CSEN 1001 Computer and Network Security

Project Report

Linux Rootkit

Ahmed Mahmoud Hathout 34-9785

Ahmed Tarek 34-2376

Gehad AbdElhalim 34-14711

Amr Ayman Elsayed Khalil 34-7026

Islam Mohamed Hamada 34-13150

1 Motivation

We chose to implement a **Linux Rootkit** for the project. We just find the Linux Kernel interesting and we wanted to learn more about how the kernel works and how challenging it'd be to design a malware for such a secure system.

2 Summary

In brief, we implemented a kernel module that is hidden, can't be removed after insertion, can hide a chosen process and hide its an open socket from the user. Also, we use another process that enables ssh communication and saves the attacker public key so the attacker can access the victim's files and manipulate them by having full control over the victim's filesystem.

3 Features

We implemented a number of features inside the module itself and for the ssh communication which we'll discuss one by one.

4 The Kernel Module

First, we had to understand what a kernel module is. A kernel module is some snippets of code that can loaded an unloaded when needed without the need to restart the system. They can extend a functionality, modify it, stop it or just **corrupt** it as in our case. By default when someone boots their linux

OS a number of modules are loaded at startup and they handle all basic functionalities utilized by the OS. For example, there are kernel modules that handle how a file is open, read and modified and how a folder is open and iterated to get all files inside, Also how the OS deals with interrupts and different input devices such as the networks card, the mouse and the keyboard. One can easily see the list of loaded modules using the following command: ”`lsmod`”.

The module has many features as mentioned before which I’m going to discuss how we implemented them.

4.1 The Kernel Version

Modules are written in different ways according to the kernel version used because implementation details change over time and also data structures that’s why one is restricted by the version used and code aren’t compatible across different kernel versions. First, we decided to start working on a virtual machine for safety and not to damage our current running version. Also, it needs a lot of restarting which can damage the system for unsynched data between caches and actual memory. So we chose Lubuntu trusty to work on with a kernel version **3.19** which introduced some difficulties because Hijacking system calls got harder since **2.6.x** versions and generally a lot of *structs* and mechanisms changed over time.

4.2 Hiding The Module

As I mentioned before, modules can be listed using simple commands, hence can be removed if discovered. So the first task was to hide the module from being discovered. There's a simple way of doing it which is using this command "$list\_del(\&THIS\_MODULE->list)$;" which ensures the module is running but not detectable at the same time even using "`lsmod`".

4.3 How to Manipulate The Kernel

First, we need to set a target function to modify and corrupt. For example, if we want to change the behavior of some command such as "`cd`", we need first to know the system calls used by that command and then modify those specific system calls to meet our desires. To know those system calls we can use the command "`strace`". The next sections will show examples that we implemented.

4.4 Hiding Process

To understand how a process is hidden, we need to understand first how to know the running processes. To know the running processes, we can simply use "`ps aux`" which will list all running processes of the system. To know the target system calls we can run "`strace ps aux`" which would list a huge sequence of commands but most importantly there's this specific line "`getdents(...,/proc,...)`". *getdents* is a system call that list the contents of a folder. That's exactly the system call needed because linux stores folders inside /proc folder where each folder represents a running process, and the folder is named according the process id (*pid*). So to list the processes we need to modify the function that reads the processes inside */proc* and somehow skip the folder (*process*) if its *pid* matches the process we'd like to hide. $getdents() \xrightarrow{calls} iterate\_dir \xrightarrow{calls} iterate \xrightarrow{calls} filldir\_t$. filldir is a function that outputs the results to the user, so we simply modify it that whenever the folder name is the *pid* we need to hide we return 0.

4.5 Hiding Socket

In a similar fashion to the previous one, we need to know how the OS gets the open sockets used in our case (TCP socket). First, we need to know the command used to list the sockets which is "$netstat - l$". Then, examine the output of "$stracenetstat - l$". We'll find that among the systems calls used there are 2 calls:

1. *open*("*/proc/net/tcp*", ..., ...) where "*/proc/net/tcp*" is the path where linux stores the tcp sockets. Note: it's sometimes *openat* instead *open* which does a similar function.

2. *read*(..., ..., ...)

the two calls: *open* and *read* are always called consecutively whenever a file is read. Typically, we call *open* on the file path then it returns a pointer that we pass to the system call *read* and then we are ready to read the file. So, we need to do the following: whenever we open the file "*/proc/net/tcp*" we need to make sure that the line recording our socket not to be returned to the user. However, after many trials and search, we couldn't find a way that doesn't lead to crashing or undefined behavior, so for simplicity we hide all tcp sockets including ours. That's done by simply returning 0 inside the *read* call whenever the last file opened, using *open*, is "*/proc/net/tcp*".
In this case, we need to modify two system calls, and modifying system calls is more complicated than modifying normal calls. Note that in the

previous section, we were trying to modify some functions used inside the system call *getdents*() not the system call itself. But here, we need to modify the system calls themselves. To do so, we need to know where they are stored inside the memory as there are no variable we can use to get those functions. To get their address in the memory, we need to get the address of the "*system_call_table*" where all system calls are stored inside.

4.5.1 System Call Table

In older versions of linux you could simply call some variable to get the system call table within the code itself. It's no more possible in newer versions including our version. But, there's a way to get the address as it's stored in the file `/boot/System.map-$kernel_version` so we can grep the line containing the address of "*sys_call_table*" and then feed it as an input to the module then we typecast that address as a pointer to an array which contains all the system calls. Then we can get the respective address of a system call in the following way: To get the address of *open* we use $sys\_call\_table[\_\_NR\_open]$, similarly for read: $sys\_call\_table[\_\_NR\_read]$ and so on. The __NR_*some_sys_call* is predefined inside the kernel space. So, we only need to modify the value stored at $sys\_call\_table[\_\_NR\_open]$ which is originally the address of the original system call to an address of our fake system call and also for $sys\_call\_table[\_\_NR\_read]$. But there's another problem, the system call table is stored in a read-only area of the memory which means that whenever we try to override a system call address by ours, the module would crash and a page fault will happen. However, this read-only restriction is enforced by some register called cr. So, we change the value of the register so that we can modify read-only addresses, then we can modify the system call table, and finally we can reset the register value to its original not to cause major problems and that's it !!!!!.

4.6 Cleaning

Every module has two functions: *init* and *exit*.

- *init* is used to define the module purpose and write whatever changes the module should achieve and is called whenever the module is loaded.
- *exit* is called whenever the module is unloaded. It's usually used to clean up after. In some case, we can use it to remove our modifications and get the system back as it was to use the original functions and system calls. **Note: in our case, it wouldn't be useful to clean after because we hide the module which makes it impossible to unload the module, except for restarting the machine of course**.

4.7 Refrences

https://info.fs.tum.de/images/2/21/2011-01-19-kernel-hacking.pdf

https://memset.wordpress.com/2010/12/28/syscall-hijacking-simple-rootkit-kernel-2-6-x/

https://yassine.tioual.com/index.php/2017/01/10/hiding-processes-for-fun-and-profit/

https://syscalls.kernelgrok.com/

https://elixir.bootlin.com/linux/v3.19/ident/linux

5 Remote Access

We wanted to give the attacker remote access to the victim's machine. The simplest way and probably the best one is to use SSH.

5.1 What is SSH

SSH stands for Secure Shell. It enables the the user to log into a remote machine with his username on that remote machine. The user can then transfer files or execute commands on that remote machine as if they were using it directly.

SSH can authenticate users either by passwords or SSH keys. SSH keys are public and private key pairs and can be generated by a number of cryptographic algorithms like RSA, DSA, or ECDSA. If a user wants to connect to a remote machine using ssh keys, their public key must be saved in some file on that remote machine. The user in that case will not be prompted to enter any password.

5.2 Design

As soon as the rootkit runs on the victim's machine, it saves the attacker's public key, downloads and runs the ssh daemon and sends the username and the IP of the victim to the attacker. The attacker can then use `ssh victim_username@victim_ip` and they are now logged in with the victim's username. if the victim is root then the attacker will also be root.

5.3 Implementation

Python was used as it has pretty good native libraries that were needed and it is much easier to grasp and implement with than C

SSH was used with RSA keys since it is the one of the most common remote access methods to Unix servers and it is the most secure way.

All the libraries used are native python libraries

Two scripts of course need to be implemented; one will be running on the victim's machine and the other on the attacker's. The one on the victim's is a standalone program and actually can be run without sudo privilege unlike the rootkit module (though it would need some luck to work as we will explain).

The attacker's script is very simple. It just waits for the rootkit to send it the username of the victim and then it displays the username and the IP of the victim (which it got when the connection was established) to the attacker. This is easily implemented using socket API. So the attacker script is a server waiting to get a message from the rootkit and when it gets the message it displays it and terminates

Most of the work is done on the victim's machine. The script adds the public key of the attacker to `~/.ssh/authorized_keys`. If the victim is root then `~` is `/root`. If not then it is `/home/victim_username`.

The script then downloads and runs the ssh daemon. This is done using the subprocess Python module which creates a Unix process given its command. After that, the rootkit gets the username and sends it to the attacker using sockets.

Note that if the victim does not have root access, there will be no way to download and run the ssh daemon. The script will not fail in that case and will still send the username to the attacker. If the ssh daemon were running on the victim machine anyway then the attacker will be able to ssh into it. That is why it is possible for this program to work without sudo privilege. If ssh daemon was not running then it is impossible for the attacker to log into the victim's machine.

5.4 References

The only thing that needed research was SSH and this is a very good article about it. https://www.digitalocean.com/community/tutorials/ssh-essentials-working-with-ssh-servers-clients-and-keys

6 Vulnerabilities

There are a couple of things that may imply that there is a rootkit installed on the machine.

6.1 ssh daemon process

This is the process that allows SSH connections. This process can be hidden easily like the dummy process we showed but unfortunately we forgot about it and noticed it a bit late :-).

6.2 authorized_keys

The victim may notice a new entry in his authorized_keys file. This is very unlikely of course since no one checks this file out anyway (unless it was created by the rootkit. It would be very obvious).

6.3 insmod

This one is harder than guessing a password. If the victim tries to insert a kernel module that has the same name as the rootkit module name, then he will see a message saying that this module is already inserted. If the rootkit name is just some random characters then it is nearly impossible to guess it