Linux Rootkit

Michael Sherif Kamel 34-12239 T-15
Karl Maged Fawzy 34-484 T-10
Ahmed Hossam Mostafa 34-722 T-13
Mohamed Maged Mohamed 34-6012 T-10
Carol Emad Fekry 34-8145 T-13

Contents:

1. Introduction:

Our project is concerned with engineering a linux rootkit. In general, a rootkit is a set of tools inserted by an intruder into a computer to allow later use for malicious purposes, without being detected. The main feature of a rootkit is its ability to hide its presence from system administrator and to continue providing system access to an attacker. A basic way in which rootkits make themselves extremely difficult to detect is by replacing several standard system utilities with modified versions. Rootkits can be classified into three categories; application, kernel, BIOS rootkits.
We are only concerned with the second type.We implemented a Linux kernel module (LKM) to act as a rootkit. A LKM are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.

2. Entry Point

`rootkit_init`:
This first calls **`hide_proc`**, which hides our module from **/proc/modules**, and remove root kit's system filesystem entry.

We then call **`hook_func`** and **`procfs_entry_init`**, if the process is neither, we don't return it.

1) **`hook_func`**: mounts the **/proc**, retrieves all fops (file operations struct), and adds an entry (to **`hooked_functions_listhead`**) for replacement iterate operation function with our replacement function. This is done in the hookedlist_append function call. Our replacement function is called **`filesystem_procfs_hook_iterate`**. This first gets actor function from dir_context, and hooks a parasite (**`dup_procfs_filldir`**) to actor function . In other words, It uninstalls original functions from hooklist (retrieves original functions), and then installs parasite. Finally, it restores original

function.
Injecting a parasite, depending on the flag `(install/uninstall)`, either replaces target function prologue bytes with root kits's parasite, or replaces target function entry (where our parasite had been installed) with original function (restore target function's prologue bytes), respectively.

2) **`procfs_entry_init:`** creates proc entry with predefined read and write functions, for communication with user.

3. Used Kernel Data Structures:

3.1. proc_dir_entry:

As mentioned above, /proc files are used as a means of communication between the attacker and the kernel. Each proc entry (file and directory) is described using a structure proc dir entry. In our rootkit, we are especially concerned with the functions responsible for reading and writing our newly created /proc file. Such functions can be modified through pointing the `procfs_read` and `procfs_write` function pointers to our custom made functions

3.2. kobject:

Sysfs is a virtual filesystem that describes the devices known to the system from various viewpoints. By default it is mounted on /sys. The basic building blocks of the hierarchy are kobjects.

3.3. hooked_function:

Each hooked function is represented by a hooked_function_info
structure that records all info related to the hooked function (its original address ,it's original first bytes,etc ...). The rootkit keeps track of all hooked functions by

implementing a doubly linked list of hooked_function_info structures.

3.4. **Hidden_pids:**

A struct that defines a kernel linked list that we will then embed in a kernel linked list and represents a list of the hidden pids, and id for each

4. **Features:**

4.1. **Communicating with the root kit:**

On writing to the /proc file, the rootkit taps into whatever command has been issued, and carries out such a command.

In our code, we first intercepted the filesystem processes. We obtained the actor procedure in the dircontext (file system process), assigned to `struct_procfs_filldir`, and we replaced it with `dup_procfs_filldir` procedure call.

`dup_procfs_filldir` intercepts the list of running processes, and iterates over them checking if they are added to the list of hidden process IDs, or if it's our rootkit communication process, in which cases, it hides them. It then calls the original function.

4.2. **Obtaining Root Access**

One of the key features of the rootkit is providing the attacker with root privileges on the compromised system, which essentially opens a window of endless possibilities to the attacker.

In our code, if the value in the buffer; the input command is equal to "root", we initialize a cred_struct (credentials structure), and sets all IDs to 0's (which is responsible for paging, and is actually part of the kernel (system boot), rather than a normal user-mode

process. This is not the same as PID 1, which is for the init process.). We then commit credentials; assign these credentials to the current calling process (the process that writes the proc file).

4.3. Hiding the rootkit

A feature which is of crucial importance is the ability to hide the rootkit, otherwise, a clever administrator could easily spot the existence of an through something as simple as running 'lsmod' and therefore, we incorporated stealth mode within our rootkit. In order to achieve complete stealth, the are two main lists from which our module had to be removed, the list of modules as well as the list of **kobjects**, which reflect the presence of the module in /proc/modules and /sys/modules respectively. The module was removed from the list of modules by simply invoking 'list del init' which given an element in a list, deletes it from the list, and was removed from the list of **kobjects** by invoking 'kobject_del'. On the contrary, showing the rootkit simply dictates that we re-insert the module in both lists through invoking list add and kobject add.

4.4. Hiding a process

Processes are usually listed in Unix through invoking the 'ps' command or the like. Such commands share the same basic infrastructure, 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 procfs functions such that the entry for the `iterate` function

actually points to our custom-made function. Having modified this function, we call the original iterate function ourselves and store the results. (everything in linux is treated as a file, which makes our lives much easier). Furthermore, the hooked function uses the overridden actor function .which returns 0 if the proc is in the list of hidden ones stores in the struct held for it or it is the proc we create for communicating with the user.

4.5. Unhiding a process

Unhiding the process is as simple as removing the id of the process stored in the hidden procs so that the overridden actor function does not return 0 for it.