Politecnico di Torino

Master of Science in Cybersecurity

A.Y. 2024/2025

Graduation Session October 2025

Detection and Mitigation of eBPF Security Risks in the Linux Kernel

Supervisor

Riccardo Sisto

Candidate

Francesco Rosucci

Company Tutors

David Soldani, Hami Bour, Saber Jafarizadeh

Abstract

eBPF has become a key technology for system observability and security, enabling efficient in-kernel execution of user-defined programs. However, its growing adoption has also introduced new attack surfaces, with several vulnerabilities leading to severe consequences such as privilege escalation, kernel panics, and attacks on eBPF maps. This thesis, developed in collaboration with Rakuten and a colleague, was divided into four use cases overall, with two presented here. The first part of the work focused on studying the fundamentals of eBPF, analyzing known security issues, exploring its integration with LSM, and reviewing market products that employ eBPF for security monitoring. This provided the foundation for the subsequent experimental phase, which investigated specific vulnerabilities and potential hardening strategies. The adopted methodology consisted of analyzing high-severity CVEs and attack vectors, examining and replicating existing proofs of concept, and proposing defensive mechanisms designed to be general enough to withstand variations of known exploits. Two case studies are discussed. The first addresses the protection of eBPF maps, which are central to both benign applications and security monitoring tools, making them an attractive target for attacks. The second focuses on privilege escalation exploits, with a detailed analysis of CVE-2021-3490 and the development of countermeasures. The results show that kernel-level protection mechanisms such as BPF-LSM, as well as higher-level security tools like Tetragon or the use of kernel modules, can be effectively leveraged to mitigate these threats, each with its own trade-offs in terms of granularity, coverage, and ease of deployment. Overall, this thesis contributes to a better understanding of eBPF-related attack surfaces and provides practical insights into designing hardening solutions for real-world environments.

Acknowledgements

Only in the printed version

Table of Contents

Chapter 1

Introduction

1.1 Background and Motivations

eBPF is a modern technology originally developed within the Linux kernel that enables safe and efficient extension of kernel functionality without modifying kernel source code or loading additional kernel modules[1]. Historically, the operating system has always been an ideal place to implement observability, security, and networking functionality due to the kernel's privileged ability to oversee and control the entire system. However, evolving an operating system kernel is notoriously challenging because of its central role and strict stability and security requirements. eBPF fundamentally changes this paradigm. It introduces the ability to execute sandboxed programs inside the operating system kernel, allowing developers to dynamically add functionality at runtime. This innovation has led to a wide range of use cases: high-performance networking and load balancing in modern data centers and cloud-native environments, fine-grained security observability at low overhead, advanced application tracing and performance troubleshooting, as well as proactive runtime security enforcement for containers and applications.

Despite its numerous advantages, the adoption of eBPF introduces significant security challenges. The ability to execute code within the kernel environment increases the attack surface, making any bug or flaw in an eBPF component potentially exploitable and capable of compromising system integrity. Several CVEs and documented vulnerabilities highlight these risks, emphasizing the need for robust mitigation strategies. For this reason, systematically studying known vulnerabilities and designing effective hardening solutions is essential when integrating eBPF-based components into production systems.

1.2 Research Objectives

This research was conducted in collaboration with Rakuten, which provided the context and technical guidance for the study. The primary objective of this thesis is to investigate the security implications of adopting eBPF technology, with a focus on known vulnerabilities and their mitigation. Specifically, this research aims to:

- Analyze documented vulnerabilities and CVEs related to eBPF to identify common attack vectors, root causes, and exploitation techniques.
- Examine available exploits to understand how attacks are implemented and to identify areas where hardening measures can be applied to block them.
- Propose and validate hardening strategies that strengthen the security posture of systems leveraging eBPF, based on insights from vulnerability and exploit analysis.
- Provide actionable recommendations and best practices to help system architects and developers securely deploy eBPF-based solutions.

By addressing these objectives, this thesis aims to deliver a clear overview of existing threats and practical mitigation strategies, supporting the safer adoption of eBPF in modern computing environments.

1.3 Overview of the document

This thesis is organized as follows:

- **Chapter 1 - Introduction** presents the motivation, research objectives, and structure of the thesis
- **Chapter 2 – eBPF Architecture**: introduces the eBPF architecture and its main components, providing the technical background for the vulnerabilities explored in later chapters.
- **Chapter 3 - Security Risks and Challenges**: details the risks and challenges introduced by the use of eBPF technology, and provides an overview of the possible attacks on the different eBPF components.
- **Chapter 4 – Linux Security Architecture**: explains the Linux security architecture, covering concepts such as DAC and MAC, Linux process credentials, Linux Security Modules (LSMs) and BPF LSM, and kernel modules.

- **Chapter 5 – eBPF-based products**: presents a popular security monitoring tool based on eBPF, discussing its functionality and relevance.
- **Chapter 6 – Analysis, Exploitation and Hardening of eBPF Vulnerabilities**: the main chapter, presenting two use cases that illustrate different types of vulnerabilities. It details the exploits, proposes hardening solutions, and evaluates their effectiveness, including potential future improvements.
- **Chapter 7 – Conclusions and Future Works**: summarizes the work, highlighting key findings, providing recommendations for best practices, and outlining possible directions for future research and development.

Chapter 2

eBPF Architecture

2.1 Key Features

2.1.1 Instruction set and hook points

eBPF (extended Berkeley Packet Filter) uses a general-purpose 64-bit instruction set [2], originally designed to allow programs written in a subset of C to be compiled into BPF instructions via a compiler back end. These instructions are executed inside an **eBPF virtual machine** implemented within the Linux kernel, which is a sandboxed environment that executes eBPF instructions safely, providing registers, a stack, and a controlled execution model. The kernel can optionally use a Just-In-Time (JIT) [1] compiler to translate eBPF bytecode into native machine instructions, achieving optimal execution performance.

The eBPF virtual machine provides 10 general-purpose 64-bit registers[2] and one read-only frame pointer register (R10) used to access the stack. The eBPF calling convention is defined as follows:

- R0: return value from function calls and exit value for eBPF programs
- R1–R5: function call arguments
- R6–R9: callee-saved registers preserved across function calls
- R10: read-only frame pointer

Registers R0–R5 are scratch registers, meaning that eBPF programs must spill and fill them if their values need to be preserved across function calls.

eBPF programs are event-driven [1], running whenever the kernel or an application triggers a specific hook. Predefined hooks include system calls, function entry and exit points, kernel tracepoints, network events, and others. If a predefined

hook does not exist for a particular need, it is possible to create a **kernel probe (kprobe)** or a **user probe (uprobe)** to attach eBPF programs almost anywhere in the kernel or user-space applications. These probes must be added explicitly to the system to serve as custom hook points.

When an eBPF program is loaded into the Linux kernel via the `bpf` system call, it passes through two important steps before attachment to the requested hook:

- Verification – ensures the program is safe to execute. The verifier checks that the process loading the program has the required privileges, that the program does not crash or otherwise harm the system, and that it always terminates (i.e., does not loop indefinitely).
- JIT compilation (unless disabled) – translates the generic eBPF bytecode into the machine-specific instruction set to maximize execution efficiency. This allows eBPF programs to run as efficiently as natively compiled kernel code or code loaded as a kernel module.

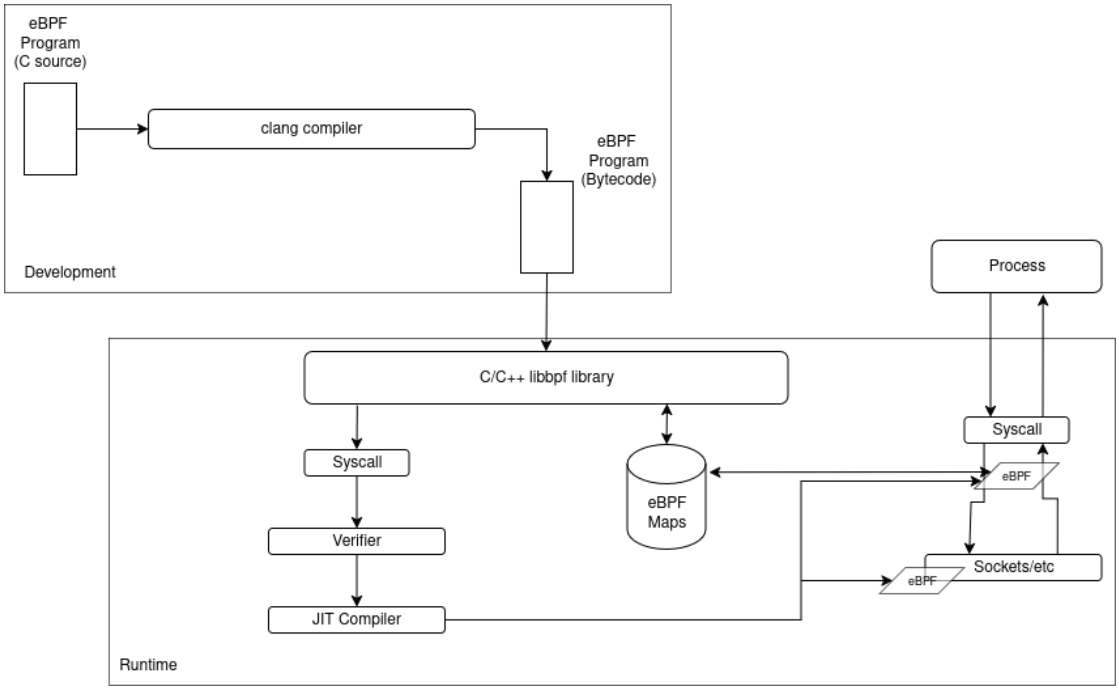

Figure 2.1: eBPF Architecture and execution flow of a C-based eBPF Program

2.1.2 eBPF Verifier

The eBPF verifier [3] is a core kernel component whose primary responsibility is to ensure that an eBPF program is safe to execute in kernel context by statically

checking it against a rich set of safety rules, and then annotating or transforming the bytecode as needed before JIT compilation or execution. The verifier exists because eBPF programs are translated to native machine code and run in kernel mode, so any unchecked program could corrupt memory, leak sensitive data, hang the kernel, or otherwise compromise system security and stability. The verifier, therefore enforces a trade-off: it favors high runtime performance (avoiding expensive dynamic checks) by performing intensive static analysis up front. A non-exhaustive list [3, 4] of restrictions enforced by the verifier includes:

- Programs must always terminate within a reasonable amount of time, so infinite loops or infinite recursion are disallowed.
- Pointer comparisons are not permitted; only scalar values can be added to or subtracted from a pointer. A scalar value in this context is any value not derived from a pointer. The verifier tracks which registers contain pointers and which contain scalars.
- Pointer arithmetic cannot exceed the "safe" bounds of a map. In other words, a program cannot access memory outside the predefined map region. To enforce this, the verifier keeps track of the upper and lower bounds of each register.
- No pointers may be stored in maps or returned as function values, in order to avoid leaking kernel addresses to user space.
- Programs must not deadlock. Any acquired spinlocks must be released, and only one lock may be held at a time to avoid deadlocks across multiple programs.
- Programs cannot read uninitialized memory, since this could leak sensitive data.

Concretely, verification proceeds in two main phases. First, control-flow analysis (CFG/DAG checks) disallows back edges, rejects loops, and detects unreachable instructions so that programs are acyclic. Second, the verifier simulates execution along every feasible path of the program, tracking the state of all registers and stack slots and validating each instruction against that state (e.g., bounds, alignment, and type rules). In particular, the verifier stores the following bound values [4] for every register in each possible execution path, in order to prevent out-of-bounds memory accesses:

- `umin_value`, `umax_value`: minimum and maximum values of the register when interpreted as an unsigned 64-bit integer.

- `smin_value`, `smax_value`: minimum and maximum values of the register when interpreted as a signed 64-bit integer.
- `u32_min_value`, `u32_max_value`: minimum and maximum values of the register when interpreted as an unsigned 32-bit integer.
- `s32_min_value`, `s32_max_value`: minimum and maximum values of the register when interpreted as a signed 32-bit integer.
- `var_off`: information about which bits of the register are known. It is represented using a structure called `tnum`, containing two 64-bit fields: `mask` and `value`. Bits set in `mask` are unknown, while unset bits are known and their values are stored in `value`. For example, `var_off = {mask = 0x0, value = 0x1}` means the register is known to equal 1. Conversely, `var_off = {mask = 0xFFFFFFFF00000000, value = 0x3}` indicates that the lower 32 bits are known to be `0x3`, while the upper 32 bits are unknown.

These bounds are continuously updated and refined. For instance, if `var_off` reveals that a register holds a known constant, the min/max bounds are tightened to reflect that value. This range-tracking for registers and stack slots is implemented through `struct bpf_reg_state` [5], defined in *include/linux/bpf_verifier.h*, which unifies scalar and pointer state tracking. Each register state has a type: `NOT_INIT` (unused), `SCALAR_VALUE` (non-pointer), or a specific pointer type.

During instruction traversal, every time the verifier encounters a branching instruction, it forks the current state [3], queues one branch for later investigation, and updates the states accordingly. For example, if register R3 is known to hold a value between 10 and 30, and the verifier sees an instruction `if R3 > 20`, one fork will constrain R3 to [10,20], and the other to [21,30]. This mechanism also propagates across linked registers: if R2 = R3 before the branch, the verifier ensures R2's range is updated consistently. Until kernel v5.2 there was a hard 4k instruction limit and a 128k complexity limit; afterwards, both were raised to one million. To balance verifier complexity with useful language features, several optimizations exist: tail calls allow large programs to be split into independently verified units, dead-code elimination removes instructions that can never execute, bounded loops (introduced in v5.3) are permitted when the verifier can prove termination (by unrolling loop paths), and function-by-function verification ensures global functions are checked once in isolation rather than at every call site.

The verifier does not actually explore all possible paths exhaustively [5]. For each new branch, it compares the current state against previously visited states at the same instruction. If the new state is a subset of an earlier one, the branch is pruned: the fact that the previous state was accepted implies the current one will be as well. For example, if in a prior state `R1` held a packet pointer with strict bounds and alignment, then any current state where `R1` has at least the

same safety guarantees is automatically accepted. Similar reasoning applies to registers in `NOT_INIT` state. This pruning mechanism, implemented in `regsafe()` and `states_equal()`, also considers stack state and spilled registers.

Finally, because purely static checks can miss subtle runtime behaviors (e.g., pointer arithmetic tricks), the verifier may insert runtime ALU sanitization patches [4]. The idea is to prevent out-of-bounds memory accesses if register values deviate from their expected ranges at runtime. For every arithmetic operation involving a pointer and a scalar, an `alu_limit` is computed, representing the maximum allowed offset. Before the operation, the verifier patches the bytecode with checks enforcing the `alu_limit`, ensuring the resulting pointer remains in safe bounds. This mechanism was introduced to mitigate verifier vulnerabilities and speculative attacks, and the computation of `alu_limit` has been progressively tightened across kernel versions.

2.1.3 JIT Compiler

The Just-in-Time (JIT) compilation step [6] translates the generic bytecode of the program into the machine-specific instruction set to optimize the execution speed of the program. JIT compilers speed up execution of the BPF program significantly since they reduce the per-instruction cost compared to the interpreter. Often instructions can be mapped 1:1 with native instructions of the underlying architecture. This also reduces the resulting executable image size and is therefore more instruction cache-friendly to the CPU. In particular, in case of CISC instruction sets such as x86, the JITs are optimized for emitting the shortest possible opcodes for a given instruction to shrink the total necessary size for the program translation.

2.1.4 eBPF maps

eBPF maps [7] provide a generic storage mechanism that allows programs to share data between kernel space and user space. Maps are mainly defined by their type, the maximum number of elements, the key size in bytes, and the value size in bytes. They serve as the primary communication channel for eBPF programs.

Maps can be accessed from user space via the bpf system call, and from eBPF programs through helper functions defined in tools/lib/bpf/bpf_helpers.h. The exact helper functions available depend on the map type.

There are different types of maps, each optimized for specific use cases. Among them, the main ones are:

- BPF_MAP_TYPE_HASH: stores entries using key-value pairs associated with a hash function.

- BPF_MAP_TYPE_ARRAY: indexed map providing direct access to elements via an index.
- BPF_MAP_TYPE_PROG_ARRAY: stores references to eBPF programs and allows tail-calling between programs.

In the Linux kernel, eBPF maps are internally represented by the struct bpf_map, which contains several fields describing the map's properties and behavior. An important field within this structure is const struct bpf_map_ops *ops;. The bpf_map_ops structure defines the set of operations that can be performed on a map, such as creation, lookup, update, and deletion of elements. These operations are later invoked by the eBPF helper functions (e.g., bpf_map_update_elem, bpf_map_lookup_elem), which serve as the public interface for programs, while the actual implementation is carried out through the corresponding functions defined in the ops table. For specific map types, different bpf_map_ops structures are defined, ensuring that each type of map has the appropriate implementation of these operations.

2.1.5 Helper functions

Helper functions are functions defined by the kernel which can be invoked from eBPF programs. These helper functions allow eBPF programs to interact with the kernel as if calling a function. The kernel places restrictions on the usage of these helper functions to prevent misuse. The potential misuse of helper functions is discussed in more detail in Section 3.1.

2.2 Privileges and capabilities

In traditional UNIX systems [8], processes are categorized as privileged (effective UID 0, referred to as superuser or root) and unprivileged (effective UID nonzero). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission verification based on their credentials, typically including the effective UID, effective GID, and supplementary group list.

Starting with Linux 2.2, the privileges traditionally associated with the superuser have been divided into distinct units known as capabilities. Each capability can be independently enabled or disabled and is a per-thread attribute, providing permission to perform specific privileged operations.

Access to eBPF can therefore be controlled through capabilities. Only privileged users or processes with the required capabilities are allowed to load and execute eBPF programs. Some important capabilities relevant for eBPF access are listed in Table 2.1.

Additionally, since Linux kernel version 4.4, the parameter kernel.unprivileged_bpf_disabled can be used to further restrict eBPF access:

- 0: Unprivileged users are allowed to load and run eBPF programs, but their capabilities are limited. They can only use programs of the type BPF_PROG_TYPE_SOCKET_FILTER and interact with associated maps [9].
- 1: Access to eBPF is restricted to privileged users only, and changing this setting requires a system reboot.
- 2: Access to eBPF is limited to privileged users as well, but this setting can be modified without rebooting the machine.

Capability	Allowed Features
No capabilities	• load and attach operations for BPF_PROG_TYPE_SOCKET_FILTER. • Map usage restricted to the ones relative to BPF_PROG_TYPE_SOCKET_FILTER.
CAP_BPF	• Employ privileged BPF operations • Introduced in Linux 5.8 to isolate BPF-related privileges from the overly broad CAP_SYS_ADMIN capability.
CAP_NET_ADMIN	• Interface configuration. • Modification of routing tables. • Attaching of networking programs, such as XDP and TC. • Stopping network traffic.
CAP_SYS_PTRACE	• Trace arbitrary processes using ptrace • Apply get_robust_list to arbitrary processes • Transfer data to or from the memory of arbitrary processes using process_vm_readv and process_vm_writev • Inspect processes using kcmp
CAP_SYS_ADMIN	• Perform core system administration tasks (mount, swap, hostname, etc.) • Manage and configure namespaces and IPC objects • Access or override system-wide resource limits • Execute privileged I/O, filesystem, and device operations • Use various powerful capabilities (BPF, perf, seccomp, etc.)

Table 2.1: Capabilities and allowed features [8]

Chapter 3

Security Risks and Challenges

As eBPF continues to gain traction in modern computing environments, understanding its security landscape is crucial. While eBPF enhances performance and flexibility, it also introduces potential vulnerabilities that could be exploited if not properly managed.

This section presents the findings of research conducted for this thesis, which examines the primary security risks and weaknesses associated with key components of the eBPF architecture, as well as notable attack techniques that have been used to exploit eBPF. The analysis is based on a comprehensive review of multiple sources, ranging from official threat modeling reports by the Linux Foundation to documented CVEs and attacks discussed in the scientific literature. The discussion begins with the risks related to architectural components of eBPF, such as helper functions, maps, and the verifier, and continues with an overview of common pitfalls and known attack vectors.

3.1 Helper Functions

If a threat actor can load and run eBPF code, the following helper functions have particular security relevance [10]:

- bpf_probe_read_user
- bpf_probe_write_user
- bpf_send_signal
- bpf_override_return

- bpf_map_get_fd_by_id

3.1.1 bpf_probe_read

This helper lets an eBPF program inspect memory belonging to either user processes or the kernel, without disrupting those processes. Because it can extract needed data non-invasively, it's often used to collect runtime information such as process state or memory contents, and to walk kernel data structures. At the same time, it can reveal sensitive information, for example, when placed in a kprobe on an authentication routine it could read a temporary buffer holding a user's password, or disclose kernel addresses that aid further exploitation. In practice bpf_probe_read has largely been replaced by the more targeted helpers [11] (bpf_probe_read_user for user memory and bpf_probe_read_kernel for kernel memory) which restrict the scope of reads.

3.1.2 bpf_probe_write_user

This helper tries, in a safe manner, to copy len bytes from a source buffer (src) to a destination address (dst) in memory. It only operates when the thread is running in user context, and dst must point to a valid user-space location. By allowing the kernel to modify user-space memory directly, it makes certain interventions, like applying immediate fixes to running processes,much simpler than alternative approaches. That capability is valuable in time-sensitive environments where changing a value on the fly is preferable to restarting a process. However, if used without strict controls it can be dangerous: unintended writes can corrupt data or open serious security vulnerabilities, and misuse may even crash the system or applications. Because of these risks, this helper is intended for experimental use; when an eBPF program employing it is attached, the kernel emits a log warning that includes the process ID and name. [11].

3.1.3 bpf_send_signal

This helper gives an eBPF program the ability to trigger signals toward user-space processes directly from within the kernel. Such functionality is valuable when immediate communication is needed, as it removes the overhead of polling the kernel for updates. Instead of constantly checking for changes, processes can react as soon as a signal is received. This mechanism is particularly useful in environments focused on security, where it can alert to suspicious activity, or in performance-sensitive systems that must respond quickly to critical events. By enabling fast notifications, it helps the system take corrective actions right away without unnecessary resource consumption. However, if abused, this capability

could seriously impact system stability, since it allows the emission of signals like SIGTERM and SIGKILL, which can terminate processes and thus be leveraged maliciously.

3.1.4 bpf_override_return

This helper enables an eBPF program to override a kernel function's return value while the system is running, directly changing how that function or system call behaves. Developers commonly use it for debugging, forcing particular outcomes lets them observe how the system responds to specific return values. It's also handy for short-term control of execution, for example to enforce security checks by blocking or altering operations that would violate policy. At the same time, modifying return values carries major hazards: forcing unexpected results can produce erratic behavior or destabilize the system. Because of these security implications, the helper is only available when the kernel is built with CONFIG_BPF_KPROBE_OVERRIDE and may be used only on functions explicitly marked with ALLOW_ERROR_INJECTION [11]. [11].

Helper Function	**Purpose**	**Required Minimum Capabilities**
bpf_probe_write_user	Write to any process's user space memory	CAP_SYS_ADMIN (& kernel lockdown[4])
bpf_probe_read_user	Read any process's user space memory	CAP_BPF & CAP_PERFMON
bpf_override_return	Alter return code of a kernel function	CAP_SYS_ADMIN
bpf_send_signal	Send a signal to kill any process	CAP_SYS_ADMIN

Table 3.1: Critical Helper functions[10]

3.2 Maps

The shared nature of the maps introduces major security risks since eBPF maps lack any isolation mechanism between programs, potentially allowing one program to interfere with one or more others.

A possible attack that exploits this attack vector is the so-called eBPF Map Tamper Attack[12] : The eBPF maps created by one program can be globally accessed by other programs via the bpf map_get_fd_by_id helper. Attackers can manipulate eBPF programs by altering their maps since eBPF programs depend on maps for receiving control configurations and exchanging data with user-space programs. Large eBPF programs like Tetragon and Datadog , which rely heavily

on eBPF tail call maps to dispatch jumps to other eBPF functions, can be fully paralyzed by deleting the map items. To exploit this type of attack a program requires CAP_SYS_ADMIN capability.

Another possible attack involves maps that are marked as read-only, meaning that neither user space nor eBPF programs should be able to modify their contents after initialization. However, a vulnerability in the verifier was discovered that allowed eBPF programs to bypass these restrictions and modify read-only maps under certain conditions. This vulnerability is tracked as CVE-2024-49861. Other potential vulnerabilities related to eBPF maps are analyzed more in depth in Section 6.1

3.2.1 CVE-2024-49861

The vulnerability [13] originated from how the BPF verifier handled arguments passed to certain BPF helpers. Some helpers accepted pointers to integers (ARG_PTR_TO_INT) or longs (ARG_PTR_TO_LONG), and the function responsible for checking these arguments (check_func_arg()) failed to mark them correctly as writable. As a result, when the verifier later checked memory access, it incorrectly treated write operations as reads, allowing unintended modifications to memory that was supposed to be read-only.

In case of registers of type PTR_TO_MAP_VALUE, the verifier treated all accesses to map values as reads. This oversight allowed certain helper functions to write to read-only maps without triggering an error, violating the expected behavior.

3.3 Verifier

There are numerous CVEs involving errors in the BPF verifier that yield high CVSS scores and can lead to severe vulnerabilities, exposing the system to attacks such as:

- Privilege escalation (e.g., CVE-2021-3490, CVE-2022-23222)
- Kernel panics (e.g. CVE-2024-56614, CVE-2024-56615)
- Information leaks (e.g. CVE-2021-4135) [14]
- Container escapes (e.g. CVE-2021-3490) [15]

Local privilege escalation due to CVE-2021-3490 will be discussed in detail in Section 6.2.

3.4 Pitfalls of relying on eBPF for security monitoring

eBPF (extended Berkeley Packet Filter) has emerged as the de facto Linux standard for security monitoring and endpoint observability. It is used by technologies such as BPFTrace, Cilium, Pixie, Sysdig, and Falco due to its low overhead and versatility. However, several documented challenges and pitfalls associated with eBPF have been identified in the literature [16] and are discussed below to ensure its safe, efficient, and informed deployment.

3.4.1 eBPF probes are not invoked

In theory, the kernel should consistently trigger eBPF probes without failure. However, in practice, there are rare instances where eBPF probes do not fire as expected when invoked by user code. This behavior is neither explicitly documented nor officially acknowledged, though indications of it can be found in bug reports related to eBPF tooling. [17]

The analysis of such bug reports provides valuable insights. First, these occurrences are infrequent and challenging to debug. Second, while the kernel may be technically correct in its execution, the observed behavior on the user side may still involve missing events, even if the immediate cause appears unrelated (e.g., an excess of probes rather than missing ones). Discussions within these reports suggest two potential explanations for missing events:

- The kernel imposes a limit on the number of active kRetProbes at any given time. As of Linux kernel version 6.4.5, this limit is set to 4,096. Any attempt to create additional kRetProbes beyond this threshold will fail, leading to missed events.
- The callback logic for kProbes and kRetProbes differs slightly, which may, in some cases, prevent a kProbe from detecting its corresponding kRetProbe, resulting in lost events.

Additional issues of this nature are likely present in the kernel, either as documented edge cases or as unintended side effects of unrelated design decisions. Since eBPF is not inherently designed as a security monitoring mechanism, there is no guarantee that probes will always trigger as expected.

Workarounds: None. The callback logic and value for the maximum number of kRetProbes are hard-coded into the kernel. While one can manually edit and rebuild the kernel source, doing so is not advisable or feasible for most scenarios. Any tools relying on eBPF must be prepared for an occasional missing callback.

3.4.2 Time-of-check to time-of-use issues

An eBPF program can and will run concurrently on different CPU cores. This is true even for kernel code. Since there is no way to call kernel synchronization functions or to reliably acquire locks from eBPF, data races and time-of-check to time-of-use issues are a serious concern.

The TOCTOU vulnerability arises when there is a discrepancy between the data checked by a tracing program and the data used by the kernel. Specifically [18], when an eBPF program is triggered at the entry point of a system call, it can inspect the arguments passed from user space. If any of these arguments are pointers, the kernel must copy the data they reference into its own memory before using it. As shown in Figure 3.1, an attacker may exploit the interval between the eBPF program's inspection and the kernel's data copying, thereby modifying the data. Consequently, the data acted upon by the kernel might differ from what the eBPF program observed.

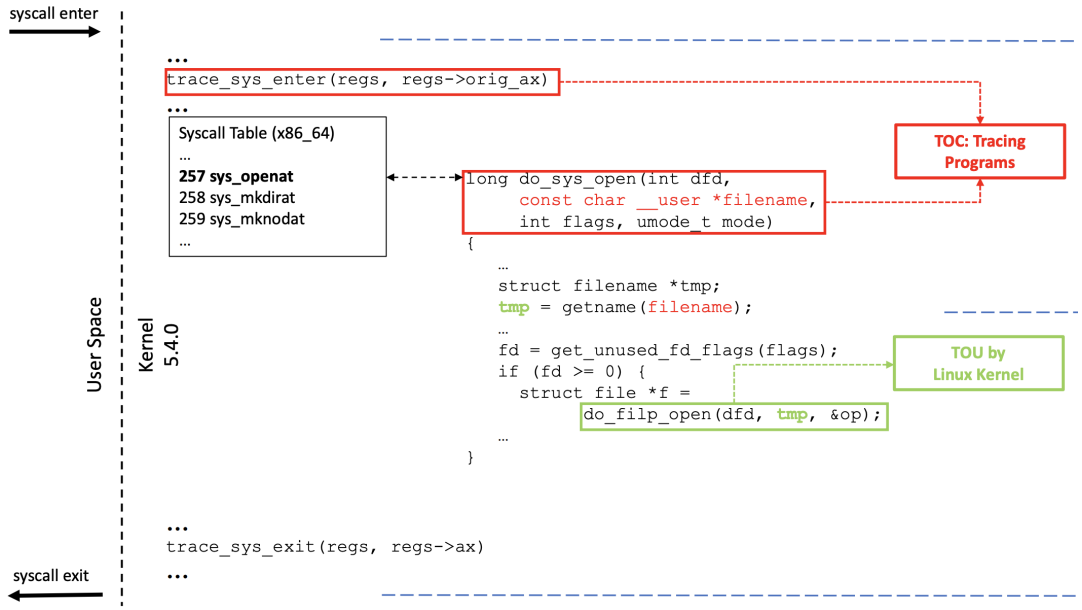

Figure 3.1: TOCTOU vulnerability in the `openat` syscall

This figure provides a more detailed overview of how a system call (such as openat, mkdirat, or mknodat) flows from user space into the Linux kernel. At the top, the static tracepoint trace sys enter(regs, regs->orig ax) captures the initial arguments passed by user space. The system call table (shown on the left) maps the call numbers (e.g., 257 for openat, 258 for mkdirat, 259 for mknodat) to their corresponding kernel handlers.

Inside the red box labeled TOC: Tracing Programs, the tracing tool (e.g., an eBPF program) can intercept and inspect the arguments before the kernel processes them. The kernel then invokes do sys open(...) (or a similar function for other calls), which copies the user-space data, like the filename pointer, into kernel memory. This kernel-side processing is highlighted in the green box labeled TOU by Linux Kernel. Here, the kernel actually uses the parameters, which may differ from those inspected by the tracing program if an attacker modifies them during the window of opportunity (real-world examples can be found in [19] or [20]). Finally, the tracepoint trace sys exit(regs, regs->ax) is called at the end of the system call, capturing the return value.

In essence, the data observed by the tracing program at syscall entry may not be the same data that the kernel ultimately operates on if it is altered after the tracing program's inspection but before the kernel copies it into its own memory. This discrepancy underscores why hooking only at the system call entry point may be insufficient for robust security applications.

Although intercepting system calls at their entry point is convenient for observability, it is not sufficient for robust security monitoring. To ensure that the eBPF program examines the same data that the kernel ultimately uses, the program should be attached to an event occurring after the parameters have been safely copied into kernel memory. Unfortunately, there is no universal interception point for all system calls because each handles data differently. However, the Linux Security Module (LSM) API offers a well-defined interface where eBPF programs can be safely attached. Unlike system call entry probes, LSM hooks are triggered after the kernel has already copied user-space data into kernel memory. This timing ensures that any subsequent checks by an eBPF program (or a similar security mechanism) will inspect the same data that the kernel will actually use. By doing so, the gap between the Time of Check and Time of Use phases is significantly reduced.

Moreover, modern security tools (such as Falco) leverage LSM hooks to detect potentially dangerous system calls or operations, for example:

- **security_bprm_check** for intercepting execve, ensuring that the executable being loaded is validated post-copy.
- **security_file_open** for monitoring file open operations (including open and openat), thereby validating pathnames already copied into kernel space
- **security_inode_unlink** for intercepting unlink calls, detecting file deletions or modifications after the kernel has performed the necessary path resolution

Because these hooks operate on data that the kernel is about to use (or has just finished preparing), they close the TOCTOU window inherent in system call entry probes. Consequently, relying on LSM hooks is an effective strategy for building

robust security tools that need to inspect system call parameters with minimal risk of data being modified in the interim.

3.4.3 Event overload

Since eBPF lacks concurrency primitives and an eBPF probe cannot block the event producer, an attach point can be easily overwhelmed with events [16]. This can lead to the following issues:

- Missed events, as the kernel stops calling the probe
- Data loss due to the lack of storage space for new data
- Data loss due to the complete overwriting of older but not yet consumed data by newer information
- Data corruption from partial overwrites or complex data formats, disrupting normal program operation

These data loss and corruption scenarios depend on the number of probes and events that are adding items into the event stream and on the extent of system activity. For instance, a docker container startup sequence or a deployment script can trigger a surprisingly large number of events.

Workarounds: The user-mode helper should treat all data coming from eBPF probes as untrusted. This includes data from your own eBPF probes, which is also susceptible to accidental corruption. There should also be some application-level mechanism to detect missing or corrupted data.

3.4.4 Page faults

Memory that has not been accessed recently may be paged out to disk, whether to a swap file, a backing file, or another location. Typically, when this memory is required, the kernel issues a page fault, loads the relevant content, and resumes execution. However, for various reasons, eBPF operates with page faults disabled [16]. If memory has been paged out, it cannot be accessed. This limitation presents a significant challenge for security monitoring tools.

Workarounds: The only workaround is to hook right after a buffer is used and hope it does not get paged out before the probe reads it. This cannot be strictly guaranteed since there are no concurrency primitives, but the way the hook is implemented can increase the likelihood of success.

3.5 BPF Door

eBPF programs are widely recognized for their powerful capabilities in monitoring and enhancing the performance of modern systems. However, as with many advanced technologies, eBPF can also be leveraged for malicious operations. One such example of the malicious use of eBPF is BPFDoor, a backdoor payload specifically crafted for Linux. Its purpose is long-term persistence in order to gain re-entry into a previously or actively compromised target environment. It notably utilizes BPF along with a number of other techniques to achieve this goal, taking great care to be as efficient and stealthy as possible. A deep analysis is conducted by the Elastic Security Team [21].

BPFDoor uses a raw socket (as opposed to "cooked" ones that handle IP/TCP/UDP headers transparently) to observe every packet arriving at the machine, including Ethernet frame headers. While this might sound like a stealthy way to intercept traffic, it actually is not: on any machine with a significant amount of network traffic, the CPU usage will be consistently high.

That is where BPF comes in, an extremely efficient kernel-level packet filter is the perfect tool to allow the implant to ignore 99% of network traffic and only become activated when a special pattern is encountered. This implant looks for a so-called magic packet in every TCP, UDP, and ICMP packet received on the system.

Once activated, a typical reverse shell, which this backdoor also supports, creates an outbound connection to a listener set up by the attacker. This has the advantage of bypassing firewalls watching inbound traffic only. However, this method is well understood by defenders. The sneakiest way to get a shell connected would be to reuse an existing packet flow, redirecting it to a separate process.

This attack is illustrated in Figure 3.2. Reading the figure from top to bottom, the initial TCP handshake is done between the attacker and a completely legitimate process,for example nginx or sshd. These handshake packets also happen to be delivered to the backdoor (like every packet on the system) but are filtered out by BPF. Once the connection is established, however, BPFDoor sends a magic packet to the legitimate service. The implant receives it, makes a note of the originating IP and port the attacker is using, and opens a new listening socket on an inconspicuous port (42391–43391).

The implant then reconfigures the firewall to temporarily redirect all traffic from the attacker's IP/port combination to the new listening socket. The attacker initiates a second TCP handshake on the same legitimate port as before, only now iptables forwards those packets to the listening socket owned by the implant. This establishes the communication channel between the attacker and the implant

that will be used for command and control. The implant then covers its tracks by removing the iptables firewall rules that redirected the traffic.

Despite the firewall rule being removed, traffic on the legitimate port will continue to be forwarded to the implant due to how Linux statefully tracks connections. No visible traffic will be addressed to the implant port (although it will be delivered there)

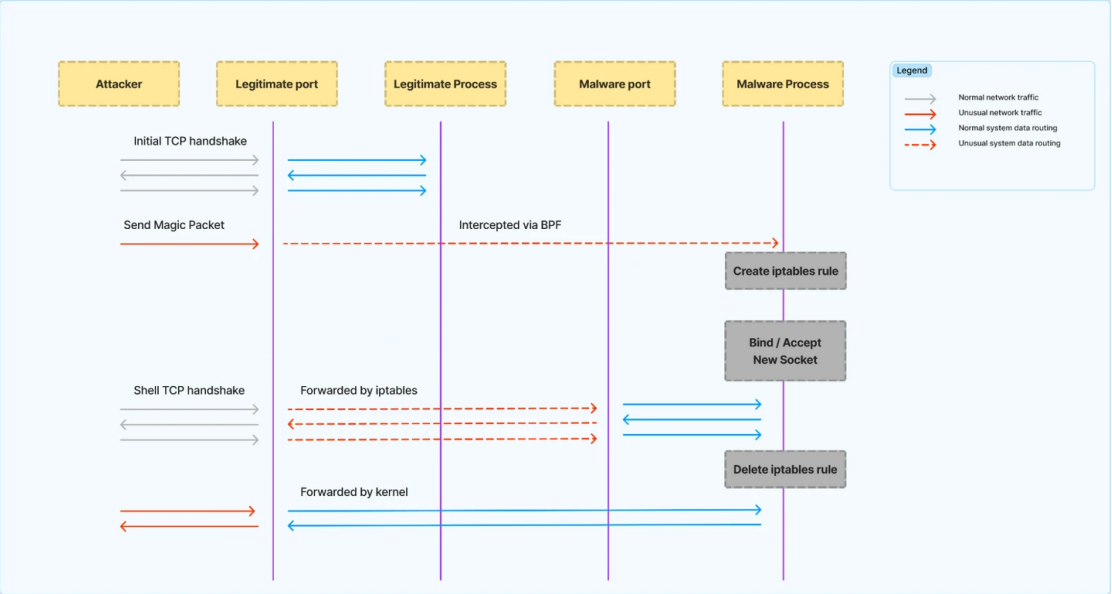

Figure 3.2: BPFDoor

Chapter 4

Linux Security Architecture

4.1 Introduction

The Linux security architecture establishes a comprehensive framework to regulate user and process access to system resources, ensuring isolation, accountability, and controlled access. Central to this architecture are two primary models: Discretionary Access Control (DAC) and Mandatory Access Control (MAC):

- **Discretionary Access Control (DAC)**: DAC [22, 23] is an identity-based access control model that gives users some control over their data. It is considered discretionary because data owners (document creators or any users authorized to control data) can define access permissions for specific users or groups of users. In other words, whom to give access to and what privileges to grant are decided at the resource owner's discretion. DAC relies on subject identification to grant access to objects. Users must supply authentication information that identifies their status before the system allows access. The access control system then decides whether the subject has the rights required to access a specific object. The basic principles of DAC are: Object characteristics (size, name, directory path) are invisible to users that aren't authorized; several failed access attempts trigger additional authentication (MFA) requirements or deny access; users can transfer object ownership to other users. The owner also determines the access type of other users. Based on these access privileges, the operating system decides whether to grant access to a file.

- **Mandatory Access Control (MAC)**: MAC [24] is a security strategy that restricts the ability individual resource owners have to grant or deny access to resource objects in a file system. MAC criteria are defined by the system administrator, strictly enforced by the operating system (OS) or security

kernel, and cannot be altered by end users. The restriction of the access to a resource (also known as an object) is based on two key factors: the sensitivity of the information contained in that resource and the authorization level of the user trying to access that resource and its information. Security teams or admins define whether a resource is sensitive or not by applying a security level, such as "Restricted," "Confidential," "Secret," or "Top Secret," to it and assigning the resource to a security category, such as "Department M" or "Project X." Together, the security level and security category constitute the security label. Admins also assign a security clearance level to each authorized user to determine which resource they can access. Once the label is applied and the MAC policy is finalized, users can only access those resources (or the information within resources) that they are entitled to access. For example, User A may be entitled to access the information within a resource labeled "Department M Restricted," but User B may not have the same authority. Similarly, User B may be entitled to access the resource labeled "Project X Confidential," but User A may not be authorized to do so. In Linux, MAC settings and policy management are often implemented through Linux Security Modules (LSMs) such as SELinux or AppArmor.

For instance, SELinux[25] enforces access control decisions based on *security labels* assigned to both system objects (such as files, directories, and sockets) and subjects (processes). Each label, also known as a *security context*, is typically composed of four fields: `user:role:type:level`. For example, a file might have the label `system_u:object_r:passwd_file_t:s0`, where `system_u` identifies the SELinux user, `object_r` represents the role, `passwd_file_t` specifies the type (the most relevant field for access control decisions), and `s0` denotes the security level.

The *security level* defines the sensitivity of the information associated with a resource, typically ranging across categories such as “Restricted”, “Confidential”, “Secret”, or “Top Secret” in multilevel security (MLS) configurations. Each process is also associated with an *authorization level* or clearance, which determines the sensitivity levels it is allowed to access.

Access control decisions are governed by SELinux policies, which define which combinations of subjects and objects (based on their types and levels) are permitted to interact. These policies are centrally defined and enforced by the operating system's security kernel, and cannot be modified by unprivileged users. Consequently, even if traditional Unix permissions allow access to a file, the MAC policy can still deny it if the security labels or clearance levels do not match the rules defined by the administrator.

In addition to access control models, process identity plays a central role in Linux security. Each process is represented internally by a data structure known as the

task_struct, which contains essential information about the process's identifiers and credentials. These identifiers include process IDs, parent process IDs, group and session IDs, as well as various user and group IDs that determine ownership and access permissions. [26]

4.1.1 Process Identifiers

Each process has a unique nonnegative integer identifier, assigned when it is created using fork, called Process ID (PID). A process can obtain its PID using getpid. PIDs are preserved across execve calls and are used in various system calls, including kill, ptrace, setpriority, setpgid, setsid, sigqueue, and waitpid.

Each process has also a Parent Process ID (PPID) identifying the process that created it. The PPID can be obtained with getppid and is preserved across execve.

Finally, processes are organized into groups and sessions to support shell job control. A process group is a collection of processes sharing the same Process Group ID (PGID), often corresponding to a single job or pipeline. A session is a collection of process groups sharing the same Session ID (SID). Children created by fork inherit their parent's PGID and SID, which are preserved across execve. New sessions can be created using setsid, with the calling process becoming the session leader. Sessions and process groups determine which process can access the controlling terminal and manage foreground and background jobs.

4.1.2 Process Credentials

Linux processes have several associated user IDs (UIDs) and group IDs (GIDs), represented as integers using the types uid_t and gid_t and stored in the cred structure (Listing 4.1. The main types of UIDs and GIDs are:

- Real UID and GID: Identify the owner of the process, accessible via getuid and getgid.
- Effective UID and GID: Used by the kernel to determine the process's permissions when accessing shared resources such as message queues, semaphores, or files. Access to files is determined by filesystem IDs. They can be obtained via geteuid and getegid.
- Saved set-UID and set-GID: Used by set-user-ID and set-group-ID programs to save the effective IDs at execution time, enabling temporary privilege changes using seteuid, setreuid, setresuid, or their GID equivalents. These saved IDs can be queried via getresuid and getresgid.
- Filesystem UID and GID: Linux-specific IDs used for file permission checks, normally aligned with effective IDs, but modifiable via setfsuid and setfsgid.

```
1  struct cred {
2      ...
3      kuid_t      uid;     /* real UID of the task */
4      kgid_t      gid;     /* real GID of the task */
5      kuid_t      suid;    /* saved UID of the task */
6      kgid_t      sgid;    /* saved GID of the task */
7      kuid_t      euid;    /* effective UID of the task */
8      kgid_t      egid;    /* effective GID of the task */
9      kuid_t      fsuid;     /* UID for VFS ops */
10     kgid_t      fsgid;     /* GID for VFS ops */
11
12     ...
13   }
```

Listing 4.1: cred structure from include/linux/cred.h

4.2 LSM Framework

The Linux Security Module (LSM) [27] framework provides a flexible hook-based infrastructure that allows various security models to be implemented as part of the Linux kernel. Despite its name, LSMs are not loadable kernel modules; they are integrated at build-time, selected via the CONFIG_DEFAULT_SECURITY configuration, and can be overridden at boot-time using the security=... boot parameter if multiple LSMs are built into the kernel.

LSMs insert security hook functions into various kernel subsystems [28], enabling access control checks that go beyond traditional Discretionary Access Control (DAC). Importantly, LSM does not enforce security policies itself; it merely supplies the necessary infrastructure. In the absence of a more specific module, the default LSM is the capabilities system, which offers basic access control mechanisms

The primary users of the LSM framework are Mandatory Access Control (MAC) extensions that enforce comprehensive security policies, such as SELinux, AppArmor, Smack, TOMOYO, and Yama.
These modules typically build upon the capabilities infrastructure, adding targeted enforcement and policy controls.

4.3 BPF LSM

Before BPF, administrators had two main ways to implement custom security policies: configuring an existing Linux Security Module (LSM) such as AppArmor or SELinux, or writing a custom kernel module. Since Linux 5.7, the LSM BPF framework [29] has introduced a more flexible alternative. It allows security policies

to be implemented as eBPF programs attached to LSM hooks, enabling fine-grained, runtime-modifiable rules without the need to develop kernel modules, which are harder to maintain and can destabilize the system.

BPF-LSM programs are verified by the kernel at load time, ensuring safety, and can attach to individual LSM hooks to make precise, context-aware decisions for specific events (e.g., checks on particular processes or files), enabling security policies with a level of detail that traditional approaches rarely achieve. Unlike conventional LSMs such as SELinux or AppArmor, which are fixed at compile time and often not stackable (i.e., mutually exclusive) [30], BPF-LSM offers dynamic loading, runtime updates, and stackability. This flexibility, combined with its granular control, makes BPF-LSM particularly well-suited for prototyping advanced security mechanisms and for environments where policies must evolve quickly.

4.3.1 Challenges and Limitations

Despite its advantages in flexibility and fine-grained control, BPF-LSM also presents some potential drawbacks. First, BPF-LSM inherently presents a larger attack surface. Writing a BPF-LSM program requires implementing full code logic, whereas traditional LSM typically only requires specifying policy rules, without complex kernel-level code. Furthermore, BPF programs pass through the verifier and, optionally, the JIT compiler, both of which can contain vulnerabilities. Historically, there have been several verifier bypasses, illustrating that the additional complexity of BPF programs can introduce potential security risks. Second, kernel compatibility can be an issue: while traditional LSM have been integrated into Linux for a long time and support a wide range of kernel versions, BPF-LSM relies on newer kernel features and may not be available or fully functional on older systems. Finally, the performance overhead of BPF-LSM should be considered; although it provides granular control and runtime flexibility, the complexity of BPF programs and verification can potentially introduce additional computational cost.

Despite these challenges, BPF-LSM was chosen as the main hardening solution in this work due to its power and fine-grained control. Traditional LSM were not well-suited to protect against vulnerabilities within eBPF itself, since they cannot fully integrate with eBPF programs or attach to eBPF hooks (sometimes being unable to access them at all, and in other cases providing only coarse-grained control). Nonetheless, alternative hardening strategies were also considered and proposed to complement BPF-LSM and provide a more comprehensive security posture.

4.4 Kernel Modules

While a monolithic kernel is generally faster than a microkernel, it traditionally suffers from limited modularity and extensibility. Modern monolithic kernels address this limitation through loadable kernel modules [31, 32, 33], which can be dynamically inserted into or removed from the kernel as needed. These modules, compiled as object files, allow new functionality to be added at runtime and can be unloaded when no longer required. Common uses include adding support for new hardware (device drivers), filesystems, or system calls. In Linux, for instance, modules can be managed using commands such as `insmod`, `rmmod`, and `modprobe`. This approach can also help reduce the size of the kernel core and improve boot time, since some drivers are only loaded when the corresponding hardware is in use.

However, the use of kernel modules also introduces some disadvantages. The main ones are:

- **Loss of stability:** poorly written or incompatible modules can cause system crashes or instability.
- **Security risks:** modules run with kernel privileges, so a malicious or vulnerable module can compromise the entire system.
- **Compatibility issues:** modules must be compatible with the specific kernel version and configuration, which can lead to maintenance challenges.

In Section 6.2, we will examine a kernel module that can be used for security monitoring and hardening, exploring both its advantages and disadvantages.

Chapter 5

eBPF-based products

eBPF has been widely adopted by companies to develop security monitoring tools. One notable example is Tetragon, which has been evaluated in this work as a potential hardening solution against specific eBPF-related vulnerabilities.

5.1 Tetragon

Tetragon [34] is an open-source runtime security observability and enforcement tool built on eBPF, developed by Isovalent as part of the Cilium ecosystem. It provides deep real-time visibility into system behavior, including process execution, system call activity, file access, network connections, and I/O operations, without requiring additional kernel modules. By leveraging eBPF hooks, as well as certain BPF-LSM hooks, Tetragon can monitor and react to security-relevant events efficiently within the kernel, avoiding the overhead and potential inaccuracies associated with user-space tracing.

Tetragon allows fine-grained filtering and policy enforcement, enabling users to specify which events are relevant based on function arguments, return values, and process metadata such as executable names and capabilities. However, this granularity has some limitations: for instance, syscall arguments that are complex structures cannot always be fully analyzed. Events can be intercepted at kernel hooks (kprobes and tracepoints) and user-space hooks (uprobes), and Tetragon supports inline enforcement by overriding function return values or sending signals to processes, providing immediate mitigation of suspicious activity.

The tool exposes captured events via gRPC or JSON logs, offering rich observability while minimizing performance impact by applying filtering and policy enforcement directly in the kernel. Its flexibility allows users to create highly specific tracing and enforcement policies, addressing a wide range of security and monitoring use cases, including intrusion detection, compliance, and runtime protection against

potential vulnerabilities.

5.1.1 Policies

At the core of Tetragon's functionality are its policies, which define how the tool observes and reacts to system events. These policies allow users to configure Tetragon's behavior by specifying which activities to monitor and what enforcement actions to take in response to particular conditions.

Tracing Policies can be loaded or unloaded at runtime, or defined at startup using configuration flags. The key elements that define a policy include the **hook point** (Tetragon supports *kprobes*, *tracepoints*, *uprobes*, and some *BPF-LSM* hooks), **selectors**, and **actions**. Users can configure hook points in the corresponding sections of the `TracingPolicy`. These hook points may expose arguments and return values, which can be specified using the `args` and `returnArg` fields.

Selectors enable per-hook, in-kernel BPF filtering and action execution. Each selector defines a set of filters and, optionally, a set of actions to be performed when those filters match. A single hook can include up to five selectors; if no selectors are defined, the default action (`Post`, i.e., log an event) is applied.

Selectors are composed of classes of filters. For example, `matchArgs` filters based on argument values, `matchData` filters based on data fields, `matchReturnArgs` filters on return values, and `matchPIDs` filters on process IDs. Actions are specified using either `matchActions`, which applies actions when a selector matches, or `matchReturnActions`, which applies actions upon return selector matching.

To illustrate a simple example of a Tetragon policy, consider Listing 5.1.

```
1  apiVersion: cilium.io/v1alpha1
2  kind: TracingPolicy
3  metadata:
4    name: "lsm-file-open"
5  spec:
6    lsmhooks:
7    - hook: "file_open"
8      args:
9        - index: 0
10         type: "file"
11     selectors:
12     - matchBinaries:
13       - operator: "In"
14         values:
15         - "/usr/bin/cat"
16       matchArgs:
17       - index: 0
18         operator: "Equal"
19         values:
```

```
20          - "/etc/passwd"
21          - "/etc/shadow"
```

Listing 5.1: Example of a Tetragon policy monitoring file access

In this example, the policy uses the `file_open` LSM hook to monitor file access events. The first selector, `matchBinaries`, ensures that the policy triggers only when the `/usr/bin/cat` binary attempts to open a file. The second selector matches the argument specified in the hook section, verifying that the file being accessed by `cat` is either `/etc/passwd` or `/etc/shadow`. Since no explicit action is defined, the default `Post` action (i.e., event logging) is applied.

Chapter 6

Analysis, Exploitation and Hardening of eBPF Vulnerabilities

For this thesis work, the methodology consisted of analyzing several known eBPF vulnerabilities with the goal of identifying and evaluating potential hardening strategies against different categories of attacks, including privilege escalation, kernel panics, and map-related attacks. The workflow followed a structured process: first, a high-severity CVE or a potential attack vector was selected; then, existing proofs of concept (PoCs) were studied and reproduced; finally, mitigation strategies were designed, implemented, tested and evaluated. The proposed solutions were developed with the intention of being as general as possible, so as to remain effective even in the presence of slight variations of known exploits.

This thesis was carried out in collaboration with a colleague, and the overall work was divided into four use cases. As practical demonstrations, two representative use cases are discussed in this thesis: the first focuses on the protection of eBPF maps against different types of attacks, while the second addresses privilege escalation exploits, with particular attention to CVE-2021-3490. The remaining two are covered in the companion thesis.

6.1 Use Case 1: Maps Protection

eBPF maps, as introduced in Section 2.1.4, serve as the primary communication mechanism for eBPF programs and are widely used to store critical data. Security monitoring tools built on eBPF often leverage maps to hold configuration parameters and global state for feature control, which makes them a particularly attractive

target for attackers for the reasons discussed in Section 3.2.

6.1.1 The Vulnerability

eBPF Rootkits

Rootkits are stealthy tools that attackers use to maintain persistent access to compromised systems, even after credential changes or vulnerability patches. By hooking into the syscall table, traditional kernel rootkits gain deep visibility and control over the operating system, enabling attackers to monitor network traffic, hide files and processes, and spawn privileged tasks. This approach offers significant stealth capabilities but introduces severe risks: even minor bugs in kernel rootkits can crash the kernel, leading to full system failure, and kernel updates frequently break rootkit functionality due to their reliance on low-level system structures.

eBPF provides a powerful alternative for building rootkits [35] that circumvent many of the stability issues inherent in kernel modules. Although originally designed for in-kernel observability and performance optimization, eBPF has been maliciously repurposed to create stealthy malware. These eBPF-based rootkits are capable of avoiding the catastrophic failures and compatibility issues tied to kernel updates. This allows attackers to perform network manipulation, stealth communication, process hiding, and privilege escalation while remaining largely invisible to standard security auditing tools.

One application of such rootkits is tampering with eBPF maps used by security monitoring tools. By modifying these maps, attackers can silently disable or alter the behavior of monitoring systems without triggering obvious indicators, such as terminating a security agent's process. This stealthy approach allows them to remain concealed while effectively neutralizing defensive mechanisms.

For these reasons, monitoring both the loading of eBPF programs and their access patterns to eBPF maps is essential as an additional layer of defense, complementing traditional privilege checks and other security controls.

File Descriptor Hijacking Attack

eBPF maps are inherently associated with file descriptors. They are exposed to user-space through file descriptors: when a process creates or accesses a map via the `bpf()` syscall, the kernel returns a file descriptor used for subsequent operations. During program loading, eBPF programs also temporarily use these file descriptors to reference maps, allowing the kernel to resolve them to internal pointers. Once the program is loaded, it interacts with the maps directly via kernel pointers and helper functions, without requiring file descriptors for execution.

Each BPF program accesses maps using the addresses of objects in kernel space via BPF helper functions, while user-space interacts with maps by issuing BPF system calls. Alternatively [36], user-space can access maps by retrieving a file descriptor and memory-mapping it (`mmap`) into its virtual address space. Once mapped, the map can be accessed through simple pointers instead of system calls.

Each map has attributes defined in a `bpf_attr` structure, including `map_flags`. Only maps created with the `BPF_F_MMAPABLE` flag can be memory-mapped. Notably, `libbpf`, one of the most widely used libraries for eBPF, implicitly creates a BPF map for global data and memory-maps it into the current process. This makes these maps memory-mappable and reduces the need to repeatedly call `bpf_map_lookup,update_elem`, improving overall performance.

Since Linux 5.6, a new syscall called *pidfd_getfd* allows a process (with `CAP_SYS_PTRACE`) to obtain a duplicate of a file descriptor from another target process. Because file descriptors within a process are allocated sequentially and are limited in number, it is possible to iterate over a process's file descriptors (brute force) to inspect them. By checking descriptor attributes, such as the `BPF_F_MMAPABLE` flag, an attacker can identify a descriptor of interest, duplicate it with the *pidfd_getfd* syscall, and use `mmap` to access the corresponding kernel object and tamper with it. Some eBPF-based security solutions store configuration in global variables, making them memory-mappable and vulnerable to this type of attack. A real-world example is the security monitoring tool Falco, which was successfully exploited in this manner in 2025 [36]:

Falco recently introduced a modern BPF probe that uses a dispatcher to delegate work to BPF programs when a system call is entered. A system call reaches the BPF programs only if it passes filtering based on interest and sampling logic, using the global table `g_64bit_interesting_syscalls_table`. This means that if an attacker gains write access to the memory region containing `g_64bit_interesting_syscalls_table`, they could disable event dispatching by setting all entries to `false`.

The exploits, as described previously, iterate over possible file descriptors looking for those marked as `BPF_F_MMAPABLE`. Once such a descriptor is found, the exploit clones the descriptor (via `pidfd_getfd`) to access a map holding global variables. Finally, by performing a memory write (for example `memset`) at the offset of the `g_64bit_interesting_syscalls_table`, the exploit zeros out all table entries and disables this security function completely. After that, all system calls that would normally pass through the dispatcher are silently dropped; for example, executing `cat /etc/shadow` no longer generates any alerts. An attacker could later re-enable the entries after causing damage or modify other settings stored in the program's global variables.

6.1.2 Exploitation

To evaluate the attacks discussed, three programs were developed. The first is a simple eBPF program called *simple_program* that accesses the map to be protected and contains a global variable for testing the file descriptor hijacking attack. The second program acts as a malicious rootkit, attempting to access the protected map value to verify whether the protection is effective. The third program is a user-space application that uses eBPF system calls to ensure that the protection also applies from user space. Additionally, the exploit proposed by Mouad Kondah [36] for the file descriptor attack was downloaded, studied, and slightly adapted to fit this simplified use case. The initial results, without any protections, showed that, given the appropriate permissions, the map used by *simple_program* was fully accessible to the other eBPF programs, and the file descriptor attack was able to tamper with the global variable, setting it to a value of choice. These attacks were tested on Ubuntu Server running kernel version `6.8.0-64-generic`. Neither attack has an upstream patch, because they do not exploit a software bug per se but rather abuse existing kernel features and require certain privileges. An eBPF rootkit exploits the shared nature of eBPF maps and can potentially affect any kernel version that supports eBPF, provided an attacker is able to install and load the rootkit. The file-descriptor-based attack leverages a system call introduced in Linux `5.6` and therefore affects all kernel versions from `5.6` onward.

6.1.3 Hardening Plan

LSM Protection

The first approach to mitigate these threats was to create an LSM-BPF program. The program used two hooks: *lsm/bpf_map*, which is triggered whenever a process obtains a file descriptor for a map and can therefore monitor both user-space access and eBPF program loading, and *lsm/mmap_file*, which is triggered during memory mapping operations. To identify the map to be protected, the loader of the *simple_program* collected information such as the map ID. These data are used in the program's checks.

To avoid completely blocking access to the protected map, which would interfere with normal communication, an allowlist solution was implemented. The allowlist is based on parameters that identify executable files authorized to access the map. Since relying solely on the file path can be error-prone, the program uses the inode number and superblock device to uniquely identify each executable. An inode number is unique within a single filesystem but may be repeated across different filesystems. The superblock device identifies the filesystem hosting the file and is unique per mounted filesystem. By combining the inode number and superblock device, a file can be uniquely identified across all mounted filesystems.

This combination is therefore sufficient to uniquely identify an executable or any file in a system with multiple mounted filesystems.

For simplicity, the allowlist was limited to a single pair of inode number (`i_ino`) and superblock device (`s_dev`), allowing, potentially, only one process to access the protected map.

The first hook, *lsm/bpf_map*, when triggered, exposes information about the `bpf_map` structure of the map for which the process is attempting to obtain a file descriptor. The logic within this hook retrieves the inode number (`i_ino`) and superblock device (`s_dev`) of the process and performs a series of checks. First, it verifies whether the `map_id` obtained from the `bpf_map` structure corresponds to the id of the map being protected. If so, it then checks whether the process attempting access matches the one allowed in the allowlist. If the last check fails, access is denied: for a malicious eBPF program, loading is blocked; for a user-space process, an "operation not permitted" error is returned.

Similarly, the second hook, *lsm/mmap_file*, exposes the `file` structure of the file being memory-mapped. Since this hook does not provide the map ID, we must rely on a less granular check. First, the `i_ino` and `s_dev` of the process that triggered the hook are extracted. Then, from the file structure, the `s_magic` value is retrieved. This is a numeric constant that identifies the type of filesystem (or pseudo-filesystem) to which the file belongs. In particular, an eBPF map can be associated with two types of filesystems:

- `BPF_FS_MAGIC = 0xCAFE4A11`: the magic number of the BPF filesystem, a pseudo-filesystem used to represent eBPF maps and programs in the kernel. When an eBPF map is created and pinned to the BPF filesystem, it can be accessed through a file descriptor pointing to a file in this filesystem.

- `ANON_INODE_FS_MAGIC = 0x09041934`: the magic number of the anonymous inode filesystem. Anonymous inodes are special inodes that do not correspond to any file in a real filesystem. They are typically used for kernel objects that need to be represented as files but do not have a persistent presence on disk, such as epoll, signalfd, and eBPF maps.

In our case, the map representing the global variable targeted by the file descriptor attack resides in the anonymous inode filesystem. The reason why we do not rely on the inode number and `s_dev` to identify the map is that, within the anonymous inode filesystem, inode numbers are not guaranteed to be unique, making them unsuitable for uniquely identifying a map.

Once this information is extracted, the hook checks whether `s_magic` corresponds to one of the two values above. If so, it means that the file being memory-mapped is potentially an eBPF map. At that point, the process is further verified against

an allowlist. If the process is not authorized, the memory mapping is blocked and an "operation not permitted" error is returned. This implies that we must maintain an allowlist of processes allowed to perform memory mappings on files belonging to `ANON_INODE_FS` or `BPF_FS`.

Together, these hooks ensure that only the explicitly allowed process can access the protected map, whether through file descriptor acquisition or memory mapping, effectively mitigating both user-space and eBPF-based attacks.

To initiate the LSM protection alongside the *simple_program* and obtain the necessary information, a simple wrapper script was created. This script first starts the *simple_program*, which writes the map information, such as the map ID, to temporary files. The wrapper then reads this value, optionally sets the parameters of the allowed process (if any), and immediately launches the LSM loader program, passing the collected values as arguments. The implementation of the LSM protection code is provided in Appendix A.1 .

Why not just eBPF

A natural question when designing kernel-level security mechanisms is: why not rely solely on standard eBPF instrumentation, such as kprobes, tracepoints, or uprobes, instead of using BPF-LSM? After all, these mechanisms already allow developers to dynamically hook into kernel functions and monitor system behavior. The answer is that BPF-LSM extends the capabilities of traditional eBPF tracing by hooking directly into the Linux Security Module (LSM) framework, allowing security policies to be enforced at critical points in the kernel's security flow (with benefits also seen in Section 3.4.2), something that generic probes cannot provide. While kprobes, tracepoints, and uprobes are powerful for observability, they are primarily designed for passive monitoring and lack a structured enforcement framework. In contrast, BPF-LSM enables active security decisions and enforcement actions to occur in real time, making it a more robust solution for hardening.

Traditional probes also suffer from a lack of context: they do not inherently provide information about user credentials, process lineage, or Mandatory Access Control (MAC) contexts, which are crucial for nuanced security decisions. The LSM framework, however, is deeply integrated into the kernel's access control mechanisms, enabling BPF-LSM programs to operate with rich metadata and apply fine-grained, context-aware policies across filesystems, networking, and process management.

For example, restricting access to sensitive files (e.g., within /etc/secret/) to specific users or processes is a classic MAC use case. Achieving this with traditional eBPF techniques, such as intercepting open() syscalls, introduces several limitations: enforcement is limited to a single syscall, leaving other operations like read, write,

or execute unchecked, requiring developers to monitor and maintain multiple syscall hooks to achieve equivalent protection. Each kernel subsystem (e.g., networking vs. filesystem) has its own functions to monitor, increasing complexity. Without a centralized framework it is difficult to ensure all execution paths are covered: missing a single syscall or internal function can bypass security checks, and implementing multi-level security policies across subsystems becomes error-prone. LSM provides this centralized framework by providing standardized hooks at key kernel points, such as inode_permission, task_alloc, and socket_connect. Every security-relevant operation passes through these hooks, regardless of the subsystem, allowing a policy to be written once at the appropriate hook and reliably enforced across the entire kernel, as the hooks are integrated into the official security flow. In contrast, relying solely on kprobes or tracepoints leads to fragmented, manual enforcement. BPF-LSM hooks, such as inode_permission, offer unified, consistent enforcement, ensuring comprehensive coverage while reducing the attack surface.

Tetragon

While powerful, as we saw in the previous section, where LSM-BPF allowed extracting very low-level information and implementing custom logic, this solution is only available starting from Linux 5.7, leaving many kernel versions unprotected. Although the FD attack is only possible in kernels from version 5.6 onward, malicious eBPF programs exist on any kernel that supports eBPF, so additional protections are desirable. For this reason, a complementary alternative solution based on Tetragon was considered. As described in Section 5, Tetragon is an eBPF-based security monitoring tool that can be configured using policies, providing real-time observability and enforcement capabilities.

In this approach, a policy was written to hook the `bpf` syscall via a kprobe and define a list of allowed binaries. Any process not in this allowlist attempting to execute the `bpf` syscall is terminated, which simultaneously prevents unauthorized userspace programs and the loading of malicious eBPF programs. Compared to the LSM solution, this approach is less granular: it cannot selectively protect individual maps while leaving others accessible, but instead blocks eBPF usage for any process not explicitly trusted. This lack of granularity, however, may be beneficial in contexts where a stricter, simpler-to-manage policy is desired.

For the FD attack, Tetragon can also monitor `mmap` syscalls, though it lacks access to the underlying file structure available to LSM, such as inode and superblock identifiers. A complete protection in this scenario would likely require a supplementary userspace program to analyze Tetragon logs and make decisions, a possibility left outside the scope of this thesis but noted as a potential theoretical alternative. The implementation of the Tetragon policy is provided in Appendix A.2

It is important to note that eBPF programs already loaded in the kernel do not rely on file descriptors or syscalls for operations (conversely to userspace programs, which always use the BPF syscall), but instead use helper functions and internal kernel pointers. As a result, such programs cannot be tracked or blocked by any LSM hooks once loaded. Consequently, if a malicious eBPF program is present before the deployment of protections, it can evade this hardening measure.

Various strategies can be employed to mitigate unauthorized or malicious usage of already loaded programs. One approach is scanning for files that contain eBPF programs, which is simpler when they are compiled using LLVM and LibBPF, although other loading methods exist. When using bpftool and LibBPF, the ELF file often embeds the eBPF bytecode in the loader's .rodata section. Another important detection method involves monitoring calls to bpf_probe_write_user, a function that can be abused for unauthorized memory modification; analysis can focus on the bytecode representation of this instruction, keeping in mind that JIT compilation may alter its appearance in the kernel. Finally, the LSM solution itself generates certain maps during program loading, but these are immediately protected by the LSM policy, ensuring that new attacks targeting these maps are prevented.

6.2 Use Case 2: CVE-2021-3490

This vulnerability, along with the analyzed PoC, targets kernels 5.8.0-25.26 through 5.8.0-52.58 and 5.11.0-16.17. It consists of an eBPF verifier bypass caused by incorrect bounds tracking in bitwise operations (AND, OR, XOR), which results in registers with invalid bounds. This flaw enables kernel information leaks, arbitrary kernel read/write, and ultimately local privilege escalation (LPE). A detailed explanation of CVE-2021-3490 is provided, following the analyses presented in [4] and [37].

6.2.1 The vulnerability

The eBPF instruction set can operate either on the full 64 bits of a register or on its lower 32 bits. For this reason, the verifier's range tracking maintains separate bounds for the lower 32 bits of each register (as discussed in Section 2.1.2): {u,s}32_min,max_value.

These bounds are updated after every operation. Each operation has two tracking functions, one for 64-bit values and one for 32-bit values, denoted as `scalar[32]_min_max_*`. For 64-bit operations, both functions are invoked within the function `adjust_scalar_min_max_vals`.

```
1  static int adjust_scalar_min_max_vals(
2                  struct bpf_verifier_env *env,
3                  struct bpf_insn *insn,
4                  struct bpf_reg_state *dst_reg,
5                  struct bpf_reg_state src_reg)
6  {
7      ...
8      case BPF_ADD:
9              scalar32_min_max_add(dst_reg, &src;_reg);
10             scalar_min_max_add(dst_reg, &src;_reg);
11             dst_reg->var_off = tnum_add(dst_reg->var_off, src_reg.
    var_off);
12             break;
13     ...
14     case BPF_AND:
15         dst_reg->var_off = tnum_and(dst_reg->var_off, src_reg.
    var_off);
16         scalar32_min_max_and(dst_reg, &src_reg);
17         scalar_min_max_and(dst_reg, &src_reg);
18         break;
19     case BPF_OR:
20         dst_reg->var_off = tnum_or(dst_reg->var_off, src_reg.
    var_off);
21         scalar32_min_max_or(dst_reg, &src_reg);
22         scalar_min_max_or(dst_reg, &src_reg);
23         break;
24     case BPF_XOR:
25         dst_reg->var_off = tnum_xor(dst_reg->var_off, src_reg.
    var_off);
26         scalar32_min_max_xor(dst_reg, &src_reg);
27         scalar_min_max_xor(dst_reg, &src_reg);
28         break;
29     ...
30         __update_reg_bounds(dst_reg);
31         __reg_deduce_bounds(dst_reg);
32         __reg_bound_offset(dst_reg);
33         return 0;
34 }
```

Listing 6.1: Excerpt from kernel source code - adjust_scalar_min_max_vals function

The bug behind CVE-2021-3490 lies in the 32-bit tracking functions for the `BPF_AND`, `BPF_OR`, and `BPF_XOR` operations, and occurs in the same way across all of them.

```
1  static void scalar32_min_max_and(struct bpf_reg_state *dst_reg,
2  struct bpf_reg_state *src_reg)
3  {
```

```
4           bool src_known = tnum_subreg_is_const(src_reg->var_off);
5           bool dst_known = tnum_subreg_is_const(dst_reg->var_off);
6           struct tnum var32_off = tnum_subreg(dst_reg->var_off);
7           s32 smin_val = src_reg->s32_min_value;
8           u32 umax_val = src_reg->u32_max_value;
9           /* Assuming scalar64_min_max_and will be called, so it is
        safe
10           * to skip updating the register for the known 32-bit case
        .
11           */
12          if (src_known && dst_known)
13                  return;
14      ...
15  }
```

Listing 6.2: Excerpt from kernel source code - scalar32_min_max_and function

As shown in Listing 6.2, if both the source and destination registers are known (i.e., they hold constant values), the function simply returns without updating the 32-bit minimum and maximum values. This behavior relies on the assumption that the corresponding 64-bit tracking function, `scalar_min_max_and`, will perform the necessary updates.

Examining the implementation of `scalar_min_max_and`, we observe the following:

```
1   static void scalar_min_max_and(struct bpf_reg_state *dst_reg,
2                                  struct bpf_reg_state *src_reg)
3   {
4           bool src_known = tnum_is_const(src_reg->var_off);
5           bool dst_known = tnum_is_const(dst_reg->var_off);
6           s64 smin_val = src_reg->smin_value;
7           u64 umax_val = src_reg->umax_value;
8
9           if (src_known && dst_known) {
10                  __mark_reg_known(dst_reg, dst_reg->var_off.value);
11                  return;
12          }
13          ...
14  }
```

Listing 6.3: Excerpt from kernel source code - scalar_min_max_and function

Here, when both `src_known` and `dst_known` are true, the function calls `__mark_reg_known`. However, the problem arises because `scalar32_min_max_and` evaluates registers using `tnum_subreg_is_const`, which returns true if only the lower 32 bits of a register are constant. In contrast, `scalar_min_max_and` uses `tnum_is_const`, which returns true only if all 64 bits are constant.

This mismatch violates the assumption in the comment of `scalar32_min_max_and`

when a register has the lower 32 bits known and the upper 32 bits unknown: the function may return without updating bounds, relying on the 64-bit counterpart to do so, but the latter might not trigger an update if only the lower 32 bits are known while the upper 32 bits are not.

Finally, as shown in Listing 6.1, the last three functions before the `return` perform one final update of the register bounds. Each of these functions has 32-bit and 64-bit counterparts.

`__update_reg_bounds` sets the minimum bounds either to the current minimum or to the known value of the register, whichever is larger. Similarly, the maximum bounds are set either to the current maximum or to the known value of the register, whichever is smaller. After that, `__reg_deduce_bounds` combines the information from the signed and unsigned bounds to update each other. Finally, the unsigned bounds are used to update `var_off` in `__reg_bound_offset`.

Now consider the instruction `BPF_ALU64_REG(BPF_AND, R2, R3)`. This instruction performs a bitwise AND operation between registers `R2` and `R3`, storing the result in `R2`.

Suppose that `R2` has `var_off = {mask = 0xFFFFFFFF00000000; value = 0x1}` (refer to Section 2.1.2 for the explanation of `var_off`), which means that the lower 32 bits are known to hold the value `1`, while the upper 32 bits are unknown. Since the lower 32 bits of the register are known, its 32-bit bounds equal that value.

Let `R3` have `var_off = {mask = 0x0; value = 0x100000002}`, meaning that all 64 bits are known and equal to `0x100000002`.

The steps to update the 32-bit bounds of `R2` are as follows.

As shown on line 14 of Listing 6.1, the function `tnum_and` is called. This performs the AND operation and updates the `var_off` field of the destination register `R2`. Recall that the lower 32 bits in both registers are known, so a normal AND operation is performed and the result for the lower 32 bits is 0. In `R3` the upper 32 bits are also known: the upper 31 bits are zero (so the AND operation will give zero independently of the value of the corresponding `R2` bits), and the 32nd bit is one (so the result of the AND operation is unknown and depends on the value of the 32nd bit of `R2`). Therefore, `R2` ends up with `var_off = {mask = 0x100000000; value = 0x0}`.

On the next line, `scalar32_min_max_and` is called. We already know that this function will return immediately without modifying the bounds, since the lower 32 bits of both registers are known. Next, `scalar_min_max_and`, as noted above, will not treat the register as unknown for the reason explained previously, and so it will not mark the lower 32 bits as known, thereby missing the correct update of the bounds.

Then, `__update_reg_bounds` is invoked. The 32-bit counterpart of this function that is responsible for updating the lower 32-bit bounds sets `u32_max_value =`

0, because var_off.value = 0 is less than the previous u32_max_value = 1. Similarly, it sets u32_min_value = 1, since var_off.value = 0 is less than the previous u32_min_value. The same occurs for the signed bounds.

Finally, the functions __reg_deduce_bounds and __reg_bound_offset do not modify the bounds further.

In this case, we end up with a register state where {u,s}32_max_value = 0 < {u,s}32_min_value = 1, i.e., with a maximum bound lower than the minimum bound. As we will see in the next section, this inconsistency can be exploited to gain arbitrary kernel memory read and write capabilities and ultimately achieve privilege escalation.

6.2.2 Exploitation

The PoC for this vulnerability was published in [4] and follows this structure: First of all, the exploit creates two maps called oob_map and store_map. The following step is to trigger the vulnerability to leak the addresses of the two maps: as discussed in Section 2.1.2, the verifier does not allow pointers to be stored into maps. However, by exploiting the bug in the verifier, it is possible to make it believe that a pointer is actually a valid integer.

We first prepare the necessary registers as follows. We perform a map_lookup_elem on both maps and store the pointer to their first element in two registers that we will call OOB_MAP_REG and STORE_MAP_REG. Then we load the value pointed to by STORE_MAP_REG into a register that we will call EXPLOIT_REG. Since the value is loaded from a map it is unknown to the verifier at compile time (it is set so that at runtime the map will contain the value 0), and thus EXPLOIT_REG will have a tnum mask of 0xFFFFFFFFFFFFFFFF. Next we load the 32-bit value 0xFFFFFFFF into another register, CONST_REG, and shift it left by 32 bits. This sets CONST_REG to 0xFFFFFFFF00000000. We then perform an AND operation between CONST_REG and EXPLOIT_REG, storing the result back into EXPLOIT_REG. After this operation EXPLOIT_REG has a tnum mask of 0xFFFFFFFF00000000, meaning the register is half-known and half-unknown. Finally, we add 0x1 to EXPLOIT_REG, obtaining for this register var_off = {mask = 0xFFFFFFFF00000000; value = 0x1}, consistent with the steps analyzed in the previous section.

To build the second register needed for the exploit, we craft a fully known register with the value 0x100000002 by setting CONST_REG = 1, shifting left by 32 bits, and adding 2. CONST_REG will then have var_off = {mask = 0x0; value = 0x100000002}. Finally, we perform the AND operation between EXPLOIT_REG and CONST_REG, storing the result back into EXPLOIT_REG (which will have value 0 at runtime). This triggers the bug in the verifier and sets {u,s}32_max_value = 0 < {u,s}32_min_value = 1 for EXPLOIT_REG.

Once this is done, to leak the addresses of the maps we must recall that the verifier tracks which registers contain pointers and which contain scalars, and it normally disallows storing pointers in maps. Looking at Listing 6.4, we see the function `adjust_ptr_min_max_vals`, which is called for arithmetic instructions involving a pointer. The register `off_reg` in the listing represents the offset scalar register being added to a pointer register. Notice that if the offset register has bounds such that `umin_value > umax_value`, the function `__mark_reg_unknown` is called on the destination (pointer) register. This function marks the register as an unknown scalar value.

Therefore, if we have `EXPLOIT_REG` with bounds `umin_value > umax_value` and add it to a destination pointer register, the pointer register will be converted to a scalar. The verifier will no longer treat the register as a pointer and will allow us to leak its value to user space by storing it in an eBPF map. We use this to save the `OOB_MAP_REG` pointer into `store_map`.

```
1  static int adjust_ptr_min_max_vals(struct bpf_verifier_env *env,
2             struct bpf_insn *insn,
3             const struct bpf_reg_state *ptr_reg,
4             const struct bpf_reg_state *off_reg)
5  {
6  ...
7  bool known = tnum_is_const(off_reg->var_off);
8    s64 smin_val = off_reg->smin_value, smax_val = off_reg->
       smax_value,
9        smin_ptr = ptr_reg->smin_value, smax_ptr = ptr_reg->
       smax_value;
10   u64 umin_val = off_reg->umin_value, umax_val = off_reg->
       umax_value,
11       umin_ptr = ptr_reg->umin_value, umax_ptr = ptr_reg->
       umax_value;
12 ...
13
14 if ((known && (smin_val != smax_val || umin_val != umax_val)) ||
15       smin_val > smax_val || umin_val > umax_val) {
16     /* Taint dst register if offset had invalid bounds derived
       from
17      * e.g. dead branches.
18      */
19     __mark_reg_unknown(env, dst_reg);
20     return 0;
21   }
22 ...
23 }
```

Listing 6.4: Excerpt from kernel source code - adjust_ptr_min_max_vals function

After leaking the address of `oob_map`, the exploit proceeds to leak the address of `map_ops` (Section 2.1.4) for the same map. First, the previous steps are repeated to obtain `EXPLOIT_REG` again, with the invalid bounds `{u,s}32_max_value = 0 < {u,s}32_min_value = 1` and a runtime value of `0`. These bounds are then used to trick the verifier into believing that the register contains the value `0` while in reality it contains the value `1`.

To achieve this, we first add `1` to `EXPLOIT_REG`. For immediate instructions, the immediate value is simply added to all bounds, provided the addition does not cause an overflow. Thus, after adding `1` to `EXPLOIT_REG`, we obtain `u32_max_value = 1` and `u32_min_value = 2`, with `var_off = {mask = 0x100000000; value = 0x1}`.

Next, we need a register with an initially unknown value. Let us call this register `UNKNOWN_VALUE_REG`. We create it as before by loading a value from an eBPF map; its real runtime value will be `0`. We then begin to manipulate the bounds of this register by adding a conditional `jmp` instruction:

```
1    BPF_JMP32_IMM(BPF_JLE, UNKOWN_VALUE_REG, 1, 1)
2    BPF_EXIT_INSN()
```

Listing 6.5: Conditional jump to manipulate bounds

The above conditional `jmp32` instruction will skip the exit instruction if the value of the lower 32 bits of the register is less than or equal to `1`. Since the actual value of the register is `0`, the exit instruction will be skipped. However, the verifier does not know the actual value of the register, so the unsigned bounds of the register will be updated to `u32_min_value = 0` and `u32_max_value = 1` for the true branch. Moreover, `jmp32` operates only on the lower 32 bits of the register, so the upper 32-bit bounds will remain unchanged and still unknown. Thus, after the `jmp32` instruction, `UNKNOWN_VALUE_REG` has `var_off = {mask = 0xFFFFFFFF00000001; value = 0x0}`, with the lower 32-bit mask set this way because the value can be either `1` or `0` from the verifier's perspective.

Next, `EXPLOIT_REG` is added to `UNKNOWN_VALUE_REG`. If the addition does not overflow either bound, the bounds of the destination and source registers are simply added together. This results in `EXPLOIT_REG` having new 32-bit bounds: `u32_min_value = 2` and `u32_max_value = 2`. Examining how `var_off` for `EXPLOIT_REG` is updated (in `adjust_scalar_min_max_vals`), we see that because the upper 32 bits of `UNKNOWN_REG` are unknown, the upper 32 bits of the result will also be unknown. The least significant bit of `UNKNOWN_REG` could be either `0` or `1`; since the least significant bit of `EXPLOIT_REG` is known to be `1`, the sum of the two least significant bits could be either `1` or `2` (at runtime this addition yields `1`). Therefore, the second least significant bit is also unknown. The remaining bits are known to be `0` in both operands. This leaves `EXPLOIT_REG` with `var_off = {mask`

`= 0xFFFFFFFF00000003; value = 0x0}`.

Finally, `__update_reg_bounds`, `__reg_deduce_bounds`, and `__reg_bound_offset` are called. The first two functions do not change the bounds, but the last one, since `u32_min_value = u32_max_value = 2`, computes that the range of possible values includes only the integer `2`. Given the current `var_off` lower 32-bit mask `0x3`, the 32-bit counterpart of `__reg_bound_offset` treats the lowest two bits as known and equal to `2`.

After these steps, `EXPLOIT_REG` has bounds `{u,s}32_min_value = {u,s}32_max_value = 2` and `var_off = {mask = 0xFFFFFFFF00000000; value = 0x2}`. In other words, the register's lower 32 bits are known and equal to `2`, while at runtime the register equals `1`. All that remains is to zero-extend the lower 32 bits of the register and perform a simple AND operation:

```
1    BPF_MOV32_REG(EXPLOIT_REG, EXPLOIT_REG)
2    BPF_ALU64_IMM(BPF_AND, EXPLOIT_REG, 1)
```

Listing 6.6: Zero extend and AND operation

The verifier will now believe that `EXPLOIT_REG = 0` because `2 & 1 = 0`, but at runtime it will actually be equal to `1` (`1 & 1 = 1`).

The final step is to bypass ALU sanitization. As mentioned earlier, ALU sanitization has undergone several changes over the years. This particular bypass works on kernel versions not patched for CVE-2020-27171 (which affects kernel versions up to `5.11.8`) and, therefore, it can also be applied to almost every kernel versio vulnerable to CVE-2021-3490) and is described in detail in Manfred Paul's blog post [37].

To understand how ALU sanitization works in this kernel version, consider the following example [37]:

- Register 1 contains a pointer `p` to the beginning of 4000 bytes of safe memory. Assume that the type of `*p` is one byte in size. This means that adding 3999 to `p` is permissible, but adding 4000 would exceed the bounds.
- Register 2 contains a scalar `a`, which the static verifier believes to be `1000`.
- Now add register 2 to register 1. The static verifier has no problem with this, since `p+1000` is still within the memory region. Because this is arithmetic involving a pointer, the ALU sanitizer will set an `alu_limit`. However, even though the verifier believes that `a` is at most `1000`, the limit is not set to `1000` but rather to `3999`, because this is the largest value still considered safe. This implies that, even if at runtime `a` is larger than `1000`, as long as it is less than `3999` the arithmetic operation will succeed.

- Add register 2 to register 1 again. This time, the static part of the verifier believes that `p` points to position `1000` in the 4000-byte memory block. The `alu_limit` will now be set to `2999`, because this is the largest legal value that can still be added. The addition will be successful as long as `a < 2999`.

Since the `alu_limit` is set based on the maximum possible value that can be added to or subtracted from `p`, independently of the verifier's believed value of `a`, it is possible to perform multiple additions or subtractions until the `alu_limit` is reached even if the runtime value of `a` differs from the value believed by the verifier. For example, if we used a bug in the static analysis to make the verifier believe `a = 1000` while in reality we set `a = 2500`, we can add `a` twice and still pass both runtime checks: `2500 < 3999` (first `alu_limit`) and `2500 < 2999` (second `alu_limit` after the first addition). However, in total we have added `5000` to the pointer `p`, resulting in an address well outside the safe memory range.

Returning to our exploit, if we want to move backwards from a pointer to the first element of a map and access metadata such as `map_ops`, we cannot do so directly because the initial `alu_limit` for subtraction is `0` (we are already at the beginning of the map). Instead, we add a dummy value to `OOB_MAP_REG` to alter the `alu_limit`:`0x1000-1` in our case. We can now subtract up to `0x1000-1` from our pointer. Then we multiply `EXPLOIT_REG` by `0x1000-1`. Since the verifier believes that `EXPLOIT_REG = 0`, it will believe that the result of the multiplication is also `0`. However, at runtime `EXPLOIT_REG = 1`, so the result of the multiplication will be `0x1000-1`. The `alu_limit` for subtraction becomes `0x1000`, so we can subtract `EXPLOIT_REG` from `OOB_MAP_REG`, moving back toward the beginning of the map. At runtime the subtraction respects the `alu_limit` and thus passes the sanitization; the verifier believes we subtracted `0`, so the `alu_limit` for subtraction remains `0x1000`. We can then repeat: multiply `EXPLOIT_REG` by any value up to `0x1000-1` and subtract it from `OOB_MAP_REG`, bypassing the runtime checks repeatedly. This allows us, in particular, to subtract `BPF_MAP_OPS_OFFSET` from the `oob_map` value pointer so that it points to `bpf_map->ops`. The value of `map_ops` is then read and stored for later use.

The final step is obtaining arbitrary read and write capabilities. For reading, we use the field `struct btf *btf` present in `oob_map`. It turns out [37] that the `btf` pointer can be conveniently accessed through the `bpf_map_get_info_by_fd` function, which is called by the `BPF_OBJ_GET_INFO_BY_FD` command of the `bpf` syscall when the file descriptor of the map is provided. This function effectively performs the following (the full function is located in `kernel/bpf/syscall.c`):

```
1 ...
2 if(map->btf){
3         info->btf_id = map->btf->id;
4         info->btf_key_type_id = map->btf_key_type_id;
5         info->btf_value_type_id = map->btf_value_type_id;
```

```
6 }
7 ...
```

Listing 6.7: Excerpt from kernel source code - bpf_map_get_info_by_fd function

This means that if we use our bug to set `map->btf` to **someaddr - offsetof(struct btf, id)**, then `BPF_OBJ_GET_INFO_BY_FD` returns `*someaddr` in `info.btf_id`. Since the `id` field of `struct btf` is a `u32`, this primitive can be used to read four bytes at a time from an arbitrary address.

For write capabilities, by overwriting `map_ops` using the verifier bug we can force the kernel to call any function we choose. However, the first argument will always be the `bpf_map` structure, which rules out many existing functions. It is therefore natural to overwrite selected entries of `array_map_ops` with different map operation pointers that accept the map pointer as their first argument. The exploit does exactly this, overwriting `map_push_elem` with `map_get_next_key`, whose implementation is shown in Listing 6.8.

```
1  /* Called from syscall */
2  static int array_map_get_next_key(struct bpf_map *map, void *key,
       void *next_key)
3  {
4    struct bpf_array *array = container_of(map, struct bpf_array,
       map);
5    u32 index = key ? *(u32 *)key : U32_MAX;
6    u32 *next = (u32 *)next_key;
7
8    if (index >= array->map.max_entries) {
9      *next = 0;
10     return 0;
11   }
12
13   if (index == array->map.max_entries - 1)
14     return -ENOENT;
15
16   *next = index + 1;
17   return 0;
18 }
```

Listing 6.8: Excerpt from kernel source code - array_map_get_next_key function

If we control `next_key` and `key`, then `*next = index + 1` can be used as an arbitrary write primitive under the condition that `index < array->map.max_entries`. If `map->max_entries` can be set to `0xffffffff` (which can be done using the same strategy to write out-of-bounds into a map), this check will always pass (except when `index = 0xffffffff`, which is actually the value we will use; this is acceptable because `*next` will still be set to `0 = index + 1` due to 32-bit wrap-around).

Note that the signature of `map_push_elem`, which we have overwritten with `map_get_next_key`, is:

```
1 int (*map_push_elem)(struct bpf_map *map, void *value, u64 flags);
```

Listing 6.9: map_push_elem signature

However, `map_push_elem` is invoked by the `BPF_MAP_UPDATE_ELEM` command only if the map has type `BPF_MAP_TYPE_STACK` or `BPF_MAP_TYPE_QUEUE`; thus the exploit again uses `EXPLOIT_REG` to modify the map type. We can then directly control the `flags` argument (which will be interpreted as `next_key`), as well as the `value` (which will be interpreted as `key`) argument. Other fields are also overwritten, such as the spinlock, to bypass additional checks. The arbitrary 32-bit write can now be triggered by passing appropriate arguments to `BPF_MAP_UPDATE_ELEM` (for example: `update_map_element(pCtx->oob_map_fd, 0, val-1, addr)`).

Once arbitrary read and write primitives for kernel addresses are obtained, the exploit locates the task structure and overwrites the credentials, then spawns a root shell to complete the privilege escalation.

To simulate more flexible behavior, the exploit was modified so it could perform attacks other than full privilege escalation (root shell spawning), such as privilege restoration, sending `SIGKILL` to processes, and reading (and potentially writing) arbitrary files. The attack type can be selected via command-line arguments.

6.2.3 Hardening Plan

This kind of exploit of the verifier is difficult to block preventively without patching the kernel itself, since it abuses a bug in the static analysis of the verifier and the exploit can have many variants. To find a solution as general as possible that can work for privilege escalation exploits even exploiting other bugs in the verifier detailed in other CVEs, a reactive approach was adopted, leveraging two different technologies: LSM and LKRG.

LSM Protection

The first solution was developing from scratch an LSM module using eBPF to monitor and block processes that performed an escalation of privileges. To do this, first of all the LSM program declares a map to keep track of processes that make use of eBPF programs. This is done by hooking the `bpf` syscall via lsm/bpf hook. Every time a process calls the `bpf` syscall to load an eBPF program, first it is checked that the process is not being already tracked, then its task_struct is retrieved and its credentials, in particular uid, gid, euid, fsuid, fsgid, suid and sgid, are stored in the map. There is a limit of processes that can be tracked at once that is determined by the max entries of the map. This limit is customizable and

it is necessary to find a good compromise between security and performance. If the map is full, no new processes that use eBPF will be loaded.

Once the process is being tracked and its credentials are stored, every time a sensitive operation is performed, such as invoking execve (so spawning a new process), trying to access a file, trying to send a signal to a process, trying to modify the credentials (e.g. with setuid), trying to use or create a socket, the process is checked against the map. If it is present in the map, its stored credentials are compared with the current ones. If they differ, it means that a privilege escalation has occurred and the process is immediately killed. If the process is not in the map, meaning that it does not use eBPF and so it cannot be an exploit of the verifier, it is allowed to proceed.

In our solution the hooks used to monitor sensitive operations are:

- **lsm/inode_permission** to monitor file access
- **lsm/cred_prepare** to monitor credential changes and calls to execve
- **lsm/task_kill** to monitor signals sent to processes
- **lsm/task_alloc** to monitor process creation (e.g. fork) and track the child process if the parent was tracked

Misssing hooks that could be useful are lsm/socket_create and lsm/socket_connect to monitor socket creation and usage.

Moreover other hooks used are:

- **lsm/bpf** to track processes that use eBPF programs as discussed before
- **lsm/task_free** to remove the process from the map when it terminates if it was being tracked

This solution has been tested against the exploit analyzed in Use Case 2 and it successfully blocked the privilege escalation and this very same solution was successfully tested also against an exploit of CVE-2022-23222 to prove that it could work against general privilege escalation exploits of the verifier.

There are pros to this solution but also limitations: the advantages reside in the fact that it is a general solution that can work against different exploits and that it is strongly customizable in terms of type of operations to monitor, number of processes to track and actions to take when a privilege escalation is detected. The limitations are that it is a reactive solution, so it can only block the privilege escalation but not the arbitrary read and write capabilities obtained before the escalation and also this solution has to include all the necessary hooks to monitor all the sensitive operations, otherwise it could be bypassed if a hook was missing. For example, in the current implementation there is no hook to monitor socket

creation and usage, so if an exploit used sockets after raising its privileges it would not be blocked. The implementation of the LSM protection code is presented in Appendix B.1.

LKRG

The second solution adopted was LKRG (Linux Kernel Runtime Guard). LKRG is a loadable, out-of-tree, kernel module that performs runtime integrity checking and exploit detection [38]. Unlike preventive security measures that try to stop attacks before they reach the kernel, LKRG acts as an observer: LKRG does not primarily aim to block attacks in advance, but instead detects and stops attacks occurring inside the kernel in real time.

Its operation relies on two main mechanisms. First, it performs runtime integrity checks, constantly verifying critical kernel data structures. If these structures are altered in unexpected ways (a common technique used to gain persistence, hide presence, or escalate privileges), LKRG can detect the change. Second, it provides exploit detection by identifying behavioral patterns typical of kernel exploits, such as memory manipulations or unusual control-flow redirections. For example, LKRG can detect an attempt to gain root privileges by exploiting a kernel flaw and block the action as it happens.

Because LKRG is a loadable kernel module, it can be added to a running Linux system without recompiling the kernel or rebooting, which simplifies deployment. Once loaded, it snapshots the current state of the CPUs, the kernel, and its most critical data, and then applies a series of integrity checks to detect possible compromise.

For kernel integrity validation, LKRG monitors:

- critical CPU metadata, such as which cores are online and whether expected security features (e.g., SMEP and SMAP) remain enabled,
- critical global kernel variables, such as whether SELinux is enabled and enforcing,
- keyed cryptographic hashes of the kernel image and of modules as they appear in memory.

When legitimate changes occur through documented means, for instance taking a CPU core offline, changing SELinux state, or loading/unloading a kernel module by the root user, LKRG transparently updates its snapshot. When unexpected changes are detected, its default response is to log the event (optionally to a remote server) and trigger a kernel panic, since milder reactions are generally ineffective against such attacks.

To validate the kernel's view of running tasks and detect exploits, LKRG keeps track of:

- process credentials, such as user and group IDs and container namespaces,
- control flow, i.e., kernel function call chains observed in stack frames at critical locations,
- use of the `usermodehelper` mechanism, restricting it to only a few approved programs (for example, `modprobe`).

When changes occur through legitimate operations, for example a process changing its user ID via the `setuid(2)` system call, LKRG transparently updates its copy of the data. When unexpected modifications to task credentials, anomalies in control flow, or misuse of `usermodehelper` are detected, the default response is to log the event (also to a remote server if configured) and kill the offending task.

LKRG's behavior, what it validates and how it responds to detected violations, can be configured through the `sysctl` interface and module parameters. Optional remote logging is performed directly from the kernel, with traffic encrypted using the log server's public key. The log server daemon and related tools are provided with LKRG.

LKRG successfully blocks many existing Linux kernel exploits and is likely to prevent many future ones, including some zero-day attacks, unless an exploit is specifically designed to bypass it [39]. While bypasses are possible, they typically require more complex and less reliable techniques. At present, LKRG is experimental, but it is a useful tool for strengthening kernel resilience against sophisticated attacks.

LKRG was tested against the exploit analyzed in the previous section and successfully blocked the resulting privilege escalation. The advantage of this solution is that it is ready to use, works across many kernel versions (including older ones that BPF LSM does not support), and is configurable in terms of detection sensitivity and response actions. By contrast, a solution based on eBPF offers greater flexibility in internal logic and avoids some shortcomings of kernel modules discussed in Section 4.4, such as the need to update the module for each kernel version to maintain compatibility and stability.

eBPF programs rely on two key mechanisms that enable portability and stability across architectures and kernel versions:

- **eBPF core instructions**, i.e., a set of architecture-independent instructions. The same eBPF program can run on different kernel versions without modification because it does not use fixed offsets of kernel objects directly, but makes use of BTF information [**btf**],

- **Just-in-time (JIT) compilation**: when an eBPF program is loaded into the kernel, it is compiled just in time into native machine code for the host architecture, ensuring the program runs efficiently across different systems without requiring recompilation.

These mechanisms make eBPF programs highly portable and stable across kernel versions and architectures, reducing the maintenance burden compared to traditional kernel modules. Moreover, since LKRG is an out-of-tree kernel module, it may introduce reliability issues, both in terms of potential bugs, as it does not undergo the same thorough review process as in-tree code, and in maintaining compatibility with future kernel versions.

Chapter 7

Conclusions and Future Work

This thesis has systematically addressed the issue of security in the adoption of eBPF technology. After analyzing the architectural mechanisms and the main risks introduced by its use, the work focused on studying documented vulnerabilities and reproducing real exploits, with the goal of identifying common patterns and proposing mitigation strategies.

Two case studies were examined in particular: the protection of eBPF maps and the exploitation of CVE-2021-3490, which demonstrates how verifier errors can be leveraged to achieve privilege escalation. For each case, hardening approaches were designed and tested, aiming to provide generalizable solutions resilient even to variants of known attacks.

The work highlighted both the potential and the limitations of these strategies. On one hand, the developed approaches show that it is possible to significantly strengthen the security of an eBPF-based system even without definitive solutions such as kernel patches or updates, which, although they provide complete protection and fix vulnerabilities, are often not applicable as they would require downtime or may introduce compatibility issues due to kernel version changes. Each proposed solution has its strengths and weaknesses: LSM BPF offers great flexibility and portability but may have issues with older kernel versions; LKRG provides strong out-of-the-box protection against attacks compromising kernel integrity and is effective against various exploits, but being an out-of-tree and still experimental kernel module, it may not be completely reliable or portable. Tetragon is easy to configure using YAML-based policies and excellent for observability, but it faces challenges when implementing more granular controls or analyzing complex data structures. On the other hand, significant challenges remain open, such as balancing performance with detection capabilities and further experimental validation in

real-world scenarios.

Since this research was carried out in collaboration with a colleague, the present thesis discusses only two of the four analyzed use cases. Nevertheless, several key principles have emerged from the overall work that can guide secure eBPF adoption:

- Verify every loaded eBPF program, ensuring it is trusted and does not access unauthorized maps.
- Monitor eBPF map creation and usage for anomalous behavior, such as unusually large map sizes or oversized keys.
- Deploy solutions that track credential IDs of processes using eBPF to detect and prevent privilege escalation attacks, thereby helping to mitigate LPE attempts across multiple CVEs.
- Stay updated on the latest critical CVEs and attack techniques, considering not only known vulnerabilities but also potential risks introduced by new Linux features.

Building on these findings, several directions for future work can be identified. One promising idea is the development of a verifier extension that leverages eBPF to implement custom detection logic, proactively identifying exploits targeting particularly severe known verifier bugs. During this thesis, a preliminary version of such a solution was developed. The idea is to create an eBPF program that monitors the loading of other eBPF programs via the `bpf` syscall. When a program is being loaded, the eBPF program retrieves its bytecode and analyzes the instructions to detect patterns that could indicate an attack. For example, in the case of CVE-2021-3490, the exploit begins by crafting a register that is half known and half unknown at compile time. This could be detected by tracking the register content through this verifier extension.

The eBPF program that retrieves the instructions of other programs must be complemented by a user-space component capable of more complex analysis, which would be difficult to implement in eBPF due to program size limitations. The current prototype implements the eBPF part that retrieves and logs the instructions, as shown in Figure 7.1. This is achieved as follows: first, the program declares a map to store the instructions of another BPF program during its loading phase. Then, using the `lsm/bpf` hook, triggered each time the `bpf` syscall is invoked, the program checks that the BPF command is `BPF_LOAD`. If so, it proceeds to read the `bpf_attr` struct, which contains information about the program being loaded, such as the number of instructions and the pointer to the instruction buffer. To balance performance, memory usage, and the strictness of security measures, a maximum number of instructions can be defined, blocking the loading of excessively large

programs. For simplicity, in this prototype, only the first `MAX_INSNS` instructions were analyzed without blocking the eBPF program if it exceeded this limit. After this check, the program loads the instructions into the map, taking into account that, unlike the usual data eBPF programs operate on, these data reside in user space. Finally, the instructions are printed, as shown in Figure 7.1. The code of this prototype solution can be found in Appendix B.2.

The user-space component is yet to be developed, and the logic for detecting attack patterns remains to be implemented. This approach represents a proactive solution capable of blocking potentially dangerous eBPF programs before they are executed, potentially overcoming the downsides of reactive solutions such as LKRG or the LSM protections analyzed in Use Case 2. However, this idea is still in its early stages, and several challenges must be addressed, such as finding a good balance between security and performance, since analyzing every eBPF program being loaded could introduce significant overhead. Moreover, it will be necessary to balance false positives and false negatives: overly strict detection logic may block legitimate programs, while overly lenient logic may fail to catch real attacks.

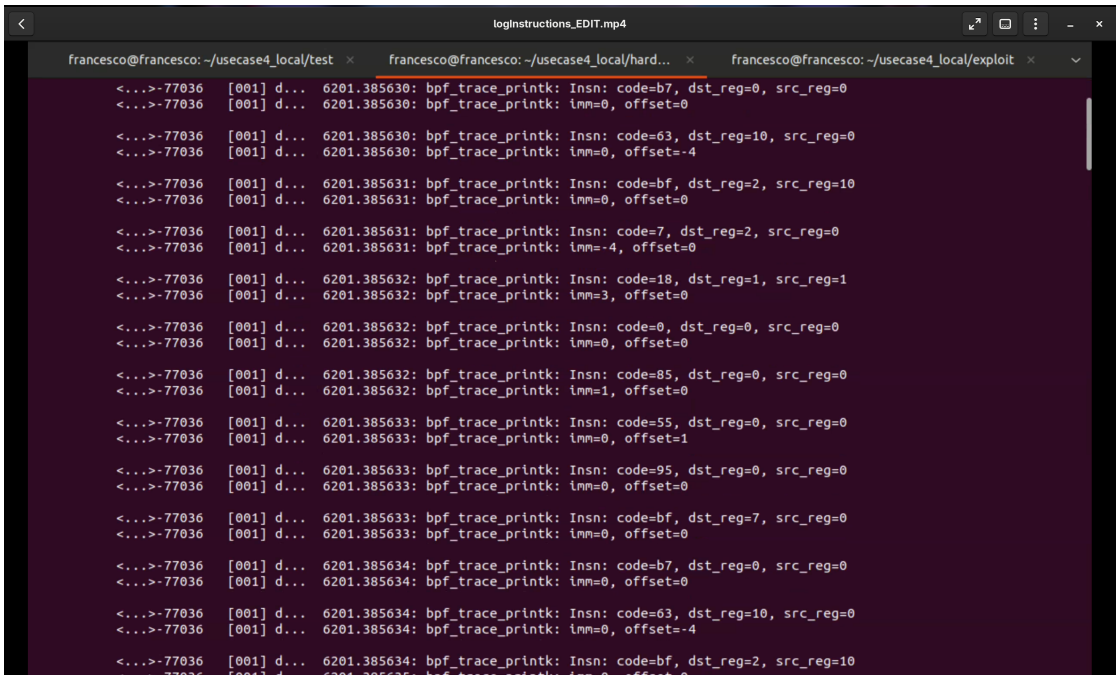

Figure 7.1: Log of the verifier extension showing the instructions of a loaded eBPF program. E.g. second line (code=63, dst_reg=10, src_reg=0, imm=0, off=-4) corresponds to storing the value of register 0 into the stack at offset -4 from the frame pointer (register 10).

Another interesting direction for future work would be to conduct a thorough

performance evaluation of the proposed mitigation strategies. While the focus of this thesis has been primarily on security effectiveness, it is equally important to assess the overhead introduced by each solution in realistic workloads. Measuring metrics such as latency, throughput, and resource consumption would help quantify the trade-offs between enhanced security and system efficiency.

In conclusion, this thesis has shown that strengthening the security of eBPF environments is possible through a combination of different strategies. Although the proposed solutions are still at a prototype stage, they represent a concrete step toward a more robust and secure adoption of eBPF technology in production systems.

Appendix A

Use Case 1

```
1  #include "vmlinux.h"
2  #include <bpf/bpf_helpers.h>
3  #include <bpf/bpf_tracing.h>
4  #include <bpf/bpf_core_read.h>
5
6  char LICENSE[] SEC("license") = "GPL";
7
8  #define EPERM 1
9
10 #define BPF_FS_MAGIC 0xCAFE4A11
11 #define ANON_INODE_FS_MAGIC 0x09041934
12
13 const volatile __u32 allowed_ino = 0;
14 const volatile __u32 allowed_sdev = 0;
15 const volatile __u32 expected_map = 0;
16
17 SEC("lsm/bpf_map")
18 int BPF_PROG(bpf_map, struct bpf_map *map, fmode_t fmode)
19 {
20     struct dentry *dentry_exe;
21     long unsigned int i_ino_exe;
22     dev_t s_dev_exe;
23
24     struct task_struct *task = (struct task_struct *)
    bpf_get_current_task();
25
26     __u32 map_id = BPF_CORE_READ(map, id);
27     char map_name[16];
28     __builtin_memcpy(map_name, BPF_CORE_READ(map, name), sizeof(
    map_name));
29
30     if (!task) {
31         bpf_printk("Error in reading task struct");
```

```
32          return -EPERM;
33      }
34
35      char task_name[16];
36      __builtin_memcpy(task_name, BPF_CORE_READ(task, comm), sizeof(
        task_name));
37
38
39      dentry_exe = BPF_CORE_READ(task, mm, exe_file, f_path.dentry);
40      if (!dentry_exe){
41          bpf_printk("Error in reading dentry_exe");
42          return -EPERM;
43      }
44      i_ino_exe = BPF_CORE_READ(dentry_exe, d_inode, i_ino);
45      s_dev_exe = BPF_CORE_READ(dentry_exe, d_inode, i_sb, s_dev);
46
47      if ( map_id == expected_map && (i_ino_exe != allowed_ino ||
        s_dev_exe != allowed_sdev)) {
48          bpf_printk("Denied access to BPF map %s from %s with mode
        %d", map_name, task_name, fmode);
49          bpf_printk("Exe: i_ino: %lu, s_dev: %u\n", i_ino_exe,
        s_dev_exe);
50          return -EPERM;
51      }
52
53      bpf_printk("Allowed access to BPF map %s from %s with mode %d
        ", map_name, task_name, fmode);
54      return 0;
55  }
56
57  SEC("lsm/mmap_file")
58  int BPF_PROG(block_mmap_bpf_map, struct file *file, unsigned long
        prot, unsigned long flags) {
59
60      struct dentry *dentry, *dentry_exe;
61      long unsigned int i_ino_exe;
62      dev_t s_dev_exe;
63      long unsigned int s_magic;
64
65
66      dentry = BPF_CORE_READ(file, f_path.dentry);
67      if(!dentry)
68          return 0;
69
70      s_magic = BPF_CORE_READ(dentry, d_inode, i_sb, s_magic);
71
72      pid_t pid = bpf_get_current_pid_tgid() >> 32;
73      struct task_struct *task = (struct task_struct *)
        bpf_get_current_task();
```

```
74      dentry_exe = BPF_CORE_READ(task, mm, exe_file, f_path.dentry);
75      if (!dentry_exe){
76          bpf_printk("Error in reading dentry_exe");
77          return -EPERM;
78      }
79      i_ino_exe = BPF_CORE_READ(dentry_exe, d_inode, i_ino);
80      s_dev_exe = BPF_CORE_READ(dentry_exe, d_inode, i_sb, s_dev);
81
82      if((s_magic == ANON_INODE_FS_MAGIC || s_magic == BPF_FS_MAGIC)
        && (i_ino_exe != allowed_ino || s_dev_exe != allowed_sdev)){
83          bpf_printk("File with magic: 0x%08x\n", s_magic);
84          bpf_printk("Exe: i_ino: %lu, s_dev: %u\n", i_ino_exe,
      s_dev_exe);
85          bpf_printk("Denied access to process with PID %d trying to
       mmap file with magic 0x%08x", pid, s_magic);
86          return -EPERM;
87      }
88
89      return 0;
90  }
```

Listing A.1: LSM BPF protection

```
1
2   apiVersion: cilium.io/v2alpha1
3   kind: TracingPolicy
4   metadata:
5     name: "bpf"
6   spec:
7     kprobes:
8     - call: "sys_bpf"
9       syscall: true
10      return: false
11      args:
12        - index: 0
13          type: "int"
14          label: "bpf command"
15      selectors:
16        - matchArgs:
17            - index: 0
18              operator: "Equal"
19              values:
20              - 7
21              - 1
22              - 2
23              - 3
24          matchBinaries:
25            - operator: "NotIn"
26              values:
```

```
27              - "/mnt/shared/usecase1/subcase1/simple/simple_user"
28              - "/mnt/shared/usecase1/subcase1/userProgram/
    user_program"
29          matchActions:
30            - action: Sigkill
31
```

Listing A.2: Tetragon Policy

Appendix B

Use Case 2

```
1  #include "vmlinux.h"
2  #include <bpf/bpf_helpers.h>
3  #include <bpf/bpf_tracing.h>
4  #include <bpf/bpf_core_read.h>
5
6  char LICENSE[] SEC("license") = "GPL";
7
8  #define EPERM 1
9  #define NULL 0
10 #define MAX_TRACKED_PROCESSES 1024
11
12 struct {
13     __uint(type, BPF_MAP_TYPE_HASH);
14     __uint(max_entries, MAX_TRACKED_PROCESSES);
15     __type(key, __u32);
16     __type(value, struct tracked_entry);
17 } tracked_processes SEC(".maps");
18
19 struct tracked_entry {
20     __u32 uid;
21     __u32 gid;
22     __u32 euid;
23     __u32 fsuid;
24     __u32 fsgid;
25     __u32 suid;
26     __u32 sgid;
27 };
28
29 /* Hook for BPF syscalls to track processes using BPF */
30 SEC("lsm/bpf")
31 int BPF_PROG(bpf_syscall, int cmd, union bpf_attr *attr, unsigned
       int size)
32 {
```

```
33      pid_t pid = bpf_get_current_pid_tgid() >> 32;
34
35      // Track any process that uses BPF syscalls
36      struct tracked_entry *entry = bpf_map_lookup_elem(&
        tracked_processes, &pid);
37      if (entry) {
38          // Already tracking this process
39          return 0;
40      }
41
42      // Get current task credentials
43      struct task_struct *task = (struct task_struct *)
        bpf_get_current_task();
44      if (!task)
45          return 0; // Don't block, just don't track
46
47      const struct cred *cred = NULL;
48      bpf_core_read(&cred, sizeof(cred), &task->cred);
49      if (!cred)
50          return 0; // Don't block, just don't track
51
52      __u32 uid_val = 0, euid_val = 0, gid_val = 0, fsuid_val = 0,
        fsgid_val = 0, suid_val =0, sgid_val =0;
53      bpf_core_read(&uid_val, sizeof(uid_val), &cred->uid.val);
54      bpf_core_read(&euid_val, sizeof(euid_val), &cred->euid.val);
55      bpf_core_read(&gid_val, sizeof(gid_val), &cred->gid.val);
56      bpf_core_read(&fsuid_val, sizeof(fsuid_val), &cred->fsuid.val)
        ;
57      bpf_core_read(&fsgid_val, sizeof(fsgid_val), &cred->fsgid.val)
        ;
58      bpf_core_read(&suid_val, sizeof(suid_val), &cred->suid.val);
59      bpf_core_read(&sgid_val, sizeof(sgid_val), &cred->sgid.val);
60
61      struct tracked_entry new_entry = {
62          .uid = uid_val,
63          .euid = euid_val,
64          .gid = gid_val,
65          .fsuid = fsuid_val,
66          .fsgid = fsgid_val,
67          .suid = suid_val,
68          .sgid = sgid_val
69      };
70
71      bpf_printk("BPF syscall tracking process %d (cmd=%d)\n", pid,
        cmd);
72      bpf_printk("Initial credentials uid=%d, euid=%d, gid=%d\n",
        uid_val, euid_val, gid_val);
73      bpf_printk("Initial credentials fsuid=%d, fsgid=%d\n",
        fsuid_val, fsgid_val);
```

```
74
75      int ret = bpf_map_update_elem(&tracked_processes, &pid, &
    new_entry, BPF_ANY);
76      if (ret < 0) {
77          bpf_printk("Failed to track BPF process %d\n", pid);
78      } else {
79          bpf_printk("Successfully tracking BPF process %d\n", pid);
80      }
81
82      return 0;
83  }
84
85  SEC("lsm/task_alloc")
86  int BPF_PROG(task_alloc, struct task_struct *task, unsigned long
    clone_flags)
87  {
88      pid_t parent_pid = bpf_get_current_pid_tgid() >> 32;
89      pid_t child_pid;
90
91      // Read the child process PID
92      int ret = bpf_probe_read_kernel(&child_pid, sizeof(child_pid),
     &task->pid);
93      if (ret < 0) {
94          return 0; // Don't block on read failure
95      }
96
97      // Check if parent is tracked
98      struct tracked_entry *parent_entry = bpf_map_lookup_elem(&
    tracked_processes, &parent_pid);
99      if (parent_entry) {
100         // Inherit tracking from parent
101         struct tracked_entry child_entry = {
102             .uid = parent_entry->uid,
103             .euid = parent_entry->euid,
104             .gid = parent_entry->gid,
105             .fsuid = parent_entry->fsuid,
106             .fsgid = parent_entry->fsgid,
107             .suid = parent_entry->suid,
108             .sgid = parent_entry->sgid
109         };
110
111         ret = bpf_map_update_elem(&tracked_processes, &child_pid,
    &child_entry, BPF_ANY);
112         if (ret == 0) {
113             bpf_printk("Inherited tracking: child %d from parent %
    d\n", child_pid, parent_pid);
114         }
115     }
116
```

```
117     return 0;
118 }
119
120
121 SEC("lsm/cred_prepare")
122 int BPF_PROG(cred_prepare, struct cred *new, const struct cred *
    old, gfp_t gfp)
123 {
124     pid_t pid = bpf_get_current_pid_tgid() >> 32;
125     // Check if this process is tracked (BPF-related)
126     struct tracked_entry *entry = bpf_map_lookup_elem(&
    tracked_processes, &pid);
127     if (!entry) {
128         // Not tracked - allow privilege escalation (sudo, etc.)
129         return 0;
130     }
131
132     // Read new credentials
133     __u32 new_uid = 0, new_euid = 0, new_gid = 0, new_fsuid = 0,
    new_fsgid = 0, new_sgid=0, new_suid = 0;
134     bpf_core_read(&new_uid, sizeof(new_uid), &new->uid.val);
135     bpf_core_read(&new_euid, sizeof(new_euid), &new->euid.val);
136     bpf_core_read(&new_gid, sizeof(new_gid), &new->gid.val);
137     bpf_core_read(&new_fsuid, sizeof(new_fsuid), &new->fsuid.val);
138     bpf_core_read(&new_fsgid, sizeof(new_fsgid), &new->fsgid.val);
139     bpf_core_read(&new_suid, sizeof(new_suid), &new->suid.val);
140     bpf_core_read(&new_sgid, sizeof(new_sgid), &new->sgid.val);
141
142     bpf_printk("LSM cred_prepare: tracked BPF pid=%d\n", pid);
143     bpf_printk("Expected uid=%d, euid=%d, gid=%d\n", entry->uid,
    entry->euid, entry->gid);
144     bpf_printk("Expected fsuid=%d, fsgid=%d\n", entry->fsuid,
    entry->fsgid);
145     bpf_printk("New cred uid=%d, euid=%d, gid=%d\n", new_uid,
    new_euid, new_gid);
146     bpf_printk("New cred fsuid=%d, fsgid=%d\n", new_fsuid,
    new_fsgid);
147
148
149     // Only block privilege escalation to root for BPF-related
    processes
150     if ((new_uid != entry->uid ) ||
151         (new_euid != entry->euid) ||
152         (new_gid != entry->gid) ||
153         (new_fsuid != entry->fsuid ) ||
154         (new_fsgid != entry->fsgid ) ||
155         (new_sgid != entry->sgid ) ||
156         (new_suid != entry->suid )
157     ){
```

```
158         bpf_printk("[!LSM ALERT] BPF process privilege escalation
    in cred_prepare!\n");
159         bpf_printk("[!LSM ALERT] UID: %d->%d\n", entry->uid,
    new_uid);
160         bpf_printk("[!LSM ALERT] EUID: %d->%d\n", entry->euid,
    new_euid);
161         bpf_printk("[!LSM ALERT] GID: %d->%d\n", entry->gid,
    new_gid);
162         bpf_printk("[!LSM ALERT] FSUID: %d->%d\n", entry->fsuid,
    new_fsuid);
163         bpf_printk("[!LSM ALERT] FSGID: %d->%d\n", entry->fsgid,
    new_fsgid);
164         return -EPERM;
165     }
166
167     return 0;
168 }
169
170 SEC("lsm/inode_permission")
171 int BPF_PROG(inode_permission, struct inode *inode, int mask){
172
173     pid_t pid = bpf_get_current_pid_tgid() >> 32;
174     // Check if this process is tracked (BPF-related)
175     struct tracked_entry *entry = bpf_map_lookup_elem(&
    tracked_processes, &pid);
176     if (!entry) {
177         // Not tracked - allow legitimate privilege escalation (
    sudo, etc.)
178         return 0;
179     }
180
181
182     struct task_struct *task = (struct task_struct *)
    bpf_get_current_task();
183
184     const struct cred *cred = NULL;
185     bpf_core_read(&cred, sizeof(cred), &task->cred);
186     if (!cred)
187         return -EPERM;
188
189     __u32 new_uid = 0, new_euid = 0, new_gid = 0, new_fsuid = 0,
    new_fsgid = 0, new_suid=0, new_sgid=0;
190     bpf_core_read(&new_uid, sizeof(new_uid), &cred->uid.val);
191     bpf_core_read(&new_euid, sizeof(new_euid), &cred->euid.val);
192     bpf_core_read(&new_gid, sizeof(new_gid), &cred->gid.val);
193     bpf_core_read(&new_fsuid, sizeof(new_fsuid), &cred->fsuid.val)
    ;
194     bpf_core_read(&new_fsgid, sizeof(new_fsgid), &cred->fsgid.val)
    ;
```

```
195     bpf_core_read(&new_suid, sizeof(new_suid), &cred->suid.val);
196     bpf_core_read(&new_sgid, sizeof(new_sgid), &cred->sgid.val);
197
198     bpf_printk("LSM inode_permission: tracked BPF pid=%d\n", pid);
199     bpf_printk("Expected uid=%d, euid=%d, gid=%d\n", entry->uid,
    entry->euid, entry->gid);
200     bpf_printk("Expected fsuid=%d, fsgid=%d\n", entry->fsuid,
    entry->fsgid);
201     bpf_printk("New cred uid=%d, euid=%d, gid=%d\n", new_uid,
    new_euid, new_gid);
202     bpf_printk("New cred fsuid=%d, fsgid=%d\n", new_fsuid,
    new_fsgid);
203     // Only block privilege escalation to root for BPF-related
    processes
204
205     if ((new_uid != entry->uid ) ||
206         (new_euid != entry->euid) ||
207         (new_gid != entry->gid) ||
208         (new_fsuid != entry->fsuid ) ||
209         (new_fsgid != entry->fsgid ) ||
210         (new_sgid != entry->sgid ) ||
211         (new_suid != entry->suid )
212         ){
213         bpf_printk("[!LSM ALERT] BPF process privilege escalation
    in inode_permission!\n");
214         bpf_printk("[!LSM ALERT] UID: %d->%d\n", entry->uid,
    new_uid);
215         bpf_printk("[!LSM ALERT] EUID: %d->%d\n", entry->euid,
    new_euid);
216         bpf_printk("[!LSM ALERT] GID: %d->%d\n", entry->gid,
    new_gid);
217         bpf_printk("[!LSM ALERT] FSUID: %d->%d\n", entry->fsuid,
    new_fsuid);
218         bpf_printk("[!LSM ALERT] FSGID: %d->%d\n", entry->fsgid,
    new_fsgid);
219
220         return -EPERM;
221     }
222
223     return 0;
224 }
225
226 SEC("lsm/task_kill")
227 int BPF_PROG(task_kill, struct task_struct *p, struct
    kernel_siginfo *info,
228     int sig, const struct cred *cred)
229 {
230     pid_t pid = bpf_get_current_pid_tgid() >> 32;
231     // Check if this process is tracked (BPF-related)
```

```
232     struct tracked_entry *entry = bpf_map_lookup_elem(&
        tracked_processes, &pid);
233     if (!entry) {
234         // Not tracked - allow legitimate privilege escalation (
        sudo, etc.)
235         return 0;
236     }
237
238     __u32 new_uid = 0, new_euid = 0, new_gid = 0, new_fsuid = 0,
        new_fsgid = 0, new_suid=0, new_sgid=0;
239     bpf_core_read(&new_uid, sizeof(new_uid), &cred->uid.val);
240     bpf_core_read(&new_euid, sizeof(new_euid), &cred->euid.val);
241     bpf_core_read(&new_gid, sizeof(new_gid), &cred->gid.val);
242     bpf_core_read(&new_fsuid, sizeof(new_fsuid), &cred->fsuid.val)
        ;
243     bpf_core_read(&new_fsgid, sizeof(new_fsgid), &cred->fsgid.val)
        ;
244     bpf_core_read(&new_suid, sizeof(new_suid), &cred->suid.val);
245     bpf_core_read(&new_sgid, sizeof(new_sgid), &cred->sgid.val);
246
247     bpf_printk("LSM task_kill: tracked BPF pid=%d\n", pid);
248     bpf_printk("Expected uid=%d, euid=%d, gid=%d\n", entry->uid,
        entry->euid, entry->gid);
249     bpf_printk("New cred uid=%d, euid=%d, gid=%d\n", new_uid,
        new_euid, new_gid);
250
251     // Only block privilege escalation to root for BPF-related
        processes
252     if ((new_uid != entry->uid ) ||
253         (new_euid != entry->euid) ||
254         (new_gid != entry->gid) ||
255         (new_fsuid != entry->fsuid ) ||
256         (new_fsgid != entry->fsgid ) ||
257         (new_sgid != entry->sgid ) ||
258         (new_suid != entry->suid )
259         ){
260         bpf_printk("[!LSM ALERT] BPF process privilege escalation
        in task_kill!\n");
261         bpf_printk("[!LSM ALERT] UID: %d->%d\n", entry->uid,
        new_uid);
262         bpf_printk("[!LSM ALERT] EUID: %d->%d\n", entry->euid,
        new_euid);
263         bpf_printk("[!LSM ALERT] GID: %d->%d\n", entry->gid,
        new_gid);
264         bpf_printk("[!LSM ALERT] FSUID: %d->%d\n", entry->fsuid,
        new_fsuid);
265         bpf_printk("[!LSM ALERT] FSGID: %d->%d\n", entry->fsgid,
        new_fsgid);
266
```

```
267         return -EPERM;
268     }
269
270     return 0;
271 }
272
273 SEC("lsm/task_free")
274 int BPF_PROG(task_free, struct task_struct *task)
275 {
276     pid_t pid;
277     int ret = 0;
278     ret = bpf_probe_read_kernel(&pid, sizeof(pid), &task->pid);
279
280     if(ret<0){
281         bpf_printk("LSM failed to read pid of dying process\n");
282         return 0; // Don't block, just return
283     }
284
285     // Check if this process is actually tracked before trying to
    remove it
286     struct tracked_entry *entry = bpf_map_lookup_elem(&
    tracked_processes, &pid);
287     if (!entry) {
288         // Process not tracked, nothing to do
289         return 0;
290     }
291
292     // Process is tracked, remove it from the map
293     ret = bpf_map_delete_elem(&tracked_processes, &pid);
294     if (ret == 0) {
295         bpf_printk("Removed tracked entry for BPF process pid=%d\n
    ", pid);
296     } else {
297         bpf_printk("Failed to remove tracked entry for pid=%d, ret
    =%d\n", pid, ret);
298     }
299     return 0;
300 }
```

Listing B.1: LSM BPF protection

```
1 #include "vmlinux.h"
2 #include <bpf/bpf_helpers.h>
3 #include <bpf/bpf_tracing.h>
4 #include <bpf/bpf_core_read.h>
5
6 char LICENSE[] SEC("license") = "GPL";
7
8 #define NULL ((void *)0)
```

```
9  #define __aligned_u64 __u64 __attribute__((aligned(8)))
10 #define MAX_INSNS 68
11 #define CHUNK_SIZE 16
12 #define MAX_CHUNKS ((MAX_INSNS + CHUNK_SIZE - 1) / CHUNK_SIZE)  //
       5
13 #define EPERM 1
14
15 struct {
16     __uint(type, BPF_MAP_TYPE_ARRAY);
17     __uint(max_entries, MAX_CHUNKS);
18     __type(key, __u32);
19     __type(value, struct bpf_insn[CHUNK_SIZE]);
20 } insn_chunks SEC(".maps");
21
22
23 SEC("lsm/bpf")
24 int BPF_PROG(bpf_syscall, int cmd, union bpf_attr *attr, unsigned
       int size)
25 {
26     if(cmd != BPF_PROG_LOAD)
27         return 0;
28
29     __u32 insn_cnt = 0;
30     __aligned_u64 insn_ptr = 0;
31     __u32 total_read = 0;
32
33     bpf_core_read(&insn_cnt, sizeof(insn_cnt), &attr->insn_cnt);
34     if (insn_cnt > MAX_INSNS) //here we could return -EPERM
35   insn_cnt = MAX_INSNS;
36     bpf_core_read(&insn_ptr, sizeof(insn_ptr), &attr->insns);
37
38     bpf_printk("BPF syscall cmd=%d, size=%u", cmd, size);
39     bpf_printk(" insn_cnt=%u, insn_ptr=%p\n", insn_cnt, (void *)
       insn_ptr);
40
41
42     __u32 chunks = (insn_cnt + CHUNK_SIZE - 1) / CHUNK_SIZE;
43     if (chunks > MAX_CHUNKS)
44         chunks = MAX_CHUNKS;
45
46     //BEGIN MANUAL LOOP TO READ INSTRUCTIONS
47
48     __u32 chunk_idx = 0;
49     struct bpf_insn buf[CHUNK_SIZE] = {};
50     if (chunk_idx < chunks){
51   __u32 to_read = CHUNK_SIZE;
52   if ((chunk_idx + 1) * CHUNK_SIZE > insn_cnt){
53       total_read += (unsigned int)(insn_cnt - chunk_idx *
       CHUNK_SIZE);
```

```
54  }else{
55        total_read += CHUNK_SIZE;
56  }
57
58  if(to_read <= CHUNK_SIZE && to_read > 0){
59      void *user_ptr = (void *)insn_ptr + chunk_idx * CHUNK_SIZE *
      sizeof(struct bpf_insn);
60      int ret = bpf_probe_read_user(buf, to_read*sizeof(struct
     bpf_insn) , user_ptr);
61      if (ret!=0)
62     return ret;
63
64      bpf_map_update_elem(&insn_chunks, &chunk_idx, buf, BPF_ANY);
65  }
66    }
67    chunk_idx++;
68    __builtin_memset(buf, 0, sizeof(buf));
69    if (chunk_idx < chunks){
70  __u32 to_read = CHUNK_SIZE;
71  if ((chunk_idx + 1) * CHUNK_SIZE > insn_cnt){
72      total_read += (unsigned int)(insn_cnt - chunk_idx *
     CHUNK_SIZE);
73  }else{
74      total_read += CHUNK_SIZE;
75  }
76
77  if(to_read <= CHUNK_SIZE && to_read > 0){
78      void *user_ptr = (void *)insn_ptr + chunk_idx * CHUNK_SIZE *
      sizeof(struct bpf_insn);
79      int ret = bpf_probe_read_user(buf, to_read*sizeof(struct
     bpf_insn) , user_ptr);
80      if (ret!=0)
81     return ret;
82
83      bpf_map_update_elem(&insn_chunks, &chunk_idx, buf, BPF_ANY);
84  }
85    }
86    chunk_idx++;
87    __builtin_memset(buf, 0, sizeof(buf));
88    if (chunk_idx < chunks){
89  __u32 to_read = CHUNK_SIZE;
90  if ((chunk_idx + 1) * CHUNK_SIZE > insn_cnt){
91      total_read += (unsigned int)(insn_cnt - chunk_idx *
     CHUNK_SIZE);
92  }else{
93      total_read += CHUNK_SIZE;
94  }
95
96  if(to_read <= CHUNK_SIZE && to_read > 0){
```

```
97        void *user_ptr = (void *)insn_ptr + chunk_idx * CHUNK_SIZE *
      sizeof(struct bpf_insn);
98        int ret = bpf_probe_read_user(buf, to_read*sizeof(struct
     bpf_insn) , user_ptr);
99        if (ret!=0)
100      return ret;
101
102       bpf_map_update_elem(&insn_chunks, &chunk_idx, buf, BPF_ANY);
103   }
104     }
105     chunk_idx++;
106     __builtin_memset(buf, 0, sizeof(buf));
107     if (chunk_idx < chunks){
108   __u32 to_read = CHUNK_SIZE;
109   if ((chunk_idx + 1) * CHUNK_SIZE > insn_cnt){
110       total_read += (unsigned int)(insn_cnt - chunk_idx *
     CHUNK_SIZE);
111   }else{
112       total_read += CHUNK_SIZE;
113   }
114
115   if(to_read <= CHUNK_SIZE && to_read > 0){
116       void *user_ptr = (void *)insn_ptr + chunk_idx * CHUNK_SIZE *
      sizeof(struct bpf_insn);
117       int ret = bpf_probe_read_user(buf, to_read*sizeof(struct
     bpf_insn) , user_ptr);
118       if (ret!=0)
119     return ret;
120
121       bpf_map_update_elem(&insn_chunks, &chunk_idx, buf, BPF_ANY);
122   }
123     }
124     chunk_idx++;
125     __builtin_memset(buf, 0, sizeof(buf));
126     if (chunk_idx < chunks){
127   __u32 to_read = CHUNK_SIZE;
128   if ((chunk_idx + 1) * CHUNK_SIZE > insn_cnt){
129       total_read += (unsigned int)(insn_cnt - chunk_idx *
     CHUNK_SIZE);
130   }else{
131       total_read += CHUNK_SIZE;
132   }
133
134   if(to_read <= CHUNK_SIZE && to_read > 0){
135       void *user_ptr = (void *)insn_ptr + chunk_idx * CHUNK_SIZE *
      sizeof(struct bpf_insn);
136       int ret = bpf_probe_read_user(buf, to_read*sizeof(struct
     bpf_insn) , user_ptr);
137       if (ret!=0)
```

```
138     return ret;
139
140       bpf_map_update_elem(&insn_chunks, &chunk_idx, buf, BPF_ANY);
141   }
142     }
143
144     bpf_printk("Total read: %u insns\n", total_read);
145
146     //BEGIN MANUAL LOOP TO PRINT INSTRUCTIONS
147
148      __u32 key = 0;
149     struct bpf_insn *chunk;
150     __u32 chunks_read = total_read / CHUNK_SIZE;
151     __u32 read_leftover = total_read % CHUNK_SIZE;
152     __u32 leftover_key = chunks_read + 1;
153
154     bpf_printk("Chunks read: %u\n", chunks_read);
155
156     if(key < chunks_read){
157   chunk = bpf_map_lookup_elem(&insn_chunks, &key);
158
159   if (chunk != NULL){
160       for(__u32 i=0; i<CHUNK_SIZE; i++){
161     bpf_printk("Insn: code=%x, dst_reg=%d, src_reg=%d ", chunk[i].
       code, chunk[i].dst_reg, chunk[i].src_reg);
162     bpf_printk("imm=%d, offset=%d\n", chunk[i].imm, chunk[i].off);
163       }
164   }else{
165       bpf_printk("Chunk %u not found\n", key);
166   }
167     }
168     key++;
169     if(key < chunks_read){
170   chunk = bpf_map_lookup_elem(&insn_chunks, &key);
171
172   if (chunk != NULL){
173       for(__u32 i=0; i<CHUNK_SIZE; i++){
174     bpf_printk("Insn: code=%x, dst_reg=%d, src_reg=%d ", chunk[i].
       code, chunk[i].dst_reg, chunk[i].src_reg);
175     bpf_printk("imm=%d, offset=%d\n", chunk[i].imm, chunk[i].off);
176       }
177   }else{
178       bpf_printk("Chunk %u not found\n", key);
179   }
180     }
181     key++;
182     if(key < chunks_read){
183   chunk = bpf_map_lookup_elem(&insn_chunks, &key);
184
```

```
185   if (chunk != NULL){
186       for(__u32 i=0; i<CHUNK_SIZE; i++){
187     bpf_printk("Insn: code=%x, dst_reg=%d, src_reg=%d ", chunk[i].
      code, chunk[i].dst_reg, chunk[i].src_reg);
188     bpf_printk("imm=%d, offset=%d\n", chunk[i].imm, chunk[i].off);
189       }
190   }else{
191       bpf_printk("Chunk %u not found\n", key);
192   }
193     }
194     key++;
195     if(key < chunks_read){
196   chunk = bpf_map_lookup_elem(&insn_chunks, &key);
197
198   if (chunk != NULL){
199       for(__u32 i=0; i<CHUNK_SIZE; i++){
200     bpf_printk("Insn: code=%x, dst_reg=%d, src_reg=%d ", chunk[i].
      code, chunk[i].dst_reg, chunk[i].src_reg);
201     bpf_printk("imm=%d, offset=%d\n", chunk[i].imm, chunk[i].off);
202       }
203   }else{
204       bpf_printk("Chunk %u not found\n", key);
205   }
206     }
207     key++;
208     if(key < chunks_read){
209   chunk = bpf_map_lookup_elem(&insn_chunks, &key);
210
211   if (chunk != NULL){
212       for(__u32 i=0; i<CHUNK_SIZE; i++){
213     bpf_printk("Insn: code=%x, dst_reg=%d, src_reg=%d ", chunk[i].
      code, chunk[i].dst_reg, chunk[i].src_reg);
214     bpf_printk("imm=%d, offset=%d\n", chunk[i].imm, chunk[i].off);
215       }
216   }else{
217       bpf_printk("Chunk %u not found\n", key);
218   }
219     }
220
221     if(read_leftover > 0){
222   chunk = bpf_map_lookup_elem(&insn_chunks, &leftover_key);
223   if (chunk != NULL){
224       for(__u32 i=0; i<CHUNK_SIZE; i++){
225     if (i < read_leftover){
226         bpf_printk("Insn: code=%x, dst_reg=%d, src_reg=%d ", chunk
      [i].code, chunk[i].dst_reg, chunk[i].src_reg);
227         bpf_printk("imm=%d, offset=%d\n", chunk[i].imm, chunk[i].
      off);
228     }
```

```
229         }
230     }else{
231         bpf_printk("Chunk %u not found\n", leftover_key);
232     }
233       }
234
235       return 0;
236 }
```

Listing B.2: Verifier Extension prototype

Bibliography

[1] eBPF Documentation. *What is eBPF?* URL: https://ebpf.io/what-is-ebpf/ (visited on 09/10/2025) (cit. on pp. 1, 4).

[2] The Linux Kernel Documentation. *BPF Documentation*. URL: https://www.kernel.org/doc/html/v6.0/bpf/index.html (visited on 09/10/2025) (cit. on p. 4).

[3] *eBPF Verifier*. URL: https://docs.ebpf.io/linux/concepts/verifier/ (visited on 09/15/2025) (cit. on pp. 5–7).

[4] Chomp. *Kernel Pwning with eBPF - a Love Story*. 2021. URL: https://chomp.ie/Blog+Posts/Kernel+Pwning+with+eBPF+-+a+Love+Story (visited on 09/17/2025) (cit. on pp. 6, 8, 38, 42).

[5] *Linux Kernel Documentation: eBPF Verifier*. URL: https://www.kernel.org/doc/html/v6.0/bpf/verifier.html (visited on 09/15/2025) (cit. on p. 7).

[6] Cilium Authors. *BPF Architecture*. URL: https://docs.cilium.io/en/stable/reference-guides/bpf/architecture/ (visited on 09/04/2025) (cit. on p. 8).

[7] eBPF.io. *eBPF Maps*. URL: https://docs.ebpf.io/linux/concepts/maps/ (visited on 09/14/2025) (cit. on p. 8).

[8] Linux manual page. *Capabilities*. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html (visited on 09/12/2025) (cit. on pp. 9, 11).

[9] *bpf(2) — Linux manual page*. URL: https://man7.org/linux/man-pages/man2/bpf.2.html (cit. on p. 10).

[10] Linux Foundation. *Control Plane — eBPF Security Threat Model*. 2021. URL: https://www.linuxfoundation.org/hubfs/eBPF/ControlPlane%20%E2%80%94%20eBPF%20Security%20Threat%20Model.pdf (visited on 09/14/2025) (cit. on pp. 12, 14).

[11] *BPF helpers*. 2025. URL: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html (visited on 09/14/2025) (cit. on pp. 13, 14).

[12] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. «Cross Container Attacks: The Bewildered eBPF on Clouds». In: *Proceedings of the 32nd USENIX Security Symposium*. Anaheim, CA, USA, Aug. 2023. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/system/files/usenixsecurity23-he.pdf (visited on 09/14/2025) (cit. on p. 14).

[13] NVD - National Vulnerability Database. *CVE-2024-49861 Detail*. 2024. URL: https://nvd.nist.gov/vuln/detail/cve-2024-49861 (visited on 09/14/2025) (cit. on p. 15).

[14] Oracle. *Oracle Linux Kernel (UEK) 7.1 Release Notes — CVE Fixes*. URL: https://docs.oracle.com/en/operating-systems/uek/7/relnotes7.1/u1-cve-fixes.html?utm_source=chatgpt.com (visited on 09/17/2025) (cit. on p. 15).

[15] CrowdStrike. *Exploiting CVE-2021-3490 for Container Escapes*. URL: https://www.crowdstrike.com/en-us/blog/exploiting-cve-2021-3490-for-container-escapes/ (visited on 09/17/2025) (cit. on p. 15).

[16] Trail of Bits. *Pitfalls of Relying on eBPF for Security Monitoring (and Some Solutions)*. Sept. 2023. URL: https://blog.trailofbits.com/2023/09/25/pitfalls-of-relying-on-ebpf-for-security-monitoring-and-some-solutions/ (visited on 09/14/2025) (cit. on pp. 16, 19).

[17] oseiberts11. *BCC Issue #2825: kretprobes are mysteriously missed*. 2020. URL: https://github.com/iovisor/bcc/issues/2825 (visited on 09/14/2025) (cit. on p. 16).

[18] Rex Guo and Junyuan Zeng. «Phantom Attack: Evading System Call Monitoring». In: *Proceedings of DEF CON 29*. 2021. URL: https://media.defcon.org/DEF%20CON%2029/DEF%20CON%2029%20presentations/Rex%20Guo%20Junyuan%20Zeng%20-%20Phantom%20Attack%20-%20%20Evading%20System%20Call%20Monitoring.pdf (visited on 09/14/2025) (cit. on p. 17).

[19] Falco Security Advisory. *TOCTOU issue that could lead to rule bypass*. 2022. URL: https://github.com/falcosecurity/falco/security/advisories/GHSA-6v9j-2vm2-ghf7?utm_source=chatgpt.com (visited on 10/09/2025) (cit. on p. 18).

[20] Olivia A. Gallucci. *How to manipulate the execution flow of TOCTOU attacks*. 2024. URL: https://oliviagallucci.com/how-to-manipulate-the-execution-flow-of-toctou-attacks/?utm_source=chatgpt.com (visited on 10/09/2025) (cit. on p. 18).

[21] Elastic Security Labs. *A Peek Behind the BPFdoor*. 2022. URL: https://www.elastic.co/security-labs/a-peek-behind-the-bpfdoor (visited on 09/14/2025) (cit. on p. 20).

[22] Syteca. *MAC vs DAC: Understanding the Difference*. URL: https://www.syteca.com/en/blog/mac-vs-dac (visited on 09/19/2025) (cit. on p. 22).

[23] NordLayer. *Discretionary Access Control (DAC)*. URL: https://nordlayer.com/learn/access-control/discretionary-access-control/ (visited on 09/19/2025) (cit. on p. 22).

[24] TechTarget. *Mandatory Access Control (MAC)*. URL: https://www.techtarget.com/searchsecurity/definition/mandatory-access-control-MAC (visited on 09/19/2025) (cit. on p. 22).

[25] *Using SELinux — Red Hat Enterprise Linux 8 Documentation*. 2025. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/using_selinux/index (visited on 10/09/2025) (cit. on p. 23).

[26] Michael Kerrisk. *credentials(7) - Linux manual page*. 2025. URL: https://man7.org/linux/man-pages/man7/credentials.7.html (visited on 09/19/2025) (cit. on p. 24).

[27] Linux Kernel Documentation. *Linux Security Modules (LSM) — Kernel Documentation*. URL: https://docs.kernel.org/admin-guide/LSM/index.html (visited on 09/10/2025) (cit. on p. 25).

[28] Wikipedia contributors. *Linux Security Modules*. URL: https://en.wikipedia.org/wiki/Linux_Security_Modules (visited on 09/10/2025) (cit. on p. 25).

[29] The Linux Kernel documentation. *LSM BPF Programs*. URL: https://docs.kernel.org/bpf/prog_lsm.html (visited on 09/10/2025) (cit. on p. 25).

[30] Star Lab Software. *A Brief Tour of Linux Security Modules*. URL: https://www.starlab.io/blog/a-brief-tour-of-linux-security-modules (visited on 09/10/2025) (cit. on p. 26).

[31] Arash Jafari. *Understanding Kernel Modules: Enhancing Flexibility in Operating Systems*. 2023. URL: https://medium.com/@arashjafariwork/understanding-kernel-modules-enhancing-flexibility-in-operating-systems-ed348807ed7b#:%20~:text=Disadvantages:%20Stability%20Risks:%20A%20fault%20in%20any,%20debugging%20more%20complex.%20Examples:%20Linux%2C%20Unix%2C%20Solaris. (visited on 09/19/2025) (cit. on p. 27).

[32] OSDev Community. *Modular Kernel — Advantages and Disadvantages*. 2025. URL: https://wiki.osdev.org/Modular_Kernel#Disadvantages (visited on 09/19/2025) (cit. on p. 27).

[33] Linux Kernel Labs. *Kernel Modules — Linux Kernel Teaching*. 2025. URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html (visited on 09/19/2025) (cit. on p. 27).

[34] Isovalent. *Tetragon Docs*. URL: https://tetragon.io/docs/overview/ (visited on 09/09/2025) (cit. on p. 28).

[35] Pat Hogan. *Warping Reality: Creating and countering the next generation of Linux rootkits using eBPF*. URL: https://media.defcon.org/DEF%20CON%2029/DEF%20CON%2029%20presentations/PatH%20-%20Warping%20Reality%20-%20creating%20and%20countering%20the%20next%20generation%20of%20Linux%20rootkits%20using%20eBPF.pdf (visited on 09/15/2025) (cit. on p. 32).

[36] Mouad Kondah. *You See Me, Now You Don't: BPF Map Attacks via Privileged File Descriptor Hijacking*. URL: https://www.deep-kondah.com/you-see-me-now-you-dont-bpf-map-attacks-via-privileged-file-descriptor-hijacking/ (visited on 09/15/2025) (cit. on pp. 33, 34).

[37] Zero Day Initiative. *CVE-2020-8835: Linux Kernel Privilege Escalation via Improper eBPF Program Verification*. 2020. URL: https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification (visited on 09/17/2025) (cit. on pp. 38, 45, 46).

[38] CIQ. *A Deep Dive into Linux Kernel Runtime Guard (LKRG)*. 2023. URL: https://ciq.com/blog/a-deep-dive-into-linux-kernel-runtime-guard-lkrg/ (visited on 09/20/2025) (cit. on p. 50).

[39] LKRG Project. *LKRG: Linux Kernel Runtime Guard*. URL: https://lkrg.org/ (visited on 09/19/2025) (cit. on p. 51).