

Linux Loadable Kernel Module Based Rootkit Detector

Andrew Stone

Implementation Details

Basic Architecture

The basic architecture for the rootkit detector remains as before. A loadable kernel module contains code where a loop is made through the task list identifying all of the processes running on the system, their PIDs, and all associated open files and sockets. In addition to this, any memory mapped files, including shared libraries, that exist in the process address space of every process running on the system are identified. Originally this was not part of the architecture, but was deemed necessary for a successful product.

The data recovered from the module is then written to a file which is compared to the output of lsof in a python script. Any anomalies will be written to the console. Originally I was going to compare the output of the module to ps, netstat, and lsof, but realized that lsof contained all the info I needed (shared libraries, PIDs, sockets etc...). Furthermore, I only need lsof to detect kernel mode rootkits, which is the real purpose of the detector. Many other systems can easily detect user mode rootkits. By just running the same programs off a write-proof cd, one can prove they have been trojaned. Furthermore, if user-mode rootkits don't trojan lsof, my program will not detect them. However, by simply running lsof and comparing the output to ps and netstat one will notice the difference. Simple functions may also be added to my python script to check for any number of trojaned binaries, including ps and netstat, but I chose not to focus on that and to focus more on gathering as much info from the kernel as possible.

The file **rk_detect.c** contains the loadable kernel module code. The python script is named **rk_checker.py**

The following information is checked by the system for being hidden:

Processes and Threads (PID)

Regular Files that are currently being accessed by running processes

Internet Domain Sockets (TCP/UDP)

Memory Mapped Files including Shared Libraries

The next subsections will detail the implementation of the kernel module routines which recover the information in the previous list.

Processes and Threads

Since all threads have separate PIDs in the Linux kernel and are scheduled independently, they will henceforth be referred to as processes.

Discovering all process names and PIDs for every process in the Linux kernel is fairly easy. There is a macro **for_each_process(task)**, that loops through every single process on the system. From there the module can retrieve the **struct task_struct* task** which corresponds to a given process. It then access

the members of the struct which correspond to the name of the process (**task->comm**), and to the PID (**task->pid**).

Regular Files and Internet Domain Sockets

In each **struct task_struct** there exists a **files (task->files)** member which is a structure containing an array of all open file descriptors for that process. These open file descriptors may represent Regular Files, Sockets, Device Special Files, or FIFOs. For each of these open file descriptors we can retrieve an in-memory inode for the given file and check whether it is a regular file with the **!special_file()** macro, a socket with the **S_ISSOCK()** macro, or any other type of file with various macros. However the rootkit detection module only deals with regular files and sockets for now.

From here the **d_path** function of the VFS subsystem of the kernel can be called on the directory entry (dentry) corresponding to the given inode of a regular file and will return the full path name of the given file.

To determine which sockets and what types are open we need to do a little more work.

There is a **SOCKET_I()** macro which extracts a **struct socket* sock** from a given inode. Then there is a **struct sock* sk** member of **sock**, which holds the domain specific socket information for the given socket. A **struct inet_sock* inet** can be extracted from **sk** by the function **inet_sk(sk)**. This holds Internet socket specific information.

The following information is extracted from these structures:

sock->ops->family gives the network family of the socket (**AF_UNIX, AF_INET, etc...**).

sock->type tells whether the socket is stream or dgram

sk->sk_state tells whether the socket is Listening or Established, etc...

inet->saddr gives the source address of the socket

inet->daddr gives the destination address

inet->sport gives the source port

inet->dport gives the destination port

Memory Mapped Files

Memory mapped files do not show up as open files with file descriptors in the **task->files** structure.

The only way to discover what they are is to look at the virtual memory mapping of the process address space.

The process address space is represented by the memory descriptor, **struct mm_struct* mm**. In each process address space, there are multiple memory areas of type **struct vma_area_struct**. Each area is a continuous range of addresses that have common permissions and attributes. Some areas specifically map to files. These are the memory mapped files.

The module loops through all of the memory areas in each process by accessing a linked list in the memory descriptor, namely **mm->mmap**. Within each memory area there is a **struct file* vm_file**

which if not NULL gives a file descriptor corresponding to the memory area, aka a memory mapped file. We then dereference the **dentry** from the file descriptor and use **d_path** to retrieve the full file path of the memory mapped file. Note once again that this file descriptor does not reside in the open file descriptor list of the task structures files object.

Python Script

The python script, **rk_checker.py**, is a simple script that first scans the the output file from the module and creates a list of data to be checked against lsof. It then executes **lsof -nP** and parses the piped output into a list with identical structure. The two lists are then compared and any anomalies are printed to the console.

One interesting fact is that some memory mapped temporary system files really screw things up. They all start with a **/SYSV**. They are currently ignored. This is a security hole that needs to be plugged.

Performance and Evaluation

The procedure used to Evaluate my program was as follows:

Clean System Run

- 1)load module: **insmod rk_detect.ko**
- 2)run **./rk_checker.py**
- 3)unload module: **rmmod rk_detect**

Output is baselined. A weird error happened where by 3 files utilized by Xorg are not recognized and are printed to the screen. Also insmod used to load the system is not recognized because it is finished by the time lsof runs in the checker script, but is the process in which the module actually performs its duties. Obviously any malicious activity that occurred between the loading of the module and running of the script cannot be observed. These will be shown in the sample files.

Rootkit System Run

- 1)Install adore-ng rootkit for 2.6 kernel: **insmod adore-ng-2.6.ko**
- 2)start ssh session to nunki: **ssh -l andrewjs nunki.usc.edu**
- 3)run **ps ax** to retrieve the pid of the ssh session
- 4)run the shell associated with adore-ng to hide the process id of the ssh session: **./ava -i PID**
- 5)run **ps ax | grep PID** and **lsof | grep PID** to make sure the process is hidden
- 6)load detection module: **insmod rk_detect.ko**
- 7)run **./rk_checker.py**
- 8)unload module: **rmmod rk_detect**

Output will show the process id of the hidden process along with all of its associated files, sockets, and memory mapped files. The items in the baseline will still appear.

An interesting fact here is that only hidden processes show up, the adore rootkit itself remains hidden! Thus it is only detectable if it is actively being used to hide processes. This occurs, because it is a kernel module, and no checking is done to see if kernel modules are hidden. A simple way to get

around this is to build a database of allowable kernel module signatures and have the kernel authenticate each one at load time, thus adore cannot get loaded. The author even admits that this works.

Another thing adore does is that it can hide files. This is undetectable by the detection module unless they are in use.

Finally, the rootkit detection module runs in the insmod process. It would have been relatively easy to load it and have it export syscalls or ioctls so that a user could invoke it at will, however this could allow rootkits to be aware of its presence and shield themselves. However, if kernel preemption is turned off it will run on load in its initialization function and the rootkits will never get a chance to see it before it is too late. Even if kernel preemption is on, the chances of a rootkit having time to run and hide itself is very slim.