Omar Sherif Farag 34-2656 David Onsy Shokry 34-4658

Karim Willian 34-6466 George Ashraf 34-864

Security Project Report

The project of our choice is implementing a Linux rootkit. A 'rootkit' is typically a malicious loadable kernel module designed to hide certain activities from the administrator of a system, or a network of systems. The complexity of rootkits arises not from the difficulty of their implementation, but rather from the difficulty to detect their presence as they could easily mislead the software intended to discover their presence.

The motivation behind this project is to implement a successful rootkit for Linux in order to show how easy it can be to exploit the Linux OS and compromise the system. Also to open the door for potential future fixes for these system security holes.

The processes in user space in Linux call a variety of interrupts to kernel space. Those interrupts are stored in predefined table, IDT or interrupt descriptor table while initialization process in Linux. It stores the registered function pointers to handle diverse interrupts. The "system call table" contains the data structure to process system calls when calling interrupt 0x80 in Linux. The idea is to manipulate the function pointers in system call table with hijacking in order to insert desired actions. The original system call has been stored somewhere to avoid unwanted system crash.

At the beginning of our implementation, we were faced by a decision. Either to modify the Kernel structures or intercept Linux system calls, modify their results, and return the modified results to the user. In most cases, the latter was preferred because it is a more robust and cleaner way of accomplishing the task. The first problem we encountered was finding the System Call Table. The System Call Table is no longer exported in modern Kernels, so its location will have to be obtained manually. Our approach is to search the memory for the system call table. A simple brute force search for the system call table returns the correct address for the beginning of the table. After retrieving the location of the table in the memory, we are faced by yet another problem. The memory page in which the system call table is located is marked as Read-only. In order to make it writable we need to make sure that the Write-Protect bit in cr0 is disabled. 'cr0' is a control register in the Intel architecture that contains a flag called WP on bit 16. When this flag is set to 1 any memory page that is set read-only cannot be changed to be writable, so we need to change this flag back to 0 before we can make the system call table writable again.

The main feature of the rootkit is the ability to grant the attacker root privileges which essentially opens a window of endless possibilities to the attacker. A process Identifier (PID) is a number used by most operating systems kernels in order to temporarily uniquely identify a process. On Unix systems, the process with PID 1 is the init process, primarily responsible for starting up and shutting down the system, and of course possesses root privileges. Promoting a certain process to root is done by substituting the credentials for that process with the credentials of the init process.

A feature which is of crucial importance is the ability to hide the rootkit, otherwise, a clever administrator could easily spot the existence of the module through something as simple as running `lsmod' and therefore, we incorporated stealth mode within our rootkit. In order to achieve stealth, we need to remove our module from the list of modules. The module was

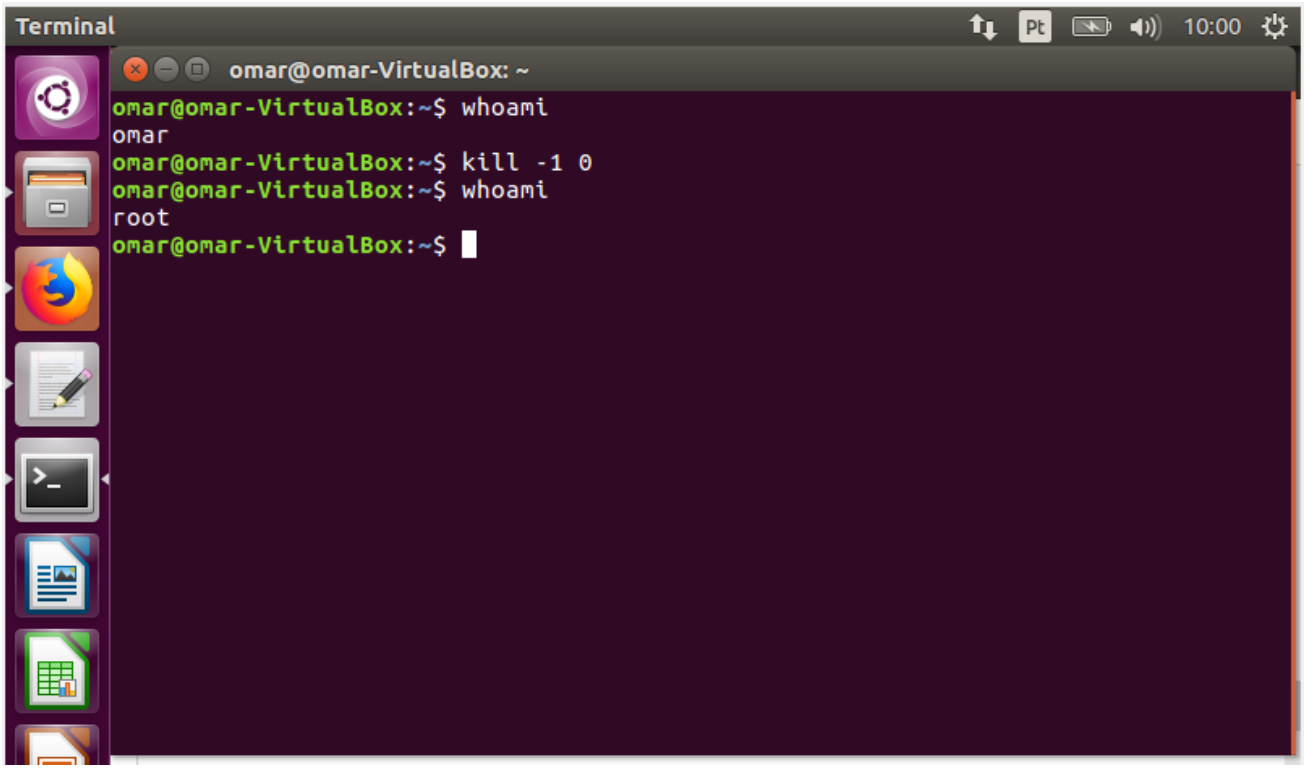
removed from the list of modules by simply invoking `list del' which given an element in a list, deletes it from the list. On the other hand, showing the rootkit simply dictates that we re-insert the module in both lists through invoking list add.

Processes are usually listed in Unix through invoking the `ps' command or the like. Such commands share the same basic infra-structure, which is the getdents system call. Such a system call reads several linux dirent structures from the directory referred to by the open file descriptor. Therefore, one of the easiest ways to hide our process would be to modify the system call table such that the entry for the getdents function actually points to our custom-made function.

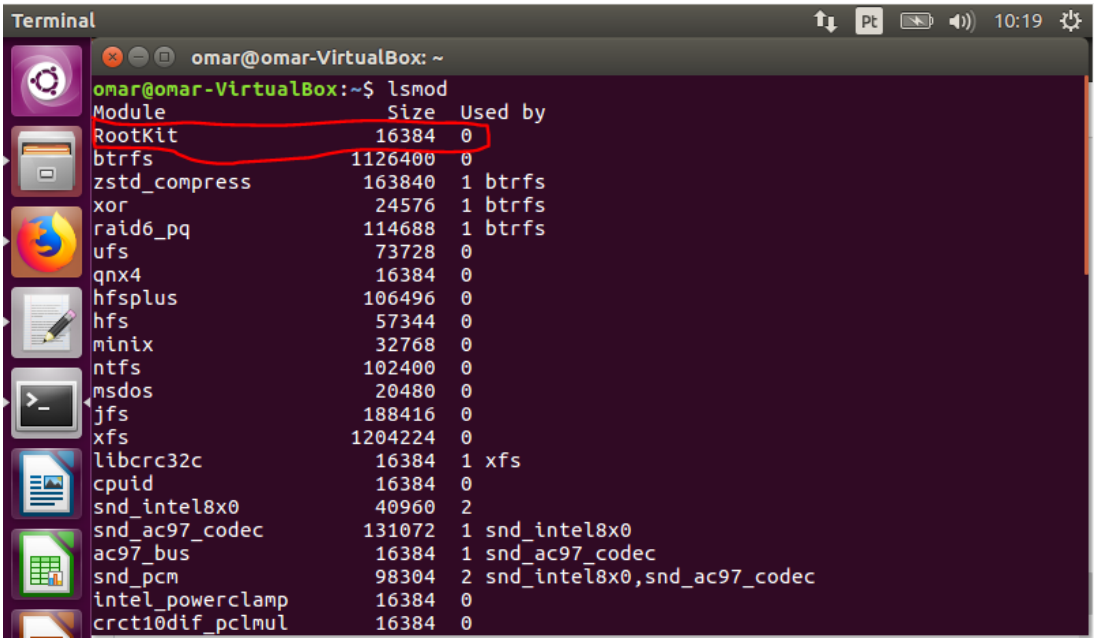
In our approach, we created a custom function which mimics the ‘kill’ command and we made the system call table point to it instead of the original ‘kill’ function. Of course we saved the original function in order to restore it later. whenever a ‘kill’ command is issued, we check for the signal appended in the command. Depending on the signal, we perform one of the 3 features in our rootkit (give root, hide module, hide process). We define these signals in our code. For example, ‘kill -0 1’ means to hide the process with PID 1. ‘-0’ is for (un)hiding process, ‘-1’ is for giving root and ‘-2’ is for (un)hiding the module.

When the module is removed, we bring back the pointers of the original functions. Specifically the getdents and the kill system calls. Also the Cr0 bit was modified accordingly before and after writing the pointers of the custom function to the system call table.

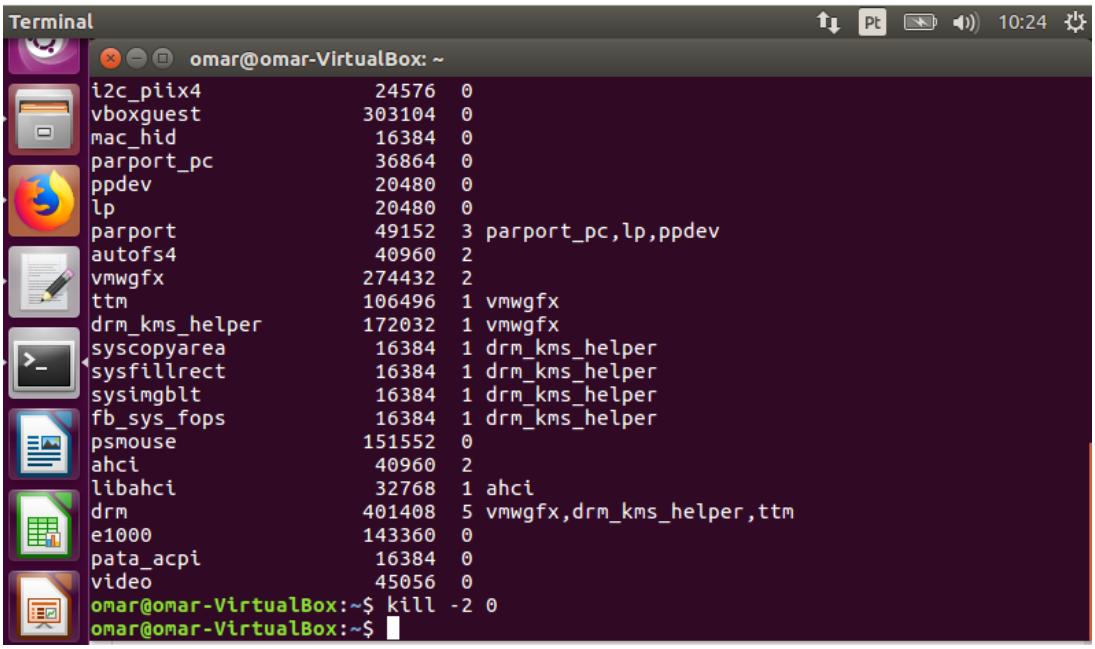
The following is an example of using the rootkit module in order to gain root access. We first use the ‘kill -1 0’ command in order to signal to the rootkit that we want to hijack the root privileges . ‘-1’ indicates the action and the ‘0’ is the PID, which can be any PID in this example since the command doesn’t require a specific process. (so we can insert any PID, its irrelevant). As we can see the system call table calls our own ‘kill’ function which receives the ‘-1’ signal and substitutes the credentials for that process with the credentials of the init process, granting root access.



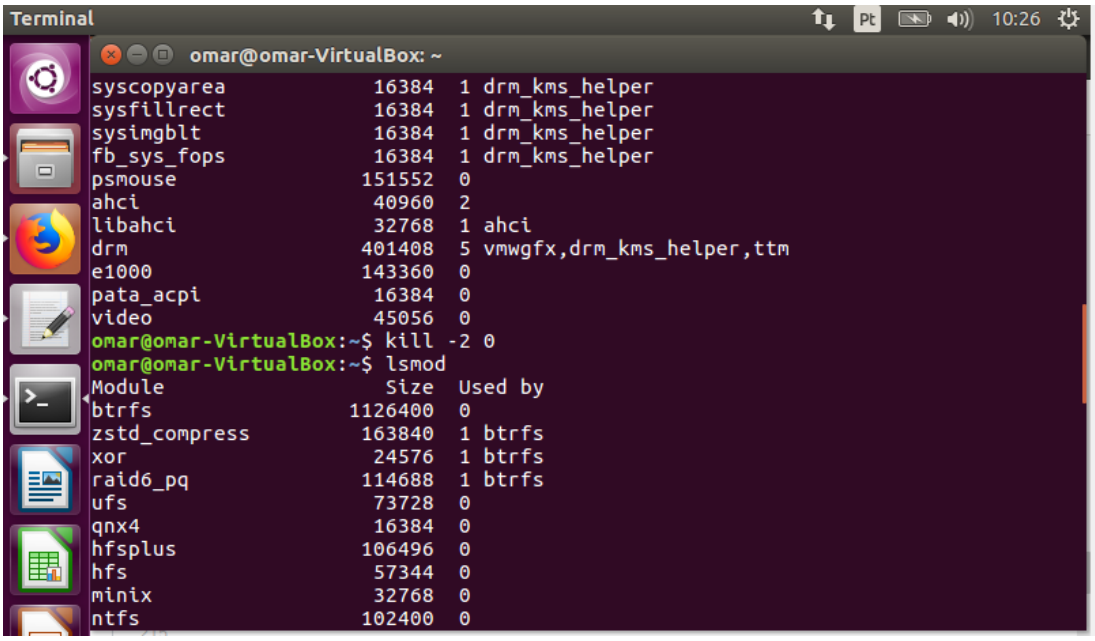
The following image shows how the same procedure could be done but this time to hide the module from the ‘modls’ list, in order to make the module invisible.



As we can see, a simple ‘lsmod’ shows all the modules running In the background , including our rootkit. So now we will signal to our custom ‘kill’ function to hide the module using the ‘–2’ signal, which essentially uses the “list_del” function to erase the module from the list. (We can use any PID number, as it is irrelevant)

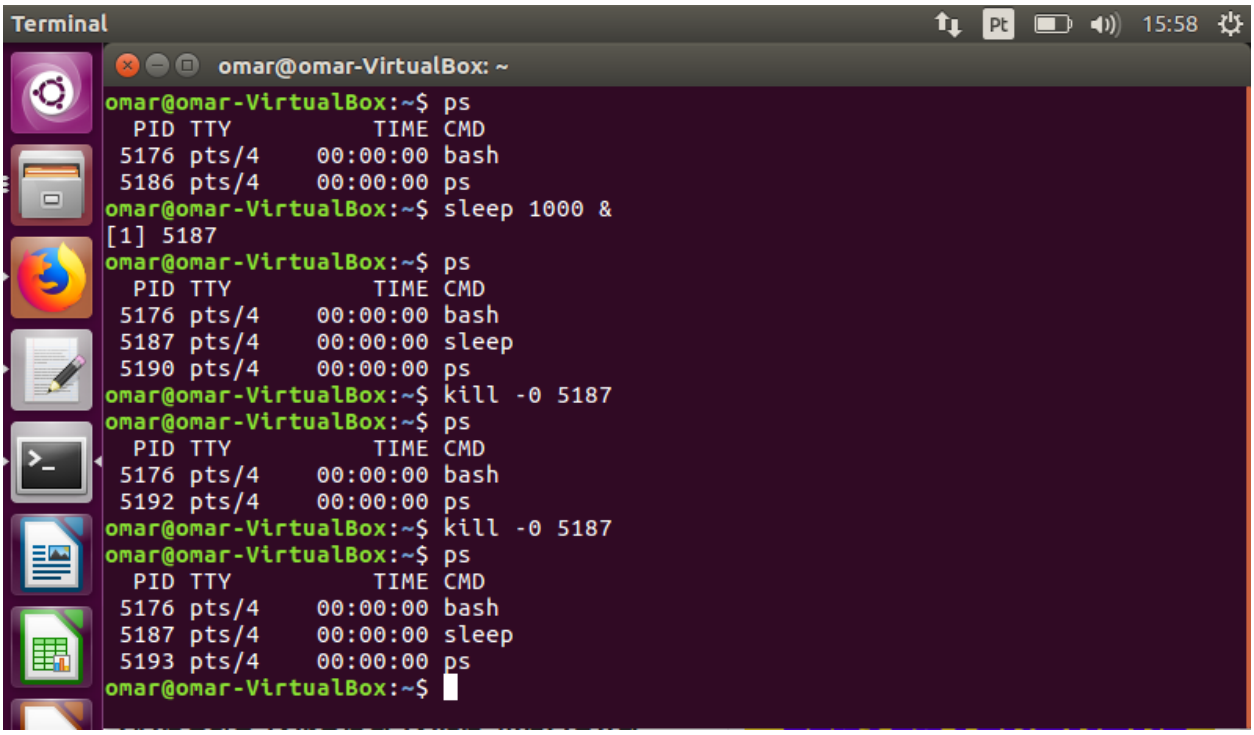


And now the rootkit module does not appear on the list anymore.



We also removed the module from the memory as soon as it is initialized so it becomes hard for scanning software to detect our rootkit.

Finally, for our last feature. We try to hide a process from the ‘ps’ list. First, we create a dummy process using the command “sleep 60000 &” which creates a process for an amount of seconds. Then we look for that process and find its PID. We issue a command to our module “kill -0 5187”. The ‘-0’ tells our module that we want to hide a process and “5187” is the PID of the process we want to hide(the one we just created). As we can see, when we try to look for the process again, it is invisible.



It is worth mentioning that in order to unhide the module or a process, we just have to call the method again.(toggle feature)

References:

1. https://en.wikipedia.org/wiki/Rootkit
2. http://dandylife.net/blog/archives/304

3. https://github.com/nurupo/rootkit
4. https://github.com/f0rb1dd3n/Reptile
5. https://github.com/m0nad/Diamorphine
6. http://lwn.net/Kernel/LDD3/
7. http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/
8. http://turbochaos.blogspot.com/2013/09/linux-rootkits-101-1-of-3.html
9. https://yassine.tioual.com/index.php/2017/01/10/hiding-processes-for-fun-and-profit/
10. https://www.howtogeek.com/107217/how-to-manage-processes-from-the-linux-terminal-10-commands-you-need-to-know/