CMK Rootkit. Identifying the "magic packet" requirements via pattern recognition

David Álvarez-Pérez[1], Manuel Fernández-Veiga[2]

[1]Gen Digital Inc. Pikrtova 1737/1a, 140 00 Praha 4, Czech Republic. [2]atlanTTic, I&C Lab, Escuela de Ingeniería de Telecomunicación. Campus universitario s/n, Vigo, 36310, Spain.

Abstract

The present paper analyzes the CMK Linux Kernel Rootkit. It demonstrates that it is possible to unpack the rootkit using emulation to avoid inserting the module in a real Linux distribution matching specific Linux kernel requirements. CMK Rootkit implements "magic packets", this study also demonstrates that it is possible to extract the requirements for the "magic packets" based on assembly language patterns. We provide the implementation for both, the Linux kernel rootkit unpacker and a Ghidra script for extracting the requirements of the "magic packets".

Keywords: cmk rootkit, magic packets, botnet, netfilter.

1 Introduction

Packed malware has the disadvantage of being not stealthy, at least such code protection will raise suspicions.

It is quite common to see Linux-packed malware samples in ring 3 but it is much less common to see packers in ring 0 Linux malware. Of course, we can put some examples of ring 0 malware using kernel space packers, for instance, the Reptile Rootkit [1] uses a custom rol32 algorithm in kernel space for decrypting the payload at the init_module function. Reptile calls its packer Kmatryoshka and Parasite to the unpacked kernel rootkit [2, 3] which is the Reptile Rootkit kernel module.

Most of the antiviruses detect Reptile Rootkit pretty well [4] because it is open source and the source code repository dates from 4 years ago, but we discovered a new rootkit in VirusTotal [5] (*malware_tr_2022-06-19.tar*), that we called CMK Rootkit because of how it reveals itself, which uses a kernel space packer [6] and remained undetected for about three months in VirusTotal [7].

Kernel space packers for Linux kernel modules can be helpful for efficiently passing under the radar. Fortunately, we are always looking for this kind of threat (i.e., Syslogk [8][9]) and, even if protected, we were able to detect this new kernel rootkit and extract the requirements for the "magic packets".

The rest of the paper is structured as follows. Section 2 describes the components of this new rootkit whose features are presented in Section 3. In Section 4, we explain some solutions to extract the "magic packet" requirements, present an implementation in Section 5, discuss next our results in 6 and some extensions are summarized in Section 7. Finally, Section 8 collects our conclusions.

2 Overviewing CMK rootkit

A machine infected with CMK rootkit contains the following three artifacts (see Fig. 16):

(a) /usr/share/man/B18D/cmk/launch

(b) /usr/share/man/B18D/cmk/module

(c) /usr/bin/systemd-hwdbsa

The launch script and the packed CMK Linux kernel rootkit (artifacts (a) and (b), respectively) are analyzed

next. At the same time, (c) is the arbitrary malicious application hidden by the rootkit.

2.1 Analyzing the launch script

The launch script (see Fig. 1) is written in bash programming language [10] and it is responsible for installing the packed CMK Linux kernel rootkit and executing the malicious hidden application.

In an intrusion post-exploitation stage scenario, the attacker copies the three components of the rootkit (the launch script, the packed CMK Linux kernel rootkit, and the arbitrary malicious application) to the compromised machine. After that, the threat actor executes the launch script (with arguments) for silently installing the packed CMK Linux kernel rootkit and, finally, the CMK Linux kernel rootkit calls the same script (without arguments) for executing the malicious hidden application: */usr/bin/systemd-hwdbsa*.

If the script is executed without arguments, it will execute the arbitrary malicious hidden application whose name can be confused with the legitimate *systemd-hwdb* (hardware database management tool [11]) and whose name appears also in the hardcoded list of hidden files of the CMK rootkit.

In case the launch script is executed with arguments, the *pkill* [12] Linux command with the flag *-STOP* will send a signal to Linux Journal [13] for stopping it and preventing the system from logging the subsequent malicious actions. Notice that all the commands are executed by redirecting the output to */dev/null* such that the command output is hidden (lines 3-5). After stopping the logging system, the launch script inserts the packed CMK Linux kernel rootkit into the kernel via *insmod* [14] Linux command (line 4). Finally, the launch script lets the Linux Journal continue its execution by sending a signal using *pkill* with the flag *-CONT* (line 5) such that the service will continue logging system events.

2.2 Analyzing the CMK kernel space packer

The */usr/share/man/B18D/cmk/module* component is a packed protected version of CMK Linux kernel rootkit. Its code consists of two parts: an encrypted version of CMK Linux kernel rootkit, and a routine or stub responsible for decrypting, loading, and executing it (see Fig. 16).

The packed CMK Linux kernel rootkit is an ELF64 x86-64 (Relocatable) file, compatible with the following kernel version [15] according to the .modinfo [16] section of the file: *vermagic=3.16.06amd64 SMP mod_unload modversions*. It can be loaded, for instance, in a Debian 8.11 Jessie [17] 64-bit VM disk image which exactly matches the kernel version. The file does not have too much code but a lot of data, it has only a few symbols (*init_module* and *kallsyms_on_each_symbol*) apart from each other introduced by the compiler. Those are indicators of packed files [18].

By analyzing the *init_module* function, we found a basic block in a loop containing instructions typically used for decryption (arithmetical, logical, and bit-shifting operations). So, taking a look at the data used in the loop, it is easy to identify (see Fig. 2) the beginning of the encrypted kernel rootkit, at the offset 0x300 (0x340 offset on disk), that is stored in the RDI register of the processor. Notice that the end of the encrypted code is at the address 0x1FD78 (0x1FDB8 offset on disk) stored in esi. The .text section has a 0x40 bytes displacement on disk, so it needs to be kept in mind when calculating the offsets. Finally, the key of the decryption algorithm is 0x24924925 which is stored at the r9 register of the processor.

The decryption routine is located just after such initialization of the registers. It applies a custom algorithm that does not use syscalls or any other external code so this routine is very easy to emulate. We implemented an unpacker that relies on Miasm framework [19, 20] to emulate the decryption routine and dump the CMK Linux kernel rootkit to disk. Miasm is a free and open-source (GPLv2) reverse engineering framework aiming to analyze/modify/generate binary programs. It also includes an x86-64 emulator that we use in our dumper tool.

2.3 Dumping the CMK Linux kernel rootkit with Miasm

It is possible to dump the CMK Linux kernel rootkit from memory after the decryption algorithm gets executed but it requires setting up the environment for loading the packed CMK Linux kernel rootkit which can be a time-consuming task. So we developed an unpacker tool that statically dumps the CMK Linux kernel rootkit via emulation. It works as follows.

Based on the disassembly, we added some information to our unpacker. For instance: the entry point of the decryption routine, the offset to the encrypted bytes, its length, and so on. Notice that all of these values were taken from the in-memory representation, the values are displaced 0x40 bytes on disk, as already mentioned in the previous section, so we added such displacement to our unpacker tool as Imagebase.

After reading the bytes of the packer, our unpacker sets up an x86-64 emulator [20][21] and loads those bytes at address 0x0. To avoid some errors that happened during the emulation, we manually set the rdi and esi registers and prepare the execution for starting at the next instruction (address 0x5C in memory).

We also developed a function for dumping the decrypted bytes to disk that is executed when hitting a breakpoint at the end of the decryption routine (address 0xA1 in memory). Then our unpacking script lets the emulator run (optionally observing the decryption trace) which ends up dropping the CMK Kernel Rootkit to disk.

2.4 Understanding how the packer inserts the decrypted rootkit

For dynamically loading the CMK Linux kernel rootkit, after unpacking, the stub of the packer calls *sys_init_module* [22] which is a Linux kernel API function for loading kernel modules. In case the *sys_init_module* symbol can't be resolved via *kallsyms_on_each_symbol* [23], it tries to load it via *__do_sys_init_module*.

3 CMK Linux kernel rootkit features

As expected, CMK Linux kernel rootkit is distributed as an x86-64 kernel module compiled for the same kernel version as the packer. It was designed to start a hidden application and to bypass the local firewall when providing a stealth reverse shell to the attacker, on-demand via magic packets.

CMK Linux kernel rootkit installs two Netfilter hooks [24, 25] for implementing "magic packets" (see Fig. 12) that execute the user-mode commands via Linux Kernel Work Queues [26]. It also hijacks some kernel functions, hides itself, hides malicious processes, and so on.

This malware is probably also based in OSOM (Out of Sight Out of Mind kernel rootkit) [27], a Linux kernel rootkit developed for academic purposes (OSOM bachelor project, Department of Computer Science, University of Copenhagen). We found great similarities between OSOM and CMK in the function hijacking technique implementation, the Netfilter hooks implementation, the use of Linux work queues [26], and the code structure (i.e., the init function of the module).

Briefly stated, OSOM consists of an *osom_init* function where the malicious functions for hijacking APIs, intercepting network traffic via Netfilter hooks (first hook, last hook), and hiding the module are called. It uses two Netfilter hooks for implementing "magic packets". The first hook parses the incoming DNS traffic [28] and, in case it fits some requirements, the malicious bash command to execute is extracted from the packet for being executed as a Linux work [26]. The other hook rewrites incoming DNS packets from the C&C server such that the communication is transparent to the host.

3.1 The rootkit hides itself from the list of modules of the system

As the majority of the rootkits does [29], CMK Rootkit removes itself from the list of the inserted Linux modules making it invisible to the lsmod Linux command (see Fig. 3). The function implementing it checks, by using the *repe cmpsb* assembly instruction, if the value of the parameter passed to it is: *hide* or *show*. Allowing both, adding or removing the module from the list.

This hiding feature is very similar to the one implemented by a didactic rootkit by Nick Newson [30], which uses the *sysfs_unlink_sibling* [31] Linux kernel API for hiding the rootkit (removing it from the list) after checking if it is already hidden and also uses *sysfs_link_sibling*[32] for adding it to the list. The main difference is that the CMK rootkit tries to use *kernfs_unlink_sibling* in case *sysfs_unlink_sibling* is not found and also tries *kernfs_link_sibling* [33] when *sysfs_link_sibling* is not found.

3.2 Persistence and execution of CMK

During the execution of the *init_module* function, the CMK rootkit declares a Linux delayed work [26] to be executed after 2 seconds. The delayed work structure pointer *cmk_launch_delayed_work_structure* is passed as an argument to the *queue_delayed_work_on* API call.

As shown in Fig. 4, it executes the file */usr/share/man/B18D/cmk/launch* without arguments. This means that it will run the malware */usr/bin/systemd-hwdbsa* as already discussed in the section *Analyzing the launch script*.

The call to the *snprintf*[34] function is important and deserves more explanation. It concatenates the substring *CMKCBAA6780FD86=* and a key allowing it to hide and protect user mode processes. It is explained next.

3.3 CMK can distinguish if the process was created by the rootkit

CMK rootkit relies on the kernel API *call_usermodehelper* for executing user mode applications. Such a call receives a string that starts with the substring *CMKCBAA6780FD86=* followed by a key (see Fig. 5). The key varies depending on the call and can be both, supplied by the attacker via "magic packets" (must match the value: *0x8A9C491F*) or hardcoded in the rootkit (value *00000001*). The resulting string is passed to *call_usermodehelper* [35] as an environment variable local to the process.

The rootkit hooks [36] the Linux kernel API load_elf_binary which is responsible for preparing the ELF files execution; such hook checks the existence of the *CMKCBAA6780FD86* environment variable.

If the variable contains any of the two mentioned values, the rootkit will hide and protect the user mode process using the mechanisms explained in the next sections.

3.4 CMK rootkit hijacks OS functions

After starting to take a look at the obfuscated functions, we noticed that some of them consisted of 0x20 NOP instructions (opcode 0x90) followed by a RET instruction (opcode 0xC3) [37].

This kind of trampoline function is also implemented by OSOM Rootkit for hijacking a set of kernel APIs. The main difference between both rootkits is that while the kernel addresses are hardcoded

and NULL data (0x00 opcodes) is used in OSOM, the CMK rootkit improves it (see Fig. 9) by resolving the addresses via kallsyms_on_each_symbol and uses NOP instructions which are less prone to produce system crashes. You can find a table enumerating the functions that are hijacked by the CMK rootkit (the original kernel API, the trampoline, and the hijack function) in Table 3 of the paper.

3.5 The rootkit hides files and folders from disk

CMK Rootkit uses an internal structure to store the names of the files and folders that will be hidden. Such a structure is composed of pairs of quad-words containing both, the length and the file/directory name to hide.

Based on this structure, the rootkit hooks [36] the following Linux Kernel APIs for hiding files and folders containing the strings in the list (see Fig. 6): user_path_at_empty, compat_fillonedir, compat_filldir64, compat_filldir, fillonedir, filldir64, filldir, do_sys_open [38, 39]. For more information on the hooked functions, please refer to Table 3. All of those hooks on file and folder-related APIs, make a call to a function that iterates on the *hidden_files* structure and compares it with the original data for determining if it should hide it or not.

3.6 CMK hides userland processes

It implements the same technique as Reptile rootkit [1] for hiding processes. It hooks the function *next_tgid* which is responsible for the */proc/PID* entries (allowing it to make the process invisible) but, it additionally hooks *copy_creds* and *exit_creds* to add or remove the flag *0x0BABA00* (see Fig. 7), on the *task_struct*, for a process being visible or invisible, respectively.

It also hooks *find_task_by_vpid* which allows to retrieve the *task_struct* for a given PID. CMK Rootkit performs checks if the *task_struct* contains the flag *0x0BABA00* established by *copy_creds* which indicates that the task is hidden (this technique is also implemented by Reptile rootkit).

3.7 The rootkit hides its CPU usage

To efficiently make the process invisible, the CMK rootkit also hides its CPU usage by hooking the kernel function *account_process_tick*. By doing so, it skips the ticks for the hidden process (it checks if the flag *0x0BABA00* is present in the *task_struct*). Probably a great example of kernel rootkits using this technique is those hiding cryptominers.

At Fig. 8, you can see the hook entry for *account_process_tick*, where *f91* is the trampoline and *sub_4960* is the function that implements the hook. Even if implementing the hiding mechanism is not complex, it requires a good understanding of Linux internals for processes. There are not too many Linux kernel rootkits thoroughly implementing it (i.e., only 4 unique Rookits on Github use this technique [40]) so this is a strength of CMK rootkit.

3.8 CMK protects the userland process from kill

CMK rootkit hooks also *kill_pid_info* (see Fig. 13), which is part of the internal flow when *sys_kill* is called (see Fig. 10). Hidden processes, namely, processes with the flag *0x0BABA00* on the *taks_struct*, cannot be killed. An implementation of this technique is publicly available in the *Kunkillable* [41] repository, which also implements this technique.

3.9 The rootkit hides the malicious TCP and UDP network traffic from the host

For hiding the network traffic, CMK Rootkit hooks the following Linux kernel APIs: *inet_stream_connect*, *inet_release*, *inet_bind*, *inet_diag_bc_sk*, *udp6_seq_show*, *tcp6_seq_show*, *raw6_seq_show*, *udp4_seq_show*, *tcp4_seq_show*, *raw_seq_show*, *tcp_time_wait*, *sockfd_lookup_light*. It allows CMK to remove the traffic related to invisible processes having the flag *0x0BABA00* in the *task_struct*.

3.10 CMK prevents the system from logging the malicious activity

As shown in Table 2, CMK Rootkit hooks do_syslog, devkmsg_read and comm_write for filteringout messages from the kernel's log.

3.11 The rootkit prevents the system from auditing the malicious processes

CMK rootkit also hooks the *audit_alloc* function in the kernel and clears the *TIF_SYSCALL_AUDIT* flag which defines whether the process is audited [42]. A similar code snippet is available in the Reptile rootkit implementation 13 [43].

3.12 CMK rootkit implements "magic packets"

CMK Rootkit sets two Netfilter hooks by calling the kernel API *nf_register_hooks* [44] (see Fig. 11). It allows the rootkit to inspect the traffic. *nf_register_hooks* receives two parameters: the array of hooks and its number of entries. Each hook entry in the array consists of a *nf_hook_ops* structure (see Table 12). The API *nf_register_hooks* will call to *nf_register_hook* for each hook entry, this second function is responsible for setting the hook.

In the next subsections, we discuss the functions *hook_1* and *hook_2* which allow spawning of a reverse shell [45] and bypass the firewall, respectively.

3.13 Analyzing the "magic packets" allowing to remotely spawn a reverse shell

When inspecting the traffic, this Netfilter hook checks that the protocol is IPv4, and the length of the header is between *0* and *9* DWORDs in size. After that, it checks whether the packet's length is *5* or *6* DWORDs. Based on these comparisons and also other subsequent checks on the fields of the packet, the instruction pointer will be able or not to reach a basic block that contains the call to the *spawn_reverse_shell* function. It is possible to reach the *spawn_reverse_shell* (see Fig. 14) function with IPv4 packets but different header lengths (*5* or *6* DWORDS). In our experiments, we used a *6* DWORDs IPv4 header length for targeting one of the two basic blocks that allow spawning of the reverse shell.

The reverse shell is not executed directly, as other Linux Kernel rootkits do (i.e., OSOM Rootkit, Reptile Rootkit, etc.), it prepares a work_struct [26] structure containing the parameters and the reverse shell function which will be finally executed by a dedicated kernel thread.

The reverse shell function is straightforward. It creates the string *CMKCBAA6780FD80=* plus the key as explained in Section 3.3. After that, it executes the reverse shell command in user space.

/bin/bash i >& /dev/tcp/IP_ADDRESS/PORT 0>&1

This one-line bash TCP reverse shell is not new at all, you can find it in many places [46].

As we already mentioned, there is more than one path for reaching the reverse shell function of the rootkit. Even if this is not the only possible solution, we suggest the following "magic packet" structure:

- IP protocol version: *0x4*
- Length of the IP header (specified in DWORDS): *0x6*
- Identification: *0x9C8A*
- Protocol: *0x6* (TCP Protocol)
- Options: *0xF7A9*
- Source Port: *0x1F49*
- Destination Port: *0x0080*
- Sequence number: Use it for passing the "magic packet" data.
- Flags: *0x2*

Using the previous packet structure as a template, triggering the reverse shell requires the following sequence of packets to be sent:

1. A packet containing the value *0x1A499C8A* in the Sequence number field such that the rootkit gets prepared for reading the IP and PORT next, at the offset *0x1A* (the Sequence number field).
2. A packet with the IP address where the reverse shell will try to connect in the Sequence number field of it.
3. A packet with the PORT where the reverse shell will try to connect in the Sequence number field followed by the value *0x9C8A*.
4. A packet containing the value *0x1F499C8A* in the Sequence number that will reach the basic block executing the reverse shell. Without the previous packets, the variables IP and PORT are not initialized such that the reverse shell will try to connect to an invalid socket: IP address *0.0.0.0* at port *0*.

Our implementation of this sequence of "magic packets" is publicly available [47] under the file name:
cmk_rootkit_magic_packet_reverse_shell.py

3.14 Analyzing the "magic packet" for bypassing the firewall

The reverse shell can bypass the local firewall (i.e., Iptables). A Netfilter hook registered by the rootkit returns *NF_STOLEN*, meaning that the reverse shell packet will be handled by the rootkit [48], preventing this way from being rejected by the local firewall. In case the packet does not need to be handled by the rootkit, it returns *NF_ACCEPT* (See Fig. 15).

An example of Iptables rule [49] that can be bypassed by the rootkit, assuming the port *1337* for the reverse shell, is the following:
iptables -A OUTPUT -p tcp –dport 1337 -j REJECT –reject-with tcp-reset

For reference, this technique is also implemented by the KoviD rootkit [50].

4 Possible solutions when extracting the "magic packet" requirements

Extracting the "magic packet" requirements consists of, given the graph representation of a program, extracting the requirements of the input for reaching a node or branch where the payload to trigger is present.

This is a well researched topic [51, 52, 53] but, for the specific case of the "magic packets", specific solutions can be more efficient due to the following reasons:

- Gaining internal information of the hooks installed in the system (i.e., *nf_hook_ops*[54] structure) aids in substantially reducing the input space.
- Luckily, the parser of the "magic packet" triggering the code is not obfuscated so, the checks on specific fields of the packet (i.ex. the IP protocol in case of IP packets) can be easily identified because those produce known assembly patterns that appear across different malware samples.

Next, let us discuss the specific problem of extracting the requirements of the "magic packets".

4.1 A pattern matching approach for extracting the "magic packet" requirements

Linux allows intercepting network packets in Kernel Mode via Netfilter hooks in a standard way. The hooks receive a *skbuff* [55] parameter, the main Linux networking structure representing a packet.

So, the requirements for the "magic packets" can be obtained from the following sources of information:

1. The *nf_hook_ops*[54] structure is passed to the appropriate kernel function for registering the Netfilter hooks. It contains relevant information, such as the hooked networking protocol. The "magic packet" networking protocol can be directly extracted from this structure.

2. The hook function. It receives the *skbuff* structure and performs the checks on the packet. The field's values on the "magic packets" can be extracted from those checks.

Notice that the function implementing the Netfilter hook can call other functions to perform the checks, so those must also be included in the analysis.

4.2 Figures and tables

launch

```
#!/bin/sh
if [ $# -ne 0 ]; then
    /usr/bin/pkill -STOP journal 2>/dev/null 1>&2
    /sbin/insmod /usr/share/man/B18D/cmk/module 2>/dev/null 1>&2
    /usr/bin/pkill -CONT journal 2>/dev/null 1>&2
else
    /usr/bin/systemd-hwdbsa
fi
```

Figure 1: CMK Rootkit launch script

```
.text:0000000000000050 mov     rdi, offset byte_300
.text:0000000000000057 mov     esi, offset unk_1FD78
.text:000000000000005C mov     r9d, 24924925h
```

Figure 2: CMK rootkit decryption routine parameters

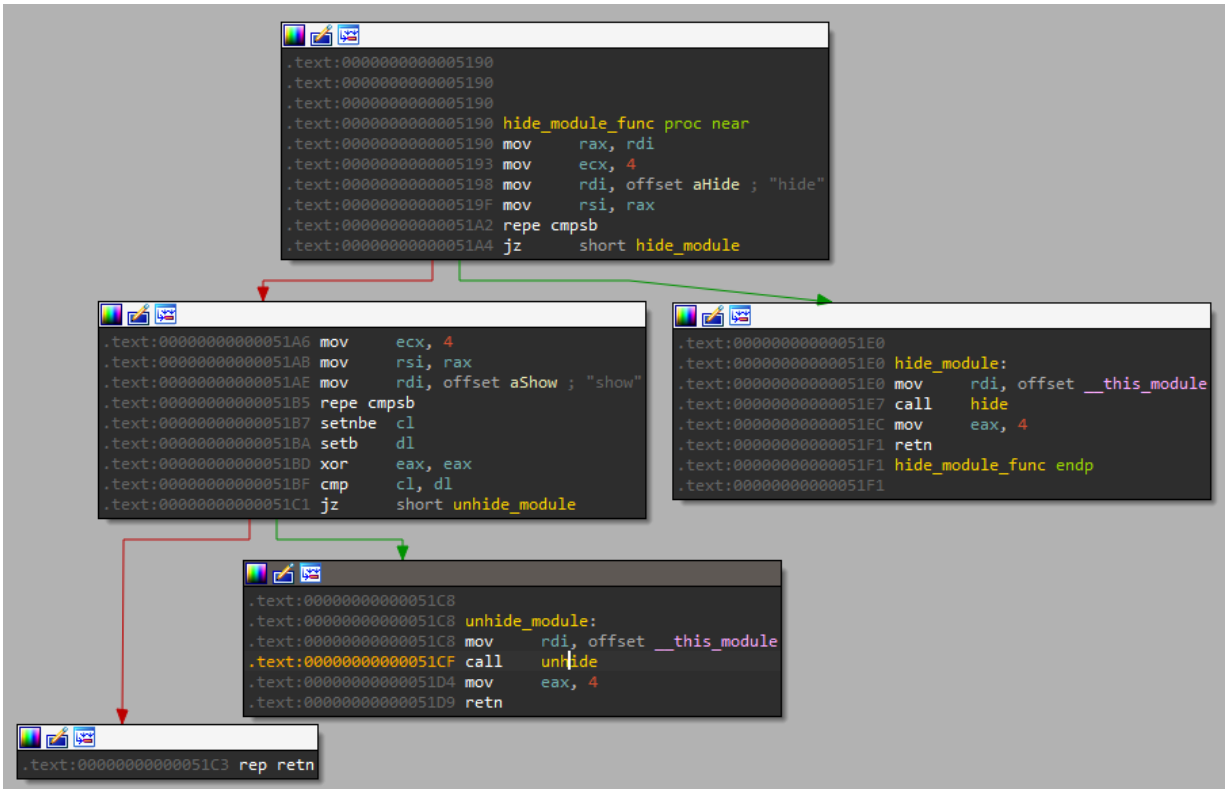

Figure 3: CMK rootkit hides itself from lsmod

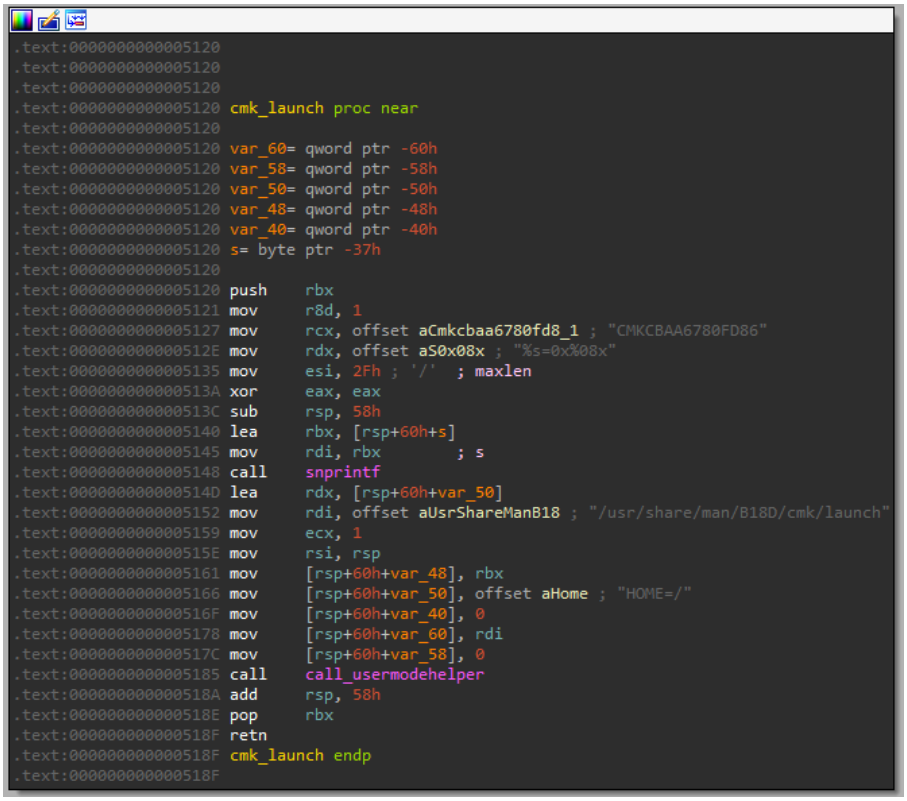

Figure 4: CMK Rootkit. Persistence and execution.

```
mov     r8d, 8A9C491Fh
push    rbx
mov     rcx, offset aCmkcbaa6780fd8_0 ; "CMKCBAA6780FD86"
mov     rbx, rdi
mov     rdx, offset format ; "%s=0x%08x"
```

Figure 5: CMK rootkit identify the processes that it creates

```
.rodata:0000000000008900 hidden_files    dq 300000000h            ; DATA XREF: sub_3290+9↑o
.rodata:0000000000008908                 dq offset aCmk           ; "cmk"
.rodata:0000000000008910                 dq 600000000h
.rodata:0000000000008918                 dq offset aCmkKo         ; "cmk.ko"
.rodata:0000000000008920                 dq 0E00000000h
.rodata:0000000000008928                 dq offset aSystemdHwdbsa ; "systemd-hwdbsa"
.rodata:0000000000008930                 dq 0C00000000h
.rodata:0000000000008938                 dq offset a99CmkRules    ; "99-cmk.rules"
.rodata:0000000000008938 _rodata         ends
.rodata:0000000000008938
```

Figure 6: CMK rootkit hide files and folders

Figure 7: CMK rootkit set unset invisible process

```
.data:000000000000A200 f107          dq offset aAccountProcess
.data:000000000000A200                                 ; DATA XREF: sub_4960↑w
.data:000000000000A200                               aAccountProcess db 'account_process_tick',0
.data:000000000000A200                                                 ; DATA XREF: .data:f107↓o
.data:000000000000A208               align 20h
.data:000000000000A220               dq offset f91
.data:000000000000A228               align 10h
.data:000000000000A230               dq offset sub_4960
.data:000000000000A238               align 20h
```

Figure 8: CMK rootkit hiding CPU usage

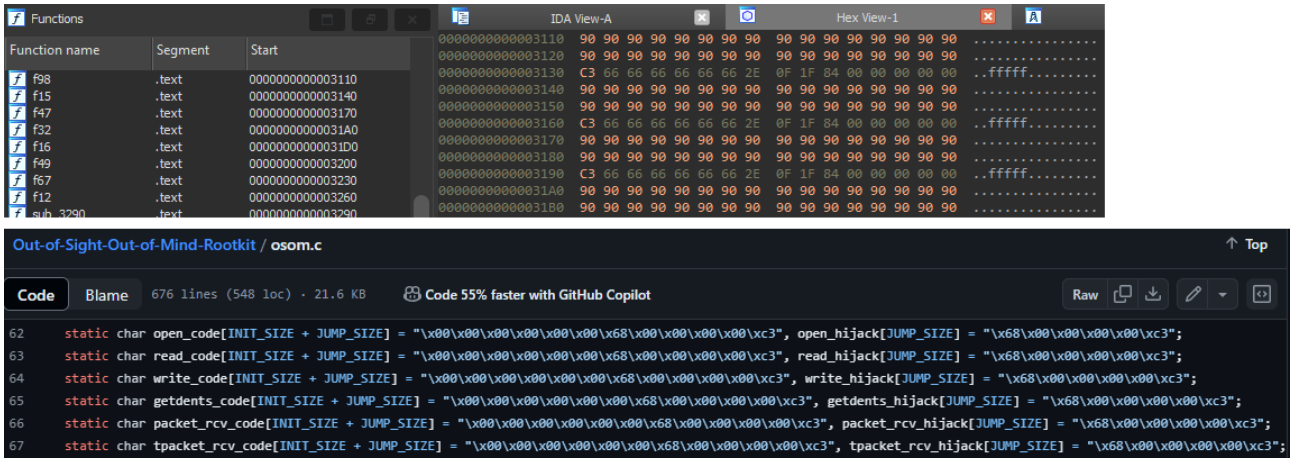

Figure 9: CMK Rootkit vs OSOM Rootkit. Trampolines.

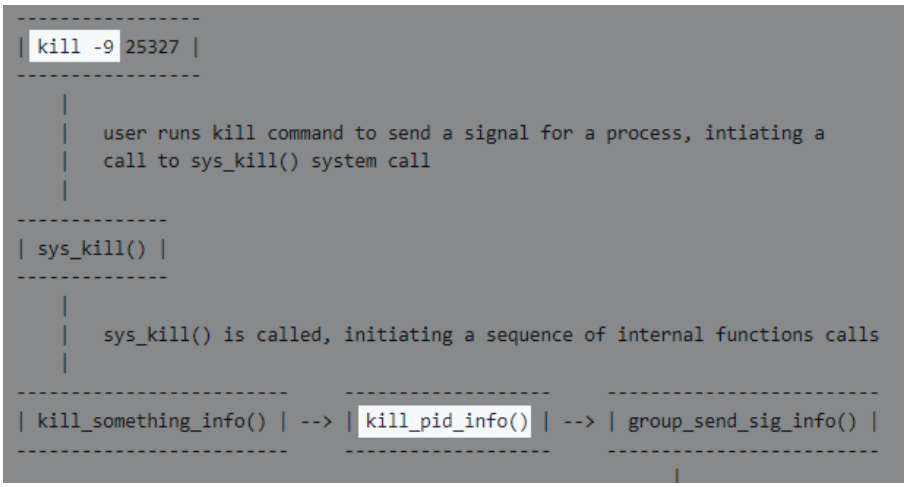

Figure 10: Unkillable process technique. Source: https://github.com/spiderpig1297/kunkillable

```
.text:0000000000004830
.text:0000000000004830
.text:0000000000004830
.text:0000000000004830 set_two_netfilter_hooks proc near
.text:0000000000004830 mov     esi, 2           ; size of array_of_hooks
.text:0000000000004835 mov     rdi, offset array_of_hooks
.text:000000000000483C jmp     nf_register_hooks
.text:000000000000483C set_two_netfilter_hooks endp
.text:000000000000483C
```

Figure 11: CMK rootkit "magic packets" hooks

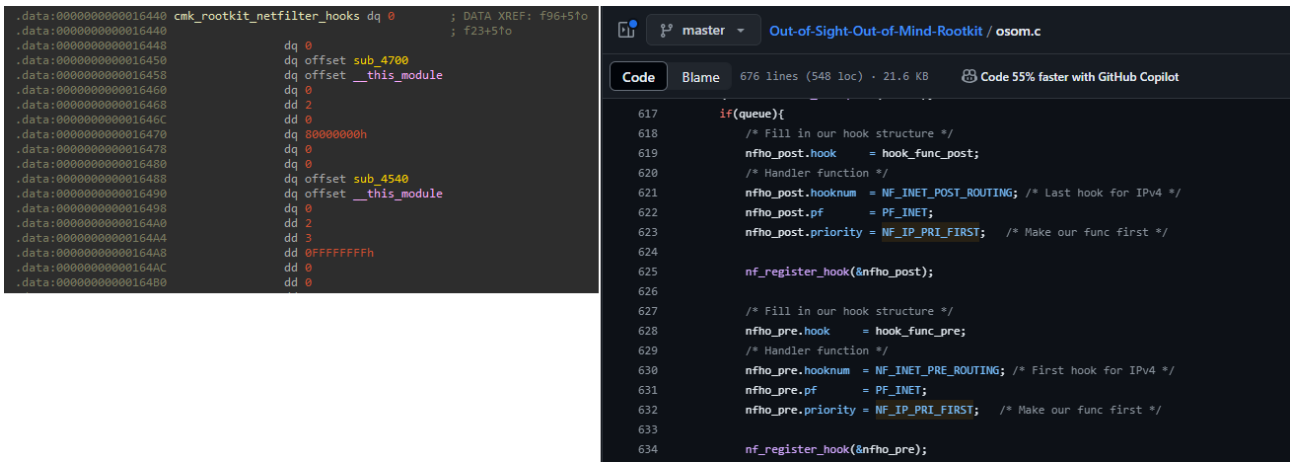

Figure 12: CMK rootkit vs OSOM rootkit. Netfilter hooks.

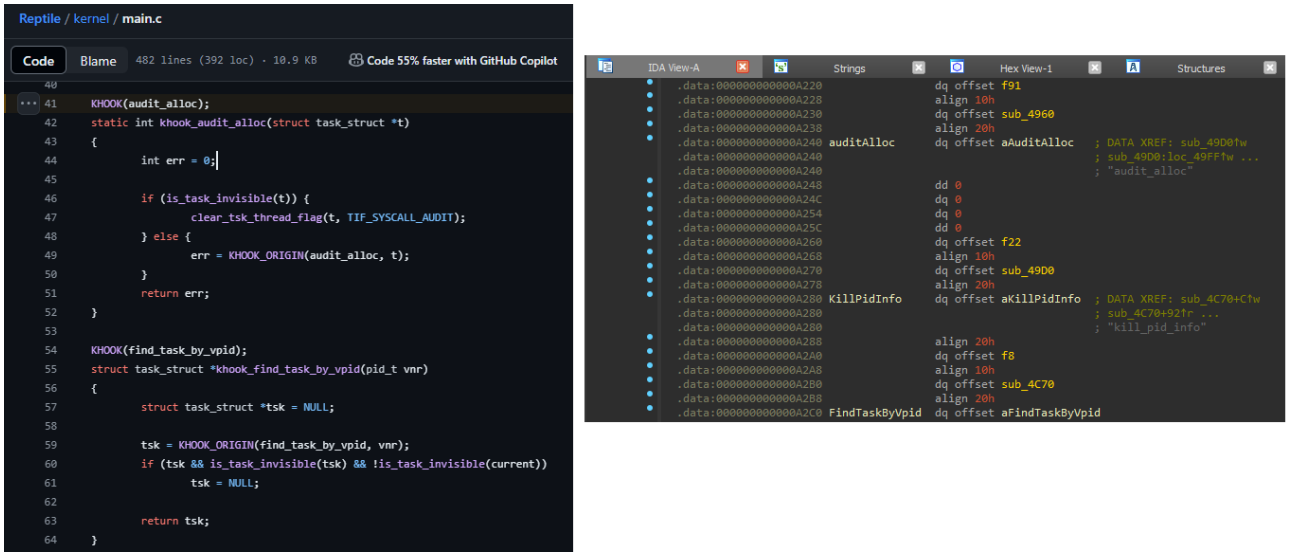

Figure 13: Reptile rootkit vs CMK rootkit. Hook on audit_alloc function.

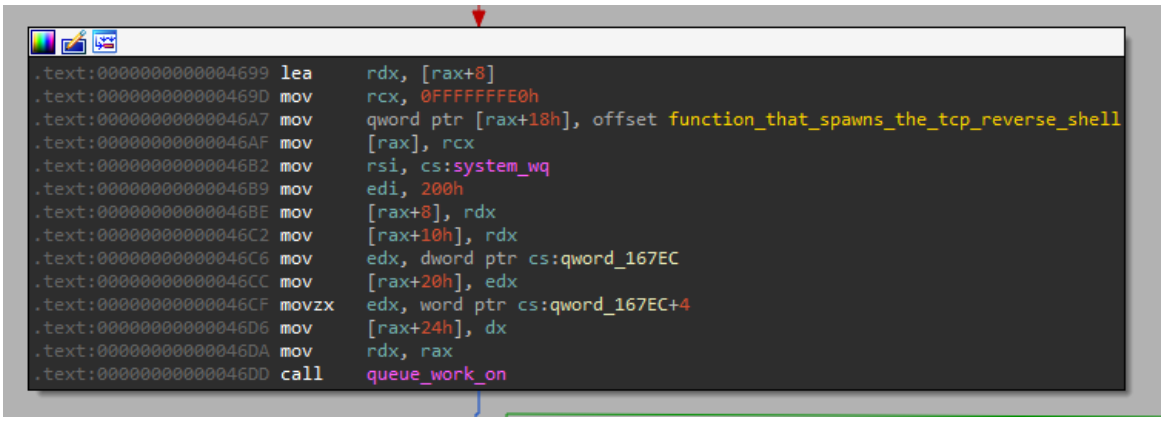

Figure 14: CMK rootkit spawn reverse shell work queue

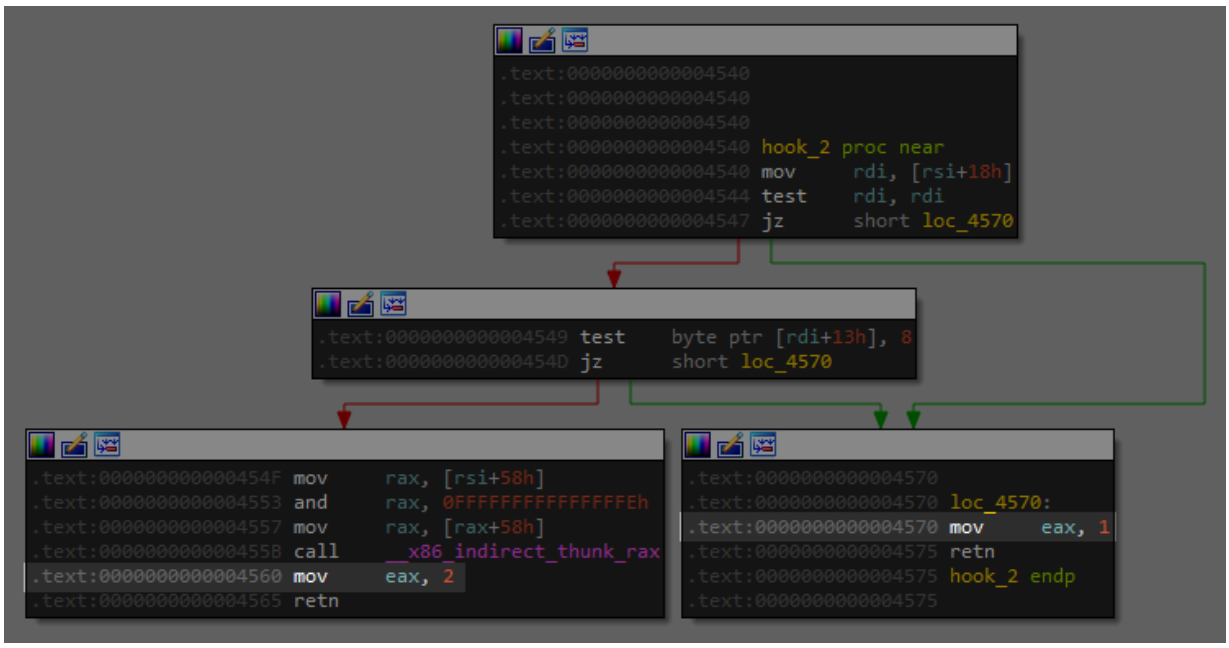

Figure 15: CMK rootkit bypass the firewall

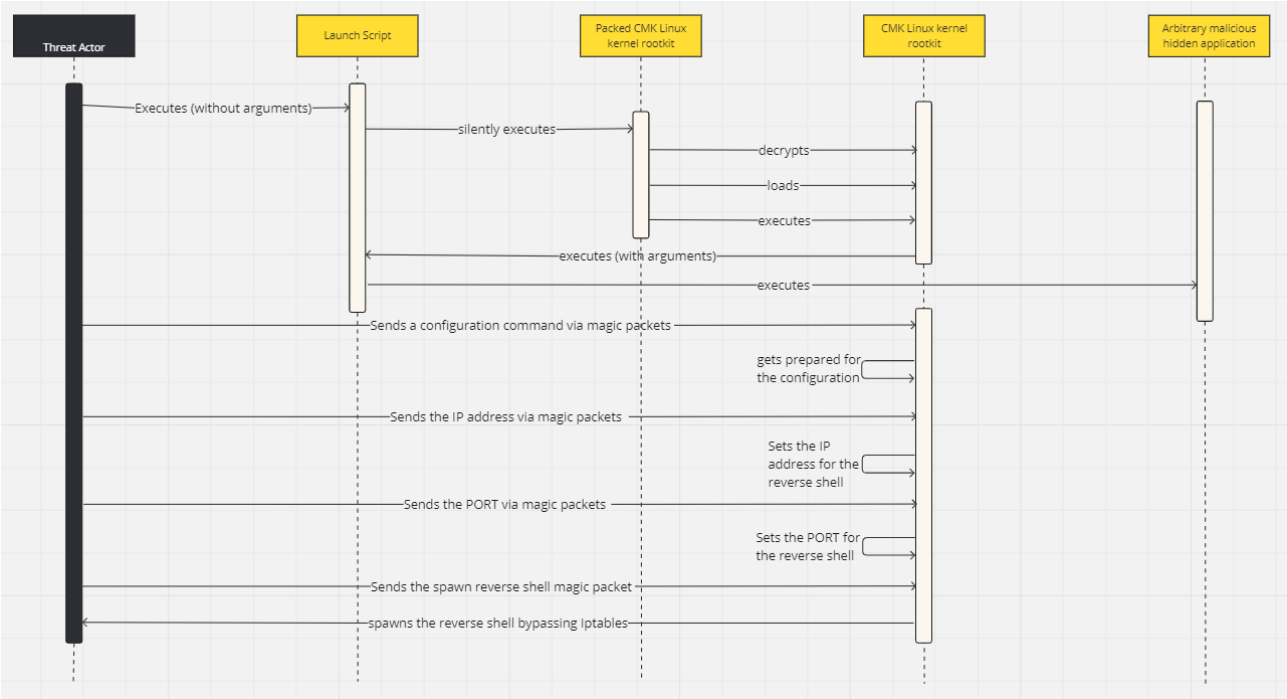

Figure 16: CMK rootkit. Sequence diagram

5 Implementation and evaluation

Our analysis for extracting the requirements of the "magic packets" is assisted by the following three tools are publicly available on Github [47]:

- The *magic_packets_analyzer.py* tool consist of a Ghidra script [56, 57, 58] that statically analyzes the target rootkit's Netfilter hooks (and the functions implementing those). Ghidra is a Software Reverse Engineering Framework created and maintained by the National Security Agency that includes disassembly, assembly, decompilation, graphing, and scripting, along with hundreds of other features.

  It searches for calls to the *nf_register_hook* and *nf_register_hooks* Linux kernel APIs in the target. If found, it analyzes its parameters (the Netfilter hook function, the network protocol being hooked, and so on). It extracts the potential values required for the magic packets (the Protocol Family, the constant values used in the hook function, and so on). The script prints all the relevant information extracted from the target during the static analysis (see Table 1). Finally, it automatically generates brute-forcing Python scripts and Linux kernel tracer modules based on KProbes, prepared with tracepoints at each basic block of the Netfilter hook functions (and other functions called from it).

- A set of brute-forcing Python scripts that aims to generate valid "magic packets". Those scripts are produced by the *magic_packets_analyzer.py* tool explained before. The brute force attempts are based on values extracted from static information of the target Linux kernel rootkit.

- A set of KProbes Linux Kernel modules prepared for tracing the "magic packet" brute-forcing process. These kernel modules are produced by the *magic_packets_analyzer.py* tool and use KProbes for tracing.

Our methodology is as follows. We automatically extract relevant information from the target (see Table 1) by using the *magic_packets_analyzer.py* tool (i.ex. information of the hooks that are relevant to the "magic packets" and the constant values that participate, directly or indirectly, in the comparisons of the packets fields).

We use brute-forcing Python scripts to try to reach the basic blocks that execute the payload of the "magic packets" while tracing the brute-forcing attempts with the KProbes Linux kernel modules (see Table 2).

After that, only a partial analysis of the target Linux Kernel rootkit is needed to produce the "magic packets". In the case of CMK Linux Kernel rootkit, we developed a Python script, publicly available on Github [47], that can spawn the reverse shell while bypassing the firewall via "magic packets".

6 Discussion

We fully analyzed CMK rootkit. A new Linux kernel rootkit that implements "magic packets" for configuring and then spawning a reverse shell bypassing the rules of the local firewall. We used structure parsing and pattern-matching [59] techniques for extracting the requirements of the "magic packets" in a semi-automatic way.

Even if it is still hard to reach the basic blocks that trigger the commands in user-mode space by simply running the scripts as is (see Table 2), those scripts covered all the Netfilter hooks present in the malware samples that were evaluated and chose the appropriate networking protocols when generating the template for the "magic packets".

7 Future work

Our pattern-matching approach substantially reduces the time spent extracting the "magic packets" requirements and writing the PoC (Proof of Concept) code.

- The current implementation of our tool successfully identifies the functions that handle the Netfilter hooks but it still fails to accurately determine the code responsible for the "magic packets" parsing. Our current implementation follows code called from the Netfilter hook function which is mostly prone to overestimation errors.

- We added support for a few x86 and x86-64 assembly language code patterns and supported only the most frequent networking protocols used for implementing the "magic packets": IP and TCP. Our implementation can be extended to cover more assembly language patterns and more networking protocols supported by Netfilter hooks.

- Kprobes [60, 61, 62] is a standard Linux Kernel tracer that can trace most of the kernel except itself. Unfortunately, we found that some dynamic calls introduced by the compiler appeared in the standard Kprobes blacklist [63] being unable to trace those.

8 Conclusion

We recently discovered the Syslogk Linux kernel rootkit, and now the CMK rootkit which also uses "magic packets" for executing the payload and went unnoticed for months in VirusTotal. Even if Linux kernel modules are harder to write than user-mode applications, threat actors can find a lot of code snippets of Linux kernel rootkits on the Internet which can be easily modified to speed up the malware development process and evade the detection.

This malware probably evaded the current detection mainly due to the packer layer. It is important to make an effort to efficiently detect all the techniques implemented by open-source kernel rootkits (i.e., Reptile Matroska), hunting new rootkit malware samples, etc. because it is expected to see, for instance, more packed Linux kernel rootkits in the future as code snippets are available on the internet.

Appendices

Appendix 1

CMK Rootkit. Script output.

```
magic_packets_analyzer.py> Running...

[*] Hook: DAT_00116440
   [-] Function: 0x104700
   [-] PF Number: 0x2
   [-] Hook Number: 0x0
   [-] Priority: -0x80000000
      [-] IP header pointed by: RSI
      [-] Protocols: set([6, 17])
         [-] TCP header pointed by : EDX
   [!] Bruteforce the IP header with the following values:
      set(['0x114840L', '0x18L', '0xc2L', '0x8L', '0x4L', '0x58L',
      '0xe28L', '0x30L', '-0x20L', '0x10L'])
   [!] Bruteforce the TCP header with the following values: set(['0x229L', '0xaL', '0x38L',
      '0x18L', '0x239L', '0x228L', '0x23bL', '0x22aL', '0x6L', '0x22cL', '0x8L', '0x23aL',
      '0x10a480L', '0x150L', '0x22fL', '0x2L', '0x4L', '0xeL', '0x168L'])
Generated:
C:\magic_packets\packet_9a1d46ef10e56195affd360c408e56e7.py
Generated:
C:\magic_packets\packet_480699c1caa3c49333829a51d3ee9e00.py

[*] Hook: None
   [-] Function: 0x104540
   [-] PF Number: 0x2
   [-] Hook Number: 0x3
   [-] Priority: -0x1
         [-] TCP header pointed by : EDX
   [!] Bruteforce the IP header with the following values: set(['0x248L', '0x228L', '0x22aL',
      '0x8L', '0x23aL', '0x170L', '0x150L', '-0x18L', '0x80L', '0x24L', '0x158L', '0x168L',
      '-0x20L', '0x20L', '0x162L', '0x10L', '0x198L', '0x38L', '0x28L', '0x18L', '0x239L',
      '0x23bL', '0x3f0L', '0x9L', '0x1c8L', '0x4L', '0x250L'])
   [!] Bruteforce the TCP header with the following values: set(['0x229L', '0xaL', '0x38L',
      '0x18L', '0x239L', '0x228L', '0x23bL', '0x22aL', '0x6L', '0x22cL', '0x8L', '0x23aL',
      '0x10a480L', '0x150L', '0x22fL', '0x2L', '0x4L', '0xeL', '0x168L'])
Generated:
C:\magic_packets\packet_78d9d5ba87dc6b9008eb0bfee4703bf5.py
Generated:
C:\magic_packets\packet_536e27cf7e99e992e59006c5ef7e7d46.py

magic_packets_analyzer.py> Finished!
```

Table 1: Ghidra script output. Static analysis on CMK Rootkit.

Appendix 2

CMK Rootkit dynamic results.

Hook Function	id	Autogenerated Test script identifier	Tracepoint addresses	Tracepoints Reached (id)
0x104700	a	9a1d46ef10e56195affd360c408e56e7	0x10471d	a,b
	b	480699c1caa3c49333829a51d3ee9e00	0x10471b	a,b
			0x104730	a,b
0x104540	a	78d9d5ba87dc6b9008eb0bfee4703bf5	0x10454f	Not reached
	b	536e27cf7e99e992e59006c5ef7e7d46	0x10454d	Not reached
			0x104549	Not reached
			0x104547	b
			0x10455b	Not reached
			0x1170b8	Not reached
			0x1045b1	Not reached
			0x117010	Not reached
			0x104570	a,b

Table 2: Dynamic results. Experiment on CMK Rootkit

Original Kernel API	Trampoline start-end offsets (Virtual Address)	Hijack (Virtual Address)	Purpose
exit_creds	0x0030 - 0x0050	0x01B0	Hide process
copy_creds	0x0000 - 0x0020	0x00A0	Hide process
user_path_at_empty	0x3260 - 0x3280	0x3560	Hide files/folders
compat_fillonedir	0x3230 - 0x3250	0x38C0	Hide files/folders
compat_filldir64	0x3200 - 0x3220	0x3A80	Hide files/folders
compat_filldir	0x31D0 - 0x31F0	0x3700	Hide files/folders
fillonedir	0x31A0 - 0x31C0	0x39A0	Hide files/folders
filldir64	0x3170 - 0x3190	0x37E0	Hide files/folders
filldir	0x3140 - 0x3160	0x3620	Hide files/folders
do_sys_open	0x3110 - 0x3130	0x34B0	Hide files/folders
inet_stream_connect	0x3D90 - 0x3DB0	0x4340	Hide network traffic
inet_release	0x3D60 - 0x3D80	0x4260	Hide network traffic
inet_bind	0x3D30 - 0x3D50	0x4140	Hide network traffic
inet_diag_bc_sk	0x3D00 - 0x3D20	0x4030	Hide network traffic
udp6_seq_show	0x3CD0 - 0x3CF0	0x3F10	Hide network traffic
tcp6_seq_show	0x3CA0 - 0x3CC0	0x4000	Hide network traffic
raw6_seq_show	0x3C70 - 0x3C90	0x3EE0	Hide network traffic
udp4_seq_show	0x3C40 - 0x3C60	0x3EB0	Hide network traffic
tcp4_seq_show	0x3C10 - 0x3C30	0x3FD0	Hide network traffic
raw_seq_show	0x3BE0 - 0x3C00	0x3E80	Hide network traffic
tcp_time_wait	0x3BB0 - 0x3BD0	0x3DC0	Hide network traffic
sockfd_lookup_light	0x3B80 - 0x3BA0	0x40B0	Hide network traffic
load_elf_binary	0x4420 - 0x4440	0x4450	Intercept ELF execution
account_process_tick	0x4930 - 0x4950	0x4960	Hide CPU usage
audit_alloc	0x4900 - 0x4920	0x49D0	Hide from kernel audit
kill_pid_info	0x48D0 - 0x48F0	0x4C70	Process kill
find_task_by_vpid	0x48A0 - 0x48C0	0x4BA0	Hide process
next_tgid	0x4870 - 0x4890	0x4A60	Hide process
do_syslog	0x4DD0 - 0x4DF0	0x4F80	Hide from kernel audit
devkmsg_read	0x4DA0 - 0x4DC0	0x4F10	Hide from kernel audit
comm_write	0x4FD0 - 0x4FF0	0x5200	Hide from kernel audit

Table 3: Linux OS functions hijacked by CMK Rootkit.

Appendix 3

Indicators of Compromise (IoC's).

- CMK Rootkit launch script
  - 3E75B22524C41083AEA9BD6CFCC222A171CD8A9817135940A1B49E2ACEE7E66D
- CMK Rootkit (packed form)
  - 54D8B09FFC15C657ABF29A0C313B377DF64988848F2C3814243B2478B4B881CC
- CMK Rootkit (unpacked by us)
  - 83F32F8330026F7CD26E9CF516C2980C1059262B1902D2809CDFFAF771BD2AFB
- TAR archive file containing both the launch script and the CMK rootkit
  - 830B41D30241453EAA1E22CE8E076EC4A0BDA3D70AC86FC84FDF82BCF002ABBE

Acknowledgment: This work was partially supported by GenDigital Inc. through the investigations that were made during the reverse engineering process of CMK Rootkit.

Author details

David Álvarez-Pérez

Gen Digital Inc.
Pikrtova 1737/1a, 140 00 Praha 4, Czech Republic.
David.AlvarezPerez@gendigital.com

Manuel Fernández-Veiga

atlanTTic, Universidade de Vigo
Campus de Vigo As Lagoas, Escola de Enxeñaría de Telecomunicación, Rúa Maxwell, s/n, 36310 Marcosende, Pontevedra
mveiga@det.uvigo.es

References

[1] "Reptile Rootkit. github repository of the project." https://github.com/f0rb1dd3n/Reptile.

[2] "Rootkit. nist glossary of terms.." https://csrc.nist.gov/glossary/term/rootkit.

[3] "Rootkits. enisa incident response glossary of terms.." https://www.enisa.europa.eu/topics/incident-response/glossary/rootkits.

[4] "Reptile Rootkit. 36/61 av engine detectionsw in virustotal." https://www.virustotal.com/gui/file/cbe9107185c8e42140dbd1294d8c20849134dd122cc64348f1bfcc90401379ec/detection. Accessed: 2024-01-07.

[5] "Virustotal." https://www.virustotal.com/.

[6] T. Brosch and M. Morgenstern, "Runtime packers: The hidden problem," *Black Hat USA*, 2006.

[7] "CMK Rootit detections (10/12/2022)." https://www.virustotal.com/gui/file/54d8b09ffc15c657abf29a0c313b377df64988848f2c3814243b2478b4b881cc/detection.

[8] D. Alvarez-Perez, "Syslogk Rootkit. executing bots via "magic packets"," *The Journal on Cybercrime & Digital Investigations*, vol. 8, 2023.

[9] "Linux Threat Hunting 'syslogk' a kernel rootkit found under development in the wild." https://decoded.avast.io/davidalvarez/linux-threat-hunting-syslogk-a-kernel-rootkit-found-under-development-in-the-wild/.

[10] Y. Dong, Z. Li, Y. Tian, C. Sun, M. W. Godfrey, and M. Nagappan, "Bash in the wild: Language usage, code smells, and bugs," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, feb 2023.

[11] "systemd-hwdb. linux hardware database management tool." https://man7.org/linux/man-pages/man8/systemd-hwdb.8.html.

[12] "pkill user command. oracle solaris 11.2 documentation." https://docs.oracle.com/cd/E36784_01/html/E36870/pkill-1.html.

[13] "systemd-journald.service linux manual page." https://man7.org/linux/man-pages/man8/systemd-journald.service.8.html.

[14] "insmod linux manual page.." https://man7.org/linux/man-pages/man8/insmod.8.html.

[15] "Linux Kernel. reproducible builds.." https://docs.kernel.org/kbuild/reproducible-builds.html.

[16] "R. Love, Linux Kernel Development: Linux kernel development _p3. pearson education, 2010.."

[17] "Debian 8.11 Jessie. press release." https://www.debian.org/News/2015/20150426.

[18] A. Devi and G. Aggarwal, "Manual unpacking of upx packed executable using ollydbg and importrec," *IOSR Journal of Computer Engineering*, vol. 16, no. 1, pp. 71–77, 2014.

[19] F. Desclaux, "Miasm: Framework de reverse engineering," *Actes du SSTIC. SSTIC*, 2012.

[20] "MIASM reverse engineering framework." https://github.com/cea-sec/miasm.

[21] P. C. T. Đang, M. S. Phan, and M. H. Phan, *Building a Binary De-obfuscation Tool with Miasm Framework*. PhD thesis, 2021.

[22] "sysinitmodule linux documentation." https://man7.org/linux/man-pages/man2/init_module.2.html.

[23] "kallsymsoneachsymbol linux kernel documentation." https://lore.kernel.org/all/20200221114404.14641-1-will@kernel.org/#rl.

[24] R. Rosen and R. Rosen, "Netfilter," *Linux Kernel Networking: Implementation and Theory*, pp. 247–278, 2014.

[25] J. Engelhardt and N. Bouliane, "Writing netfilter modules," *Revised, February*, vol. 7, 2011.

[26] "Linux Kernel documentation. workqueues and kevents.." https://www.kernel.org/doc/html/v4.13/driver-api/basics.html#workqueues-and-kevents.

[27] "Out-of-sight-out-of-mind-rootkit linux kernel rootkit.." https://github.com/NinnOgTonic/Out-of-Sight-Out-of-Mind-Rootkit/blob/master/osom.c#L211. Accessed: 2023-02-26.

[28] J. Magnusson, "Survey and analysis of dns filtering components," *arXiv preprint arXiv:2401.03864*, 2024.

[29] J. Junnila, "Effectiveness of linux rootkit detection tools," Master's thesis, J. Junnila, 2020.

[30] "Nick Newson Linux Kernel Rootkit." https://github.com/nnewson/km/blob/master/src/rootkit.c.

[31] "Linux Kernel documentation, sysfs_unlink_sibling." https://lore.kernel.org/netdev/m1lk0iihhp.fsf_-_@frodo.ebiederm.org/raw.

[32] "Linux Kernel documentation, sysfs_link_sibling." https://lkml.kernel.org/netdev/20091029233848.GV3141@kvack.org/.

[33] "Linux Kernel documentation, kernfs_link_sibling." https://lore.kernel.org/lkml/747aee3255e7a07168557f29ad962e34e9cb964b.camel@themaw.net/.

[34] "snprintf, _snprintf, _snprintf_l, _snwprintf, _snwprintf_l microsoft documentation." https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/snprintf-snprintf-snprintf-l-snwprintf-snwprintf-l?view=msvc-170.

[35] "Linux Kernel documentation, call_usermodehelper." https://archive.kernel.org/oldlinux/htmldocs/kernel-api/API-call-usermodehelper.html.

[36] D. Andriesse, *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. no starch press, 2018.

[37] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: system programming guide, Part*, vol. 2, no. 11, pp. 1–64, 2011.

[38] "Linux Kernel documentation. filldir.." https://lore.kernel.org/lkml/lsq.1578512578.759211401@decadent.org.uk/.

[39] "The Linux Documentation Project. chapter 8. system calls.." https://tldp.org/LDP/lkmpg/2.4/html/c937.htm.

[40] "account_process_tick rootkits search on github." https://github.com/search?q=account_process_tick+rootkit&type=code.

[41] "KUnkillable github repository." https://github.com/spiderpig1297/kunkillable.

[42] I. A. Ilya V. Matveychikov, "Linux kernel rootkits advanced techniques," 2018.

[43] "A collection of Linux kernel rootkits. linux-rootkits.." https://github.com/R3x/linux-rootkits.

[44] R. Rosen, *Linux kernel networking: Implementation and theory*. Apress, 2014.

[45] E. Eliando and A. B. Warsito, "Lockbit black ransomware on reverse shell: Analysis of infection," *CogITo Smart Journal*, vol. 9, no. 2, pp. 228–240, 2023.

[46] H. Sharma and H. Singh, *Hands-on red team tactics: a practical guide to mastering red team operations*. Packt Publishing Ltd, 2018.

[47] "CMK_Linux_Kernel_Rootkit cmk linux kernel rootkit - research tools." https://github.com/dalvarezperez/CMK_Linux_Kernel_Rootkit.

[48] K. Thang and A. Nyberg, "Impact of fixed-rate fingerprinting defense on cloud gaming experience," 2023.

[49] N. Gandotra and L. S. Sharma, "Exploring the use of iptables as an application layer firewall," *Journal of The Institution of Engineers (India): Series B*, vol. 101, pp. 707–715, 2020.

[50] "KoviD github repository."

[51] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, pp. 8–9, IEEE, 2017.

[52] "Angr open-source binary analysis platform for python." https://angr.io/.

[53] X. Zhang, C. Zhang, X. Li, Z. Du, Y. Li, Y. Zheng, Y. Li, B. Mao, Y. Liu, and R. H. Deng, "A survey of protocol fuzzing," *arXiv preprint arXiv:2401.01568*, 2024.

[54] B. T. ODEHNAL, "Port block allocation for network address translation,"

[55] "Linux Kernel documentation. skbuff.." https://docs.kernel.org/networking/skbuff.html.

[56] A. David, *Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems*. Packt Publishing Ltd, 2021.

[57] C. Eagle and K. Nance, *The Ghidra Book: The Definitive Guide*. no starch press, 2020.

[58] "Ghidra a software reverse engineering (sre) suite of tools developed by nsa's research directorate in support of the cybersecurity mission." https://ghidra-sre.org/.

[59] T. Æ. Mogensen, "Machine-code generation," in *Introduction to Compiler Design*, pp. 161–172, Springer, 2024.

[60] "Kernel Probes documentation.." https://docs.kernel.org/trace/kprobes.html. Accessed: 2023-02-26.

[61] "R. Krishnakumar, Kernel korner: kprobes-a kernel debugger, linux journal, vol. 2005, no. 133, p. 11, 2005.."

[62] "Spotify KProbes examples linux kernel module.." https://github.com/spotify/linux/tree/master/samples/kprobes. accessed: 2023-02-26.

[63] "Kprobes. blacklisted functions." https://docs.kernel.org/trace/kprobes.html#kprobes-blacklist. Accessed: 2024-01-07.