Lab 7: Kernel Rootkits ***Darshan K (A20446137)***

Task 1 : Understand the code. You should realize that the kernel module's entry point (b4rn_init()) is invoked after the module is loaded by the kernel (e.g. by insmod or modprobe). Start there and read comments carefully.

The code once cloned has 3 files. The C file, which only works on linux hides the malicious library, and resident due to code injection, it hides itself from the filesystem and from the /proc/modules and implements a local backdoor. Because of the total control of it on what is exposed to userspace, it is more effective. If it can be subverted, information presented to the user can be manipulated.

As per the norms of any API, which has to have an **init** and **exit** functions defined in it, the C file provided in the git repo too has these functions defined in them. The first and the last functions, that is the entry and the exit points are defined by **module_init(b4rn_init)** and **module_exit(b4rn_deinit).**

Task 2 : The Backdoor:

Could an attacker use the backdoor exposed by the rootkit to remotely get access? Explain why or why not?

Accessing the machine remotely requires special softwares or permissions. But the backdoor is only used to grant permissions to the processes. When this Rootkit is installed as a kernel module, it can open the backdoor TCP ports and by making use of the reverse shell, the attackers gain remote access to the machine.

Realise that this is a pretty rudimentary backdoor. There are certainly more stealthy ways to do this. Can you think of any?

The aim for us at the moment is to make rootkit run any process it wants to when the OS is run. For that to be achieved we want the rootkit to reside in the master boot loader. This can be done instead of adding the device file. And for the rootkit to obscure the information about the bind shell, it has to run it in the background as a daemon process. This can be done once the attacker gains complete access to the system. And we can also make machines communicate with the attacker to get a reverse shell.

Task 3 : Hiding in Plain Sight:

Explain why we must (1) use function pointers and kallsyms_*() functions to call certain routines and (2) manipulate cr0 and page protections to install our function overrides.?

(1) Calling **kallsyms_*()** is a standard way of coding loadable kernel modules and a way for the rootkit to use routines that live inside the kernel. For this to be achieved, it has to call **kallsyms_*() API** with the symbol name to get the pointer of that function. Kernel functions symbols are unresolved when we create our module so we must call **kallsyms_*()** to dereference the pointers.

(2) Only be first removing the protection as the concerned memory locations have no write access. For hiding, If the kernel tries to modify the read handler function, and exception will be raised as we need to modify the read handler function but the kernel wont allow us to do so. To overcome this we need to revoke the write access, override it and grant the write access to it. Hence there is a write protection bit present at cr0.