Creating a Rootkit using a Linux Kernel Module

Finn Wilkinson
University of Bristol

Charlie Haslam
University of Bristol

Adam Deegan
University of Bristol

Abstract

The open sourced nature of the Linux kernel, whilst providing major benefits such as community driven development, bespoke customizability, and mass compatibility, to name a few, also opens the door to vulnerabilities such as kernel rootkit tampering. We aim to expose this by creating a rootkit module for Ubuntu 12.04 *(based on Linux kernel* 3.2*) and implementing example features such as: gaining access to restricted resources, having remote access to the system without the system user being aware, and hiding certain malicious file paths. A full detail of our rootkit implementation is outlined in the paper below, including its architecture, how it operates, its effectiveness, and how easily it is detected.*

1 Introduction

As with any complex projects, oversights, as well as unexpected behaviours, are bound to happen when designing and creating software. This is especially true of operating systems which need to manage every part of a system, hardware, passwords and users, personal files and programs. Along with all of this management, there is a need for security at every stage - we do not want one user to access another user's files without permission, and we definitely do not want a program looking into files it is not meant to. Here is where a large problem lies; people are always on the lookout for new security vulnerabilities, whether this be for improving the current security system and discovering unthought of exploits, or for personal gain by infecting a system to hold somebody's data at ransom. We aim to showcase a major exploit that can be difficult to prevent and detect in Linux operating systems - the kernel rootkit [8].

In this paper, we go over our implementation of a kernel module rootkit, decisions we have made along the way and features we have implemented. We hope that this can be used to highlight the security risk rootkits pose (in this case, specifically with Linux systems), as well as educate people on just how easy they are to create and install. We will evaluate the effectiveness of our rootkit by conducting system commands and seeing if we are able to find any evidence of a malicious kernel module or accompanying user-space programs.

2 Background

Rootkits can be implemented in various ways, from Direct Memory Access (DMA) to accessing the `/dev/kmem` virtual device, or by using Linux Kernel Modules [21]. Most notably, the key differences between User-level and Kernel-level rootkits. Within these two categories, there are many sub-types of rootkit available.

2.1 User-Level Rootkits

User-level rootkits exploit the API's and libraries that user-programs use, this is called API hooking or DLL Injection [13]. The rootkit accesses program memory address space and manipulates information. However, these are very easily detectable via methods such as signature-based, behaviour-based, or integrity-based detectors [18]. The rootkit aims to replace function addresses in program address space with the address of the rootkit, meaning that the rootkit is executed instead of the original function.

2.2 Kernel-Level Rootkits

Kernel-level rootkits are much harder to detect as they execute at the same level as anti-rootkit software. They work by hooking the native API of the kernel they are operating on to override system calls and behaviours [13]. A Linux Kernel Module is an example of a Kernel-level rootkit that, if designed in the right way, can be difficult to detect.

A Linux Kernel Module is an object file which contains code that can extend the kernel functionality at runtime [2]. These are used by most people to customise the base Linux kernel to their bespoke needs. The main difference of rootkits is that they modify pre-existing system calls and functionality in order to perform some malicious activity, rather than for legitimate reasons. This could be to: give root access to a user-space process; enable a backdoor into a system; or hide certain files/directories within user-space.

It can be hard to notice or detect if you have a rootkit installed. For one, many rootkits will alter system commands frequently used by system admins like `last` or `netstat` [21] in order to hide their network activity, as well as `ls` and `dir` to hide their malicious user-space code. Additionally, although there are certain mitigations that can be used such as static analysis and automatic verification using cryptographic signatures [21], these are not always implemented on home and workstation computers, meaning that rootkit modules can easily operate unnoticed on a large number of machines.

3 Design & Implementation

For our rootkit, we decided to create a kernel level rootkit module for Ubuntu 12.04, kernel version 3.2. Our choice for this type of rootkit was based off of the fact that they are harder to detect, as well as being able control machine functionality rather than just individual programs, making it compatible with more machines.

3.1 Creating a basic module

First, we created a simple module that would form the basis for our rootkit. This module could be loaded using the command `insmod` and removed by using `rmmod`. Listing currently loaded modules with `lsmod` shows that our rootkit is working and examining the kernel ring buffer (with `dmesg`) shows a small output message from the module. This functionality was removed after testing in order to not bring attention to the rootkit.

3.2 Hooking system calls

A lot of the functionality of our rootkit relies on being able to override system calls. This means that when a program tries to run certain kernel-level functions, our rootkit actually runs its own version of these functions. These could be used generally to gather data on an infected machine or, could be used by our malicious payload as a way of interacting with the rootkit and the functionality it provides.

When the rootkit is first initialised it locates the system call table in memory and stores a reference to the table. Then, it replaces the addresses of certain system calls with the addresses of our own modified functions. Since the table now points to our functions, any time a program uses a function we have overridden it runs the code specified by the rootkit instead. When the rootkit is uninstalled, the system call table is reverted back to its original state, to prevent errors and hide evidence of there ever having been a rootkit.

There are many way to communicate with a Linux Kernel Module from a user space program [16]. Initially, we thought about implementing this using a file which could be written to or read from by both the rootkit and a program in user space. We decided that the file approach was quite tricky to get right and would mean having an extra file which could be detected, leading to the detection of the rooktit.

Instead, we decided that a good way to enable a program to interact with our rootkit (in theory, for malicious uses) was to override the `kill(pid_t pid, int sig)` system call. In a standard Linux distribution, functionality is only defined for values of `sig` from 1 to 31 inclusive, meaning that providing any other values causes the function to fail. We override the `kill()` system call with our own function that accepts additional values of `sig`. Each value (Appendix A.2) performs a different function, thus expanding the use of the `kill` signal and allowing programs to interface with the rootkit.

3.3 A system call to allow root access

Being able to grant our malicious programs root access privileges without the user knowing is very important; it allows the malicious program to be much more capable. Our overridden `kill` provides this functionality. Using the system call `kill()` with a `sig` value of 32 results in root access being granted to the process that called it. This is done by creating a new set of credentials [12] and assigning an id of 0, the root user.

3.4 Hiding and Un-hiding the Rootkit

One of the main ideas behind the rootkit is to remain undetected. This means that it is important to hide the rootkit module from the list when `lsmod` is run. The rootkit contains a variable `module_list` of type `list_head`. This is a linked list which contains information about modules currently installed. To be able to hide and un-hide the module, the linked list needs to be altered. Changing this variable to point to the previous module in the linked list[1] hides the rootkit and adding the rootkit back to the linked list un-hides it.

A `sig` value of 33 and 34 can be used with the `kill` system call to hide and un-hide the rootkit on command, respectively.

3.5 Backdoor access to target machine

Whilst being an integral part of the rootkit, we took the decision to make the backdoor part of the accompanying malicious code rather than part of the rootkit module. Part of this motivation comes from the fact that some information needs to be read from an accompanying text file `backdoor_config.txt`, and this is much easier to do and conceal with a traditional user space program.

Much of the backdoor setup is automated on both the attacker's machine (which will be used to access the compromised machine) as well as the compromised machine itself. Before installation of the rootkit, the attacker must set up port forwarding on their router (`att_rt` from Figure 1), as well as execute the `backdoor_config.py` python program, which stores the attackers public rsa key, user name, and public ip address in `backdoor_config.txt` file.

Figure 1 shows a simplified diagram of the exchanges that happen between the attacker's computer `att_pc`, the attacker's router `att_rt`, the compromised user's router `comp_rt` and the compromised user's computer `comp_pc`, in order to set up the persistent backdoor.

When the backdoor program is executed on `comp_pc`, it starts by reading in the information from the previously filled `backdoor_config.txt`. After this, the attacker's public rsa key is copied into the `~/.ssh/authorized_keys` file, allowing the attacker to ssh into the compromised computer without the need for their password.

From here, three software packages are installed: `openssh-client`, `openssh-server`, and `autossh`. Both of the openssh packages should be pre-installed on Linux, but this will ensure the latest versions are installed. Autossh extends the ssh operation; it can open and then monitor ssh connections ensuring that if they go down, they are restarted again automatically [3]. The choice to use autossh was to ensure that our backdoor connection was stable and reliable, meaning we would not lose access in case of error.

After package installation, a check is made to see if the ssh rsa key pair exist on `comp_pc` - if they do not then they are generated. Prior to setting up the ssh connection, `comp_pc`'s public rsa key is copied to the attackers own computer using the ip address and username from `backdoor_config.txt`. On the first setup of the backdoor, the attacker will need to enter the password for their personal computer. For every other backdoor setup after this (i.e. re-establishing the backdoor on system restart) the password is not required as the shared rsa keys are used for authentication instead.

Finally, our auto ssh command is executed: `autossh -f -nNT -i ~/.ssh/id_rsa -R 7000:localhost:22 attUser@attPublicIp`. The

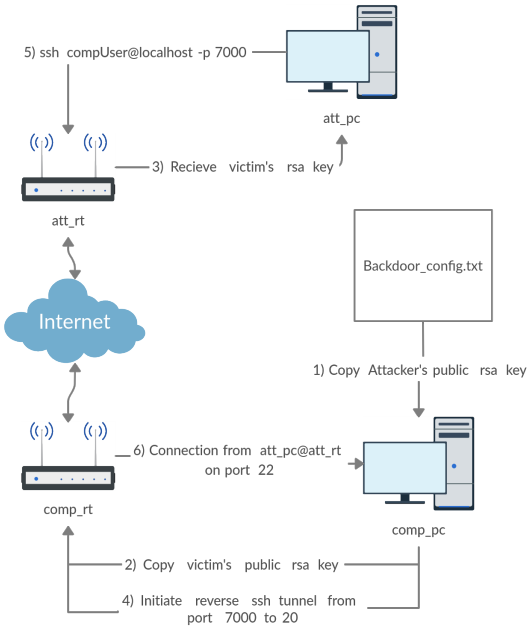

Figure 1: Diagram depicting steps taken to initialise backdoor

key parts of this command are the following [6]:

1. `-i` →dictates the ssh key to use for authentication
2. `-R` →makes this a reverse tunnel request, bypassing any firewall and getting around any port forwarding requirements on `comp_rt`
3. `7000:localhost:22` →tells `att_rt` that when an ssh request is made to `comprimisedUser@localhost` on port 7000, to connect it to `comp_pc@comp_rt` on port 22
4. `attUser@attPublicIp` →the attacker's username and public ip, previously loaded in from `backdoor_config.txt`

After the ssh tunnel has been established, the attacker can ssh into the compromised machine using the command `ssh infectedUsername@localhost -p 7000 -i ~/.ssh/id_rsa`, where `infectedUsername` is the username of the account we loaded our malicious code onto.

3.6 Hiding Backdoor Network Usage

Now that we have a working and reliable backdoor to gain access to a compromised system, we want to hide the fact that we are doing so. There are two main ways that a user or system administrator could detect us.

First is through the log files that Openssh writes to. By default, most ssh activity is logged, such as connection requests, failed attempts, and successful connections to name a few. This obviously is not good for us, as the user could read the log files and see our ip address and for how long we were connected. To try and mitigate this, we added some code to our backdoor script to execute a terminal command to change the *LogLevel* parameter to `QUIET` [1]; the lowest log level available [7]. This change is made every time the backdoor is established (i.e. on system start-up), and so even if the user changes this manually, it will likely be reverted back within a day or two, concealing our activity.

Secondly, and maybe most obviously, our ssh connection can be detected through network information commands such as `ss`, `netstat`, and `ps`, or active user commands such as `who`, `w`, and `last` [5]. Initially, it was hoped that we could take a similar approach to other parts of our rootkit and hook the syscalls associated with these commands. However, many of these types of commands get their data from files instead of kernel data structures, and so do not use syscalls that we could replace. After using `strace` to locate which files each command look at, we noticed when writing to the standard output, syscall `write` is always given 1 as its first parameter (the file descriptor)[4]. This information was used to hook the `write` syscall. If the file descriptor is 1, we look at the buffer passed to the syscall, and if it contains any of our keywords, we simply do not print to terminal and return.

The list of keywords is: [*':22', 'localhost', '127.0.0.1', ':ssh'*]. The reasoning for including *':22' and ':ssh'* is probably clear - these are the ports used for ssh connections. The reasoning for the other two keywords is specific to our backdoor. As seen in Section 3.5, the attacker executes the command: `ssh user@localhost -p 7000`. Because our reverse ssh tunnel has told the attacker's router to send any traffic to localhost on port 7000 to our compromised machine on port 22, this is also the address displayed by commands such as `last`. The attacker's real public ip is displayed by commands like `netstat`, however this is always with an accompanying port number (i.e. *':ssh'*). Therefore, by refusing to write rows of text to the terminal that include identifying information about our backdoor access, we eliminate most commands from displaying our information.

3.7 Hiding Directories and Files

We wanted the ability to hide files with a certain prefix, so that our malicious payload is not easily visible.

We used these sites [11], [10] extensively for the implementation. The first step was to hide files from appearing when using `ls`. Using `strace`, we could tell that `getdents` was the syscall being used for `ls` - `getdents` takes in a file-descriptor and populates a `struct linux_dirent` with more details, returning the number of bytes that it read.

The hooked `getdents` filters out files from being shown by adjusting the number of bytes read, starting by getting the original number of bytes using the original `getdents`. We then loop through each file; if the filename is something we want to hide we modify the returned bytes and `dirent` struct accordingly, otherwise we continue as per usual. The modified number of bytes read and `dirent` struct are then returned.

To hide multiple files, whenever a filename is checked, a list containing prefixes to ignore is looped through and checked instead.

3.7.1 Hiding Individual Queries

This does not prevent individual files from being queried. For example, `ls hiddenfile.txt` will still reveal its presence.

To hide individual queries, we again needed to look at what system calls occured with `strace`. This time, `stat` and `lstat` were the main culprits. For these, we just needed to return `-ENOENT` if the file needed to be ignored. The only complication was that full filepaths also needed to be parsed, but this was easy enough - we stripped the path before checking.

3.8 Hiding CPU and memory usage

Once our malicious payload is running, it is important to be able to hide its CPU and memory usage from the system. This helps prevent detection of the malicious program or the rootkit. To achieve this the rootkit needs to be able to hide programs for the `ps` command and `top` program. Linux uses a 'psuedo-file system' `/proc` to store information about running programs [15].

Essentially, each numbered directory in this folder stores the information relating to the associated process with that PID. This means that for `ps` and `top` to gather data about processes, they must inspect these folders. As expected, running `strace ps` shows that it makes use of the `getdents` call to inspect these folders.

This allows us to use our previous work and simply hide the directories of programs we wish to hide by hiding their respective folders. Our malicious payload does this for itself automatically, but we also added the capability of hiding a program using the overridden `kill` command. Using a sig value of 35 tells the rootkit to hided the program with the PID supplied in the `kill` call.

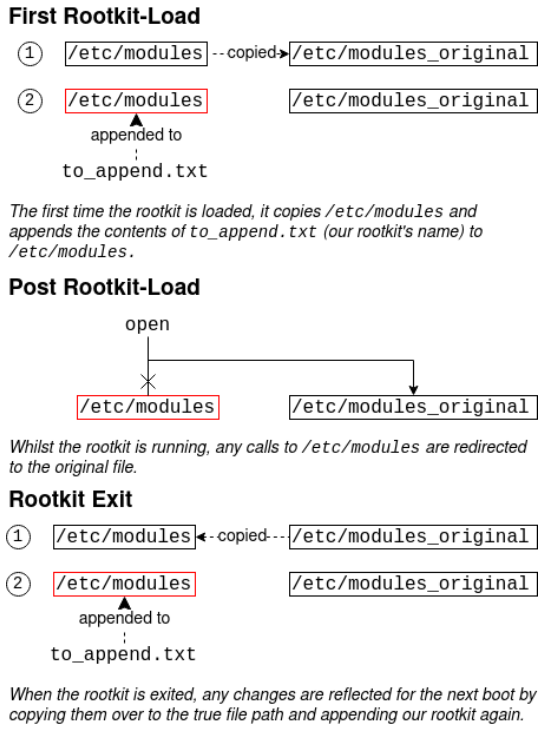

Figure 2: Diagram showing how access to \sbin\init is diverted.

3.9 Loading the Rootkit on Boot

There are several different methods of loading the rootkit on boot. We went for an approach similar to one mentioned in this paper [9], but modified. The main idea is that you want to restrict access to \etc\modules by redirecting all calls to a dummy copy of the file (i.e. to \etc\modules_original). \etc\modules contains a list of kernel modules to be loaded on boot; the true file at \etc\modules has rootkit appended on the end, loading our rootkit but hiding this from users and not allowing this true file to be edited. See 2 for a more intuitive diagram.

On first run of the rootkit, \etc\modules is copied to \etc\modules_original and rootkit is appended on the end. On rootkit exit, \etc\modules _original is copied back to \etc\modules, to reflect any changes the user has made, and rootkit is appended on the end of the true file again. Whilst it may seem redundant to keep rootkit appended on the true file at all times, it ensures the rootkit will still be loaded if the system suddenly stops or crashes, preventing the rootkit exit function not to be called. Functions to cloning and rename files within the kernel were implemented using Linux VFS functions [14].

To redirect any access to the original file after the rootkit is loaded, the open syscall is modified. open retains its original functionality, but the pathname passed into the original call is switched to a replacement if desired, i.e. calling open with \etc\modules will run open on \etc\modules _original instead.

4 Evaluation

To test our rootkit we start from a clean vagrant virtual environment and use our running instructions to install the rootkit and our malicious payload manually. We then test our various features to verify how well they work using the malicious payload. Whilst our program does not do anything malicious, it shows that we could easily install and hide something that did, such as crypto-mining, DDOS attacks or stealing information.

4.1 Loading the rootkit and hooking calls

The rootkit is designed to be used on Kernel version 3.2, in our case our VM is using Linux 12.04. The first time the rootkit is installed, it needs to be installed manually. The rootkit is compiled and then installed using insmod.

Once this has been completed, the next time the machine is restarted the module will automatically be loaded. Instead of loading the module through \etc\modules, \sbin\init can be replaced with a custom loader function - if done correctly, this does not require as many copies between the original and true files, and changes to this file are less explicit (e.g. you can not just check the file for the string rootkit).

The module correctly hides itself from the list of modules when running lsmod and hooks the relevant system calls. These are tested using our malicious payload to ensure they work as intended. Since the module is hidden from the module list, the rmmod command cannot be used to remove the rootkit.

Newer versions of Linux now require a password when running sudo commands. This would make it trickier to install the rootkit in its current state. However, it would be possible to use another exploit (ROP [17], for example) to gain root access to then install the rootkit. Although hooking system calls is a powerful way of altering system behaviour, it is possible to counteract these exploits with systems such as HookSafe or HookMap[19] [20], which detect hooking attempts and block them if they are believed to be malicious. Our rootkit heavily relies on being able to hook system calls correctly so countermeasures would severely impact our rootkit's effectiveness.

4.2 Hiding resource usage

Testing showed that the rootkit properly hides resource usage of the malicious payload form the user. Hooking

getdents allows the rootkit to hide the files and folders that our malicious program uses. Files and directories are hidden from the ls and dir commands, even upon inspecting a particular file (ls hidden.txt, for example). However, if the name of a hidden directory is known, it is still possible for us to access it with the cd command via the backdoor connection. Additionally, the hooks for getdents and stat struggle to deal with both full and relative path names. If many filenames need to be checked, the speed of the hook could become an issue. Using a linked list instead of a normal list would alleviate this.

This also enables the hiding of processes from the ps command. Processes are also correctly hidden from top. Hiding process from the user makes it much harder for the malicious program and rootkit to be detected.

4.3 Using and hiding the backdoor

To ensure our backdoor functioned correctly we accessed our VM at varying time intervals after setup. On all occasions we were able to access the system via ssh, meaning that autossh was working correctly by keeping our reverse ssh tunnel active. There are some drawbacks with our backdoor implementation however. From the attackers side, if their public ip address changes then the backdoor becomes obsolete. Additionally, if the compromised user generated new ssh keys, the backdoor would fail to authenticate with the attacker's computer and so the ssh tunnel would not be established.

Hiding the ssh logs also worked successfully. We checked the log files and found no ssh data, of which none had any identifying information such as ip addresses. A drawback with this method is that the user could easily change the ssh_config file to make ssh log verbosely again - possibly leading us being identified if this was done before we had accessed the machine. A way to counter act this would be to modify the open syscall to divert access from the ssh_config file to an identical dummy file would mean that the user could never change the real LogLevel.

In order to test that our network activity is hidden, we ran a series of different commands in terminal to ensure our information was not displayed. Using netstat, ss, last, who, and w, we verified that no information was displayed when referring to ssh ports or our ip/localhost. This method is more powerful than attacking individual commands as it works for any text lines that include our keywords. This is done by hooking the write syscall, which whilst effective, is subject to the same hooking countermeasures as mentioned before.

One drawback of this is that there is potential for overlap with legitimate text. If a user was performing their own ssh commands into another machine, they would not be able to see their connections also and would suspect that some malicious activity was going on. Additionally, other text such as times with '22' as the minutes would not be displayed - also raising concern. Furthermore, any third party application that is not terminal based would be able to display information we would like to be hidden, again countering our concealment efforts.

4.4 Communicating with the rootkit

Kill signals are used to communicate with the rootkit but a device driver would offer more flexibility. For example, the filenames which need to be hidden could be passed through and adjusted at runtime, or text which needs to be filtered could be dynamically changed.

5 Conclusion

The development of this rootkit has successfully led to a fairly well-rounded and robust rootkit. The rootkit is quite hard to detect and upon detection, there are some measures in place to prevent the user from simply removing the rootkit.

Although some aspects of development were quite tricky, our solution demonstrates a high level of functionality, exposing key exploits in the Linux kernel and how dangerous rootkits can be. However, there are countermeasures available for rootkits such as ours that can be used to expose, nullify, and eventually remove them from an infected system.

A Appendix

Overall we feel as though our project has gone well and we have managed to meet the objectives we set out in our project proposal. We were able to complete all of the essential functionality we laid out and even managed to complete most of our 'stretch goals', such as hiding network activity, and launching rootkit on system boot.

Unfortunately, due to time constraints we were unable to experiment with a different kernel version, as altering our code to work with newer kernel versions poses a fair few technical challenges. We did manage to get the rootkit to reload automatically on boot but we also wanted to load the backdoor automatically on boot. Unfortunately, we were unable to get this working in time due to technical difficulties with executing user-space programs from the kernel. However, we feel as though we have demonstrated a good, robust project through our other achievements.

Project Repo: `https://github.com/FinnWilkinson/S3CW`

A.1 Team contributions

Each of us was responsible for a subset of features. Once a feature was implemented we presented it to the rest of the group and were responsible for writing the relevant design and implementation sections of the report. We worked together to complete sections such as the abstract, introduction, background, evaluation, and conclusion.

A.1.1 Finn Wilkinson - 33%

- Backdoor access to the target machine
- Hiding backdoor network usage
- Writing README instructions and demo video work

A.1.2 Charlie Haslam - 33%

- Hiding and Un-hiding the rootkit
- Hooking system calls
- A system call for allowing root access
- Hiding CPU and memory usage
- Developed the malicious payload used to demonstrate the features

A.1.3 Adam Deegan - 33%

- Hiding directories and files
- Loading the rootkit and backdoor on boot

A.2 Kill signals

A table displaying the different `sig` values passed to our hooked kill syscall, and what functionalities these invoke.

`sig` value	Functionality
32	Gain root access
33	Hide rootkit mofule from list
34	Un-hide rootkit module from list
35	Hide program with PID of `pid` supplied in call

References

[1] https://kennethghartman.com/change-config-settings-using-a-bash-script/.

[2] https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html.

[3] https://linux.die.net/man/1/autossh.

[4] https://man7.org/linux/man-pages/man2/write.2.html.

[5] https://www.golinuxcloud.com/list-check-active-ssh-connections-linux/.

[6] https://www.ssh.com/ssh/command/.

[7] https://www.ssh.com/ssh/config/.

[8] BEEGLE, L. E. Rootkits and their effects on information security. *Information Systems Security 16*, 3 (2007), 164–176.

[9] BUNTEN, A. Unix and linux based rootkits techniques and countermeasures.

[10] CLARK, C. Rootkit for hiding files.

[11] CLARK, C. System call hooking.

[12] HOWELLS, D. Credentials in linux.

[13] KIM, S., PARK, J., LEE, K., YOU, I., AND YIM, K. A brief survey on rootkit techniques in malicious codes. *J. Internet Serv. Inf. Secur. 2* (2012), 134–147.

[14] KROAH-HARTMAN, G. Driving me nuts - things you never should do in the kernel, 2005.

[15] LINUX-FILESYSTEM-HIERARCHY. https://tldp.org/ldp/linux-filesystem-hierarchy/html/proc.html.

[16] MANISH LADDHA, A. C. M. Controlling linux kernel module through user application, 2019.

[17] SATHYANARAYAN, S., POURZANDI, M., AND ALIYARI, K. Return oriented programming - exploit implementation using functions.

[18] TODD, A., BENSON, J., PETERSON, G., FRANZ, T., STEVENS, M., AND RAINES, R. Analysis of tools for detecting rootkits and hidden processes. In *Advances in Digital Forensics III* (New York, NY, 2007), P. Craiger and S. Shenoi, Eds., Springer New York, pp. 89–105.

[19] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering kernel rootkits with lightweight hook protection. 545–554.

[20] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering persistent kernel rootkits through systematic hook discovery.

[21] ÉRIC LACOMBE, FRÉDÉRIC RAYNAL, V. N. Rootkit modeling and experiments under linux. *J Comput Virol 4* (2008), 137–157.

Notes

[1] Deleting the list from the rootkit's own struct is also required.