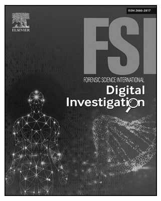
Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi



DFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA

Detecting hidden kernel modules in memory snapshots[☆,☆☆]

Roland Nagy

CrySyS Lab, Budapest University of Technology and Economics, Müegyetem rkp., Budapest, 1111, Hungary

ARTICLE INFO

Keywords:
Kernel rootkit
Memory forensics
Kernel module
LKM

ABSTRACT

Rootkit infections have plagued IT systems for several decades now. As non-trivial threats often employed by sophisticated adversaries, rootkits have received a large amount of attention, from both the industrial and academic communities. Consequently, rootkit detection has a rich literature, but most papers focus on only detecting the fact that an infection happened. They rarely offer mitigation, let alone identifying the piece of malware. We aim to solve this by not only detecting rootkit infections but by finding the malware as well. Our paper has three main goals: extend the state of the art of cross-view-based detection of Loadable Kernel Modules (the de-facto delivery method of Linux kernel rootkits), provide a memory forensics tool that implements our detection method and enables further investigation of loaded modules, and publish the dataset we used to evaluate our solution. We implemented our tool in the form of a Volatility plugin and compared it to the already existing module detection capability of Volatility. We tested them on 55 rootkit-infected memory dumps, covering 27 different versions of the Linux kernel. We also provide compatibility analysis with different kernel versions, ranging from the initial release to the latest (6.13, at the time of writing).

1. Introduction

Malware infection is a longstanding problem that affects almost every facet of today's interconnected, global IT infrastructure. The first computer viruses were experiments and pranks, but they quickly grew destructive, and in time, criminals found a way to monetize on malware-infected machines (Liao et al. (2016)).

Apart from a few notable exceptions, like ransomware infections, it is in the best interest of malware to remain hidden, and thus operational as long as possible. To avoid detection, many malware samples employ simple tricks, like using packers or renaming processes (Cozzi et al. (2018)). There is a type of malware, however, that takes hiding to another level and tries to achieve total invisibility: this subgroup of malware is called rootkits, and they often put a considerable amount of effort into hiding. They usually hide different resources associated with malware components, like running processes, files, or open network connections.

As rootkits evolved, so did the tools that aimed to detect them. The challenge was twofold: any detection solution must not only outsmart its target but must be able to protect itself from the rootkits it is hunting for. The complexity of these challenges comes from the advanced techniques, that rootkits apply, often in the deep layers of the operating system, and the total control they have over the infected machine.

Many rootkit detection solutions published in the past decades suffer from the same deficiency: they identify rootkit infections, typically by detecting traces of some technique applied by rootkits. However, they cannot take actions to mitigate the issue or identify the rootkits themselves. This limits their usefulness as parts of antivirus solutions, and even more so in the field of digital forensics. During an investigation, if the suspicion of rootkit infection arises, the rootkit should be found and analyzed, before the case can be closed.

This paper addresses this issue by providing forensics experts with a tool that is capable of finding hidden kernel modules in memory dumps of Linux machines. Kernel modules are the most popular and convenient way to implement rootkit functionality (Li et al. 2015). Our[1] solution falls into the category of cross-view-based (Nadim et al. (2023)) detection, where information is collected from different data sources, and these sources are cross-referenced with each other, to find hidden resources by identifying inconsistencies. We also address a shortcoming of cross-view-based detection: when a rootkit does not try to hide its module, cross-view-based methods will not flag it as suspicious. For this, we offer a tool that can be used to identify suspicious modules among the

☆ Code: https://github.com/CrySyS/ModXRef.☆☆ Data: https://www.crysys.hu/rnagy/datasets/rootkit.html.
E-mail address: rnagy@crysys.hu.

[1] Throughout this paper, first person plural was used, as it is customary in the field of computer science, despite of the paper having a sole author. The author worked alone on the paper, all colleagues who helped with proofreading the manuscript are listed in the Acknowledgement Section.

https://doi.org/10.1016/j.fsidi.2025.301928

benign ones and conduct a deeper investigation, to determine their maliciousness.

The main contributions of our paper are the following:

- We extend the capabilities of cross-view-based detection, regarding Loadable Kernel Modules. The method we propose works across a wide range of Linux kernel versions and surpasses the hiding capabilities of any LKM rootkit we encountered so far. It was tested against a large number of rootkits (both open-source and found in the wild), and it proved to be robust against all the techniques that rootkits apply to hide modules. We give a detailed description of this method in Section 3.
- We address a major shortcoming of cross-view-based rootkit detection: if a certain resource is not hidden, despite being able to find it, cross-view-based detection cannot flag it as suspicious. To overcome this issue, our Volatility plugin also reports a wide variety of information about the modules it finds and facilitates their deeper investigation. To demonstrate this, in Section 4, we present a case study about finding the THOR open-source rootkit.
- To test our Volatility plugin and compare its effectiveness against other, already existing solutions, we collected and compiled many rootkits and infected Linux machines with them, to acquire memory dumps. Our dataset consists of 35 memory dumps infected with open-source rootkits, and another 20 infected by rootkits we acquired from VirusTotal. We make this dataset publicly available (both the memory dumps and their corresponding symbol tables), because we believe it might be useful for the community to evaluate rootkit detection solutions.

The rest of this paper is organized the following way: in Section 2, we provide some background on rootkits, the different approaches of rootkit detection, and position our solution compared to other cross-view-based solutions that detect hidden modules. In Section 3, we give a detailed description of our approach and the data sources it uses. In Section 4 we present a case study about identifying and analyzing modules, that cross-view-based detection does not flag as suspicious. Section 5 details how we evaluated our detection method. In Section 6, we discuss the limitations of our solution and provide a compatibility analysis between our method and different versions of the Linux kernel, and finally, in Section 7, we conclude our paper.

2. Background and related works

2.1. Rootkits

Rootkits come in two flavors depending on where they operate in the software stack.

User-space rootkits, as the name suggests, stay in the user space; they often patch existing system administration tools, hook library functions, or abuse certain features of the operating system. They can hide certain resources from the users or system administrators, but not from a forensics expert, inspecting a memory dump of the infected system.

Kernel-space rootkits attack different layers of the kernel to hide certain resources and apply a wide range of techniques. The feasibility of detection depends on the applied techniques, but generally speaking, the deeper the rootkit goes into the layers of the kernel, the harder it is to detect its traces.

In this paper, we only focus on kernel-space rootkits, developed to the Linux kernel. Similar techniques might be applicable to other operating systems as well, but these are out of the scope of our research.

2.2. Rootkit attack vectors

First, we must discuss how kernel-space rootkits can be implemented on Linux. To be able to implement such functionality, an attacker must be able to execute code in the context of the kernel, or the kernel's memory must be modified (Sd, 2001).

Reading and writing the kernel memory is possible through the file `/dev/kmem`, but since no legitimate application uses it, only attackers, it is often disabled on modern versions of popular Linux distributions. Another similar file is `/dev/mem`, which provides access to the physical memory, but due to frequent abuse, it is locked as well.[2] Only certain parts of the physical memory are accessible this way, thus rootkits can no longer modify the kernel memory using these files.

Yet, executing code in kernel space is still possible: if, for example, the attacker finds a memory corruption vulnerability, it might be possible to divert the execution and use Return-Oriented Programming (ROP) to implement arbitrary functionality (Roemer et al. (2012)).

A much more convenient way is to use Loadable Kernel Modules (LKMs). These can be loaded into or unloaded from the running kernel at any time. Many drivers are implemented in this way, which allows the kernel to be built into a smaller binary file, and not loading unnecessary drivers also reduces its attack surface. This is by far the most popular and convenient way to implement kernel rootkit functionality, and as an act of self-preservation, many rootkits attempt to hide the loaded module in which they are implemented.

Another possibility is abusing the extended Berkeley Packet Filter (eBPF). This is a relatively new trend among kernel rootkits. Using eBPF, it is possible to inject bytecode into the kernel memory, which will be executed when triggered by any tracing subsystem of the kernel. This allows the implementation of rootkit-like functionality similarly to earlier, hooking-based solutions, but without using a kernel module.

Once a rootkit is capable of performing modifications in the memory of the kernel, it may begin its operation. Many techniques exist to hide certain resources, but most of them can be divided into two categories:

Function hooking happens when a rootkit can modify the kernel in such a way, that instead of the original function, the attackers implementation will be executed. For example, the `read` system call can be hooked, so when it is called on a specific file, certain lines can be omitted (e.g. by doing this with the file `/proc/modules`, certain modules can be made invisible).

Direct Kernel Object Manipulation (DKOM)[3] is a technique where data structures are modified in the kernel memory to achieve a certain goal. For example, if a list is used to store information about a set of resources, removing an element from the list will effectively hide a certain instance of said resource from all parties that rely on that specific list. An example could be the list of modules, that is used to generate the content of the file `/proc/modules`; removing a module from this list is a simple and widely used technique to hide modules.

2.3. Rootkit detection

Since rootkits are not a new problem, and their detection has a considerable literature, naturally, multiple approaches were explored.

Signature-based rootkit detection works just like detecting any other malware using signatures: signatures or fingerprints are extracted from known samples and a database is constructed from them. This can later be used to detect known threats (e.g Yamauchi and Akao (2017)).

Behavior-based detection looks for abnormal behavior or other anomalies, that might be caused by a rootkit infection. Such anomalies can include timing discrepancies, unusual memory access patterns, and many more. This approach relies on a priori measurements and might not be accurate enough outside a controlled environment (e.g Li et al. (2019)).

Integrity-based detection can be a powerful approach against rootkits that modify static sections of memory, but it typically requires a priori knowledge about the protected system (e.g. Deyannis et al. (2020)).

[2] https://lwn.net/Articles/267427/ (Last visited: 2024.10.10).

[3] In this case, the term "kernel object" refers to any C struct in the kernel memory, not just `kobject`s, which we will detail in Section 3.2.

Cross-view-based solutions work by collecting information about the same set of objects, from different sources. These different sources are cross-referenced to identify inconsistencies, most likely caused by a rootkit infection. It can detect DKOM attacks effectively, but it has limitations: it cannot detect resources, that are associated with the rootkit, but were not hidden. Also, if a certain resource is removed from all possible data sources, that the detector scans, it will not be able to detect the rootkit. This approach was taken by Wang et al. (2005), Jones et al. (2008), Xu and Jiang (2011) and our solution belongs in this category as well.

2.4. *State of the art*

Cross-view-based detection is a well-established method to detect rootkits, however, applying it to find kernel modules is less common, as most solutions focus on other resources, like hidden processes and files.

The kernel communicates information to the user space about modules in two ways: through a file, named `/proc/modules`, whose content is populated by iterating a list inside kernel memory (commonly referred to as “module list”), and via a directory named `/sys/modules`. In this directory, every module has its own subdirectory, and these are created by traversing a list called `module_kset`. We will give detailed descriptions of these data structures and Section 3, but knowing about them is necessary in order to compare the already proposed solutions. The `lsmod` utility, which we use to list loaded modules, processes the content of `/proc/modules`, and consults `/sys/modules` for additional information (i.e. modules missing from `/proc/modules` will not be present in the output of `lsmod`).

The weakest among cross-view-based module detectors are solutions that compare data from the same source but on different architectural levels. `rkhunter` (Boelen, Michael (2003)), for example, compares the output of `lsmod` to the content of `/proc/modules`, making it capable of detecting attacks in the user space only. Quynh and Takefuji (2007) apply a similar approach: their solution was implemented by the Xen hypervisor, and it compares the output of `lsmod` and the content of the module list in the kernel memory. This means that it can detect modules hidden by hooking certain kernel functions, but not the ones that remove their modules from the module list.

Other solutions rely on artificial data sources: Rhee et al. (2010) track allocation and de-allocation events inside the kernel to maintain a shadow copy of the module list, which they can periodically compare to the real module list, while Lu et al. (2023) use Kprobe[4] technology, a built-in way of tracing inside the Linux kernel to monitor lists in the memory. They also watch certain system calls, like the ones responsible for initializing and removing kernel modules, and raise an alarm, if a module was removed from the list of modules without the invocation of the `delete_module` syscall. These approaches might be robust against rootkits if implemented properly, but they must be deployed proactively, making them unsuitable for detecting rootkits in a typical memory forensics scenario.

Of course, not only artificially crafted data sources can be used for cross-view-based detection: `modreveal` (Lihi, Jafar (2023)), for example, compares the output of `lsmod` to the list of modules in the `kset`, which it acquires through a kernel-space component. This technique is closely related to the one implemented by Volatility (Volatility Foundation), where the content of the `kset` is compared to the content of the module list, both directly parsed from the kernel memory. Both versions of Volatility implement this functionality, in plugins called `linux_check_modules` at version 2, and `linux.check_modules` at version 3. This approach can detect if a module is removed from the module list only, but as Appendix A shows, many rootkits can evade this check by tampering with the `kset` as well.

Two solutions that stand out in terms of detection capability, are `tracee` (Aqua Security (2020)) and SigGraph (Xu and Jiang (2011)). `tracee` is an open-source project, that utilizes eBPF to monitor various security-related events throughout the Linux kernel. It is capable of capturing module load and unload events, and dumping the module to the disk, while it also claims to be able to detect hidden modules. The documentation about this feature is limited: it is a “self-triggered hook”, that “periodically checks for a hidden module”. Based on the source code, it collects modules from the module list, the `kset` and another source called the module layout tree. Like many solutions on this list, this one is not applicable directly to memory images either, but the concept it implements can be ported. It is a subset of what our Volatility plugin implements.

SigGraph, on the other hand, generates graphs of the data structures inside the memory, that are linked together by pointers. It works on both live machines and memory dumps and it is capable of finding hidden processes and modules. It was only tested on a narrow range of kernel versions (2.6.12-6 – 2.6.34-2), but based on its description, if it is compatible with later versions of the Linux kernel, it should be capable of finding modules via the module layout tree as well. On the other hand, it would not be able to find modules, if they are not directly accessible from any global variable.

Compared to the above-mentioned solutions, ours uses different data sources, not just different views of one data source. It only uses sources that appear in the kernel memory naturally, no artificial ones are used. It cross-references 7 different sources in total, containing ones, where modules are not directly accessible from global variables (i.e. ones the SigGraph would not be able to find). In the next section, we give a detailed overview of all these data sources, how they work, and how we can extract module information from them.

3. Detection approach

Our solution collects kernel modules from 7 different sources throughout the memory of the Linux kernel. The modules are collected into one list per source. When traversing all these sources is done, the modules from all these sources are collected into one unified list. We iterate through this unified list, and for each element, our tool reports, in which sources could it be found. Thus hidden modules can be identified, as they are not present in every sources’ list. For example, many rootkits remove their module from the module list (Section 3.1), but not from every other source we will present in this section.

In this section, we give a detailed description of the sources we used to enumerate the loaded kernel modules: what they are used for, how they work, how modules can be accessed through them, and their evolution across the different versions of the Linux kernel.

3.1. *The list of modules*

When a module is loaded into the Linux kernel, memory is allocated for an object of type `struct module`, to store all relevant information about the freshly loaded module. All these module structures are placed in a doubly linked list for accounting purposes, which is accessible through the global variable `modules`.

As it is customary in the kernel, `module`s contain a `list_head` called `list`. These `list_head`s are used to implement the doubly linked list and the location of the next module in the memory can be computed from the `list_head`’s `next` pointer. Fig. 1 illustrates, how the `module`s are organized into a doubly linked list.

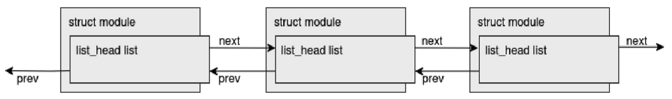


Fig. 1. The doubly linked list of modules.

[4] https://lwn.net/Articles/132196/ (Last visited: 2025.01.22).

This list is used to populate the file /proc/modules, which in turn is used by lsmod, the utility designed to obtain information about currently loaded modules.

This list was already present in the kernel at version 2.6.12-rc2, the first version that GitHub tracks. During our search for open-source rootkits to test our detection method, all rootkits that used DKOM to hide their modules tampered with this list. It is also commonly used by cross-view-based detectors.

3.2. The kset of modules

Whenever reference counting is necessary for something in the kernel memory, kernel objects (kobjects) are used. These are embedded into other structures, and kernel objects of the same type can be organized into kernel sets (ksets). modules contain a structure of type module_kobject, and each of these has an embedded kobject. All kobjects, that are embedded into modules, are collected into a kset, called module_kset.

By iterating through the doubly linked list of kobjects associated with module_kset, we can enumerate modules. By knowing the base address of a kobject, we can compute the location of the module_kobject, and thus the location of the module as well. Fig. 2 shows how the different structures are embedded in each other and how the kobjects form a doubly linked list.

The list of kobjects in module_kset is used to populate the directory /sys/modules, through which the kernel offers additional information about the loaded modules.

This data structure first appeared in kernel version 2.6.25-rc1, and rootkits often remove their modules from the module_kset, since many rootkit detection solutions cross-reference the content of this kset with the module list.

3.3. The module layout tree

Besides the module structure, other areas of memory are also associated with modules, for example, to store the code and data loaded from the module's file. In the earlier days of the Linux kernel, to find out which module a memory address belongs to, the module list was traversed and certain members of the module structure were checked. To make this lookup faster, the module layout tree was created: the description of the allocated memory was moved to the structure module_layout, and the module_layouts are organized into a latched red-black (basically, two red-black trees side-by-side, to avoid the need for locking). This tree can be traversed from the global variable mod_tree, and through the nodes of the latched red-black tree, we can access the module they are associated with. Fig. 3 details how the tree nodes are embedded into the module layout, and how the module can be accessed by traversing the tree.

This data structure was added to the Linux kernel at version 4.2-rc1. Later, at version 6.4-rc1, the module_layout structure was replaced by module_memory, but the tree remains, and the technique is still applicable. We did not find any rootkit, that tried to hide its presence from the module layout tree; on the other hand, we know about one solution (tracee), that utilizes this data structure for detection.

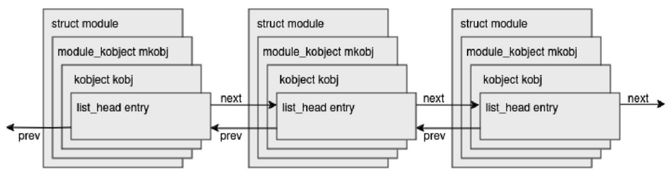


Fig. 2. The modules and their embedded kernel objects.

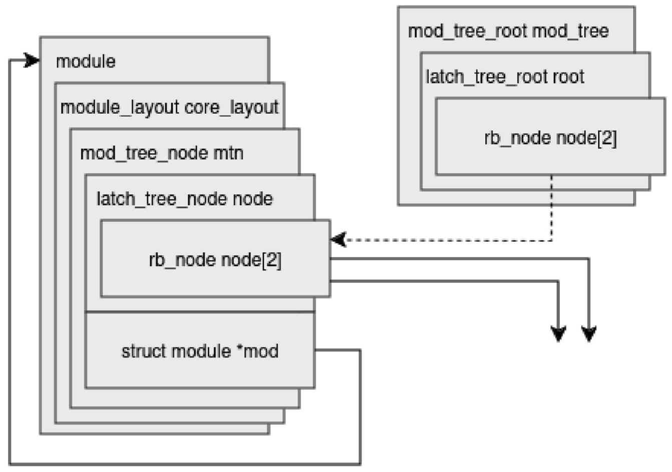


Fig. 3. The relationship of the different structures related to the module layout tree.

3.4. Virtual memory areas (VMAs)

There are two memory allocators available in the Linux kernel: kmalloc and vmalloc. The former is used when the allocated memory must be continuous both in the virtual and physical address space, while vmalloc can allocate continuous virtual memory that is mapped to arbitrary physical memory pages.

All memory associated with modules gets allocated by vmalloc, and these memory areas are also accounted for. vmap_area structures are used to describe them, and despite the lack of direct connection to modules, through memory scanning, they can be identified.

vmap_areas are placed in a doubly linked list, accessible from the global variable vmap_area_list. This list is used to populate the file /proc/vmallocinfo, which can be used to obtain information about the memory allocations.

vmap_areas are also organized into a red-black tree, accessible from vmap_area_root. This allows faster lookup if one wants to determine, which memory area an address belongs to.

The module structures themselves are stored in such memory areas, thus by scanning the memory ranges pointed by a vmap_areas, we can find them. For kernels pre-4.2, it is possible to match the module_init and module_core members of the module. The first one is a NULL pointer, if the module finished initialization, while the second one is the same as the start address of the vmap_area. For kernels between 4.2

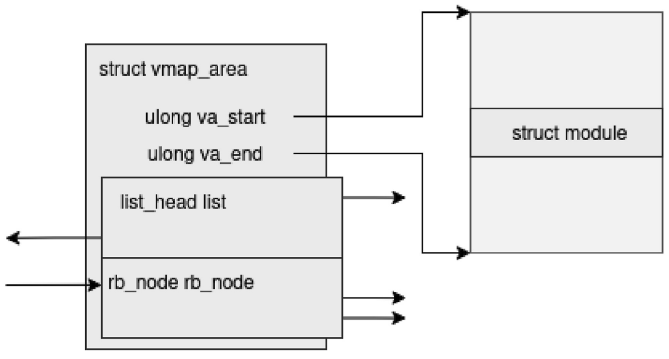


Fig. 4. The relationship of vmap_areas and modules.

and 6.4, where the module_layout structure exists, the head of this structure can be matched: its first member is the same as the start address of the vmap_area, while the second one is its size. After 6.4, the same can be done with module_memory structures. Fig. 4 illustrates, how vmap_area structures are related to modules.

We collect vmap_area structures from both the list and tree and for every memory area, we attempt to locate a module structure inside.

The vmap_area_list and vmap_area_root first appeared in the version 2.6.28-rc1 of the Linux kernel. They remained for many years until they were removed in version 6.9-rc1. Since then, vmap_area structures are accessible from an array-like data structure called vmap_nodes. This contains vmap_node structures, that contain both list_heads and red-black tree roots, so all vmap_area structures are accessible from these as well, just like they were from vmap_area_list and vmap_area_root.

There are a small number of rootkits, that attempt to hide their corresponding memory areas. However, as far as we know, these structures were never used for detection before.

3.5. The bug list

The Linux kernel code uses macros like BUG and BUG_ON to signal errors that the kernel is unable to recover from. These macros usually expand into an undefined instruction, and if execution reaches this instruction, the stack trace is dumped into the kernel log, and the process, in which this happened, is terminated.

In order to help debugging, each module has a bug table, containing information about the BUGs used by the module. To help identify, which module was responsible, if a BUG happens, the modules are chained together into a list, called module_bug_list. This list contains all modules, even if their bug table is empty, thus by traversing this list, one can collect all loaded kernel modules.

The bug list works the exact same way as the list of modules, except that the list head where it starts is called module_bug_list instead of modules, and the list_head embedded into modules is named bug_list.

This list was added to the kernel at version 2.6.12-rc2, but it was available only for the PowerPC architectures (both 32 and 64-bit). In version 2.6.20, its definition was moved to another source file, enabling its use for other architectures as well. It was not removed since. As far as we know, this data structure was never tampered with, to hide a kernel module, and no detection solution has ever used it either, explicitly. SigGraph, however, might have been able to find it, since the bug list appeared in a kernel version that is part of the interval, where SigGraph was tested.

3.6. Ftrace

Ftrace stands for Function Tracer, and it is an internal tracing utility of the kernel, designed to help debug and analyze performance issues in the kernel. Rootkits, on the other hand, often use it as a universal utility to hook arbitrary functions in the kernel.

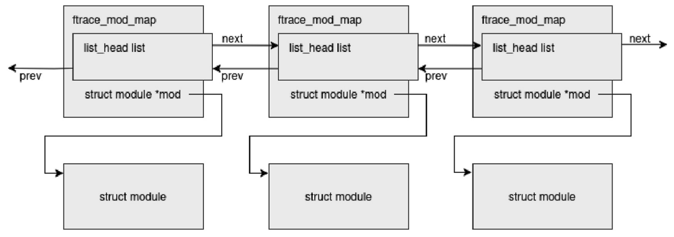


Fig. 5. The relationship of ftrace_mod_maps and modules.

At version 4.15, the functionality of ftrace was extended, and a data structure called ftrace_mod_maps appeared in the kernel. This maps ftrace hooks to kernel modules, to help keep track of them. This is essentially a doubly linked list of ftrace_mod_map structures, that contain pointers to both their corresponding function hooks, and the module that implements them as well. Fig. 5 illustrates this relationship.

Unlike any other data source we described in this section, this list only contains elements for modules that utilize ftrace, not all loaded modules. Rootkits, however, often use ftrace to hook arbitrary functions in the kernel, meaning that many of them can be found by traversing this list.

Although ftrace is part of the kernel since 2.6.27, the module mapping feature only appeared at 4.15-rc1.

As far as we know, no rootkit ever attempted to hide its ftrace hooks and we do not know of any rootkit detection solution either, that tried to use ftrace_mod_maps to find hidden modules.

4. A case study: THOR

In this section, we demonstrate the analysis capabilities of our Volatility plugin through a case study, where we analyze a memory dump from a Linux machine that was infected with an open-source rootkit named THOR.[5] It was chosen to be the subject of this case study because it attempts to hide its module, but it uses function hooks instead of DKOM, thus it does not cause any inconsistency among the data structures we analyze, so it is not trivial to spot it.

The first time, we execute our plugin, it gives us a list of loaded modules. For each of them, it is printed whether they can be found in any of the available data sources. Since the machine whose memory dump we analyze was running on a kernel of version 3.16.0–6, the following data sources are available: the module list, the kset of the modules, and the bug list. With a separate flag, it is possible to trigger the analysis of the vmap_area_list and vmap_area_tree structures as well (these are disabled by default due to performance reasons), but they do not show any inconsistency either.

The system had 63 loaded modules in total. A possible way to discard less interesting ones could be by looking at the signatures and taints associated with each module. Since 3.16 is a relatively old version of the kernel, none of the loaded modules were signed, but taints are more informative in this case. Taints represent a set of events that might be relevant for investigating problems regarding the kernel. The kernel has a bit vector to store if any of these events occurred, but each module has its own taints as well. For modules, a bit is only set if the module was responsible for a specific event. In this case, 4 modules have taints: their names are thor, vboxguest, vboxsf and vboxvideo. All of them have the taint value of 0x1000, which means they were built externally or out-of-tree (i.e. they were compiled separately from the kernel).

The plugin also displays how many other modules depend on a certain module and how many other modules a certain one depends on. These values tell us that the vbox* modules partake in such dependency relationships, while thor does not. While it is not a requirement or consequence of maliciousness, rootkits typically do not depend on other modules and they are not dependencies of other modules either. On the other hand, by querying the dependency trees of the 4 tainted modules, we can see that vboxsf depends on vboxguest and vboxvideo depends on some DRM-related modules. This information improves the credibility of the vbox* modules, so a logical next step would be the deeper analysis of thor

Our plugin is also capable of listing all the symbols defined by a certain module. The C code of kernel modules is compiled to an ELF file, which the kernel can load, and symbols (function and variable names) are not stripped during the compilation (in fact, the kernel refuses to load stripped module files). Moreover, the kernel keeps these symbols

[5] https://github.com/W4RH4WK/THOR (Last visited: 2025.01.24).

when it loads a module, thus we can query them. Rootkit developers often use descriptive names while writing code; in this case, this specific module contains symbols like `replace_tcp_seq_show` and `replace_udp_seq_show` (these most likely hook well-known function pointers to hide open network connections), `pidhider_init` and `pidhider_cleanup` (these most likely implement hiding certain processes) and `my_hide_module` (which most likely hides the kernel module that implements the rootkit). At this point, the maliciousness of the `thor` module is fairly certain.

To achieve absolute certainty, the actual code of the rootkit must be examined. To support this, our plugin is capable of partially reconstructing the `ko` file that implemented the module. We achieve this by dumping all sections of the module into a file, the symbol table we already parsed in the previous paragraph, and auxiliary information to create a valid ELF file, that can be analyzed by any reverse engineering framework, like Ghidra or Radare2.

Unfortunately, some parts of the original file are discarded once a module is fully loaded, so only partial reconstruction is possible this way. Thus we recommend recovering the module file from the file system cache, if possible, but if it is not an option, our plugin can still reconstruct much of the original module file.

Also, if the rootkit can be identified, and the source code is available, for example, on GitHub, it might be possible to find the exact version the attacker used. Modules typically include a so-called srcversion hash, which is a 24-byte long hexadecimal value. During compilation, this is computed from the source files, the module was built from. Our tool also reports this hash for each module, if available, so by compiling different versions of a rootkit, and comparing their srcversion hash to the found one, an analyst might be able to determine which version was used by the attacker.

5. Evaluation

In this section, we describe the datasets we used to evaluate our solution and the environment we used for infecting machines with different rootkits. We also discuss the results of these experiments, which we detail even further in Appendix A.

To test our detection method against the rootkits we collected, we had to infect a running system and create a memory dump from it. To do so, we used a virtual environment, specifically Vagrant[6] with VirtualBox.[7] Memory acquisition was done by the `debugvm` command and the `dumpvmcore`[8] subcommand of VirtualBox. It can save the entire physical memory of the virtual machine to an ELF file, which we could later analyze with Volatility.

We evaluated our solution on two sets of rootkits: one set of open-source rootkits, collected from GitHub, and another set of rootkits we collected from VirusTotal. In the case of GitHub, we had to compile the rootkits, while from VirusTotal, we downloaded binaries only. Based on the analysis of these wild rootkits, probably there is some overlap between the two sets. We didn't remove these overlapping samples for two reasons: they were compiled for different kernel versions, so they might function slightly differently, and it is also possible that an attacker used an open-source rootkit as a base and extended it with new functionality.

5.1. Open-source rootkits

This set of test rootkits consists of 35 open-source rootkits we found on GitHub. To collect test rootkits, we manually reviewed the source code of many rootkit projects. First, we discarded rootkits that were implemented in the user space, and then we removed the ones that did not use any technique to hide their kernel module. The remaining kernel-space rootkits used one or more of the following techniques to hide their modules: removing it from the module list, removing it from the module `kset`, removing their `vmap_area` structures from both the list and the tree, and hooking functions in upper layers, e.g., to stay invisible, when someone reads the `/proc/modules` file. For 35 of these rootkit projects, we could successfully compile the malware, infect a Linux machine with it, and create a memory dump we could analyze. Rootkits hiding by function hooking do not show inconsistencies among the different data sources, but they are present in the list that our plugin shows. Appendix A details the rootkits we used, the kernel versions, where we could compile them, and how each rootkit in our dataset attempted to hide its module.

5.2. Wild rootkits

This set of rootkits comes from VirusTotal. We collected and analyzed kernel object files that were considered to be malicious by at least 5 anti-virus vendors, and discarded those that did not attempt to hide their module. These files contain a section called `.modinfo`, from which we were able to determine, what kind of kernel were they compiled to. For 20 of these samples, we were able to prepare an environment, that we could infect and analyze. These rootkits only used the module list and the `kset` of modules to hide. We also provide their MD5 hashes and links to their VirusTotal reports.

5.3. Results

Among the 35 open-source rootkits, only 4 of them tried to hide via function hooking, the rest tried to use DKOM. All that used DKOM removed their module from the module list, and modifying the `kset` is common as well. Tampering with the virtual memory is rare, but 3 rootkits implemented this functionality. None of them tried to hide from the module layout tree, the bug list, or the ftrace module mappings; however only 4 used ftrace among those that we could compile for kernels, where it is supported. 13 of the open-source rootkits could be detected by Volatility's `linux.check_modules` plugin, while our plugin could detect all 35.

Some of them were unstable and crashed during loading, but due to our external memory acquisition process, we could still examine the kernel memory. 3 of these crashed before the rootkit could hide its module, and another 4 rootkits crashed after hiding. In 2 cases, we could not determine if the crash happened before or after hiding, since these rootkits tried to hide themselves by hooking functions in upper layers, thus they did not show inconsistencies among our data sources.

All of the 20 wild rootkits attempted to hide by using DKOM: 10 of them used only the module list, while the other 10 tampered with the module list and the `kset` as well. None of them touched any of the other 5 data sources, however, only 4 of them used ftrace. For this dataset, Volatility's `linux.check_modules` could detect those 10, that modified only the module list. Our solution, on the other hand, could successfully detect all 20 of them.

A details the results of our experiments: for each rootkit, we show how it attempted to hide its module, which kernel version could be used to test it, whether it could be detected by Volatility's `linux.check_modules` plugin, and whether it was detected by ours. We also provide links to their GitHub repositories or their VirusTotal reports, for the sake of transparency and reproducibility. We also share the code of our plugin,[9] so anyone can test and use it, and the memory dumps as well[10].

[6] https://www.vagrantup.com/ (Last visited: 2025.01.20).
[7] https://www.virtualbox.org/ (Last visited: 2025.01.20).
[8] https://www.virtualbox.org/manual/topics/vboxmanage.html/#vboxmanage-debugvm (Last visited: 2025.01.20).

[9] https://github.com/CrySyS/ModXRef.
[10] https://www.crysys.hu/~rnagy/datasets/rootkit.html.

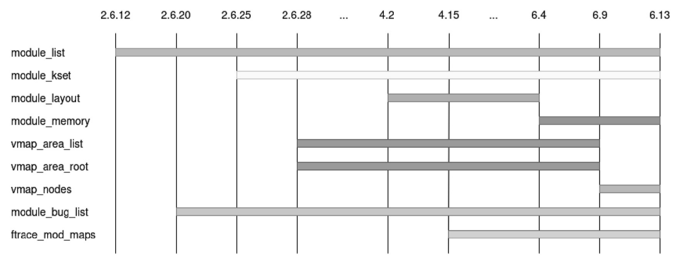


Fig. 6. The evolution of the different data structures through the different versions of the Linux kernel. Spacing between the kernel versions is not proportional to the time elapsed between the versions.

6. Discussion

In this section, we discuss some of the limitations of our solution, its relation to eBPF, a recent trend in kernel rootkit development, and we provide compatibility analysis with different versions of the Linux kernel.

6.1. Limitations

Of course, as any solution, ours is not perfect either. Data acquisition is arguably the most important part of any memory forensics project, as it directly influences the capabilities of every tool that is later used to extract information from the memory dump. This affects our Volatility plugin as well: in incomplete or inconsistent data, we might not be able to identify hidden kernel modules.

This, however, is a limitation of the implementation, not the technique: the same cross-view-based detection method could be implemented to defend live systems, as long as it can be protected from rootkits, e.g. by using a Virtual Machine Manager (e.g. Rhee et al. (2010)) or a Trusted Execution Environment (e.g. Nagy et al. (2021)).

Unfortunately, cross-view-based solutions also suffer from weaknesses of their own. One of these is the case, when the rootkit is hiding in plain sight, i.e. when the module is not hidden. To mitigate this issue, our plugin reports a wide range of information about each found module; these can help to identify suspicious ones, as it was shown in Section 4.

Another edge-case, where cross-view-based solutions are helpless is when something is hidden thoroughly: if a module cannot be found in any of the data sources our plugin examines, it cannot be detected this way.

6.2. eBPF

Since eBPF does not rely on kernel modules, our solution cannot detect eBPF-based rootkits. On the other hand, compared to kernel modules, the capabilities of eBPF programs are limited. They cannot, for example, read arbitrary memory addresses. If, however, an attacker would need such functionality, it must be implemented in the form of a loadable kernel module. In this case, it is again in the attacker's best interest to hide this module. If this would be implemented using eBPF, it would happen through hooking functions, that provide information about modules to the upper layers of the kernel, and eventually to the user space. In such a case, our tool would be able to find the hidden kernel module; although it would not mark it as suspicious, since all the used data sources would be consistent.

6.3. Compatibility analysis

The detection capability of our solution also depends on the kernel version of the analyzed machine: between 2.6.12 and 2.6.20, only the module list can be checked. From 2.6.20 to 2.6.25, only the module list and the bug list are available. The `kset` appears at 2.6.25, and later, at 2.6.28, the structures related to the virtual memory appear as well. From 4.2 upward, we can detect by using the module layout tree as well. At version 4.15, ftrace mappings appear in the kernel. At 6.4, `module_layout`s are replaced with `module_memory` structures, but this does not affect our detection capability. At 6.9, the `vmap_area_list` and `vmap_area_root` global variables are removed, but an array-like data structure called `vmap_nodes` is introduced instead. From this, `vmap_area` structures are still accessible, so this, again, does not impact our tools detection capability.

Fig. 6 depicts when these data structures were introduced to or removed from the kernel.

7. Conclusion

In this paper, we presented a new, cross-view-based rootkit detection solution for Linux systems to find hidden kernel modules. It does not require modification of the kernel and it does not rely on any a priori information or measurement. It is implemented as a Volatility plugin, but the method itself could be ported to defend live systems as well. Additionally, we implemented a toolkit for analysts to address a major shortcoming of cross-view-based detection, when a rootkit is hiding in plain sight.

To collect modules from as many sources as possible, our plugin re-implements well known techniques (module list, `kset`, `mod_tree`), and implements 4 more, that were never used in detection before (`vmap` list & tree, bug list, ftrace mappings).

We tested our solution against a large number of rootkits, on a wide variety of Linux kernels, and it proved to be robust enough even against the most modern rootkits: all 55 of the tested rootkits were successfully detected, outperforming Volatility's current module detection mechanism. We also investigated our tool's compatibility with different versions of the Linux kernel, and it proved to be effective on some of the oldest kernel versions as well, while supporting the most recent ones too. Additionally, we publish both the plugin we developed, and the dataset we used for evaluation.

Acknowledgement

The research presented in this paper was supported by the European Union's Horizon Europe Research and Innovation Program through the DOSS Project (Grant Number 101120270).

The presented work also builds on results of the project no. 2018-1.2.1-NKP-2018-00004 (SETIT Project), which was implemented with the support provided from the National Research, Development and Innovation Fund of Hungary.

The authors are also thankful for the following colleagues for proofreading and their suggestions about the manuscript: Dr. Dorottya Papp, Dr. Balázs Pejó and Dr. Levente Buttyán.

Last but not least, we are thankful to VirusTotal for the academic license provided for research purposes.

Appendix. A Test results on open-source and wild rootkits

Table A.1
This table details the open-source (first part of the table) and wild (second part of the table) rootkits that we used to test our solution, detailing how they hide their modules, against which kernel version were they tested, and wether they could be detected by Volatility's linux.check_modules plugin, and ours. For open-source rootkits, we provide the GitHub user and repository name, and for wild rootkits, their MD5 hash.

Name	Hiding technique					ftrace	Kernel version	check_mod	Detected by us	Notes	GitHub Repository/VirusTotal Report
	Mod. list	kset	Vmaps		Hooks						
			List	Tree							
brokepkg	✓					✓	6.2.0–39-generic	✓	✓		R3tr074/brokepkg
bds_lkm	✓	✓	✓	✓		✓	5.15.0–116-generic	×	✓		bluedragonsecurity/bds_lkm
bds_lkm_ftrace	✓	✓	✓	✓		✓	5.15.0–116-generic	×	✓		bluedragonsecurity/bds_lkm_ftrace
j-rootkit	✓					×	5.15.0–116-generic	✓	✓		JakeGinesin/j-rootkit
LKM-Rootkit	✓					×	5.15.0–116-generic	✓	✓		MatthiasCr/LKM-Rootkit
reveng_rtkit	✓	✓				×	5.15.0–116-generic	✓	✓	1	reveng007/reveng_rtkit
wkit	✓					×	5.15.0–116-generic	✓	✓		ngn13/wkit
Zhang1933/linux-rootkit	✓					✓	5.15.0–116-generic	✓	✓		Zhang1933/linux-rootkit
Diamorphine	✓					×	5.4.0–155-generic	✓	✓		m0nad/Diamorphine
dorosch/rootkit	✓	✓				×	5.4.0–155-generic	×	✓		dorosch/rootkit
LilyOfTheValley	✓	✓				×	5.4.0–155-generic	×	✓	▼	En14c/LilyOfTheValley
lkm-hidden	✓	✓	✓	✓		×	5.4.0–155-generic	×	✓		sysprog21/lkm-hidden
AFkit	✓	✓				×	4.15.0–206-generic	×	✓		t0t3m/AFkit
dmliscinsky/lkm-rootkit	✓					×	4.15.0–206-generic	✓	✓	▼	dmliscinsky/lkm-rootkit
Nuk3Gh0stBeta	✓					×	4.15.0–206-generic	✓	✓		juanschallibaum/Nuk3Gh0stBeta
puszek					✓	×	4.15.0–206-generic	×	✓	◆	Eterna1/puszek-rootkit
rickrolly	✓	✓				×	4.15.0–206-generic	×	✓	▼	miagilepner/rickrolly
rootfoo/rootkit	✓	✓				×	4.15.0–206-generic	×	✓		rootfoo/rootkit
suterusu	✓	✓				×	4.15.0–206-generic	×	✓	▼	mncoppola/suterusu
Reptile	✓					–	4.9.0–13-amd64	✓	✓		f0rb1dd3n/Reptile
ah450/rootkit					✓	–	4.4.0–210-generic	×	✓	◆	ah450/rootkit
liinux	✓	✓				–	4.4.0–210-generic	×	✓		a7vinx/liinux
m0hamed/lkm-rootkit	✓	✓				–	4.4.0–210-generic	×	✓		m0hamed/lkm-rootkit
nurupo/rootkit	✓					–	4.4.0–210-generic	✓	✓		nurupo/rootkit
soad003/rootkit	✓					–	4.4.0–210-generic	✓	✓		soad003/rootkit
swiss_army_rootkit	✓	✓				–	3.16.0-6-amd64	✓	✓	2	nnedkov/swiss_army_rootkit
THOR					✓	–	3.16.0-6-amd64	×	✓		W4RH4WK/THOR
wukong	✓	✓				–	3.16.0-6-amd64	×	✓		hanj4096/wukong
maK-it	✓	✓				–	2.6.32–754.35.1.el6	×	✓		maK-/maK_it-Linux-Rootkit
adore-ng	✓	✓				–	2.6.32-5-amd64	×	✓		yaoyumeng/adore-ng
brootus	✓	✓				–	2.6.32-5-amd64	×	✓	▲	dsmatter/brootus
ivyl/rootkit	✓	✓				–	2.6.32-5-amd64	×	✓		ivyl/rootkit
kevinkoo001/rootkit	✓	✓				–	2.6.32-5-amd64	×	✓	▲	kevinkoo001/rootkit
moo_rootkit	✓	✓				–	2.6.32-5-amd64	×	✓	▲	matteomattia/moo_rootkit
the_colonel	✓	✓				–	2.6.32-5-amd64	×	✓		bones-codes/the_colonel
brokepkg	✓					✓	6.10.9-amd64	✓	✓		5f549aa8ac43363599e1ed0d7f6ddbaf
dropper	✓					✓	6.8.0–31-generic	✓	✓		f9a678e518d50d844694288fa8e3d4b2
r8152_helper	✓					×	6.2.0–1012-aws	✓	✓		8502513518aa626b4bcb2f3a7dc8bdbe
graphic_card	✓					×	6.2.0–26-generic	✓	✓		ba9d6a6bbde602fd414cea09fcbd1aa0
iptable_reject	✓					×	6.2.0–26-generic	✓	✓		edc8916a4593cc8598bf9d9990cc3111
clientking	✓					✓	5.15.0–88-generic	✓	✓		80b7038ce7f5b82c8f41d8c35ea393b8
template	✓	✓				×	5.4.0–137-generic	×	✓		57d851cfdd653bce225743c279058673
ghoul	✓					✓	5.4.0–122-generic	✓	✓		ba9b483a3005e13e35839a3ed4d7080e
panix	✓					×	4.19.0–27-amd64	✓	✓		000a0065b6c33c373f929c6163e9a410
netlink	✓	✓				×	4.18.0–147.el8	×	✓		b2eade99d74995c22f7773a0dda9cf58
Diamorphine	✓					–	4.9.0-9-amd64	✓	✓		ace0ff660bf42028a862d84989f59f67
123	✓					–	3.10.0–1160.119.1.el7	✓	✓		eeb89a61e09d24c400fd4983d3b497e6
vmi	✓	✓				–	3.10.0–1160.114.2.el7	×	✓		14095c657097409db4cd5a0b5406fccd
rr	✓	✓				–	3.10.0–1127.el7	×	✓		45a74c7b4242c704c3562db0a07327ca
cryptov2	✓	✓				–	3.10.0–123.9.3.el7	×	✓		2ee204622154a0f969ed72f2812ba2f0
iproute	✓	✓				–	3.10.0–123.9.3.el7	×	✓		a36460ead268ce98095fb03aa5e1a9ca
suterusu	✓	✓				–	2.6.32–754.17.1.el6	×	✓		9d337c95034db706070045b7d3444f6a
inl	✓	✓				–	2.6.32–754.11.1.el6	×	✓		81b8ff1710160289956de28ffcdec8e8
tmp_spVMqi	✓	✓				–	2.6.32–696.el6	×	✓		5c37a233316d32b83c4e6a185abf54ea
504	✓	✓				–	2.6.32–504.el6	×	✓		de169851aca0998b01ccb585f26b8dc8

Notations used in the notes column:
▲: During loading, the rootkit crashed, before it could hide its module.
▼: During loading, the rootkit crashed, but the module is already hidden.
◆: During loading, the rootkit crashed, but we cannot determine, wether it happened before or after hiding the module.
1: The rootkit contains code to hide from the kset, but it's not used.
2: A collection of rootkits written as part of a university course. Assignment 5 was used for testing, but all rookits that implement module hiding use the same code. It implements hiding from the kset, yet, for some unknown reason, our plugin can detect it there.
The Volatility plugin is available at https://github.com/CrySyS/ModXRef.
The dataset is available at https://www.crysys.hu/~rnagy/datasets/rootkit.html.

References

Aqua Security, 2020. Modreveal. https://github.com/aquasecurity/tracee.

Boelen, Michael, 2003. Rkhunter. URL. https://rkhunter.sourceforge.net.

Cozzi, Emanuele, Graziano, Mariano, Fratantonio, Yanick, Balzarotti, Davide, 2018. Understanding linux malware. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 161–175. https://doi.org/10.1109/SP.2018.00054.

Deyannis, Dimitris, Karnikis, Dimitris, Vasiliadis, Giorgos, Ioannidis, Sotiris, 2020. An enclave assisted snapshot-based kernel integrity monitor. In: Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking, pp. 19–24.

Jones, Stephen T., Arpaci-Dusseau, Andrea C., Arpaci-Dusseau, Remzi H., 2008. VMM-based hidden process detection and identification using lycosid. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 91–100.

Li, Richard, Du, Min, Johnson, David, Ricci, Robert, Van der Merwe, Jacobus, Eide, Eric, 2019. Fluorescence: detecting kernel-resident malware in clouds. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp. 367–382.

Yu Li, Xiang, Zhang, Yi, Tang, Yong, 2015. Kernel Malware Core Implementation: A Survey. In 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, pp. 1–15. https://doi.org/10.1109/CyberC.2015.26.

Liao, Kevin, Zhao, Ziming, Doupé, Adam, Ahn, Gail-Joon, 2016. Behind closed doors: measurement and analysis of CryptoLocker ransoms in Bitcoin. In: 2016 APWG Symposium on Electronic Crime Research (eCrime). IEEE, pp. 1–13.

Lihi, Jafar, 2023. modreveal. https://github.com/h2337/modreveal.

Lu, Yan, Zhang, Da-Long, Hu, Chuan-Ping, Zhu, Kai-Lin, Zhuang, Yan, Shi, Li-Nong, 2023. Rootkit detection mechanisms for linux systems. In: 2023 9th International Conference on Computer and Communications (ICCC), pp. 2149–2154. https://doi.org/10.1109/ICCC59590.2023.10507534. https://ieeexplore.ieee.org/abstract/document/10507534.ISSN:2837-7109.

Nadim, Mohammad, Lee, Wonjun, Akopian, David, 2023. Kernel-level rootkit detection, prevention and behavior profiling: a taxonomy and. Surveyor. URL. http://arxiv.org/abs/2304.00473.

Nagy, Roland, Németh, Krisztián, Papp, Dorottya, Buttyán, Levente, 2021. Rootkit detection on embedded IoT devices. In: *ACTA CYBERNETICA*, Special Issue of the 12th Conference of PhD Students in Computer Science. https://doi.org/10.14232/actacyb.288834. URL. https://m2.mtmt.hu/api/publication/32468427.

Anh Quynh, Nguyen, Takefuji, Yoshiyasu, 2007. Towards a tamper-resistant kernel rootkit detector. In Proceedings of the 2007 ACM symposium on Applied computing, SAC '07. Association for Computing Ma-chinery, New York, NY, USA, pp. 276–283. https://doi.org/10.1145/1244002. https://dl.acm.org/doi/10.1145/1244002.1244070. ISBN 978-1-59593-480-2.

Rhee, Junghwan, Riley, Ryan, Xu, Dongyan, Jiang, Xuxian, 2010. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In: Jha, Somesh, Sommer, Robin, Kreibich, Christian (Eds.), Recent Advances in Intrusion Detection. Springer, Berlin, Heidelberg, pp. 178–197. https://doi.org/10.1007/978-3-642-15512-3_10. ISBN 978-3-642-15512-3.

Roemer, Ryan, Buchanan, Erik, Shacham, Hovav, Savage, Stefan, 2012. Return-oriented programming: systems, languages, and applications. ACM Trans. Inf. Syst. Secur. 15 (1), 1–34.

Sd, Devik, 2001. Linux on-the-fly kernel patching without LKM, 11. Phrack Magazine, 58. https://phrack.org/issues/58/7.html. (Accessed 18 September 2024).

Volatility Foundation. Volatility 3. URL https://github.com/volatilityfoundation/volatility3.

Wang, Y.-M., Beck, Doug, Vo, Binh, Roussev, Roussi, Verbowski, Chad, 2005. Detecting stealth software with strider GhostBuster. In: 2005 International Conference on Dependable Systems and Networks (DSN'05). IEEE, pp. 368–377.

Xu, Dongyan, Jiang, Xuxian, 2011. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-Based Signatures.

Yamauchi, Toshihiro, Akao, Yohei, 2017. Kernel rootkits detection method by monitoring branches using hardware features. IEICE Trans. Info Syst. 100 (10), 2377–2381.