to it with "echo hide" results in an error but since the rootkit has intercepted the write function for this file it can receive the "echo hide" and interpret it as a command, in this case, a command to hide the kernel module from users (Fontanini, 2011).

❺ "echo show" is interpreted by the rootkit as the command to stop hiding the module; allow it to be seen by anyone.

5.3.2. Privilege escalation

The attacker can achieve escalation of privilege to root by sending the command "root" along with the current shell's process id to /proc/buddyinfo as shown in Figure 19

Figure 19



```
$ whoami
sally  <──────── unprivileged user
$ id
uid=1000(sally) gid=1000(sally) groups=1000(sally)
$ ps
  PID TTY          TIME CMD
 1986 pts/0    00:00:00 bash
 4857 pts/0    00:00:00 ps
$ echo $$<───── show pid of current shell
1986
$ echo "root $$" > /proc/buddyinfo<──elevate privs for pid 1986
bash: echo: write error: Operation not permitted
$ whoami
root
$ id
uid=0(root) gid=0(root)  <──── privileged user
```

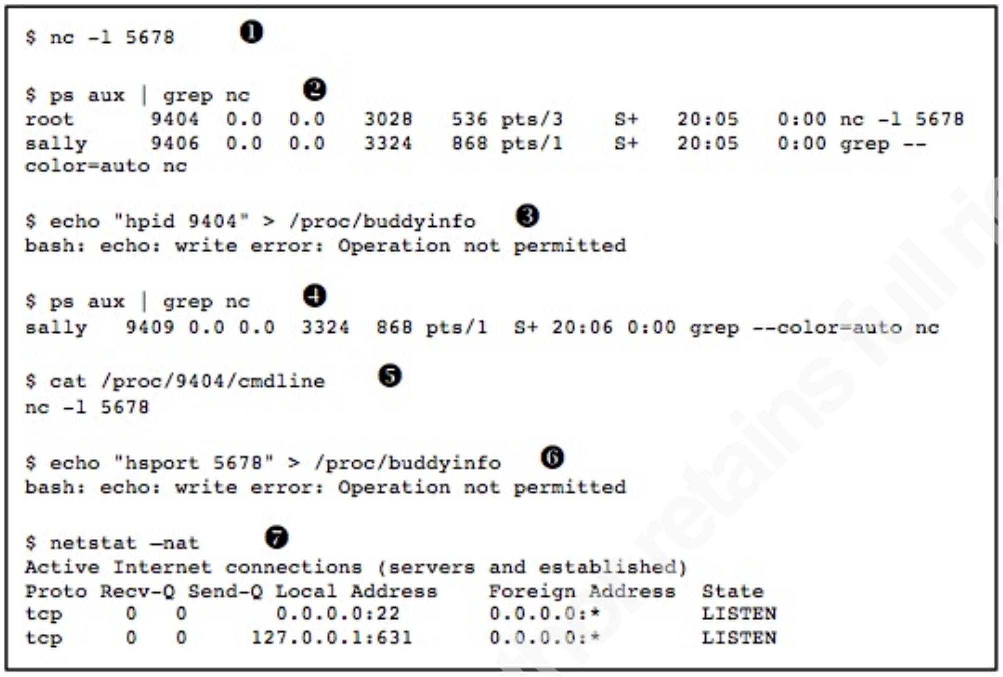
Sending privilege escalation command to rootkit

The user "sally" is an unprivileged user as shown by the UID associated with her account, 1000. The PID of the running shell is 1986 and is stored in the built-in variable "$$". Sending the command "root" along with the PID of the user's shell signals the rootkit to elevate the privileges for this user. This elevation of privileges goes unnoticed by OSSEC in our test system.

5.3.3. Hiding processes and ports

To test the OSSEC's detection of hidden ports and processes, a *netcat* listener was started on port 5678. Both the "nc" process and TCP port 5678 were subsequently hidden using the rootkit commands as illustrated in Figure 20. The hidden process was detected by OSSEC, however, the hidden port was not.

Figure 20



```
$ nc -l 5678       ❶

$ ps aux | grep nc       ❷
root       9404  0.0  0.0   3028   536 pts/3    S+    20:05    0:00 nc -l 5678
sally      9406  0.0  0.0   3324   868 pts/1    S+    20:05    0:00 grep --
color=auto nc

$ echo "hpid 9404" > /proc/buddyinfo       ❸
bash: echo: write error: Operation not permitted

$ ps aux | grep nc       ❹
sally    9409 0.0 0.0  3324  868 pts/1  S+ 20:06 0:00 grep --color=auto nc

$ cat /proc/9404/cmdline       ❺
nc -l 5678

$ echo "hsport 5678" > /proc/buddyinfo       ❻
bash: echo: write error: Operation not permitted

$ netstat –nat       ❼
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address      Foreign Address  State
tcp      0    0       0.0.0.0:22       0.0.0.0:*        LISTEN
tcp      0    0     127.0.0.1:631      0.0.0.0:*        LISTEN
```

Both the process and the port are hidden from the user mode utilities *ps* and *netstat*, however, they can still be found in the /proc directory as shown in Figure 20. Recall from section 5.1 that configuring OSSEC to monitor the /proc directory would be impractical due to the high occurrence of false positives.

❶ The *netcat* listener is started in the elevated shell

❷ The PID of the nc process is 9404

❸ Hide the *nc* process by sending the *hpid* command along with the PID to hide

❹ confirm that the process is hidden from user mode tool, *ps*

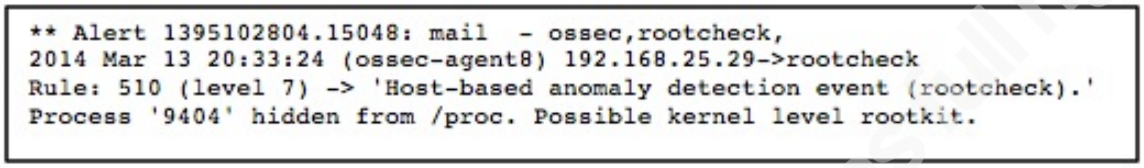
❺ The process still exists and the command line invocation can be viewed if the PID is explicitly given

❻ Hide the listening port to prevent *netstat* or *lsof* from revealing it

❼ *Netstat* does not see the listening port 5678

Figure 21 shows the alert that was logged when OSSEC's *rootcheck* module detected that there may be a hidden process. It detects this because PID 9404 is not found in the output from the *ps* command but it is found in output from one or more of the getsid(), getpid() or kill() system calls.

Figure 21



```
** Alert 1395102804.15048: mail  - ossec,rootcheck,
2014 Mar 13 20:33:24 (ossec-agent8) 192.168.25.29->rootcheck
Rule: 510 (level 7) -> 'Host-based anomaly detection event (rootcheck).'
Process '9404' hidden from /proc. Possible kernel level rootkit.
```

The OSSEC alert when it discovers the hidden nc process

5.3.4. OSSEC detection

OSSEC detected the initial privilege escalation when the rootkit was injected and the *rootcheck* module was able to detect the hidden process, however, it was unable to detect the privilege escalation exploit. OSSEC's *rootcheck* module uses the bind() system call and the output from *netstat* to find hidden ports, however, it was not able to detect the port hidden using this method.

5.4. KBeast

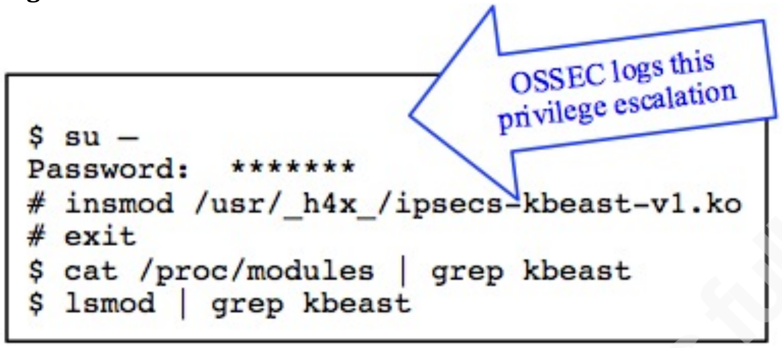
The Kernel Beast rootkit was written by Ipsecs and uses the LKM method of injection. Kbeast hides the loaded kernel module as well as files, processes and ports. In addition, this rootkit offers a password protected backdoor (Ipsecs, 2012). Kbeast was installed on CentOS running Linux kernel version 2.6.18.

5.4.1. Inserting and hiding the kernel module

Like the other rootkits, an initial privilege escalation is required in order to insert the malicious kernel module. In this test it was accomplished with the command "su –" and was detected by OSSEC and logged.

The kernel module was loaded using the *insmod* command and is hidden from /proc/modules as shown in Figure 22. Since *lsmod* uses the /proc/modules output as its source of information about loaded modules it does not see the kbeast kernel module.

Figure 22



```
$ su -
Password:   *******
# insmod /usr/_h4x_/ipsecs-kbeast-v1.ko
# exit
$ cat /proc/modules | grep kbeast
$ lsmod | grep kbeast
```

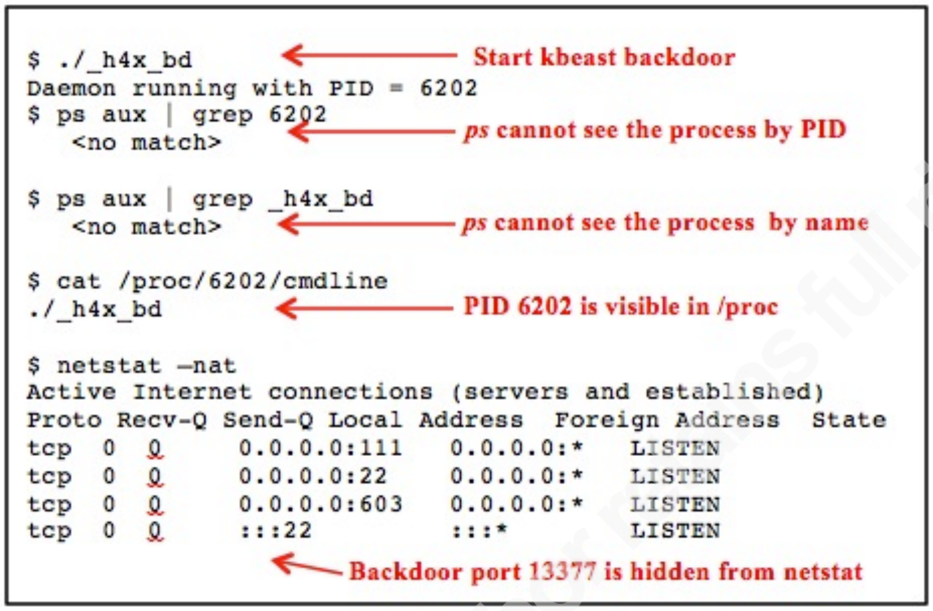
root inserts malicious kernel module and confirms it is hidden from users

The kbeast rootkit has a configuration file that can be edited before compiling. It contains many settings for the rootkit, including the port number that the rootkit will automatically hide, the backdoor password and the prefix to use for hidden files, directories and processes. For this test, the default values were used. The hidden port number is 13377, the backdoor password is "h4x3d", and the special prefix is "_h4x_".

5.4.2. Hiding files process and ports

How the rootkit hides files, processes and ports can be illustrated by setting up the backdoor feature of kbeast. After starting the backdoor, Figure 23 shows that the running process is hidden from *ps* even though the /proc virtual file system contains information about PID 6202, for example, Figure 23 shows the command line invocation for the process. The kbeast backdoor port number was defined in the configuration file as TCP/13377. Figure 23 also shows that netstat does not recognize the listening port 13377.

Figure 23



```
$ ./_h4x_bd
Daemon running with PID = 6202
$ ps aux | grep 6202
   <no match>

$ ps aux | grep _h4x_bd
   <no match>

$ cat /proc/6202/cmdline
./_h4x_bd

$ netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address    Foreign Address   State
tcp   0   0      0.0.0.0:111     0.0.0.0:*     LISTEN
tcp   0   0      0.0.0.0:22      0.0.0.0:*     LISTEN
tcp   0   0      0.0.0.0:603     0.0.0.0:*     LISTEN
tcp   0   0      :::22           :::*          LISTEN
```

Start kbeast backdoor
ps cannot see the process by PID
ps cannot see the process by name
PID 6202 is visible in /proc
Backdoor port 13377 is hidden from netstat

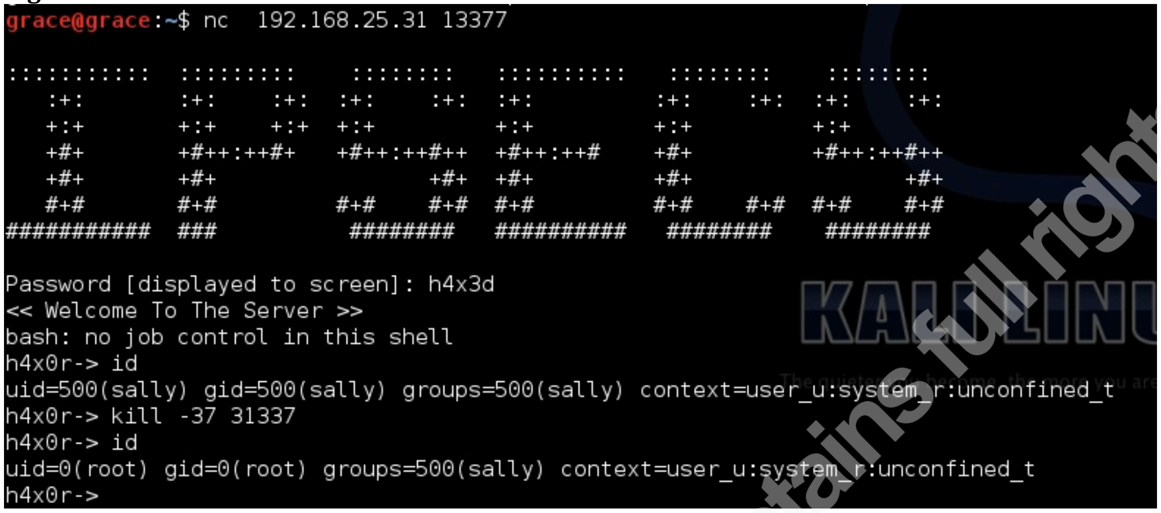
Kbeast's process and port hiding capabilities

As described in section 3.2.2, OSSEC compares the results from the system calls getsid(), getpid() and kill() looking for discrepancies in order to discover hidden processes. The kill() call is actually hooked by the rootkit. This can be discovered in the comments from the rootkits configuration file and confirmed with an experiment. The configuration file shows how to use the kill command to elevate privileges:

```
/* Magic signal & pid for local escalation */
#define _MAGIC_SIG_ 37 //kill signal
#define _MAGIC_PID_ 31337 //kill this pid
```

Indeed, sending the signal 37 to the *kill* command results in a privilege escalation as demonstrated in Figure 24.

Figure 24



```
grace@grace:~$ nc  192.168.25.31 13377

:::::::::::  :::::::::    ::::::::   ::::::::::   ::::::::    ::::::::
   :+:       :+:    :+:  :+:    :+:  :+:         :+:    :+:  :+:    :+:
   +:+       +:+    +:+  +:+         +:+         +:+         +:+
   +#+       +#++:++#+   +#++:++#++  +#++:++#    +#+         +#++:++#++
   +#+       +#+                +#+  +#+         +#+                +#+
   #+#       #+#         #+#    #+#  #+#         #+#    #+#  #+#    #+#
###########  ###          ########   ##########   ########    ########

Password [displayed to screen]: h4x3d
<< Welcome To The Server >>
bash: no job control in this shell
h4x0r-> id
uid=500(sally) gid=500(sally) groups=500(sally) context=user_u:system_r:unconfined_t
h4x0r-> kill -37 31337
h4x0r-> id
uid=0(root) gid=0(root) groups=500(sally) context=user_u:system_r:unconfined_t
h4x0r->
```

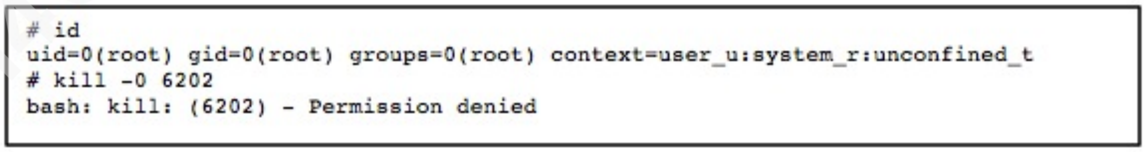
Logging in to the infected system via the kbeast backdoor and elevating privilege to root with kill

Additionally, the kbeast configuration file states that any processes starting with the defined prefix of "_h4x_" will be protected from *kill*:

```
/* All files, dirs, process will be hidden
Protected from deletion & being killed */
#define  _H4X0R_  "_h4x_"
```

Figure 25 illustrates that using the *kill* command as root to query a process ID that we know exists results in an error message. If the kill() system call has been intercepted by the rootkit then it is also possible that OSSEC is receiving incorrect information and is therefore unable to detect the hidden backdoor process.

Figure 25

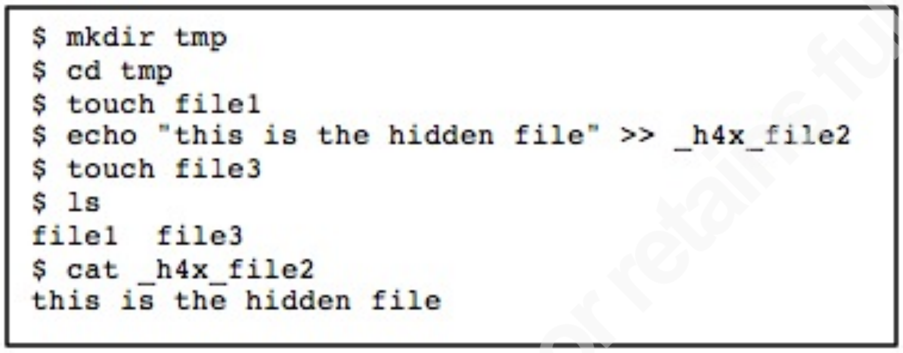


```
# id
uid=0(root) gid=0(root) groups=0(root) context=user_u:system_r:unconfined_t
# kill -0 6202
bash: kill: (6202) - Permission denied
```

Attempting to kill a process protected by the kbeast rootkit

Any files that begin with the special prefix "_h4x_" are successfully omitted from *ps* output, but they are not hidden from OSSEC if the files are in a directory that OSSEC has been told to scan. Figure 26 shows that files with the special prefix are hidden from *ls* although the file exists as demonstrated by the *cat* command.

Figure 26

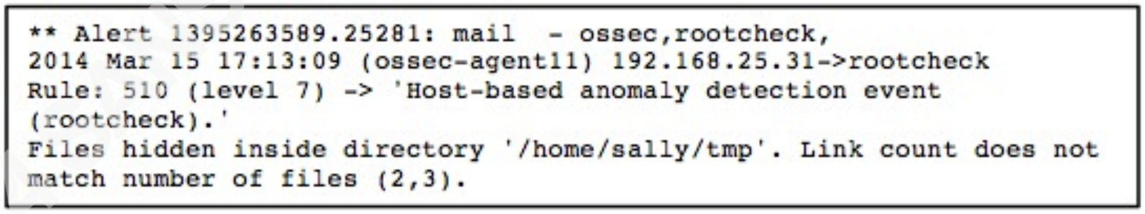


```
$ mkdir tmp
$ cd tmp
$ touch file1
$ echo "this is the hidden file" >> _h4x_file2
$ touch file3
$ ls
file1  file3
$ cat _h4x_file2
this is the hidden file
```

Kbeast hides files that begin with special prefix

Figure 27 shows the OSSEC alert when the *rootcheck* module discovers that the results from stat() and readdir() are not the same. When utilities or system calls report a different number of files for a directory, OSSEC dutifully reports this discrepancy.

Figure 27



```
** Alert 1395263589.25281: mail  - ossec,rootcheck,
2014 Mar 15 17:13:09 (ossec-agent11) 192.168.25.31->rootcheck
Rule: 510 (level 7) -> 'Host-based anomaly detection event
(rootcheck).'
Files hidden inside directory '/home/sally/tmp'. Link count does not
match number of files (2,3).
```

OSSEC notices that a file may be hidden in this directory

5.4.3. OSSEC detection

On the CentOS test system, OSSEC was able to detect the hidden files but only in the directories that it is configured to check. By default, the *syscheck* module examines the /bin, /sbin, /usr/bin, /usr/sbin and /etc directories. OSSEC was not able to detect the hidden process, the hidden ports or the rootkit's privilege escalation using the *kill* command.

6. How Did OSSEC Fare?

After extensive testing done on several different rootkits, OSSEC proves to be an excellent tool for rootkit detection. It will find user mode rootkits quite readily providing it is configured to look in the relevant directories that rootkits would likely hide. The relevant directories may vary among systems but generally include the directories in users' paths. It may be necessary for a system administrator to edit the OSSEC configuration file, ossec.conf, to include the correct directories for each system. OSSEC will also pick up many "hints" when a kernel mode rootkit has infected a system. It accomplishes this primarily with the *rootcheck* module because it is looking at behaviors that many rootkits share such as attempting to hide files but only from some system calls. OSSEC uses multiple methods to find files in a directory and compare the results. Discrepancies in this output are abnormal and OSSEC alerts on that, giving us another hint. Similarly, rootkits attempt to hide running processes but OSSEC can detect it if the attacker has not hidden the process from multiple different system calls. OSSEC was less successful detecting hidden ports.

Since OSSEC is not 100% successful in finding hidden objects, false negatives, (the problem exists on the system, but the scanning tool either does not find it, or does not report it), could occur. For this reason, perhaps OSSEC should not be the only tool utilized by a system administrator for rootkit detection.

One aspect that all rootkits have in common and that OSSEC was always able to detect in the above tests is the initial privilege escalation during installation of the rootkit. Certainly, there are stealthy privilege escalation exploits that could be launched that would evade OSSEC but it is worth noting that rootkits cannot get installed without root access. Tuning OSSEC to alert on as many forms of privilege escalation as possible could be helpful in detecting rootkit infections.

OSSEC is able to give the system administrator many hints that something is amiss with the system, however, it still takes some sleuthing to confirm or deny that a rootkit has been installed.

References

Analiz, (2012). *Linux 2.6 Kernel /proc Rootkit [Software]. Available from http://packetstormsecurity.com.*

Bovet, D. P., Cesati, M. (2006). *Understanding the Linux Kernel.* 3rd ed. Sevastopol, CA: O'Reilly.

Cid, D. (2011) *OSSEC HIDS* (Version 2.7.1) [Computer program]. Available at http://www.ossec.net/files/ossec-hids-2.7.1.tar.gz (Accessed 15 December 2013)

Cid, D. B. (n.d.). *OSSEC*. Retrieved January 5, 2014, from http://www.ossec.net/doc

Corbet, J., Rubini, A. (2005). *Linux device drivers.* 3rd ed. Beijing: O'Reilly.

devik, & sd (Dec 12 2001). *Linux on-the-fly kernel patching without LKM*. retrieved Jan 06 2014, from Phrack magazine Web Site: http://www.phrack.org/issues.html?issue=58&id=7#article

ErrProne, (2012). *Jynx-Kit Release 2 [Software]. Available from http://packetstormsecurity.com.*

Fedora security features matrix. (n.d) retrieved Jan 06 2014, from Fedora wiki Web Site: https://fedoraproject.org/wiki/Security_Features_Matrix

Fontanini, M. (2011). *Average coder Linux rootkit* [Software]. *Available from https://github.com/mfontanini/Programs-Scripts/tree/master/rootkit*

Hay, A., Cid, D. (2008). *OSSEC host-based intrusion detection guide.* Burlington, Mass.: Syngress Pub.

Henderson, B. (2001, Jun). *Introduction to Linux loadable kernel modules*. retrieved Jan 05 2014, from http://www.tldp.org/HOWTO/Module-HOWTO/x73.html

Ipsecs (2012, Jan). *Kernel Beast Linux Rootkit*. [Software]. Available from *http://packetstormsecurity.com/files/108286/KBeast-Kernel-Beast-Linux-Rootkit-2012.html*

Kerrisk, M (2013, April). *Linux system calls*. retrieved Jan 05 2014, from Linux programmer's manual Web Site: http://man7.org/linux/man-pages/man2/syscalls.2.html

Lineberry, A (Mar 27, 2009). *Malicious code injection via /dev/mem*. retrieved Jan 06 2014, from Blackhat Web Site: http://www.blackhat.com/presentations/bh-europe-09/Lineberry/BlackHat-Europe-2009-Lineberry-code-injection-via-dev-mem.pdf

Mcclure, S., Scambray, J., Kurtz, G. (2012). *Hacking Exposed 6: Network Security Secrets and Solutions.* 6th ed. San Francisco: McGraw-Hill Osborne.

Nguyen, B. (2004, July 30). *Linux filesystem hierarchy*. Retrieved January 4, 2014, from http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html

Perla, E., Massimiliano, O. (2010). *A guide to kernel exploitation: attacking the core.* Burlington, MA: Syngress.

Skoudis, E., Zeltser, L. (2003). *Malware: Fighting Malicious Code.* E Rutherford: Prentice Hall PTR.

Tanenbaum, A. S.. (2008). *Modern operating systems.* 3rd ed. Upper Saddle River, N.J.: Pearson Prentice Hall.

Terrehon, B., & Bauer, B. ((1999, October 7)1999, October 7). *The /proc filesystem*. Retrieved from https://www.kernel.org/doc/Documentation/filesystems/proc.txt

Ubuntu features. (n.d) retrieved Jan 06 2014, from Ubuntu wiki Web Site: https://wiki.ubuntu.com/Security/Features#dev-kmem

Loadable_modules. (2014, March 11). Retrieved from https://help.ubuntu.com/community/Loadable_Modules

Wheeler, D. (2000). *Program Library HOWTO*. retrieved Mar 2 2014, from http://tldp.org/HOWTO/Program-Library-HOWTO/introduction.html