RKUltra — A Linux Kernel 5.4 Rootkit

Fergus Longley
fl17431

Joshua Turner
jt17624

Abstract

This paper details the creation of a Linux Loadable Kernel Module (LKM) rootkit for the latest x86_64 LTS version of Ubuntu [7] (Kernel 5.4). Features include process and file hiding, privilege escalation, keystroke logging, and a hidden interface for controlling the active features.

The full source code is available on GitHub at `https://github.com/furgoose/SysSec-Rootkit`

1 Introduction

Rootkits are a post-exploitation tool designed to hide their presence and achieve persistence on a system, whilst providing a set of tools for further potentially malicious activity such as hiding files and processes, gathering user information, and privilege escalation. There are various forms of rootkit, both for kernel space (LKMs, modifying kernel memory) and user-space (replacing binaries, abusing `LD_PRELOAD` [1]), along with many others [15] [18].

In this project, we focus on a Linux Loadable Kernel Module (LKM) rootkit. As an LKM operates in Ring 0, the most privileged feature set, this means we can alter almost all of the system behaviour, adding necessary features and avoiding detection as needed.

2 Background

For this project, we proposed to create an LKM rootkit that contains a large set of useful features, as well as remaining undetected on the infected system. To achieve this, we implement numerous techniques, including hiding files and folders with certain prefixes, processes, removing `/proc` and `/sys/module` entries, and hiding network connections.

For an added level of challenge and relevancy, we wanted to ensure that the rootkit would target a more recent kernel version. We settled on the kernel version packaged with the latest LTS version of Ubuntu (20.04), an operating system version that makes up a large proportion of Linux machines - approximately 23% of Linux systems with Steam installed as of November 2020 [6].

3 Design & Implementation

3.1 Rootkit Features

3.1.1 System Call Hooking

A large amount of the features in our rootkit rely on being able to intercept and modify system calls. To do this we hook the system call table and replace the pointers in the table to our owns functions, which can, in turn, forward the call to original function if required [13]. However, to modify the system call table, we must first locate it in memory. Conveniently, the Ubuntu kernel is compiled with the `CONFIG_KALLSYMS` option enabled an option that enables all debugging features in the kernel. This means files such as `/proc/kallsyms` are exposed, and also means the function `kallsyms_lookup_name` is available, allowing us to look up the location of symbols in the kernel by name. Looking up the `sys_call_table` allows us to find the first entry in the system call table.

Once we have found the table, we can find the functions we want to modify by adding an offset to the address of the table in memory. These offsets are exported for use in the system generally and we can use them to change areas in kernel memory, to point to our functions that will be executed upon a system call. A small caveat to this method is that the system call table is usually read-only, however, we explore a way to circumvent this in Section 3.3.1.

3.1.2 Hidden Module

Hiding from `lsmod`

We can hide our module from being shown by utilities such as `lsmod` and from the kernel itself by removing it from the kernel's doubly-linked list of modules. Each module loaded to the kernel is added to a list in kernel memory, to prevent modules being added twice to the system and give so that the

user can list currently loaded modules. However, as we are in kernel mode and it is exported to our kernel space via the `THIS_MODULE` variable, we can modify this.

```
u8 module_hidden = 0;
static struct list_head *module_list;
void module_hide(void)
{
    if (module_hidden)
        return;
    module_list = THIS_MODULE->list.prev;
    list_del(&THIS_MODULE->list);
    module_hidden = 1;
}
```

The above code hides the module from the kernel by deleting the member for the current module from the module list. It also saves the module list, so that we can re-add our module later, useful for if we want to remove the module later as the kernel can't remove a module it doesn't know exists.

Hiding from `/sys/module`

Another place to view loaded kernel modules in the Linux filesystem is in `/sys/module`. Whilst we can remove our module from this list in a similar way to the above method, this time by unlinking the kobject (seen below), this means we cannot easily toggle the hiding of modules, as `kobject_del` results in a call to `sysfs_remove_dir`, which results in many files and folders being irreversibly removed.

```
kobject_del(&THIS_MODULE->mkobj.kobj);
list_del(&THIS_MODULE->mkobj.kobj.entry);
```

Instead, we opted to hook the file operations of the `/sys/module` folder, and simply remove the folder from the file list when `filldir` is called. This does mean that the folder could still be discovered if the user knew what to look for, but we can mitigate against this by choosing a rootkit name that is effectively undiscoverable.

3.1.3 Hiding Processes

Each process has its own directory in the proc filesystem, named by the PID of the process, containing details such as the current executable and working directory [11]. The `/proc` directory is filled by the kernel, using the `iterate_shared` file operation. We can hook this function, allowing us to filter out sub-directories that match the PID of any of our hidden processes. This means that utilities such `ps` and `htop` will not display our hidden processes.

We do this by hooking the `iterate_shared` file operation for the `/proc` folder. From reading the Linux source code, we can see that `iterate_shared` has the signature `int (*iterate_shared) (struct file *, struct dir_context *);`. The `dir_context` struct provides an actor of type `filldir_t`, which is the responsible function for outputting to userland.

When we hook the file operations for the `/proc` folder, we only[1] have to replace this actor with our own function to achieve our desired behaviour. This function checks if the currently checked file/directory is for a process that is meant to be hidden and will not return include this if it is. It does, however, allow for hidden processes to see other hidden processes.

We maintain a doubly-linked list of process that we wish to hide, this can be kept up to date by hooking process management system calls. The process management system calls that we hook can be split into two categories, those that spawn new processes (`clone`, `fork`, `vfork`), and those that end processes (`exit`, `exit_group`, `kill`) — the `execve` and `execveat` system calls do not create new process and maintain the same PID so do not need modifying. As such, these groups can be treated in largely the same way, when a new process is spawned, if the process spawning it is hidden then add the PID of the new process to our list. When ending a process, if the process being ended is currently hidden then remove it from the list, this way if the PID is recycled in the future then that process won't be hidden.

An example snippet for creating a process, when the `fork` system call is called, is shown below. The `orig_fork` function is the original function that is executed when calling `sys_fork`, it returns the PID of the new process to the new process. Our hooked fork checks if the current process is hidden and if the fork succeeded, and if both are true then it hides the new process.

```
asmlinkage long rk_fork(
const struct pt_regs *pt_regs)
{
    long i = orig_fork(pt_regs);
    if (is_hidden_proc(current->pid)
        && i != 1)
        hide_proc(i);
    return i;
}
```

When a process is ended, we remove its PID from the list by calling a function that looks through our list of hidden processes and removes the member for that PID, if it exists.

3.1.4 Hiding Files

To hide files, we hooked the `getdents64` system call, the call that gets the entries of a given directory, commonly used by utilities such as `ls`. First, we call the original `getdents64` system call and filter from the results any record that matches

[1] There is some additional work to be done saving the original memory addresses of these functions so that we can safely unload the rootkit

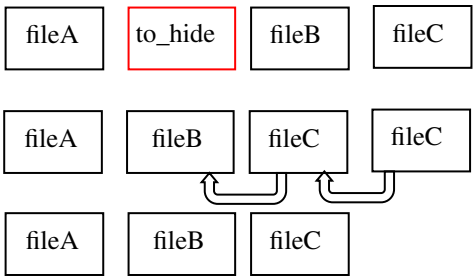

Figure 1: Diagram showing the hiding of files from a `linux_dirent64` structure. Identify the file to hide, move the entries after it forward in its place and decrease the size of the structure so as to not include extras files.

a given prefix. This way if `ls` is called for a particular directory, it will filter out all sub-directories and files that have the given prefix. If, however, the process that called `getdents64` is currently hidden, the filtering is not performed and that process will have any hidden files included in the output.

The filtering is performed by iterating over a `linux_dirent64` structure. If an entry name in the structure has a predefined prefix, we move all the memory following this entry over the entry and decrease the size of the structure we are returning. This is shown in Figure 1.

3.1.5 Hiding Sockets

There are two main ways of finding which sockets are in use, the `recvmsg` system call and the `/proc/net/` filesystem, alongside two common utilities that use each method, `ss` and `netstat`. The proc filesystem method of discovering open connections was not implemented, however, it has been explored in Section 4.1.2.

Hiding from `ss`

The `recvmsg` system call is used to receive messages from a socket and, more relevant to this paper, returns data about open sockets. `ss` uses this method to discover open sockets and the connection details [5]. When hooking this system call, we process the data that it returns to filter out any connections that we wish to be hidden, in this case, any connections that communicate to a specific remote port used by our rootkit. This port is used later when gaining remote access via port knocking, detailed in Section 3.2.2.

3.1.6 Keystroke Logging

There are many ways to create a keystroke logger from kernel space [16]. It would have been possible to use hooked system calls as we used in other sections to intercept keystrokes as they are read, however, the easiest solution is to use a method built into the Linux kernel, the keyboard notifier API.

The keyboard notifier API allows us to register a function that will be called with every keystroke on the machine. From here, we can use maps to convert the keycode to plain-text and log to a file. In order to not noticeably slow down the system, a small buffer is maintained within the rootkit, and the buffer is written out to a file every time a newline is received. We utilise the previously implemented file hiding method described in Section 3.1.4 to ensure that this file cannot be discovered by any unintended users.

3.2 Interaction

3.2.1 Local Interface

Local user-space interaction with the rootkit implemented via the proc filesystem, by using a hidden proc file (`/proc/rootkit`).

In kernel mode, Linux allows for pseudo-file creation in the proc filesystem, utilising your own functions for file operations. We define a function for write operations, allowing different commands for interaction with the rootkit. For example, writing "`root`" to the file will give the process root permissions. The "`toggle`" command will toggle the hiding of prefixed files, processes, the proc filesystem interface and the LKM itself.

Writing "`hide<PID>`" will add `<PID>` to the list of hidden processes, so this process and all its subsequent children will be hidden until it exits. If the command "`echo hide$$ > /proc/rootkit`" is run in bash, the current bash shell, and subsequent commands, will be hidden from the rest of the system. Writing "`unhide<PID>`" will remove the PID from the list of hidden processes, this means it will be visible to the rest of the system, however, any hidden children of the process will remain hidden.

3.2.2 Remote Access

As part of the design, we wanted to have a method of being able to gain root remote access to the machine. To achieve this we use port knocking.

Linux provides a framework called Netfilter to facilitate various networking operations, such as packet filtering and NAT [17]. It allows us to register our own net hooks which we can use for port knocking [14]. We built a hook which runs on incoming TCP/IPv4 connections to the machine. Upon a new incoming connection to one of the pre-specified knocking ports, we check if the port is the next in the sequence for remote address, and if it is we move the sequence forward for that address. If it is incorrect, we can reset the sequence for that address.

When the sequence is completed, a hidden connection is opened to the remote address. This is done by scheduling the kernel to start the following program in user mode:

```
/bin/bash -c 'echo hide$$ > /proc/rootkit &&
↪  bash -i >& /dev/tcp/<REMOTE_ADDR>/<RPORT>
↪  0>&1'
```

This program hides itself from the rest of the system (`echo hide$$ > /proc/rootkit`) and, if that succeeds, starts a reverse bash shell to the machine performing the knocking (`bash -i >& /dev/tcp/<REMOTE_ADDR>/<RPORT> 0>&1'`). This command is run as root, so the knocker will have root access to the machine.

This allows the knocker to gain remote access, without being dependent on any userspace programs such as OpenSSH. Whilst it is dependent on bash being installed on the system, it installed by default on Ubuntu, along with the majority of Linux distributions, and `apt` warns that bash is an essential program when attempting to remove it. The knocker needs to be listening on the defined `RPORT` on the address it is used to knock, for example, `nc -lvp 4444`.

3.3 Design Challenges

3.3.1 Circumventing Write Protection

Linux enables write protection to critical areas of the system memory, and to modify the system call table to point to our own hacked functions, we need a way to disable this protection. Previously it was possible to change the write protection bit in register `cr0` using `write_cr0`, however since Linux 5.3 checks have been added to ensure that any write operation does not turn off this bit [3]. Since this check is only used when calling the function `write_cr0`, we can avoid it entirely by using assembly language and writing directly to the register, seen below.

```
asm volatile("mov %0, %%cr0"
             : "+r"(cr0), "+m"(__force_order));
```

3.3.2 System Call Stubbed Functions

An issue we came across when trying to hook some system calls was that methods that were reliable for previous kernel versions no longer worked. Since Linux kernel version 4.17, register arguments for system calls are first copied into a `pt_regs` struct, and then manually unpack the arguments from the fields [2]. For example, for `getdents64`, the original system call had the signature

```
asmlinkage long sys_getdents64(unsigned
↪  int fd, struct linux_dirent64 __user
↪  *dirent, unsigned int count);
```

However, due to the stubbed functions, in our hacked function we need to unpack the arguments manually as follows.

```
asmlinkage long rk_getdents64(const struct
↪  pt_regs *pt_regs)
{
    struct linux_dirent64 __user *dirent =
    ↪  (struct linux_dirent64 __user
    ↪  *)pt_regs->si;
    ...
}
```

4 Evaluation

Our original aim was to build a rootkit for a modern kernel release that is resistant to reasonable detection attempts, provides remote root access, and could capture user input data for extraction.

4.1 Detectability

4.1.1 Hiding Files and Processes

Hiding files, and processes, is achieved by removing them from directory entries they would be in normally, this means that hidden entries can be discovered by enumerating every possible entry in the directory and calling `access` on them. For file hiding, the exported hidden file prefix can be changed to a long random string of alphanumeric digits, such as the hash of some known keyword. This can make enumeration much more difficult as the size of the space that would be needed to enumerate would become very large.

However, it is more reasonable to enumerate the process entries in the proc filesystem. Comparing the results of a `getdents64` system call with an `access` system call for all the possible PIDs (up to 4194304 on x86_64), and comparing PIDs you could access with those that didn't appear in the directory entries. This method would have a good chance of finding hidden processes but would also discover many false positives — processes that had started since the `getdents64` system call was made. Overall, we believe this to be a reasonable method for process hiding as the process for detection would have to be run continuously to discover a hidden process, as by default for the rootkit there are no running user-space processes (just the keystroke logging).

4.1.2 `netstat` and `/proc/net/tcp`

One of the ways in which our rootkit does not stay hidden is in the discovery of open connections. We currently hide connections being reported by the `recvmsg` system call, but open TCP/IPv4 connections can also be discovered in the `/proc/net/tcp` file. This provides an extra challenge to hiding connections since the data structures that are used to generate and control the output of this file are not exported in kernel 5.4 and have a randomised layout. The `seq_operations` and `proc_dir` structures have been used in rootkits aimed at older versions of the kernel to achieve proc filesystem network hiding. `netstat` [4] is a utility that uses the proc filesystem to

discover open connections, so will discover connections we would like to hide.

Whilst we cannot access the structures to generate the `/proc/net/tcp` file, we can modify the system calls used to access them. Specifically, `netstat` uses the `openat` system call to access the file. There is a method we can use to change the result from this system call to point to a different file. Into this file, we can write the contents of `/proc/net/tcp`, ignoring lines that are connections we want to hide. This method is detailed in Algorithm 1.

Algorithm 1 Hooked `openat` function for `/proc/net/tcp`

```
1:  procedure OpenAtProcNetTcp
2:      proc_fd ← openat("/proc/net/tcp")
3:      proc_buffer ← read(proc_fd)
4:      while proc_buffer has next line do
5:          line ← next_line(proc_buffer)
6:          if line contains hidden connection then
7:              move subsequent lines forward over this line
8:      if found hidden connection then
9:          new_fd ← new_hidden_file
10:         write(new_fd, proc_buffer)
11:         return new_fd
12:     else
13:         return proc_fd
```

This method would work to hide our hidden connection from `netstat`, however, would make for very slow accesses. Each time a user access `/proc/net/tcp`, the could be a read of a file, and then writing the contents to a new file write. This would slow down the operation massively but would hide the open connections from userspace.

4.1.3 Capturing of User Input Data

Capturing input data from the user and making it easily available in a usable format was an important aim for this rootkit. By making use of a kernel-mode keystroke logger that writes the readable form of the keystrokes into a user-space file for viewing, for example writing as a capital letter when capslock is on or if the shift key is being held. This makes for a very readable file for extraction and analysis. This was a very successful portion of the rootkit.

4.1.4 Remote Access

The use of port knocking and opening a reverse bash shell made for successful remote access as root. Since this solution was entirely kernel-based (and didn't rely on just opening ports or using a binary that may not be installed, e.g. OpenSSH), we can be very confident that it will work on target systems. This section shows a very strong application of LKM based rootkits.

5 Conclusion

Overall, We were successful in completing our original goals of creating a capable rootkit for an up to date version of the Linux kernel. Whilst recent changes to the kernel have made certain aspects of rootkit development more challenging, we have shown that it is still a viable solution when creating a rootkit.

There will always be an uphill struggle against attackers working around any mitigations. More threatening to the possibility of future rootkit development, however, is the ongoing research of detection strategies [9, 13]. In order to circumvent these newer methods, we can turn to more advanced methods of rootkit [10, 15], such as attacking hard drives [18], targeting SMM [8], or even virtualising the entire operating system to avoid detection [12].

Glossary

Hook Alter the behaviour of functions by intercepting calls to them. 1

LKM Loadable Kernel Module. 1, 3, 5

Port Knocking Opening a connection to a series of specified ports on a remote machine. 3

Proc Filesystem A virtual filesystem (`/proc`) that, in Linux, presents runtime information, such as resource usage and active processes. 2–4

SMM System Management Mode. 5

System Call Table A table in kernel memory containing pointers to functions that carry out system-level operations on behalf of the user. 1

References

[1] Jynx2 - an LD_PRELOAD rootkit. https://github.com/chokepoint/Jynx2. Accessed 9/12/2020.

[2] The linux kernel - adding a new system call. https://www.kernel.org/doc/html/latest/process/adding-syscalls.html?highlight=syscall. Accessed 9/12/2020.

[3] Linux source code (v5.3) - arch/x86/kernel/cpu/common.c. https://elixir.bootlin.com/linux/v5.3/source/arch/x86/kernel/cpu/common.c#L372. Accessed 8/12/2020.

[4] Netstat man page. http://net-tools.sourceforge.net/man/netstat.8.html. Access 10/12/2020.

[5] SS binary source code. https://fossies.org/linux/iproute2/misc/ss.c. Accessed 9/12/2020.

[6] Steam hardware & software survey: November 2020. https://store.steampowered.com/hwsurvey?platform=linux. Accessed 6/12/2020.

[7] Ubuntu 20.04 release notes. https://wiki.ubuntu.com/FocalFossa/ReleaseNotes. Accessed 26/11/2020.

[8] Shawn Embleton, Sherri Sparks, and Cliff Zou. Smm rootkit: A new breed of os independent malware. *Security and Communication Networks*, 6, 12 2013.

[9] Owen Hofmann, Alan Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with osck. *ACM SIGARCH Computer Architecture News*, 39:279, 03 2011.

[10] Ralf Hund, Thorsten Holz, and Felix Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. pages 383–398, 01 2009.

[11] M Tim Jones. Access the linux kernel using the/proc filesystem. *IBM developerWorks*, 2006.

[12] Samuel King, Peter Chen, Yi-Min Wang, Chad Verbowski, Helen Wang, and Jacob Lorch. Subvirt: Implementing malware with virtual machines. volume 2006, pages 314–327, 01 2006.

[13] John Levine, Julian Grizzard, and Henry Owen. A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table. In *Second IEEE International Information Assurance Workshop, 2004. Proceedings.*, pages 107–125. IEEE, 2004.

[14] Rusty Russell and Harald Welte. Linux netfilter hacking howto. *Disponıvel em http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO. letter. ps (Junho de 2005)*, 2002.

[15] Wonjun Song, Hyunwoo Choi, Junhong Kim, Eunsoo Kim, Yongdae Kim, and John Kim. Pikit: A new kernel-independent processor-interconnect rootkit. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 37–51, 2016.

[16] Christopher Wood and Rajendra Raj. Keyloggers in cybersecurity education. In *Security and Management*, pages 293–299. Citeseer, 2010.

[17] Qing-Xiu Wu. The research and application of firewall based on netfilter. *Physics Procedia*, 25:1231–1235, 12 2012.

[18] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. Implementation and implications of a stealth hard-drive backdoor. pages 279–288, 12 2013.