INCOGNITO - LKM Rootkit

CSE 2431 Systems II

Authors: Kordell Stewart, Suvaion Das, Alex Hassenbein

Github: https://github.com/MiltonMan/Incognito

Intro

For our semester project, we decided to implement a rootkit as a linux kernel module. A rootkit is a clandestine computer program designed to provide continued privileged access to a system while actively hiding its own presence. The term rootkit is actually a connection of the two words "root" and "kit." By developing a kernel rootkit instead of an userspace rootkit, we have created a simple malware module that, by its basic nature, is far more difficult to detect and remove as the very kernel itself can’t be trusted once it is initialized. We decided to name our program, quite fittingly, **Incognito**.

In order to accomplish a fully realized rootkit, Incognito would need to have two basic functionalities. It would need to hide itself (the module that is) and be able to grant root access. From this base, different and varied capabilities could be developed. On its own, it’s entirely harmless, but it is intended to be used with a broad variety of other malicious tools to create a potent combination. For example, a stealthy remote shell to issue commands to the rootkit or a malicious file that is actively hidden by the rootkit. The whole idea is that the rootkit prolongs access to a system by hiding and obfuscating common methods of detection and navigation employed by a naive user. Once control and privilege is gained on a system there isn’t a limit to the damage an attacker could do.

Development

In terms of concept, our goto reference was an online kernel hacking manual. This manual organized and analyzed popular old 2.x kernel rootkits and their specific exploits. While the functionality was very outdated, it provided valuable insight to the various techniques, exploits, and methods our rootkit could utilize to achieve our functionality goals. With this in mind, we decided to make Incognito more of an open platform. Something simple enough to be expanded in the future. These possible expansions will be discussed later.

The rootkit was developed over multiple different stages, each stage taking a different approach. In the initial stage, the rootkit was developed with an attempt at using proc to communicate with the rootkit from a client program, which would then ask the kernel to hide files, processes, and directories by intercepting the appropriate system calls and modifying the data returned.

Over the course of the project, a number of minor problems were found. First, proc proved to be infeasible. Therefore, the rootkit was switched to working by instead hijacking the “kill” command and detecting what signals were carried on it. This effect had to propagate through the other functions, switching them over to rely on the hijacked kill instead of the initial plan as well. A config file was removed, for example. To deal with the limited nature of our input at this stage, we switched from targeting user input files (i.e., user types in “data.txt” to hide data.txt) we switched to a more limited, but just as functional means of hiding files and directories. Via our hijack of the getdents system call, we also check if files have a target prefix. This had a dramatic effect of streamlining our code so that instead of having to hijack multiple systems calls such as ls or lstat, we can get by quite comfortably with our couple of hijacks. One lingering issue is due entirely to implementation. Our command handler's root command is unstable. It doesn't work as intended with relative frequency and ends up targeting the 'sh' process which doesn't accomplish the intended functionality. As of project completion, the kill command for root should be used to gain root access reliably.

Functionality

The rootkit has a few main functions. First, the rootkit is capable of hiding itself from the user. Second, the Rootkit is capable of protecting itself from deletion. Third, the rootkit is capable of hiding and protecting other active processes. Fourth, the rootkit can hide other files and directories in the system.

These functions are called through a use of a hijacked "kill" system call. By using "kill" on a specific PID and using the designated Signals, the kill command will instead either hide, unhide, or grant root privileges to the process. By using Kill with the PID of 0, the module will target itself for hiding or protecting.

The signal -64 allows Root access to the PID specified, while 31 hides it instead. -63 hides the rootkit, specifically, and -1 protects it. Finally, -2 prints a help message, and repeating any of the previous signals simply undoes their action. This also provides a passive disguise. By resembling a commonly used system call, seeing "kill" on the list of previous calls is unsuspicious to the user, allowing the hacker to work with greater freedom.

Hiding functions by hijacking the getdents command. Upon retrieving the list, the module edits it before passing it to the user, removing any files that have been marked by a magic prefix (which, ideally, will never show up outside of this system) and by removing any PIDs and related processes that have been previously marked.

Protection works by incrementing the usage count of the module. By keeping it above zero, the system refuses to easily unload the module, as it assumes it's in use at the moment by another system, and that removing it would cause long-term problems.

Finally, root access is granted by changing the creds of the chosen task to the appropriate ones with create_creds().

All of this can also be accessed with the command_handler, allowing hiding, protection, and other such effects with a more intuitive command line series in user space.

Future Expansions

As mentioned earlier, Incognito is rather simple in nature and intended to be used with other malware or expanded into other, more advanced, functionality. Due to this, we believe Incognito is ripe for expansion in a few key areas. Namly, a proper infection method, key logging/file logging, and remote shell integration.

At the moment, it's highly unlikely that this module will infect any machine (an academic project on github isn't the best means of spreading malware). In order to take full advantage of Incognito and to simulate real world use of rootkits, Incognito would require a proper injection method. Whether a "dirty" download, a script, or injecting another module with our code, this would make a logical next step in the development of this module.

Another functionality that makes sense given the nature of rootkits, is a keylogger/filelogger. We have a hidden module that is intended to evade detection and usurp control over a system, it might as well keep track of all keystrokes and keep a detailed log of file modification. This could be a userspace system that Incognito hides or an added capability of the module itself.

All of the above would help flesh out the "ultimate" extension this project could utilize. At the moment, the user needs to input commands for Incognito to execute and designate what files need to be hidden. This is obviously not ideal for attackers trying to utilize Incognito. That is, the integration of a remote shell would solve this. This remote shell could be utilized to issue commands to Incognito over network from another machine. In combination with the hiding capabilities of Incognito, such capability would enable Incognito to operate maliciously and would approach the usage of real world rootkits that users fall victim to everyday.

Conclusions

Kernel rootkits are dangerous. They are relatively simple (there are ample resources online to ensure that even green kernel coders can accomplish functionality almost identical to rootkits deployed by real world attackers). This potent combination became increasingly clear as we developed this project. This combination allows attackers to hide their tracks and to keep privileged access to the machines they control. They can be running on a system and no naive user can even know they have been running (funnily enough, while developing Incognito, there were several instances where we even forgot it was running, quietly hidden in the background). The rootkits which use LKM in the fashion we implemented seem to be one of the worst ones a Linux user can get. This project also illustrated just how important practicing good security is whether it's not trusting dirty downloads or employing software specifically for targeting and removing rootkits in the kernel space.