POLITECNICO DI TORINO

Master of Science in Cybersecurity

Master Degree Thesis

Detection and Mitigation of eBPF Security Risks in the Linux Kernel

Supervisor
prof. Riccardo Sisto
Company Tutors
dr. David Soldani
dr. Hami Bour
dr. Saber Jafarizadeh

Candidate
Vincenzo Costanzo
matricola: 329008

Academic Year 2024-2025

Abstract

The continuous adoption of cloud-native architectures and the widespread use of containerization have increased the demand for powerful, low-overhead observability and monitoring tools. eBPF (extended Berkeley Packet Filter) has emerged as a cornerstone technology in this domain, enabling the dynamic injection of user-defined programs into the Linux kernel to implement high-performance networking, tracing and security functionalities. However, executing code at kernel level inherently carries significant security risks and enlarges the system attack surface: verifier bugs, misused helper functions, map tampering, and interactions with pre-existing kernel vulnerabilities are among the threats that may lead to privilege escalation, denial-of-service, and container escapes.

This thesis investigates the security implications of eBPF with the goal of analyzing critical vulnerabilities and developing systematic hardening strategies. The first part provides a compact but comprehensive background, including eBPF technology and its primitives, the Linux security architecture (capabilities and the LSM framework), and an overview of monitoring and hardening tools such as Tetragon and LKRG (Linux Kernel Runtime Guard). Building on this foundation, the core of the work presents two in-depth case studies of high-impact vulnerabilities, examining their root causes, exploitation techniques, and reproducibility in a controlled environment, followed by the design of practical mitigation strategies. Proposed countermeasures include kernel and configuration recommendations, runtime detection policies for Tetragon, integration with LKRG, and the development of custom LSM BPF programs to enforce security policies directly within the kernel. Each mitigation approach is evaluated in terms of security effectiveness, operational practicality, and limitations.

The research concludes by outlining practical recommendations for reducing the attack surface of eBPF-enabled systems and proposes a general framework for strengthening defenses against future vulnerabilities, based on common exploitation patterns identified in the case studies.

Acknowledgements

Only in the printed version

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The extended Berkeley Packet Filter (eBPF) emerged as an innovative kernel mechanism that fundamentally transforms how programmability is achieved within Linux systems. Born from the need to extend operating system capabilities dynamically, eBPF addresses a perennial tension in kernel development: while the kernel's privileged position makes it optimal for implementing networking, security, and observability features, the traditional approach of modifying kernel code or deploying loadable modules presents significant challenges. Kernel evolution demands exceptional rigor due to stringent stability requirements and security constraints, often creating bottlenecks that slow innovation compared to user-space development.

eBPF resolves this dilemma through a sophisticated verification and execution framework. It permits applications running in user space to submit bytecode programs for kernel-side execution, subjecting these programs to rigorous static analysis that guarantees memory safety and termination. Once verified, programs execute in a protected runtime environment, achieving near-native performance through Just-In-Time compilation while maintaining kernel integrity through sandboxing. This architecture unlocks capabilities previously constrained by the traditional kernel development cycle, enabling rapid deployment of advanced monitoring, packet processing, and security enforcement logic without kernel recompilation or system restarts.

The impact has been transformative across cloud-native infrastructure. Organizations operating container orchestration platforms, public cloud services, and security monitoring solutions have embraced eBPF as essential infrastructure, leveraging its unique combination of safety, performance, and flexibility to build next-generation observability and security tools that were impractical with conventional approaches.

However, this increased functionality introduces significant security challenges. As eBPF programs execute with kernel privileges, vulnerabilities within the eBPF subsystem, particularly in the verifier component, responsible for ensuring program safety, can lead to severe security breaches including privilege escalation, kernel memory corruption, and complete system compromise. In fact, several recent Common Vulnerabilities and Exposures (CVEs) are discovered and published, demonstrating that even subtle flaws in the eBPF verifier or map handling can be exploited to bypass fundamental kernel security boundaries.

Modern eBPF implementations' complexity, combined with continuous subsystem evolution, creates ongoing challenges for maintaining security. Traditional kernel hardening approaches often prove inadequate for addressing eBPF-specific attack vectors, requiring specialized defensive techniques that understand unique characteristics of eBPF programs and their kernel interaction patterns.

This thesis presents an in-depth security analysis of the eBPF ecosystem, paired with practical kernel hardening strategies for the mitigation of emerging threats. The research begins with a detailed vulnerability taxonomy based on comprehensive examination of CVE records, leading security conference papers, and documented real-world attacks such as BPFDoor and the TripleCross rootkit. This investigation classifies flaws across all major eBPF components and analyzes operational constraints and container-specific risks, establishing the theoretical foundation for subsequent defensive implementations.

Building upon this systematic analysis, the thesis examines in detail two specific security flaws recognized for their elevated Common Vulnerability Scoring System (CVSS) scores and substantial potential consequences. The first case focuses on a privilege-escalation vulnerability enabling non-privileged users to obtain root permissions by leveraging defects in the eBPF verifier's pointer arithmetic handling. The second investigates vulnerabilities that produce kernel crashes and denial-of-service conditions through out-of-bounds memory operations in specific eBPF maps configurations.

Following characterization of these flaws and their context, open-source proof-of-concepts were studied and reproduced. Targeted modifications were introduced to improve observability, increase determinism, and provide clearer evidence of the exploitation process. Controlled and isolated virtual machines were established to evaluate and reproduce the attacks, using specific Linux distributions and kernel releases known to be vulnerable, alongside instrumented environments for syscall tracing, kernel source review, and runtime

debugging.

For each vulnerability scenario, comprehensive hardening plans were developed and evaluated. These plans begin with upstream patches that resolve root causes and extend to supplementary mitigations that strengthen system security posture. Existing security solutions were systematically assessed: Linux Kernel Runtime Guard (LKRG) as a broad runtime integrity guard particularly effective for credential anomalies, Tetragon as an eBPF-powered observability tool for behavioral monitoring, and crucially, custom BPF LSM programs as precise enforcement layers. These custom Linux Security Module (LSM) implementations represent the principal defensive contribution of this work: context-aware, in-kernel policies that align enforcement to actual exploitation mechanics while preserving legitimate workloads.

This thesis is the result of a collaborative research effort conducted in partnership with *Rakuten Mobile Inc.* and a fellow university colleague. The comprehensive study encompasses four distinct eBPF vulnerability use cases, with this thesis focusing on the scenarios described above, while the remaining cases are addressed in the companion thesis. This collaborative approach enables a broader and more systematic analysis of eBPF security vulnerabilities across different attack vectors and exploitation techniques.

The work is organized as follows:

Chapter 2 establishes technical foundations by examining eBPF's core components including verifier, maps, helper functions, and Compile Once, Run Everywhere (CO-RE) capabilities, with particular emphasis on security-relevant aspects and inherent attack surfaces.

Chapter 3 presents the security research conducted on the literature, articles and other sources regarding eBPF component-specific vulnerabilities, operational challenges in security monitoring deployments, container-related attack vectors, and malicious eBPF usage patterns.

Chapter 4 explores the broader security context within which eBPF operates, covering the Linux privilege model, capabilities system, and the LSM framework that provides the foundation for defensive implementations.

Chapter 5 surveys existing defensive technologies including LKRG (Linux Kernel Runtime Guard) and Tetragon, describing their architectures and operational principles.

Chapter 6 presents the core research contributions through the detailed

vulnerability case studies: CVE-2022-23222 demonstrating privilege escalation via eBPF verifier bypass, and CVE-2024-56614/56615 illustrating kernel panic and denial of service through integer confusion in eBPF map operations. Each case includes comprehensive exploitation analysis, custom hardening implementations using BPF LSM programs, Tetragon monitoring policies, LKRG deployment, and comparative evaluation of mitigation strategies.

Chapter 7 synthesizes the research findings and delivers operational directives for minimizing eBPF attack surfaces based on the studies performed during the thesis. It concludes by proposing future research directions, including systematic performance and overhead evaluation of the mitigation solutions developed in this work, and the formulation of a comprehensive defensive methodology framework that can be applied to emerging eBPF threats beyond the specific vulnerabilities examined.

Chapter 2

eBPF Technology

2.1 Definition and Purposes

The extended Berkeley Packet Filter (eBPF) [2] is a revolutionary technology in the Linux kernel that allows the execution of custom, user-defined programs within kernel space. Originally introduced as an enhancement of the classic Berkeley Packet Filter (cBPF), a mechanism designed in the early 1990s to efficiently filter network packets in user space, eBPF has evolved into a general-purpose execution engine closely integrated with the Linux kernel.

Unlike its predecessor, which was limited to packet filtering, eBPF provides an in-kernel virtual machine capable of executing programs in response to a wide range of kernel events, including system calls, network activity, tracepoints, and security hooks. These programs are verified for safety by the eBPF verifier before execution, ensuring that they cannot crash the kernel or perform unsafe operations.

Traditionally, the kernel has served as a strategic location for implementing features related to monitoring, security, and networking, thanks to its comprehensive oversight and control of system resources. However, modifying the kernel is challenging due to its critical functions and the necessity for reliability and security.

eBPF addresses this challenge by extending the Linux kernel dynamically without requiring modification of kernel source code or loading traditional kernel modules. This capability enables developers and system administrators to implement custom monitoring, tracing, networking, and security functionality at runtime, with minimal overhead and safety guarantees, by running small sandboxed programs, the so called eBPF programs.

When an eBPF program is introduced into the Linux kernel, it must

pass through a validation procedure to guarantee safety and security. The eBPF verifier examines the program for potential vulnerabilities including endless loops, unauthorized memory operations, and compliance with resource constraints. Additional information regarding the validation procedure is available in the corresponding subsection.

Upon successful validation, the program undergoes Just-in-Time (JIT) compilation, converting the platform-independent bytecode into architecture-specific machine instructions to enhance performance within the kernel environment. This process ensures the eBPF program operates with maximum efficiency and is optimized for the particular system where it executes, achieving performance comparable to natively compiled kernel code or kernel modules.

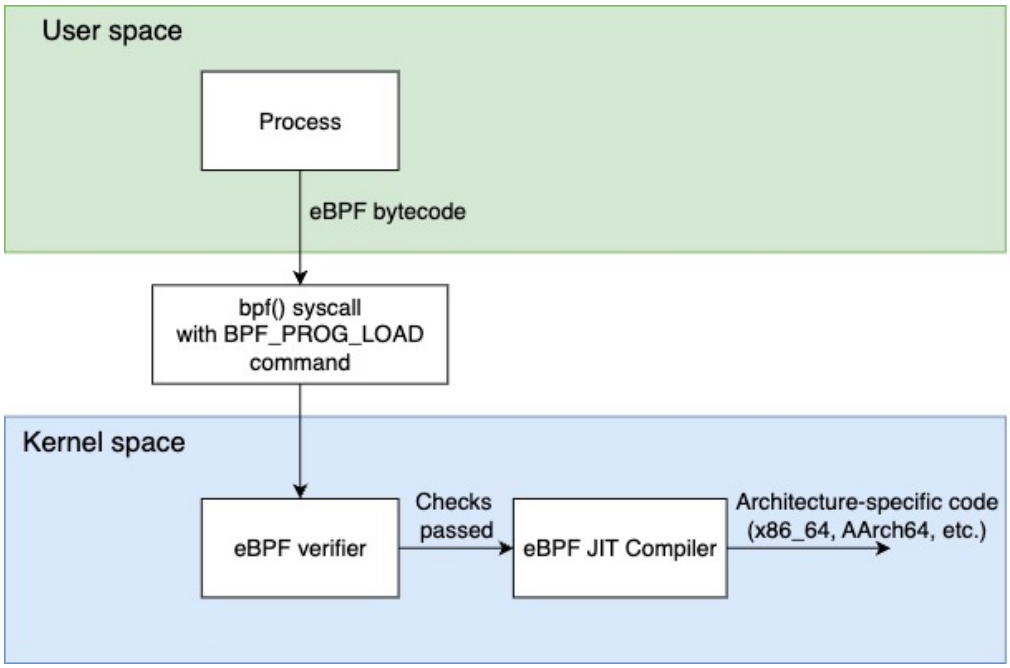

Figure 2.1. eBPF Program Lifecycle

By deploying eBPF programs across kernel subsystems, Linux exposes a programmable interface that supports advanced observability tools, fine-grained security policies, and high-performance networking. This versatility has made eBPF one of the most effective mechanisms for operating system extensibility in modern computing.

2.2 Key Components and Features

This chapter examines the fundamental architectural elements and operational mechanisms that enable eBPF programs to execute successfully. The analysis encompasses the eBPF virtual machine (VM) infrastructure, comprehensive security validation performed by the verifier, performance optimizations achieved through just-in-time compilation, and bidirectional communication between user space and kernel space through eBPF maps and helper functions.

2.2.1 eBPF Virtual Machine

The eBPF virtual machine [3] differs fundamentally from conventional virtualization technologies that emulate complete operating systems. Instead, it constitutes a secure, lightweight runtime environment embedded within the Linux kernel, specifically designed to execute eBPF programs safely.

Programs operating within this environment are dynamically loaded into kernel space and activated in response to various system events, including incoming network packets or system call invocations. The in-kernel execution model eliminates performance penalties associated with system call overhead and user-kernel space transitions, providing direct access to kernel resources.

The architecture implements a register-based design utilizing a specialized 64-bit Reduced Instruction Set Computer (RISC) instruction set. This configuration supports Just-in-Time compilation of eBPF programs into native machine code, enabling high-performance execution while maintaining controlled access to kernel functions and memory regions. Importantly, this implementation represents a complete virtual machine distinct from the Kernel-based Virtual Machine (KVM), which serves as a hypervisor enabling Linux to host other virtual machines.

Although kernel modules can technically accomplish the same operations as eBPF programs, direct kernel code execution presents significant risks. Unverified kernel code may cause system lockups, memory corruption, process crashes, security breaches, and other critical failures. The eBPF virtual machine addresses these concerns by providing a secure execution environment that combines the performance benefits of JIT-compiled kernel code with comprehensive safety guarantees.

The eBPF architecture deliberately incorporates limitations to ensure program safety and system stability. Initially, loops were prohibited entirely, though bounded loops were introduced in Linux 5.3 [4]. These constraints guarantee program termination and prevent infinite execution. Additional

safety measures include bounded memory access with comprehensive type checking, elimination of null pointer dereferences, an instruction count limits (typically 4096 instructions), and standardized function signatures requiring a single context argument. These restrictions enable efficient static analysis when programs are loaded, allowing the verifier to construct directed acyclic graphs for comprehensive validation.

2.2.2 eBPF Verifier

The eBPF verifier [5] represents an essential element within the eBPF framework, responsible for ensuring that eBPF programs can safely operate in the kernel environment. Its core responsibility involves examining eBPF bytecode prior to kernel loading, identifying potential vulnerabilities that could threaten system stability or security. The verifier conducts static analysis, inspecting the eBPF bytecode without executing it. The decision to examine bytecode instead of source code arises from the fact that source code can undergo different compilation processes, and bytecode represents the actual format that gets interpreted during execution.

Verification Process

The verifier utilizes a comprehensive multi-stage methodology encompassing parsing, control flow validation, type verification, and data flow examination to systematically assess all potential execution paths.

During the initial parsing stage, the verifier examines the eBPF bytecode structure, confirming adherence to expected formats and detecting syntactical anomalies. Subsequently, the control flow validation stage generates a control flow graph (CFG) that maps potential execution pathways within the program. This graph ensures well-defined program control flow, preventing infinite loops and unreachable code segments, thereby minimizing exploit chain risks. Due to the verifier's exhaustive path exploration, it may occasionally reject legitimate programs when encountering state explosion scenarios.

The type verification stage confirms that program operations maintain type safety, guaranteeing correct and consistent data type usage throughout the program. This approach prevents type conflicts, invalid memory access attempts, and buffer overflow conditions, ensuring proper operations within designated data structures.

At the core of this process lies the concept of "safety." The general principle is that eBPF programs must never compromise kernel stability or violate the

system's security model. This implies strict restrictions on what programs are allowed to do. They must always terminate within a reasonable amount of time, which excludes infinite loops and infinite recursion. They cannot arbitrarily read memory, since such access could lead to the disclosure of sensitive information; exceptions exist for tracing programs, which may rely on helpers to read memory in a controlled way, but these require root privileges and therefore do not represent a security risk. Similarly, network programs are restricted from accessing memory outside of packet bounds to avoid exposure of adjacent memory regions. Programs must also avoid deadlocks by ensuring that held spinlocks are released and that only one lock is held at any given time, and they are prohibited from reading uninitialized memory, which could otherwise leak confidential data.

The verifier also monitors register relationships, maintaining consistent constraints when one register receives another's value. Furthermore, the verifier distinguishes between various data types, including numeric values, pointers, and complex structures like map values, while validating memory structure offsets and ensuring proper null checking before pointer dereferencing. Importantly, pointer arithmetic operations should only be permitted when pointers are confirmed as valid and non-null.

These restrictions are not uniform across all program types. Different categories of eBPF programs are subject to distinct rules regarding accessible helper functions and permissible context fields. Before bounded loops were introduced, the verifier rejected any program containing loops, forcing compilers to unroll them. While this approach ensured termination, it increased program size and was not always feasible. Bounded loops now provide a more flexible solution, as the verifier is able to confirm that the loop will always terminate. However, this verification process is computationally expensive: a loop with one hundred iterations, a body of twenty instructions, and multiple branches can translate into several thousand instructions when the verifier accounts for all possible permutations, thereby contributing significantly to the program's overall complexity.

Capability-Based Access Control

Furthermore, the verifier validates that the attaching process possesses the necessary privileges for both the specific eBPF program type and the target hook. As an example, connecting an eBPF program to networking-related hooks typically requires the `CAP_NET_ADMIN` privilege, whereas alternative

hooks may demand distinct authorization levels. This access control framework guarantees that exclusively privileged processes can bind eBPF programs to critical kernel operations, consequently strengthening the overall system security posture. Additional details regarding Linux capabilities can be found in the corresponding section.

2.2.3 Just-in-Time Compiler

Following successful verification and safety validation by the verifier, programs proceed to the Just-in-Time (JIT) compilation phase. The JIT compiler performs bytecode transformation, converting platform-independent eBPF instructions into native machine code tailored to the host processor's instruction set architecture (ISA). This compilation process delivers substantial performance improvements by eliminating interpretation overhead, thereby reducing execution costs per instruction compared to runtime interpretation.

The translation mechanism frequently enables direct one-to-one mapping between eBPF instructions and corresponding native processor instructions, resulting in compact executable code with reduced memory footprint. Specifically, when targeting Complex Instruction Set Computing (CISC) architectures like x86, JIT compilers employ optimization strategies focused on generating minimal opcode sequences for each instruction, effectively minimizing the overall program size and enhancing cache efficiency during execution [6].

2.2.4 eBPF Maps

Maps represent data storage mechanisms that accommodate key-value pair collections within eBPF programs. They enable eBPF programs to preserve state information and facilitate data exchange among various program instances operating in the kernel environment or between userspace and kernel space [7, 8].

While maps are commonly defined within eBPF programs, they can alternatively be instantiated and controlled from userspace through the `bpf()` system call. The `BPF_MAP_CREATE` command exemplifies this capability, enabling userspace applications to generate new maps with designated characteristics. Among these characteristics, the map type stands as a critical attribute that governs storage behavior and access methodology.

Multiple map types exist, each optimized and suitable for distinct applications. Two map types hold particular significance for this thesis:

- `BPF_MAP_TYPE_DEVMAP`: device maps that facilitate network device management and support advanced packet routing and forwarding operations. These maps empower eBPF programs to communicate directly with network interfaces, accommodating scenarios like load balancing, traffic regulation, and specialized forwarding algorithms.
- `BPF_MAP_TYPE_XSKMAP`: eXpress Data Path (XDP) socket maps engineered specifically for XDP program integration. XDP [9] constitutes a high-efficiency packet processing architecture functioning at the network stack's foundational layer, delivering exceptionally rapid packet handling capabilities. XSK maps facilitate seamless interaction between XDP programs and userspace applications through shared data and state mechanisms.

Additional essential and required parameters for map creation include:

- `key_size`: the size (in bytes) of keys utilized within the map.
- `value_size`: the size (in bytes) of values contained in the map.
- `max_entries`: the maximum number of entries the map can accommodate.

Maps, functioning as data structures, support content modification and dynamic updates through various `bpf()` system call commands. The primary operations include:

- `BPF_MAP_UPDATE_ELEM`: modifies an existing map element or creates a new element when absent.
- `BPF_MAP_LOOKUP_ELEM`: extracts an element from the map using its corresponding key.
- `BPF_MAP_DELETE_ELEM`: locates and removes an element by key.

2.2.5 eBPF Helper Functions

eBPF programs can utilize numerous helper functions, which represent predefined routines provided by the kernel that simplify operations and interactions, functioning similarly to standard function calls.

These helper functions come with strong stability guarantees, as they are part of the Userspace API (UAPI), ensuring consistent availability across different kernel versions.

A significant consideration is that each program type supports only a specific collection of helper functions. For instance, XDP programs access a distinct helper function set compared to tracing programs.

These utilities cover an extensive array of functionalities [10], however, within the scope of this thesis, the subsequent helper functions are of particular importance:

- `bpf_get_current_pid_tgid`: provides a 64-bit return value where the current task's Process ID (PID) occupies the lower 32 bits while the TGID (thread group ID) fills the upper 32 bits. This utility enables eBPF programs to determine process and thread group identity, proving beneficial for monitoring individual threads or complete processes while implementing thread-specific regulations.

- `bpf_map_lookup_elem`: searches the map for an entry corresponding to the specified key, returning either a pointer to the associated value or NULL when the key remains absent. This function requires the map definition pointer and key pointer as input parameters.

- `bpf_map_delete_elem`: removes the map entry associated with the provided key, returning 0 upon successful completion or a negative error code when operations fail. Input parameters include the map definition pointer and key pointer.

- `bpf_map_update_elem`: stores a value in the map using the corresponding key, returning 0 for successful operations or a negative error code during failures. Required parameters encompass the map definition pointer, key pointer, value pointer, and update flags.

- `bpf_probe_read_user`: securely attempts to extract specified byte quantities from userspace addresses, storing retrieved data in the designated destination. Returns 0 upon success or negative error codes during failures. Parameters include the destination buffer pointer, userspace address, and data size specifications.

- `bpf_ringbuf_output`: transmits data to a ring buffer structure, returning 0 for successful operations or negative error codes when failures occur. Required parameters include the ring buffer pointer, data pointer, and data size specifications.

- `bpf_ringbuf_reserve`: allocates space within a ring buffer, returning either a pointer to the reserved area or NULL during allocation failures.

Input parameters comprise the ring buffer pointer and size requirements for space reservation.

- `bpf_skb_load_bytes`: loads a specified number of bytes from a packet associated with a socket buffer (skb) into a destination buffer. This helper copies len bytes from offset from the packet into the buffer pointed by to. Returns 0 on success, or a negative error in case of failure. Input parameters include the skb pointer, offset within the packet, destination buffer pointer, and the number of bytes to copy.

2.2.6 eBPF Programs

As previously mentioned, eBPF facilitates the runtime enhancement of kernel functionalities. This capability is realized through eBPF programs, which are compact code modules engineered to execute securely within the kernel space while delivering targeted features.

These programs operate on an event-triggered basis, activating when the kernel or applications encounter specific hooks. Pre-established hooks include system calls, function entry and exit points, kernel tracepoints, networking events, as well as numerous additional trigger mechanisms [11]. When pre-existing hooks do not meet particular requirements, developers can establish custom hooks through kprobes and uprobes. Kprobes facilitate the deployment of monitoring points within kernel functions, whereas uprobes provide equivalent capabilities for user-space applications. There exist also Kretprobes, a variant of kprobes, which are used to attach to the return point of a kernel function, enabling the capture of function return values.

After identifying the suitable attachment point, developers can bind eBPF programs to it, loading them into the kernel via the `bpf()` system call. When eBPF programs are connected to system call hooks, system administrators gain the ability to monitor, record, and modify the execution of these calls. This functionality proves especially valuable for security oversight, system auditing, and performance evaluation. For instance, an eBPF program can be connected to the `execve` system call to track all file access operations, recording information such as file names, user identifiers, and process identifiers.

Various eBPF program types exist, each optimized for particular applications and attachment mechanisms. The eBPF verifier may impose restrictions or grant permissions for specific features based on the program type, determined by its execution location within the kernel. Within this thesis context, understanding `BPF_PROG_TYPE_SOCKET_FILTER` is essential. This

program category focuses on filtering or altering packets passing through sockets, excluding egress/outbound packet processing. The complete list of eBPF program types is available in the eBPF documentation [12].

2.2.7 BPF CO-RE

BPF CO-RE represents an innovative framework implementing the "Compile Once, Run Everywhere" paradigm. This technology tackles a fundamental challenge within the eBPF ecosystem: enabling eBPF programs to execute consistently across varying kernel versions without necessitating version-specific recompilation.

eBPF programs frequently access kernel-originated data structures and memory regions. Across different kernel releases, these structures undergo modifications, including field offset changes, attribute renaming, or structural additions. Traditionally, such changes would require developers to maintain distinct program versions for each kernel release, creating substantial complexity and maintenance burdens. This challenge becomes particularly pronounced considering the Linux kernel's rapid development cycle and the heterogeneous deployment environments where eBPF programs operate.

CO-RE solves this by relying on the BPF Type Format (BTF) metadata [13], the compiler, and the libbpf library [14]. The kernel exposes authoritative BTF data, from which a header file, typically named `vmlinux.h`, can be generated, providing a complete description of kernel types.

At runtime, libbpf leverages BTF information embedded in both the compiled BPF program and the target kernel. It matches types and fields, applies relocations, and adjusts offsets as needed to ensure the program logic remains correct regardless of structural changes in the kernel.

In this way, BPF CO-RE eliminates much of the overhead traditionally associated with eBPF development and enables developers to ship portable BPF applications that run correctly across diverse kernels without code modifications or runtime recompilation.

2.2.8 In-depth Analysis

This section delves into more advanced aspects of eBPF, which, although not required for a high-level understanding, are particularly useful to understand the security implications and use cases that will be discussed in later chapters.

Verifier Internal: OR_NULL pointers

Within its operation, the verifier performs detailed monitoring of register contents, categorizing values into two primary classifications: pointers and scalars. The verification system bears responsibility for validating pointer boundaries, guaranteeing that stack memory reads occur only after corresponding writes (this requirement exists because the BPF stack implementation does not initialize memory to zero, creating potential information leakage vulnerabilities), and preventing pointer values from being stored within stack memory to avoid unauthorized pointer exposure.

The eBPF framework employs multiple pointer classifications, several of which include the `OR_NULL` identifier. This designation indicates operations with uncertain outcomes that might successfully return valid pointer references or encounter runtime failures that produce `NULL` results. Because the success or failure of these operations cannot be determined during static analysis, `OR_NULL` pointer classifications function as transitional states that facilitate runtime validation against zero values.

During null validation procedures, the verifier establishes two distinct execution paths: the first branch removes the `OR_NULL` designation from the pointer type following successful non-null confirmation, while the second branch treats the register as containing a scalar zero value. As an example, `PTR_TO_MEM_OR_NULL` becomes `PTR_TO_MEM` once null validation succeeds.

Within the alternative execution branch where the register holds NULL, arithmetic operations on pointers should be prevented. Nevertheless, as demonstrated in the first use case, for same old kernel version this restriction fails to apply consistently across various `OR_NULL` pointer classifications, creating opportunities to deceive the verifier into assuming no pointer manipulation occurred while the actual value underwent modification.

Verifier Internal: ALU Sanitation

This security mechanism emerged as a response to numerous vulnerabilities stemming from verifier implementation flaws. The fundamental concept involves augmenting the verifier's static boundary analysis with dynamic validation of actual runtime values processed by programs. Remember that pointer calculations are restricted to addition or subtraction operations involving scalar values. When arithmetic operations combine pointer and scalar registers (rather than immediate values), the system calculates an `alu_limit` representing the maximum absolute value that can be safely added to or subtracted from the pointer while maintaining acceptable bounds.

Given its protective nature, Arithmetic Logic Unit (ALU) sanitation serves as a critical safeguard against malicious pointer arithmetic in BPF programs. Consider this operational sequence:

1. establish a map with 4-byte values;
2. transfer a map value pointer into `REG_1`;
3. increment `REG_1` by 4;
4. perform a write operation to the memory location referenced by `REG_1`.

ALU sanitation intervenes during this process by maintaining pointer boundary awareness, recognizing the 4-byte allocation limit for the map, and subsequently blocking the operation due to memory boundary violations, as illustrated in the following listing.

```
invalid access to map value, value_size=4 off=4 size=1
R1 min value is outside of the allowed memory range
R1 pointer arithmetic of map value goes out of range, prohibited for
   !root
```

Listing 2.1. ALU sanitation in action

A crucial aspect of ALU sanitation involves its behavior when encountering arithmetic operations between pointer containing registers and scalar holding registers where the verifier possesses knowledge of the scalar's value. In such cases, the system modifies the operation to preserve its intended effect while eliminating dependency on the scalar register.

Internal Structure of eBPF Maps

From a kernel perspective, every eBPF map is represented by a `struct bpf_map`, defined in `include/linux/bpf.h`. This object encapsulates metadata about the map, including information for memory management, and pointers to auxiliary structures that govern map behavior. Importantly, the `bpf_map` object does not only hold map contents; instead, the contents are placed immediately after the main structure in memory, effectively making the map a flexible container that hosts both control data and storage.

```
1 struct bpf_map {
2     /* The first two cachelines with read-mostly members of
    which some
3       * are also accessed in fast-path (e.g. ops, max_entries)
    .
```

```
4       */
5      const struct bpf_map_ops *ops ____cacheline_aligned;
6      struct bpf_map *inner_map_meta;
7      void *security;
8      enum bpf_map_type map_type;
9      u32 key_size;
10     u32 value_size;
11     u32 max_entries;
12     u32 map_flags;
13     int spin_lock_off; /* >=0 valid offset, <0 error */
14     u32 id;
15     int numa_node;
16     u32 btf_key_type_id;
17     u32 btf_value_type_id;
18     struct btf *btf;
19     /* ... */
20 };
```

Listing 2.2. Definition of `bpf_map`

A central member of `struct bpf_map` is the `ops` field. This field is a pointer to a constant table of function pointers, formally described by `struct bpf_map_ops`. Each supported map type (e.g., hash maps, array maps, Least Recently Used maps, stacks, queues) provides its own implementation of this table, which defines the behavior of fundamental operations such as insertion, deletion, lookup, and key iteration. For instance, array maps use the statically defined `array_map_ops` structure, located in the kernel's read-only data segment (`.rodata`).

Another relevant member of `struct bpf_map` is the `btf` pointer, which can optionally reference a `struct btf` containing debug type information (used for BPF Type Format metadata). In practice, this field is typically unused and defaults to `NULL`. Its limited functional role and absence in normal operation make it an appealing target for attackers: overwriting this pointer is less likely to crash the system, while still providing a controlled kernel memory pointer that can be subsequently dereferenced.

The kernel exposes partial visibility into `struct bpf_map` via the syscall interface. In particular, the command `BPF_OBJ_GET_INFO_BY_FD`, invoked through `bpf_map_get_info_by_fd`, retrieves metadata from the map structure. This includes fields such as `map_type`, `max_entries`, and, crucially, information derived from the `btf` pointer.

Similarly, by redirecting `map->ops` to a crafted operations table placed in attacker-controlled memory (e.g., within another map's data region), it becomes possible to hijack the function dispatch mechanism. Since the kernel

assumes the `ops` table is immutable, it blindly dereferences these function pointers whenever user-space triggers a map operation through `bpf()` syscalls such as `BPF_MAP_UPDATE_ELEM` or `BPF_MAP_LOOKUP_ELEM`.

The practical exploitation of these techniques will be examined in detail in the first use case.

Chapter 3

eBPF Security Risks and Challenges

While eBPF provides powerful capabilities for in-kernel inspection, filtering, and extensibility, its deep integration into the kernel significantly broadens the attack surface. This chapter presents the results of a comprehensive security analysis of the eBPF ecosystem conducted for this thesis, examining the principal classes of vulnerabilities, illustrates them with representative examples, and discusses operational pitfalls and recurring attack patterns.

The research was conducted using a combination of systematic CVE database analysis, academic literature review, and industry security documentation. Specifically, the study examined CVEs from the National Vulnerability Database (NVD) to identify recurring vulnerability patterns and classify them by affected eBPF component (verifier, JIT compiler, helper functions, maps). Industry white papers and threat models, particularly the eBPF Security Threat Model published by the Linux Foundation [15], were analyzed to understand inherent security controls and design limitations. Academic publications from leading security conferences such as USENIX Security and DEF CON [16, 17] were examined to document practical attack scenarios and operational challenges. Technical reports and security advisories from cybersecurity research firms and practitioners [18, 19] were reviewed to identify real-world exploitation patterns and malicious usage. Documentation from eBPF-based security tool vendors such as Tetragon [20] and Datadog [21] was consulted to assess operational security challenges in production deployments.

This structured approach enabled systematic categorization of security risks across different eBPF components and usage contexts.

3.1 eBPF Component Vulnerabilities

The eBPF subsystem consists of several critical components, each responsible for specific functionalities that enable its operation within the Linux kernel. These components include eBPF maps for data sharing, helper functions for kernel interactions, the verifier for program safety validation, and the Just-In-Time compiler for performance optimization. More details are already presented in the previous chapter. While each component is designed with security considerations, the study identified that vulnerabilities can arise from implementation flaws, design trade-offs, or the inherent complexity of their operations. This section examines the security vulnerabilities discovered in each major eBPF component, analyzing their root causes, high-level exploitation mechanisms, and potential impacts on system security.

3.1.1 eBPF Maps

eBPF maps are shared kernel data structures that facilitate communication between eBPF programs and user-space controllers, primarily for state and configuration exchange. These maps are globally accessible through interfaces such as the `bpf_map_get_fd_by_id` helper function, but they inherently lack mechanisms for per-program isolation.

This design choice has significant security implications, particularly with respect to map tampering. Any actor with sufficient privileges can modify or delete map entries used by other programs, potentially disrupting control flows or corrupting program state. The issue becomes especially critical for eBPF-based security tools such as Tetragon [20] and Datadog [21], which rely on maps to coordinate execution across multiple small programs. In such cases, deletion or poisoning of map entries can paralyze the entire tool, making them highly susceptible to targeted map manipulation attacks.

3.1.2 Helper Functions

BPF helper functions expose a wide range of interactions with both kernel and user memory, including reading and writing user data, overriding return values, and sending signals. Vulnerabilities often arise from insufficient validation, ambiguous flag semantics, or misinterpretation of metadata passed as arguments. The study identified critical security flaws that demonstrate the risks associated with inadequate input validation and semantic ambiguities in helper function interfaces.

Out-of-bounds Writes via Metadata Parsing (CVE-2022-0500)

The vulnerability affects the `BPF_BTF_LOAD` operation, which handles the loading of BPF Type Format data structures into kernel space [22]. BTF metadata provides type information that enables advanced eBPF features such as CO-RE (Compile Once, Run Everywhere) by describing the structure layout and type definitions of kernel data structures.

The security flaw originates from the kernel's failure to enforce appropriate boundary validation when processing BTF data submitted by userspace applications. An attacker with eBPF loading privileges can exploit this weakness by submitting carefully crafted BTF metadata that triggers writes beyond allocated memory boundaries. The consequences of this vulnerability range from immediate system destabilization through kernel panics to more severe scenarios where attackers leverage the memory corruption to modify critical kernel data structures, potentially achieving arbitrary code execution with kernel privileges. The vulnerability received a CVSS score of 7.8 (HIGH), reflecting its serious impact on system security.

Memory-Buffer Semantic Ambiguities (CVE-2024-50164)

This vulnerability demonstrates how the evolution of kernel interfaces can introduce subtle security weaknesses [23]. The `MEM_UNINIT` flag was initially designed to signal that buffers passed to helper functions don't need to contain initialized data before use. As the eBPF subsystem evolved, developers extended this flag's semantics to additionally indicate write permission for the referenced buffer, creating a dual-purpose flag with ambiguous interpretation contexts.

The security issue manifests within `check_mem_size_reg()`, a verifier function responsible for validating memory access patterns. When analyzing memory operations with non-constant size offsets, the function deactivates certain validation modes by nullifying the metadata tracking structure. This implementation decision created an unintended consequence: essential write permission checks were inadvertently disabled, allowing eBPF programs to modify memory regions explicitly marked as immutable, including `.rodata` sections that should remain constant throughout execution. This vulnerability exemplifies how semantic drift in flag definitions can undermine the verifier's foundational security guarantees, enabling unauthorized modification of protected kernel memory regions.

3.1.3 eBPF Verifier

The verifier acts as the central safety mechanism for eBPF programs, performing static analysis to ensure that loaded code cannot compromise kernel integrity. However, the analysis identified that discrepancies between the verifier's computed value and the actual runtime ones, combined with inherent constraints in tracking complex execution paths, enable sophisticated bypass attacks. The following vulnerabilities illustrate how subtle implementation flaws in verification logic can undermine the entire security model.

Register State Desynchronization in 32-bit Operations (CVE-2020-8835)

This vulnerability affects the verifier's handling of mixed 32-bit and 64-bit arithmetic operations on 64-bit registers [24]. The core issue stems from incomplete state propagation when 32-bit operations modify registers: while the verifier correctly updates bounds information for the lower 32 bits, it fails to recalculate constraints for the upper 32 bits, creating a divergence between its internal representation and the actual register contents at runtime.

This desynchronization allows attackers to construct eBPF programs that manipulate the verifier's bounded range assumptions. By carefully orchestrating sequences of 32-bit and 64-bit operations, an attacker can force the verifier to conclude that a register holds a safe, bounded value when it actually contains an address pointing to arbitrary kernel memory regions. The vulnerability was demonstrated at Pwn2Own 2020 by Manfred Paul [25], who successfully leveraged this flaw to read and modify kernel memory structures, ultimately achieving privilege escalation by overwriting process credential structures to obtain root access.

Path Pruning Logic Errors in State Space Exploration (CVE-2023-2163)

To manage the computational complexity of verifying programs with numerous conditional branches, the verifier employs a path pruning optimization. When the verifier determines that a particular program state has already been proven safe through analysis of an equivalent state, it prunes the redundant exploration path. This optimization relies on "precise tracking" to identify which registers influence control-flow or memory safety decisions.

The vulnerability arises from incomplete propagation of precision markers across register assignments [26]. Specifically, when a register's precision

depends on values from another register through intermediate operations, the verifier may fail to mark all contributing registers as requiring precise analysis. This causes the verifier to incorrectly classify distinct program states as equivalent and prune execution paths that should have been analyzed. The result is that certain code paths execute at runtime without ever being subjected to verification checks. This vulnerability was discovered by Buzzer, Google's eBPF fuzzer, which generates large volumes of syntactically valid programs to stress-test verification logic, as described in the blog post [27]. The exploit allows execution of pointer arithmetic operations with register values that differ from those the verifier assumed during its analysis.

ALU32 Bitwise Operation Boundary Miscalculation (CVE-2021-3490)

The verifier maintains value ranges representing the possible bounds each register may contain. For bitwise operations on 32-bit ALU instructions (AND, OR, XOR), the verifier must compute how these operations affect the minimum and maximum bounds of the resulting value. The vulnerability stems from imprecise boundary calculations for these operations: the verifier's approximation of output ranges was insufficiently conservative, allowing the actual runtime values to exceed the assumed boundaries [28].

Exploiting this flaw enables attackers to establish unauthorized kernel memory read capabilities, which can then be leveraged to leak critical kernel addresses and locate essential kernel data structures. By carefully navigating through kernel memory using the obtained addresses, attackers can identify and modify process credential structures, ultimately setting their user and group identifiers to zero to achieve root privilege escalation.

Sign-Extension Inconsistencies in Immediate Value Interpretation (CVE-2017-16995)

This vulnerability exploits a semantic inconsistency in how the verifier and runtime interpreter handle immediate constant values [29]. During verification, 32-bit immediate values are interpreted as unsigned quantities, but at runtime, these same immediates undergo sign-extension when loaded into 64-bit registers, producing different values than the verifier analyzed.

The exploit leverages this discrepancy by loading the immediate value `0xFFFFFFFF`, which the verifier treats as the 64-bit value `0x00000000FFFFFFFF` but the runtime sign-extends to `0xFFFFFFFFFFFFFFFF`. A carefully constructed

conditional comparison between a register and this immediate appears to the verifier as always-true, causing it to skip analyzing the false branch. However, at runtime, the sign-extended values differ, causing the comparison to fail and the program to execute along the unverified path. This allows arbitrary operations in the unanalyzed code segment, including stack frame pointer manipulation to locate the `thread_info` structure, traverse to the `task_struct`, and ultimately modify the credentials structure to achieve privilege escalation.

3.1.4 Just-In-Time Compiler

The JIT compiler translates verified BPF bytecode into native machine instructions specific to the target architecture, enabling near-native execution performance. However, the complexity of this translation process, particularly in managing instruction size estimation and control flow displacement calculations, introduces potential security vulnerabilities. Analysis revealed that errors in these critical compilation phases can result in memory corruption or execution of unintended instruction sequences.

Architecture-Specific Instruction Size Misprediction (CVE-2024-50203)

This vulnerability manifests as a heap buffer overflow in the Linux kernel's BPF subsystem when executing on ARM64 architecture with tag-based Kernel Address Sanitizer (KASAN) enabled [30]. The flaw resides within the BPF trampoline mechanism, which generates optimized function call sequences at runtime.

The root cause stems from inconsistent address usage across the compiler's two-phase process: the size calculation phase uses stack addresses to estimate code length, while the code generation phase operates with heap addresses. When KASAN tags are applied to heap addresses on ARM64, the `emit_a64_mov_i64()` function may generate longer instruction sequences than initially estimated, as the number of required instructions depends on specific bit patterns in the immediate values. This size mismatch causes buffer overflow when the pre-allocated space proves insufficient for the actual generated code, potentially enabling privilege escalation attacks.

Control Flow Displacement Calculation Errors (CVE-2021-29154)

This vulnerability affects the JIT compiler's computation of branch displacement values when translating conditional and unconditional jump instructions from BPF bytecode to native machine code [31]. Branch instructions in compiled code require precise offset calculations to determine the target instruction address relative to the current program counter.

The flaw occurs when the compiler incorrectly calculates these displacement values, causing jump instructions to transfer control to unintended memory locations within the compiled code. An attacker can exploit this miscalculation by crafting BPF programs with specific branching patterns that trigger the erroneous offset computation. When executed, these programs cause control flow to deviate from the intended execution path, potentially landing in the middle of instructions or executing instruction sequences that were never validated by the verifier. This enables the execution of arbitrary instruction sequences within kernel context, bypassing all safety guarantees provided by the verification process and potentially leading to privilege escalation or arbitrary kernel memory access.

3.2 Operational Pitfalls for Security Monitoring

Although eBPF is extensively used for observability and security monitoring, the research identified intrinsic limitations that complicate its reliable deployment for security enforcement [18]. These operational constraints stem from fundamental design decisions in the eBPF subsystem and reflect trade-offs between performance, safety, and functionality. The following challenges illustrate how architectural limitations can undermine the reliability of eBPF-based security monitoring systems.

3.2.1 Probe Invocation Reliability Issues

While eBPF probes are theoretically designed to trigger consistently whenever their associated kernel events occur, practical deployments reveal occasional probe firing failures. This behavior, though infrequent and challenging to reproduce, has been documented in bug reports from eBPF tooling projects [32]. Analysis of these reports identifies two primary causes for missed events.

First, the kernel enforces a hard limit on the maximum number of concurrent kretprobes that can be active simultaneously. As of recent kernel versions, this

threshold is set to 4,096 active kretprobes. Any attempt to register additional kretprobes beyond this limit results in silent failures, causing events to go unmonitored without explicit notification to user-space applications. This limitation becomes particularly problematic in complex monitoring scenarios involving numerous instrumentation points across multiple subsystems.

Second, subtle implementation differences exist between the callback mechanisms for kprobes and kretprobes. These differences can, under specific circumstances, prevent a kprobe from correctly detecting or pairing with its corresponding kretprobe, leading to event loss. Since eBPF was not originally designed as a comprehensive security enforcement mechanism but rather as a performance monitoring tool, there exists no guarantee that all probe invocations will occur as expected, particularly under high system load or when approaching architectural limits.

3.2.2 Concurrency and Event Management Limitations

eBPF programs execute in a restricted kernel environment that prohibits access to traditional kernel synchronization primitives such as mutexes, semaphores, or spinlocks. Furthermore, eBPF probes cannot block event-producing kernel code paths, as doing so would introduce unacceptable latency and potential deadlock scenarios. This architectural constraint creates several operational challenges when event rates exceed processing capacity.

When attachment points experience high event volumes, multiple failure modes can manifest. The kernel may cease invoking probes entirely once internal buffers saturate, resulting in complete event loss for that monitoring point. Alternatively, when storage structures such as eBPF maps or ring buffers become full, new events overwrite older, unconsumed data before user-space applications retrieve them. In scenarios involving complex data structures spanning multiple map entries, partial overwrites can corrupt data integrity, disrupting monitoring program operation. These issues become particularly acute during system state transitions such as container initialization sequences or automated deployment scripts, which can generate unexpectedly large event bursts.

3.2.3 Memory Access Constraints

The Linux kernel's virtual memory subsystem can page out infrequently accessed memory regions to disk-backed storage (swap space or memory-mapped files) to optimize physical memory utilization. Under normal circumstances,

when paged-out memory is accessed, the kernel handles the resulting page fault transparently by loading the required content back into physical memory. However, eBPF programs execute with page fault handling explicitly disabled due to the atomic execution context in which they operate.

This restriction means that any attempt by an eBPF program to dereference a pointer to paged-out memory immediately fails without the possibility of fault recovery. For security monitoring tools attempting to inspect process memory, command-line arguments, or file contents, this limitation constrains the set of reliably accessible data. While instrumentation points can be strategically positioned immediately after the kernel copies data into memory (before it has opportunity to be paged out), this approach provides no strict guarantees given the absence of concurrency primitives to enforce memory residency.

3.2.4 Time-of-Check to Time-of-Use Race Conditions

eBPF programs routinely execute concurrently across multiple CPU cores, processing events in parallel. This concurrent execution, combined with the inability to acquire kernel locks or employ synchronization mechanisms, creates fundamental data race vulnerabilities. A particularly severe manifestation involves time-of-check to time-of-use (TOCTOU) attacks on system call parameters [17].

When an eBPF program inspects system call arguments at the syscall entry tracepoint, it examines pointers to user-space memory buffers. However, between the time of this inspection and the moment when the kernel copies the data into kernel space for actual processing, malicious user-space code can modify the buffer contents. This race window enables attackers to present benign data for security checks while substituting malicious payloads for actual kernel consumption, effectively bypassing eBPF-based security policies. This limitation fundamentally undermines the reliability of eBPF for security enforcement scenarios requiring tamper-proof parameter validation.

3.3 Container-Related Threats

Even in the absence of kernel-level vulnerabilities, the standard eBPF feature set introduces significant attack vectors in containerized environments. A fundamental design characteristic of eBPF tracing programs is their lack of namespace awareness: unlike network-oriented eBPF features such as XDP

and Traffic Control (which operate only within their container's network interfaces), tracing programs can monitor and manipulate processes system-wide, transcending container boundaries to observe activities on the host or within other containers.

Research presented at the 32nd USENIX Security Symposium demonstrates the severity of these cross-container attack capabilities [16]. The analysis reveals that while the `CAP_SYS_ADMIN` capability requirement theoretically restricts eBPF access, over 2.5% of container images in Docker Hub repositories possess the necessary permissions for eBPF-based attacks. Once compromised, these privileged containers become platforms for cross-boundary attacks that can compromise entire Kubernetes clusters. Linux's capability system proves insufficient for mitigating these threats, as it can only enable or disable eBPF functionality entirely, making selective restriction impractical for environments where legitimate services require eBPF access. The main attack vectors are the following:

- **Hijacking host processes:** Attackers can inject eBPF tracing programs at system call dispatch points using RAW Tracepoints, enabling comprehensive monitoring of all system calls across namespace boundaries. By identifying privileged processes on the host system (such as Bash shells running with elevated privileges), attackers can exploit helper functions like `bpf_probe_write_user` to inject malicious commands into shell script buffers during read operations. Combined with `bpf_override_return` to manipulate system call return values, this technique ensures execution of injected payloads, achieving container escape and arbitrary command execution with root privileges on the host system.

- **Cross-container information theft:** Helper function such as `bpf_probe_read_user` enables eBPF programs to extract system call arguments and read process memory across namespace boundaries. Attackers can leverage this capability to steal sensitive data from processes running in other containers or on the host system, including file contents, cryptographic keys, authentication tokens, and other confidential information. Additionally, this primitive facilitates kernel address leakage, providing attackers with information necessary for exploiting memory corruption vulnerabilities or bypassing kernel address space layout randomization (KASLR) protections.

- **Denial-of-service and manipulation:** eBPF tracing programs can intercept system calls across all containers and the host, enabling various

forms of denial-of-service attacks. Using `bpf_probe_write_user`, attackers can corrupt system call arguments to trigger application crashes. The `bpf_override_return` helper allows manipulation of system call return codes, causing programs to perceive failures where none occurred. Most critically, the `bpf_send_signal` helper enables arbitrary signal delivery to target processes, allowing attackers to systematically terminate critical system services such as `systemd` or `dockerd`, potentially rendering the entire host system inoperable. These capabilities extend to eBPF map tampering attacks, where globally accessible eBPF maps can be manipulated via `bpf_map_get_fd_by_id`, disrupting the operation of security monitoring tools like Tetragon that depend on map-based control flow and data exchange mechanisms.

3.4 Malicious eBPF Usage

While eBPF is widely promoted as a tool for monitoring and performance optimization, the research identified that its kernel-level power also makes it attractive to attackers seeking persistence or stealth. Two notable cases analyzed were BPFDoor and eBPF-based rootkits.

3.4.1 BPFDoor

BPFDoor [19] represents a sophisticated Linux backdoor that employs eBPF for stealthy network monitoring and persistent access. The malware uses raw sockets to monitor all network packets, which would normally cause high CPU usage. eBPF solves this by providing efficient kernel-level filtering, allowing the implant to ignore 99% of traffic and activate only on specific magic packets.

Rather than creating obvious reverse shells, BPFDoor employs connection hijacking. It performs an initial handshake with legitimate services, such as nginx and sshd, then sends a magic packet to identify the attacker's IP and port. The implant temporarily reconfigures iptables to redirect traffic from the attacker to a listening socket on an inconspicuous port (42391-43391). After establishing the command channel, firewall rules are removed, but Linux's stateful connection tracking ensures continued traffic forwarding, providing exceptional stealth.

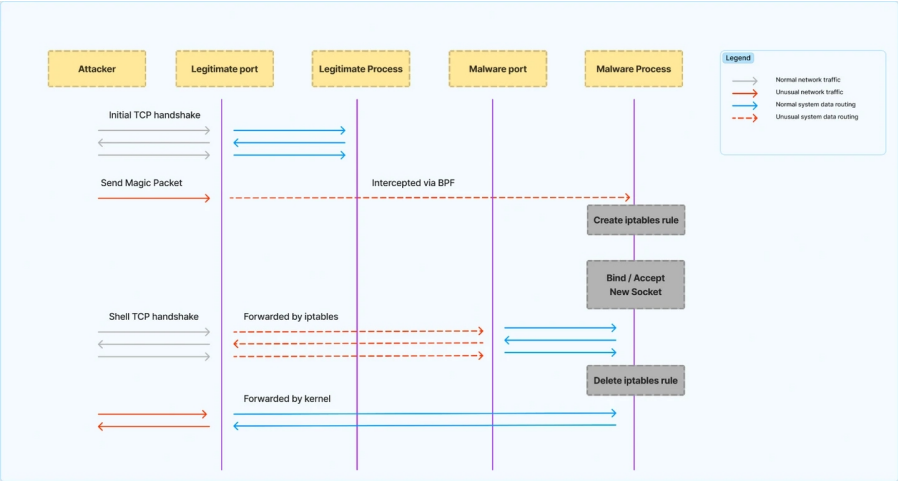

Figure 3.1. BPFDoor network flow.

3.4.2 eBPF-Based Rootkits

Rootkits represent stealthy tools that attackers employ to maintain persistent access to compromised systems despite credential changes or vulnerability patches. By integrating with the syscall table, these malicious programs acquire comprehensive visibility and control, enabling attackers to monitor network traffic, conceal files and processes, and spawn root-level processes while remaining undetected.

Traditional kernel rootkits provide formidable stealth capabilities but introduce significant risks. Minor bugs can cause catastrophic system failures, and kernel updates may destabilize rootkit functionality.

eBPF offers a powerful alternative for developing rootkits that circumvent traditional kernel-level risks. Malicious programs can leverage its hooks to intercept syscalls, manipulate network traffic, and tamper with user-space data while avoiding the risks of kernel modules. A notable example of this threat is TripleCross [33], an eBPF-based rootkit that demonstrates the practical exploitation of these capabilities in real-world attack scenarios.

Attackers employing eBPF-based rootkits can alter network packets before they reach system firewalls, disguising external traffic as internal and evading detection. They can attach to OpenSSL functions to intercept and modify encrypted Transport Layer Security (TLS) data in real time, compromising secure communications without leaving traces in traditional logs. By intercepting syscalls such as `open`, `read`, or `write`, attackers can present falsified data to applications and security tools, for instance bypassing Secure Shell

(SSH) multi-factor authentication by forging reads from `pam.d` configuration files. Additionally, they can supply fake `/etc/passwd` or `/etc/shadow` entries to escalate privileges while keeping on-disk files intact, ensuring that file integrity checks and security audits reveal no anomalies.

These manipulations are achieved using helpers like `bpf_probe_write_user` and `bpf_probe_read_user`, allowing attackers to tamper with syscall buffers under precise conditions such as specific process IDs or User IDs (UIDs). Importantly, underlying files remain untouched, so standard forensic tools display authentic data, leaving security auditors unaware of tampering.

Additional stealth techniques include using `bpf_send_signal` to terminate inspection processes that might detect the rootkit's presence and employing `fmod_ret` programs to override kernel functions by returning fabricated results, effectively creating a parallel reality where malicious activities remain invisible to both users and security monitoring systems.

Chapter 4

Linux Security Architecture

The Linux operating system employs a multilayered security framework that integrates conventional mechanisms with advanced capability-driven access control and kernel extensibility frameworks. This chapter presents an overview of the primary elements: user and process identity management, the capabilities framework, and the Linux Security Module (LSM) architecture.

4.1 Identity and Privilege Model

Within Linux systems, access control decisions originate from the notion of *credentials*, which establish the security context associated with users or processes. Credentials dictate whether operations on system *objects* receive authorization or face denial. *Objects* within Linux are system elements that userspace programs can directly manipulate, including *tasks, files/inodes, sockets, message queues, shared memory segments, semaphores, and keys*, all carrying credentials as integral metadata components.

Access control within Linux encompasses the regulation of actions (*operations*) that subjects (primarily processes) may execute on *objects*. This regulation occurs through access policy definitions that specify operation permissions and restrictions.

From a user control standpoint, two fundamental access control models demonstrate relevance:

- **Discretionary Access Control (DAC):** Within this framework, resource owners determine access privileges and permissible actions for

other entities. Traditional Unix file permissions (`rwx` bits) and ownership mechanisms exemplify DAC implementations. Although flexible, DAC relies on individual users to manage access appropriately.

- **Mandatory Access Control (MAC):** This restrictive framework enforces access policies established by system administrators or security specialists. Users cannot modify or override these regulations.

Within Linux environments, DAC and MAC operate concurrently: DAC establishes baseline, user-controlled permissions, while MAC frameworks, when activated, apply supplementary, system-enforced constraints to achieve detailed and mandatory control.

Types of Credentials

The Linux kernel supports multiple credential types associated with tasks and *objects*:

- **Real User ID (UID) and Group ID (GID):** These identifiers establish process ownership and represent the authentic identity of the initiating user. They frequently serve as the *objective context* for actions. Processes obtain their real user (group) ID through `getuid(2)` (`getgid(2)`). During process creation via `fork(2)`, children inherit copies of their parent's user and group identifiers.
- **Effective User ID (EUID) and Effective Group ID (EGID):** The kernel utilizes these identifiers to determine process permissions when accessing shared resources. They form the *subjective context* for tasks. For instance, when executing a set-UID binary, the EUID may diverge from the UID. Processes obtain their effective user (group) ID using `geteuid(2)` (`getegid(2)`).
- **Saved set-user-ID and saved set-group-ID:** These identifiers function within set-user-ID and set-group-ID programs to preserve copies of corresponding effective IDs established during program execution. Set-user-ID programs can acquire and drop privileges by alternating their effective user ID between real user ID and saved set-user-ID values. This alternation occurs through `seteuid(2)`, `setreuid(2)`, or `setresuid(2)` calls.
- **Filesystem UID (FSUID) and GID (FSGID):** These Linux-specific identifiers determine file access permissions. When a process's effective

user (group) ID changes, the kernel automatically updates the filesystem user (group) ID to match. Consequently, filesystem IDs typically mirror corresponding effective IDs, maintaining file permission check semantics consistent with other UNIX systems. Filesystem IDs can differ from effective IDs through `setfsuid(2)` and `setfsgid(2)` calls.

- **Supplementary groups:** This represents additional group ID collections used for permission verification when accessing files and shared resources.

Task Credentials and the `struct cred`

Each Linux task references its credentials through the `task_struct`. The `task_struct` instances reside in a linked list, with the global kernel variable `init_task` indicating the initial entry.

Beyond linking data, the `task_struct` maintains a reference-counted structure named `struct cred`, which consolidates all task identity information, encompassing UIDs, GIDs, group memberships, keyrings, and LSM-specific security data. Specifically, it includes the IDs previously discussed as follows:

```
struct cred {
  /* ... */

  kuid_t     uid;     /* real UID of the task */
  kgid_t     gid;     /* real GID of the task */
  kuid_t     suid;    /* saved UID of the task */
  kgid_t     sgid;    /* saved GID of the task */
  kuid_t     euid;    /* effective UID of the task */
  kgid_t     egid;    /* effective GID of the task */
  kuid_t     fsuid;     /* UID for VFS ops */
  kgid_t     fsgid;     /* GID for VFS ops */

  /* ... */
};
```

Listing 4.1. `struct cred` showing user and group IDs

To maintain consistency, once credential sets receive commitment to tasks through functions including `commit_creds()`, they become *immutable*. The system permits only reference count modifications or associated keyring changes. Credential modifications follow the *copy and replace principle*:

1. Current credential copies are generated using `prepare_creds()`.
2. Modifications apply to the copy.

3. New credentials receive atomic commitment through `commit_creds()`.

This design guarantees that no task can directly modify another task's credentials, thereby enhancing process isolation. For instance, the `capset()` system call restricts modifications to the current process's credentials exclusively.

Accessing Credentials

Tasks can freely access their own credentials using functions including:

- `current_uid()` and `current_gid()` for real IDs,
- `current_euid()` and `current_egid()` for effective IDs,
- `current_fsuid()` and `current_fsgid()` for filesystem checks.

Accessing other task credentials requires special handling due to concurrency concerns. The kernel enforces *Read-Copy Update (RCU)* mechanisms to ensure safe reads without race conditions. For persistent access, the `get_task_cred()` function safely obtains references.

These identifiers establish the foundation for traditional discretionary access control (DAC) models. Processes executing as root (`EUID 0`) possess unrestricted system resource access, motivating the need for more detailed privilege separation.

4.2 Linux Capabilities

To address the constraints of the omnipotent root account, Linux implemented the *capabilities framework* [34], which segments superuser privileges into discrete units that can be independently assigned to processes, threads, or executables. This approach enables more precise control over the operations that privileged tasks may execute.

Capability Sets

For capability implementation, the kernel structures them into distinct collections that govern capability acquisition, utilization, and inheritance:

```
typedef struct __user_cap_data_struct {
        __u32 effective;
        __u32 permitted;
        __u32 inheritable;
```

```
} *cap_user_data_t;
```

Listing 4.2. `struct __user_cap_data_struct` defining capability sets

Each process maintains three primary capability sets that control privilege acquisition and usage:

- **Effective set (pE):** Contains capabilities currently applied to privilege checks. These represent the capabilities that are "active" and will be used by the kernel when evaluating whether the process can perform privileged operations.
- **Permitted set (pP):** Contains capabilities that the process may move into its effective set via the `capset(2)` system call. The effective set is never a superset of the permitted set. Once a capability is removed from this set, it cannot be regained without executing a privileged program.
- **Inheritable set (pI):** Used in the calculation of capability sets at file execution time. Unlike the other sets, the inheritable set remains unchanged across `execve(2)` unless explicitly modified by the process itself using `capset(2)`.

When a new process is created via `fork(2)`, its capability sets are identical to its parent's. The inheritance occurs at the moment of process creation, establishing the initial privilege context for the child process.

File Capabilities

Since Linux 2.6.24, executable files can possess associated capability sets stored within extended attributes. Files have three capability sets: inheritable (fI), permitted (fP), and effective (fE). The file permitted set (fP) is also called the *forced set* because capabilities in that set will be in the process's new permitted set regardless of whether the process previously had them. The file inheritable set (fI) is sometimes referred to as the *optional set* because the program will only acquire capabilities from it if the invoking process includes them in its inheritable set.

During `execve(2)`, new capability sets are calculated based on the interaction between process and file capabilities. The process inheritable set remains unchanged across execution. The new permitted set combines capabilities from the file's forced set (masked by the bounding set) with capabilities that are present in both the process and file inheritable sets. The file effective capability (fE) functions as a boolean flag rather than a set and when set, all permitted capabilities become effective in the new process.

Key Capabilities

Among the numerous available capabilities, several demonstrate particular relevance for security and eBPF:

- **CAP_SYS_ADMIN** – Frequently regarded as the "new root" capability, providing extensive administrative operations.
- **CAP_NET_ADMIN** – Permits network configuration including interface settings, routing, or firewall rule management.
- **CAP_BPF** – Introduced in Linux 5.8 to separate BPF operations previously permitted under `CAP_SYS_ADMIN` into a more secure model. It enables basic operations including map creation, program loading, and attachment to various hooks. Not all map types and program categories are permitted; for instance, creating the previously discussed `BPF_MAP_TYPE_DEVMAP` requires `CAP_NET_ADMIN` or `CAP_SYS_ADMIN`. Additional information is available here [35].

Unprivileged eBPF

Originally, specific BPF programs could be loaded without special privileges, but following the identification of multiple vulnerabilities within the eBPF infrastructure, unprivileged loading faced progressive restrictions. In contemporary Linux kernels, loading most eBPF programs requires `CAP_BPF` and `CAP_SYS_ADMIN`, demonstrating this mechanism's sensitivity.

Unprivileged eBPF access operates under regulation by the `kernel.unprivileged_bpf_disabled` sysctl parameter, which governs whether the `bpf(2)` system call can be invoked without elevated privileges. This parameter accepts the following values:

- **0:** Unprivileged `bpf(2)` usage is permitted.
- **1:** Unprivileged usage becomes permanently disabled for the running kernel's lifetime. Once configured to 1, the value cannot revert to 0, even through administrator intervention. Any unprivileged `bpf(2)` invocation will produce `-EPERM` errors.
- **2:** Unprivileged usage is disabled, but the configuration remains modifiable during runtime. Administrators may subsequently write 0 or 1 to reactivate or permanently disable it. This value was introduced in Linux 5.13.

When the kernel is built with `CONFIG_BPF_UNPRIV_DEFAULT_OFF`, the parameter's default value becomes 2 rather than 0, ensuring unprivileged usage is disabled by default while maintaining administrative control. Without this configuration option, the default remains 0.

This mechanism provides administrators with detailed control: either permanently securing unprivileged eBPF usage, or temporarily disabling it while preserving reactivation options when needed.

4.3 Linux Security Modules (LSM)

The Linux Security Modules (LSM) framework [36] was introduced to support the implementation of flexible and fine-grained security policies inside the kernel, without hardcoding a single access control model. Instead of embedding complex security logic directly into the kernel, LSM provides a generic interface that decouples mechanism from policy. This design enables different major security models, such as SELinux [37], or AppArmor [38], to be implemented, though typically only one major LSM can be active at a time.

The core of the framework consists of *security hooks*, which are callback functions strategically placed throughout the kernel at points where sensitive operations occur. Typical hook locations include file system operations (`open`, `mmap`), process management (`execve`, `ptrace`), networking (socket creation, packet transmission), and inter-process communication. When a hook is triggered, the kernel invokes the registered security function of the active module before completing the operation. The module can then allow, deny, or alter the request according to its policy. For example, the `security_inode_permission()` hook is called whenever a process attempts to access a file, giving the LSM module the chance to enforce additional checks beyond the standard DAC and capability-based mechanisms.

To maintain state across kernel objects and their lifetimes, the LSM framework provides *security blobs*: opaque `void*` fields added to many core structures (e.g., `task_struct`, `inode`, `sock`). These blobs are allocated, initialized and freed by the active security module; the kernel itself treats them as opaque pointers and does not interpret their contents. Security modules therefore use the blob to attach module-specific metadata to objects. When an LSM hook is invoked, the module dereferences the blob to obtain the contextual information needed to evaluate a policy decision. Because the framework offers only the storage slot, modules are responsible for lifecycle management (allocation and cleanup), synchronization (RCU primitives or

explicit locking when replacing or reading blobs), and memory accounting. This design gives modules fast, persistent access to per-object policy state while keeping the core kernel agnostic about the specific policy data structures.

All hooks are organized into the global `security_operations` table, a structure grouping related function pointers by kernel object category (e.g., task, inode, file, or networking). When the kernel executes a sensitive operation, the corresponding hook is invoked through this table. This mechanism makes it possible to implement system-wide security enforcement in a modular fashion, with the base kernel providing only the infrastructure while the actual policies are defined by the loaded security module.

From a policy perspective, LSM was designed to be policy-neutral: the framework itself does not impose a particular model of access control, but rather provides the mechanism upon which diverse policies can be built. For example, SELinux implements mandatory access control (MAC) based on Type Enforcement, and AppArmor enforces pathname-based confinement. The flexibility offered by LSM enables experimentation with novel security paradigms, while simultaneously allowing administrators to select a model that best fits their operational requirements. This separation of mechanism and policy was emphasized in the original design rationale [39].

Traditionally, the major LSMs are mutually exclusive, only one can be active at boot time as the primary security module. However, the framework does support limited stacking: certain "minor" LSMs (such as Yama, LoadPin, or SafeSetID) can operate alongside a major LSM, and more recently, BPF-LSM has been designed to be fully stackable with existing security modules.

The LSM framework's operational model emphasizes kernel-resident, synchronous decision making. Hooks are placed at critical kernel locations so that each security decision is processed immediately and atomically within the kernel context when the associated system call or kernel operation executes. By handling decisions in-kernel via lightweight callbacks, LSM minimizes latency and avoids the risk that enforcement can be bypassed by delays or failures in external components, such as TOCTOU. In practice, major LSM implementations (for instance, SELinux or AppArmor) perform policy evaluation entirely in kernel context: they read attributes attached to kernel objects (security blobs on inodes, tasks, etc.) and quickly decide whether an operation complies with the loaded policy.

Interaction with user space is intentionally minimized in the standard LSM model. The kernel–user interface is primarily used for configuration and monitoring (for example, exposing policy state and audit data through `/proc`, `/sys`, or netlink to user-space daemons such as `auditd`). This asynchronous,

indirect channel avoids per-decision round trips to user space and preserves the determinism and performance characteristics required for robust enforcement.

LSM hooks are intended solely for making rapid, in-kernel decisions. The framework's focus on minimal overhead and a compact interface has allowed it to support a wide array of security modules without forcing a single unified policy model. This modularity permits systems to be hardened through complementary policies (for instance, combining a base MAC system with additional auditing or logging mechanisms), while leaving composition and more complex decision logic to the individual modules themselves.

4.3.1 BPF LSM

Historically, security modules were compiled into the kernel as substantial, monolithic extensions. Although powerful, this approach necessitated kernel rebuilding or patching to introduce or alter security policies. Prior to BPF emergence, users possessed fundamentally two alternatives: configure existing LSMs including AppArmor or SELinux, or develop custom kernel modules. Both methodologies proved effective yet lacked adaptability, as policies remained static and could only undergo modification through recompilation or module reloading.

Starting from Linux 5.7, *BPF LSM* became available. This feature enables security professionals to design custom policies that are injected into the kernel and activated through LSM hooks. Leveraging eBPF programs [40], administrators can implement detailed security policies that support runtime modification, eliminating kernel module requirements, which may introduce bugs or system instability. These programs undergo verification during loading and may utilize BPF maps and helper functions to enforce systemwide mandatory access control (MAC) and auditing policies.

BPF LSM development received substantial motivation from the *Kernel Runtime Security Instrumentation (KRSI)* project, presented by KP Singh at the Linux Security Summit North America 2019 [41]. KRSI investigated how eBPF could integrate signal concepts (suspicious events, including binary execution with unusual environment variables) and mitigation strategies (policy actions, including mount blocking or binary blacklisting) within a unified runtime mechanism. The prototype demonstrated that eBPF attachment to LSM hooks could deliver low overhead, precise, and context aware enforcement, with this vision ultimately integrated as BPF LSM in Linux 5.7.

Moreover, BPF LSM is designed with stackability in mind. Traditional LSMs are often mutually exclusive due to how they are integrated into the

kernel's security framework. In contrast, BPF LSM can run alongside these modules, enabling administrators to layer dynamic policies on top of a base LSM. This is particularly useful in scenarios where, for example, SELinux enforces a fixed mandatory access control policy, and additional filtering or logging is needed without modifying the core SELinux policy.

While traditional eBPF mechanisms (e.g., probes, tracepoints, syscall filters) provide powerful monitoring, they lack the structured semantics and enforcement guarantees provided by LSM hooks. LSM hooks are tightly integrated into the kernel's security model and deliver contextual arguments about the process and operation being performed; this enables eBPF programs attached to these hooks to implement context-aware, fine-grained decisions.

This design provides several advantages:

- **Granular enforcement:** Hooks such as `inode_permission` apply to all file operations (read, write, execute), ensuring consistent policy coverage beyond simple syscall interception.

- **Context-aware policies:** Enforcement can depend on the full kernel context, process identity, credentials, and existing LSM labels (e.g., SELinux), rather than on a single intercepted event.

- **Kernel integration:** Because decisions remain kernel-resident and are made atomically at hook points, BPF LSM preserves the speed, determinism, and security properties expected of traditional LSM enforcement while providing enhanced programmability.

Naturally, this flexibility involves tradeoffs. Expanding programmability increases potential attack surfaces, and runtime loading, verification, and dynamic eBPF program execution introduces performance and complexity overhead compared to carefully tuned static modules. Nevertheless, BPF LSM represents a pragmatic middle ground: it retains kernel enforcement semantics of LSM while enabling rapid policy evolution and experimentation.

Writing BPF LSM Programs

Prior to developing or attaching BPF LSM programs, verifying kernel support and feature enablement becomes essential. This verification involves checking the kernel configuration to confirm that BPF LSM support was compiled into the kernel, typically indicated by the `CONFIG_BPF_LSM=y` setting in the kernel configuration file. Additionally, administrators must verify that BPF

LSM appears in the active LSM list within the `/sys/kernel/security/lsm` file, which displays currently enabled security modules.

When BPF LSM does not appear in the active security modules list, manual activation becomes necessary through kernel command line modification. For Ubuntu systems, this process involves editing the GRand Unified Bootloader (GRUB) configuration file `/etc/default/grub` to incorporate BPF LSM into the LSM parameter list alongside existing modules.

Multiple approaches exist for BPF program development, encompassing C with libbpf, higher-level languages such as Go with eBPF bindings, or direct BPF assembly utilization. Throughout this thesis, all programs employ C using the libbpf library, which delivers userspace assistance for loading, verifying, and attaching eBPF programs to kernel hooks.

Complete LSM hook definitions reside within the Linux kernel source tree under the header file `lsm_hooks.h` [42]. Comprehensive comments within that file describe individual hook semantics and intended applications. A corresponding file, `lsm_hook_defs.h` [43], presents hook definitions using the following format:

```
LSM_HOOK(<return_type>, <default_value>, <hook_name>, args...)
```

Most hooks return integer values, while others return `void` (indicating result ignored). Standard return codes for LSM hooks include:

- `0`: Authorization granted.
- `ENOMEM`: Denial due to memory allocation failure.
- `EACCES`: Access explicitly denied by security policy.
- `EPERM`: Operation requires permission not granted.

BPF LSM programs are implemented as specialized eBPF program types, annotated using the keyword `SEC("lsm/<hook>")`, where `<hook>` represents the targeted LSM hook name (e.g., `inode_permission`, `task_alloc`). The general structure follows:

```
1 #include "vmlinux.h"
2 #include <bpf/bpf_helpers.h>
3 #include <bpf/bpf_tracing.h>
4 #include <errno.h>
5
6 char LICENSE[] SEC("license") = "GPL";
7
```

```
8  SEC("lsm/bpf")
9  int BPF_PROG(lsm_bpf, int cmd, union bpf_attr *attr, unsigned
       int size, int ret)
10 {
11     // ret is the return value from the previous BPF program
      or 0 if it's the first hook.
12     if (ret != 0)
13         return ret;
14
15     bpf_printk("LSM: block bpf() worked");
16     return -EPERM;
17 }
```

Listing 4.3. Example of a BPF LSM program

Within this example, the BPF program intercepts the `bpf` hook, which triggers during process invocation of the `bpf(2)` system call. The program examines return values from previously attached BPF LSM programs (or 0 for initial execution) and, when no prior program has denied the operation, logs a message and returns `-EPERM` to block the call.

From userspace perspectives, such programs typically undergo compilation with `clang` and kernel loading via `libbpf`. Attachment occurs through the helper `bpf_program__attach_lsm()`, which binds programs to selected hooks. Links can subsequently be removed using `bpf_link__destroy()`, ensuring safe cleanup.
Complete BPF LSM programs typically encompass the following components:

- **Program definition:** One or multiple functions annotated with `SEC("lsm/...")`, corresponding to LSM hooks.
- **Helper functions and maps:** Utilized for state storage, counter sharing, or userspace communication.
- **Userspace loader:** A compact application that loads BPF object files into the kernel, attaches programs, and manages lifecycle (using `libbpf` or bpftool).

This section has provided an overview of the Linux Security Modules framework, emphasizing its modular design, kernel integration, and the innovative BPF LSM extension that enables dynamic, programmable security policies. These techniques are deeply used in the subsequent chapters to implement advanced security monitoring and enforcement mechanisms.

Chapter 5

Kernel Security and Monitoring Tools

The Linux kernel presents powerful yet fragile execution environments where a single compromise can destroy the entire system. This reality drove development of rich security and monitoring mechanism ecosystems protecting kernel integrity, enforcing policies, and observing runtime behavior.

This chapter examines two technology families: Loadable Kernel Modules (LKMs) and eBPF-based tools, which represent different kernel security and observability approaches. LKMs like Linux Kernel Runtime Guard (LKRG) [44] offer deep kernel integration and can enforce strong invariants, though at complexity and risk costs. eBPF-based systems such as Tetragon [20] use safer, more dynamic instrumentation models that minimize kernel exposure while enabling rich monitoring and enforcement capabilities.

Together, these mechanisms show Linux kernel security evolution: from static, privileged modules with complete kernel access toward safer, dynamic, fine-grained instrumentation that eBPF enables. This chapter analyzes these tools thoroughly, beginning with LKRG as representative kernel module defense, then examining Tetragon as eBPF-based monitoring and enforcement system.

5.1 Loadable Kernel Modules and LKRG

Loadable Kernel Modules (LKMs) represent Linux's standard mechanism for adding or removing kernel functionality during runtime. LKMs are binary objects linked against running kernels that can register new device

drivers, filesystem code, networking features, or security and observability logic. Userspace tools like `insmod`, `modprobe` and `rmmod` typically insert and remove modules, which interact with kernels through well-defined module APIs and exported symbols.

Security and monitoring perspectives reveal several attractive LKM properties. These modules provide complete kernel context by executing in kernel space, which enables kernel state observation and manipulation with minimal mediation. LKMs also offer extensive Application Programming Interface (API) surfaces through their ability to register hooks, intercept kernel events, and export user space interfaces via `procfs`, `sysfs`, character devices, netlink, and other mechanisms. Additionally, their flexibility and performance advantages stem from in-kernel implementations that avoid user/kernel transitions for many operations, thereby enabling low-latency monitoring and rapid enforcement capabilities.

However, LKMs also create significant risks and operational burdens that must be carefully considered. These modules present stability and compatibility challenges because they must match kernel ABI (Application Binary Interface) and data layout expectations; buggy or mismatched modules can crash systems or cause subtle corruption that may be difficult to detect. The security surface expands considerably since modules run with complete kernel privileges, meaning vulnerabilities in module code may prove as damaging as core kernel bugs. Research by Haizhi Xu et al. demonstrated this risk by showing that malicious or buggy LKMs can compromise system integrity by executing exploit code in kernel space [45].

Deployment complexity adds another layer of difficulty, as shipping, signing, and maintaining modules across kernel upgrades demands careful operational processes including module signing, distribution, and extensive testing. Furthermore, the similarities between legitimate LKMs and rootkit behavior create trust issues, since malicious rootkits frequently use LKMs for persistence and stealth. This similarity means that deploying third-party modules often requires additional trust and verification steps that can complicate system administration.

Modern deployments balance in-kernel capability needs with safer alternatives because of these trade-offs. Nevertheless, LKMs remain essential where direct kernel internal access or maximum performance is required. Complex device drivers, some high-assurance monitoring solutions, and specialized security primitives still depend on the deep kernel integration that only LKMs can provide, making them an indispensable part of the Linux kernel ecosystem despite their inherent risks.

5.1.1 Linux Kernel Runtime Guard (LKRG)

Linux Kernel Runtime Guard (LKRG) [44] represents a loadable kernel module developed by Openwall project that performs runtime integrity checking of Linux kernels and security violation detection. Unlike traditional intrusion detection systems operating in user space, LKRG resides within kernels themselves and continuously monitors critical kernel object and control flow integrity. Its main goal involves detecting kernel exploits and rootkits real-time and, depending on configuration, either logging incidents or taking immediate countermeasures.

LKRG's architecture comprises several interconnected building blocks that work together to provide comprehensive kernel protection. The core component is an integrity verification engine that computes kernel code and critical data structure checksums at load time, continuously verifying them during operation. This engine protects kernel `.text` and `.rodata` regions including syscall tables, procedures and functions, and exported symbols, while also monitoring dynamically loaded modules and their internal structure order, plus global system variables that represent critical protected objects.

Beyond static integrity checking, LKRG incorporates sophisticated exploit detection mechanisms that monitor kernel runtime state to identify privilege escalation indicators, hidden processes, or kernel credential and task structure tampering. These mechanisms work in conjunction with a configurable enforcement layer that responds to detected anomalies through various actions: logging only, blocking offending actions, killing compromised processes, or triggering kernel panics. This flexibility allows administrators to balance availability and security requirements according to their specific operational needs.

To ensure its own survival against sophisticated attacks, LKRG implements comprehensive self-protection measures that resist tampering attempts. These protections include shielding its code from being unloaded, restricting write access to its memory regions, and validating its own control flow to prevent attackers from disabling or corrupting the monitoring system itself. This multi-layered approach creates a robust defense mechanism that operates entirely within the kernel space while maintaining continuous vigilance against various forms of compromise.

Runtime Flow

Typical LKRG operation proceeds as follows:

1. **Module Initialization:** During `module_init`, LKRG sets up internal data structures, computes baseline kernel code and module hashes, and registers its periodic checker.

2. **Monitoring:** LKRG operates using both event-driven and periodic mechanisms:

   - Periodic integrity scans verify kernel text and loaded modules remain unchanged.
   - Runtime hooks monitor kernel events like process credential changes, module loading, or suspicious control flow modifications.

3. **Detection:** Whenever mismatches or suspicious activity appear (e.g., altered kernel text, hidden modules, unexpected privilege escalation), LKRG flags them as potential compromises.

4. **Reaction:** Depending on configuration, LKRG logs events, terminates offending processes, or in high-assurance deployments, panics kernels to halt further exploitation.

5. **Cleanup:** On `module_exit`, LKRG unregisters its checks and frees memory; however, it may be configured preventing unloading entirely to ensure continuous protection.

Use Cases and Limitations

LKRG's primary goal involves detecting kernel exploitation attempts by monitoring corruption signs in critical kernel data and execution state. It observes both processes and kernel attributes identifying patterns typically corresponding to exploitation or privilege escalation. The system demonstrates particular effectiveness in detecting illegal privilege escalation attempts, including efforts to swap tokens or pointers, unauthorized `commit_creds()` invocations, or tampering with `cred` structures that could grant unauthorized access.

Beyond privilege escalation, LKRG excels at identifying sandbox and container escape attempts through detection of seccomp configuration or rule corruption, namespace isolation flaw exploitation, or container breakout attempts in environments like Docker or Kubernetes. The system also monitors CPU state manipulation, catching unauthorized changes to critical control registers or flags such as Supervisor Mode Execution Protection (SMEP) and Supervisor Mode Access Prevention (SMAP), that attackers commonly abuse

to disable hardware-enforced protections. Additionally, LKRG maintains vigilance over kernel code and module integrity, detecting modifications to kernel text regions or loaded modules that break established integrity baselines.

Perhaps most importantly, LKRG identifies control-flow violations where kernel code executes from non-`.text` pages, dynamically generated executable memory, or even user-space pages. These patterns typically indicate sophisticated attacks like Return-Oriented Programming (ROP) or stack-pivoting aimed at bypassing SMEP protections.

While LKRG raises the bar significantly for attackers, it has inherent limitations and isn't immune to bypass attempts. Operating within the same trust boundary as kernels themselves, sophisticated adversaries can attempt evading detection or disabling LKRG directly. Attackers might modify kernel metadata not currently protected by LKRG, or race against its periodic checks to avoid detection windows. More advanced approaches could target LKRG's internal synchronization or data structures to corrupt its monitoring database or disable active checks entirely. Some attackers leverage kernel vulnerabilities to attack user space directly without triggering the invariants LKRG monitors.

These limitations highlight that LKRG shouldn't be considered complete defense in isolation. Rather, it proves most effective when deployed as part of layered security strategies, complementing other kernel hardening mechanisms and user-space monitoring tools. This approach creates multiple defensive barriers that significantly increase the complexity and resources required for successful system compromise.

5.2 Tetragon

Tetragon [20] represents an open-source runtime security observability and enforcement platform constructed on top of eBPF. Developed by Isovalent within the Cilium ecosystem, it provides fine-grained, low-overhead visibility into process execution, file access, and network activity without requiring kernel modules. This architecture supports both monitoring and active security policy enforcement directly inside kernels, making Tetragon especially suited for cloud-native and containerized environments. A notable characteristic involves its Kubernetes-awareness: Tetragon natively understands workloads through Kubernetes namespaces, pods, and identities, enabling policy enforcement at individual container and service granularity.

Embedding filtering, policy enforcement, and event processing directly within kernels through eBPF programs allows Tetragon to drastically reduce overhead typically caused by frequent kernel/user space transitions. Rather than forwarding all kernel events indiscriminately, it uses fine-grained in-kernel filters based on various parameters including file paths, socket addresses, process names, namespaces, and capabilities. This selective forwarding ensures only relevant events reach user-space agents for further processing.

Unlike conventional syscall-based tracing that can suffer from argument inconsistencies and manipulation, Tetragon hooks into kernel functions and data structures that user processes cannot easily alter. This deep kernel instrumentation bypasses common pitfalls associated with user/kernel boundary interactions, including incorrect argument capture or page faults.

5.2.1 Policy Architecture and Specification

The TracingPolicy abstraction serves as Tetragon's primary interface for defining monitoring and enforcement behaviors. Each policy encapsulates three fundamental components working together:

- hook points that define kernel attachment locations;
- selectors that implement in-kernel filtering logic;
- optional `matchActions` that specify enforcement responses.

This architecture's flexibility allows users to craft highly specific policies tailored to particular threat models while maintaining system performance.

Hook points determine where eBPF programs attach within kernel execution flows, and several options exist depending on specific monitoring

requirements and portability needs. When maximum flexibility is needed, kprobes provide dynamic instrumentation capabilities for arbitrary kernel functions, though this flexibility comes at the cost of kernel version dependency. For more stable implementations, tracepoints offer a better alternative since they represent statically defined instrumentation interfaces that maintain stability across kernel versions, making them the preferred choice for portable policy implementations. Monitoring extends beyond kernel space through uprobes, which bring observability capabilities to functions in user-space binaries or libraries, thereby enabling comprehensive observability across entire system stacks. Additionally, LSM BPF Programs create another important avenue by enabling hooks on Linux Security Module interfaces, which proves essential for implementing system-wide Mandatory Access Control and audit policies.

Once hook points are determined, selector mechanisms implement core filtering logic determining event relevance and policy matching. Selectors operate on function arguments, process metadata, file system objects, network endpoints, and capability sets, among other kernel data structures. Evaluation logic supports boolean expressions, enabling filtering scenarios that reduce false positives while maintaining detection efficacy. Critically, selector evaluation occurs entirely within kernel space, ensuring filtering decisions get made with minimal performance impact. As a drawback, Tetragon's filtering capabilities face inherent constraints from eBPF verifier safety restrictions, preventing operations like pointer dereference or direct access to struct members passed as function arguments. These limitations will be demonstrated practically in the second use case.

When selectors match specified criteria, Tetragon can execute `matchActions`, enabling both observational and enforcement capabilities. The action framework supports comprehensive response ranges operating either directly within kernels or through coordinated user-space operations. Process termination actions include the `Sigkill` action, which synchronously terminates offending processes from within kernels, and the more flexible `Signal` action allowing arbitrary signal sending to processes. The `Override` action provides function-level enforcement by modifying intercepted call return values, effectively preventing original operations from completing while returning specified error codes to callers.

The enforcement model operates on synchronous policy evaluation and response principles. When monitored kernel functions get invoked, attached eBPF programs evaluate configured selectors against current execution contexts. Upon detecting matches, specified actions execute immediately within

the same kernel contexts, ensuring policy violations get addressed with minimal latency.

To illustrate practical application of these concepts, consider a policy designed to monitor and potentially restrict eBPF program loading operations:

```yaml
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "monitor-bpf-syscall"
spec:
  kprobes:
  - call: "sys_bpf"
    message: "Generic sys_bpf call with command
   BPF_PROG_LOAD"
    syscall: true
    args:
    - index: 0
      type: "int"
    - index: 1
      type: "bpf_attr"
    selectors:
    - matchArgs:
      - index: 0
        operator: "Equal"
        values:
        - "5"  # BPF_PROG_LOAD command
      matchCapabilities:
      - type: "Effective"
        operator: "NotIn"
        values:
        - "CAP_SYS_ADMIN"
        - "CAP_BPF"
      matchActions:
      - action: "Override"
        argError: -1
      - action: "Sigkill"
```

Listing 5.1. Tetragon TracingPolicy Example

This policy demonstrates several key concepts in Tetragon's design philosophy. The hook point attaches to the `sys_bpf` kernel function, which handles all eBPF-related system calls. Selector logic filters for program loading operations (command value equals to 5) while examining process effective capability sets. When processes attempt loading eBPF programs without appropriate capabilities (CAP_SYS_ADMIN or CAP_BPF), the policy triggers both enforcement mechanisms: overriding system call return values with error

codes and delivering SIGKILL signals to terminate offending processes.

This approach's efficiency lies in completely eliminating user-space involvement in enforcement decisions. eBPF programs evaluate policy conditions, make enforcement decisions, and apply configured actions entirely within kernel execution contexts.

Chapter 6

Analysis, Exploitation and Hardening of eBPF Vulnerabilities

This chapter presents the fundamental research conducted for this thesis, encompassing the examination of two distinct scenarios centered on severe security flaws, the exploitation of these weaknesses, and the deployment of security reinforcement methods to address them.

The security flaws chosen for this examination have been recognized as possessing elevated CVSS score [46], demonstrating substantial potential consequences for system security. Specifically, two of these have been selected for comprehensive examination, from which two use cases were developed. The initial scenario concentrates on an elevation of privileges vulnerability (CVE-2022-23222) that enables non-privileged users to obtain root permissions by leveraging defects in the eBPF verifier's pointer arithmetic management. The subsequent use case investigates two connected vulnerabilities (CVE-2024-56614 and CVE-2024-56615) that result in kernel crashes and service disruption conditions through out-of-bounds memory operations in particular eBPF map configurations.

Following the examination of these vulnerabilities' characteristics and comprehending the complete context surrounding them, the investigation proceeds with discovering publicly available and open-source exploitation methodologies. For both scenarios, public demonstration exploits exist; however, certain modifications were implemented to the original implementations to provide

enhanced capabilities and superior evidence of the exploitation process. Additionally, thorough investigation is necessary to completely comprehend the exploitation methodologies and understand the restrictions and limitations of the exploits, while also establishing the foundation for developing mitigation approaches. Subsequently, a controlled and secure testing environment is established to evaluate and reproduce the exploits. This requires constructing virtual machines with particular Linux distributions and kernel releases known to be vulnerable to the defect. The testing environment is meticulously configured to ensure seamless exploit execution while enabling comprehensive monitoring and examination of system behavior throughout and following the exploitation process. This configuration permits examination of system calls, kernel source analysis, and runtime debugging to obtain insights into the vulnerabilities and their exploitation methods.

Ultimately, for each scenario, an hardening plan is recommended. This begins with the patch that resolves the particular vulnerability, then extends further by suggesting supplementary mitigation approaches to strengthen the system's overall security posture. This encompasses utilizing available security solutions, including LKRG (Linux Kernel Runtime Guard) for the initial scenario, and Tetragon for the subsequent scenario. In addition, the thesis introduces custom BPF LSM programs as an alternative enforcement layer. These policies complement existing tools, provide gains in precision and timeliness, and at the same time involve clear limitations and scope constraints.

The recommended solutions were designed to be as broadly applicable as feasible; nevertheless, identifying universal solutions applicable to any Linux distribution and kernel release that can address various vulnerabilities remains extremely challenging.

6.1 Use Case 1 – Privilege Escalation

6.1.1 Vulnerability Description

The first use case is founded on CVE-2022-23222 [47], representing an input validation flaw within the Linux kernel's eBPF verifier that enables unsafe pointer arithmetic operations on particular `OR_NULL` pointer categories.

As detailed in Section 2.2.8, the verifier creates two execution branches when an `OR_NULL` pointer undergoes NULL validation. In the NULL branch, pointer arithmetic operations should be explicitly forbidden since no valid pointer exists.

The vulnerability exploits an incomplete enforcement of this restriction in certain kernel versions. When an `OR_NULL` pointer type is in its NULL branch, the verifier incorrectly permits arithmetic operations that should be blocked entirely. This creates a type confusion scenario: the verifier's internal tracking believes it is managing an `OR_NULL` pointer type progressing through standard null-check validation logic, but the actual register content is a scalar zero value that can be manipulated through operations intended exclusively for pointer arithmetic. This discrepancy allows an attacker to construct arbitrary scalar values within a register that the verifier incorrectly assumes contains only NULL or bounded pointer offsets.

By exploiting this primitive, an attacker can craft arbitrary memory addresses, enabling out-of-bounds kernel memory read and write capabilities. These capabilities can then be leveraged to modify critical kernel data structures, including process credentials within the `task_struct`, ultimately achieving privilege escalation to root.

The security flaw obtained a CVSS v3.1 base rating of 7.8, classified as HIGH severity, and impacts the following Linux kernel releases:

- from 5.8.0 (inclusive) through 5.15.37 (exclusive)
- from 5.16.0 (inclusive) through 5.16.11 (exclusive)

6.1.2 Exploitation Process

The exploitation approach leverages a maliciously designed eBPF program that misleads the verifier into accepting a manipulated register value as a secure register. The procedural steps are outlined as follows:

1. The helper function `bpf_ringbuf_reserve` is employed to create a `PTR_TO_MEM_OR_NULL` stored within a register, designated as `REG_0`, whose actual runtime value equals NULL.

2. The value contained in `REG_0` is transferred to an additional register, named `EXP_REG`.

3. An offset of $x - 1$, where x represents the adversary's target arbitrary value, is added to `EXP_REG`. Due to the vulnerability, the verifier permits this arithmetic operation and continues to incorrectly track both `REG_0` and `EXP_REG` as containing equal values, despite `EXP_REG` now holding the scalar $x - 1$.

4. A conditional branch evaluates whether `REG_0` equals NULL. Within the branch where `REG_0` equals NULL, the verifier incorrectly considers `EXP_REG` as NULL as well, despite containing the scalar $x - 1$. In the alternative branch, the program is configured to terminate.

5. Subsequently, 1 is added to `EXP_REG`, yielding `EXP_REG` containing the arbitrary value x, despite the verifier's incorrect presumption that it maintains a secure pointer.

Following this manipulation, the verifier is deceived into permitting `EXP_REG` to be utilized for arbitrary kernel memory operations. However, direct pointer arithmetic using the invalid register would still be blocked by ALU sanitation. As explained in Section 2.2.8, when the verifier detects arithmetic operations between a pointer and a scalar register whose value is known, it patches the instruction to replace the scalar register with an immediate value, rendering the invalid register useless.

To bypass ALU sanitation, the adversary must leverage eBPF helper functions that accept the invalid register as a size or length parameter rather than using it in direct pointer arithmetic. These helpers have runtime-determined behavior opaque to static analysis: the verifier cannot predict their execution paths or apply ALU sanitation because no direct pointer arithmetic operation appears in the bytecode. Instead, the invalid register is passed as a function argument, and the memory operations occur within the helper's internal implementation.

However, the available helper functions are constrained by the eBPF program type being loaded. Each program type supports a distinct subset of helpers, and this selection is further restricted by capability requirements. When unprivileged eBPF is enabled (`kernel.unprivileged_bpf_disabled=0`), only `BPF_PROG_TYPE_SOCKET_FILTER` and `BPF_PROG_TYPE_CGROUP_SKB` programs can be loaded by non-privileged users. The exploit targets `BPF_PROG_TYPE_SOCKET_FILTER`, which provides access to these two specific helpers capable of achieving the required out-of-bounds memory operations while circumventing the sanitation mechanism:

- `bpf_skb_load_bytes`, which retrieves data from a socket buffer into memory. By providing `EXP_REG` as the length parameter, the program instructs the helper to copy an attacker-controlled number of bytes. Since this occurs within the helper's internal implementation and not through visible pointer arithmetic in the BPF program, ALU sanitation cannot

intervene. This enables out-of-bounds writes by copying more data than the allocated buffer can contain.

- `bpf_ringbuf_output`, which transfers data to a ring buffer accessible from user space. Utilizing `EXP_REG` as the size parameter instructs the helper to read an attacker-controlled amount of data. Similarly, because the actual memory read operation happens inside the helper function's implementation, ALU sanitation mechanisms designed to validate pointer arithmetic at the BPF instruction level are completely bypassed. This achieves out-of-bounds reads by extracting data beyond the intended boundaries.

At this stage, the exploit has achieved out-of-bounds read and write capabilities. However, to gain fully arbitrary kernel memory read and write, the adversary must identify a suitable memory structure whose manipulation can elevate these primitives into unrestricted kernel access. BPF maps and their corresponding `bpf_map` structure serve as an ideal target for this purpose, as they contain memory pointers and internal fields that, when strategically modified, enable broader kernel memory access.

Specifically, by overwriting the `btf` pointer member and leveraging the helper `bpf_map_get_btf_id` (which internally calls the `bpf` syscall with the `BPF_OBJ_GET_INFO_BY_ID` command), the adversary can retrieve arbitrary kernel memory locations, establishing the arbitrary read primitive.

Achieving arbitrary write, however, requires a more sophisticated approach. The adversary cannot directly invoke kernel functions with arbitrary parameters, as the syscall interface enforces strict validation. The solution exploits `map_get_next_key`, a function that writes key values to a user-supplied buffer pointer as part of its normal operation. To redirect this write capability to arbitrary kernel addresses, the adversary hijacks the `map_ops` function pointer table within the `bpf_map` structure. Using the arbitrary read primitive, the attacker duplicates this table, replaces the `map_push_elem` pointer with `map_get_next_key`, and updates the original `map_ops` pointer to reference the corrupted table. When `bpf_map_update_elem` is subsequently invoked from user space, the kernel executes `map_get_next_key` with attacker-controlled parameters, enabling 32-bit writes to arbitrary writable kernel memory addresses.

With both arbitrary read and write primitives established, the final privilege escalation step becomes straightforward. The adversary locates the `task_struct` of the executing process using kernel symbols, retrieves the

`cred` pointer to the credentials structure, and overwrites essential fields including `uid`, `gid`, `euid`, and `egid` with zero. This grants the process root privileges, completing the privilege escalation attack.

The proof of concept exploit implementation is accessible on GitHub [48] and is specifically validated with appropriate symbols for Ubuntu 20.04 with kernel version `5.13.0-27-generic`. The exploit implements other techniques to bypass additional checks and kernel protections, such as freelist randomization, but they are out of scope for this discussion. The complete analysis can be found in the original technical report [49].

Modifications were implemented to the original implementation to provide enhanced functionalities, including:

- **Complete privilege escalation attack**, configuring all credential IDs (uid, gid, euid, egid, suid, sgid, fsuid, fsgid) to zero.
- **Privilege restoration attack**, which involves configuring only the saved IDs (`suid`, `sgid`) to zero. This approach is regarded as stealthy, since the user maintains the appearance of a normal non-privileged user, but possesses the ability to obtain full root privileges by executing the `seteuid(0)` system call.
- **File access attack**, configuring file system IDs (fsuid, fsgid) to zero, providing access to root-protected files including `/etc/shadow` or similar.
- **Process termination attack**, configuring effective IDs (euid, egid) to zero, enabling termination of any process, including those owned by root.

To properly execute the exploit, `CAP_BPF` is necessary, unless unprivileged BPF is enabled within the kernel configuration. In the Appendix A.1, the execution of the exploit is demonstrated with screenshots, showcasing the successful privilege escalation to root.

6.1.3 Hardening plan

Various defensive approaches are recommended to address the security flaw exploited in CVE-2022-23222.

Primarily, it remains essential to ensure that the system operates with a kernel release that has been updated to resolve this particular security issue. The remediation for CVE-2022-23222 was published in kernel releases 5.15.37 and 5.16.11, therefore upgrading to a minimum of these releases is fundamental. Alternatively, when kernel upgrading proves unfeasible, it is

advisable to apply the remediation manually and subsequently recompile the kernel.

An additional effective defensive approach involves restricting eBPF program utilization to authorized users exclusively. This objective can be accomplished by constraining the capabilities necessary to load eBPF programs to a designated group of users, including system administrators. This configuration can be achieved by configuring the `kernel.unprivileged_bpf_disabled` sysctl parameter to 1.

When the previously mentioned measures prove impractical, it becomes necessary to deploy supplementary security mechanisms. One such mechanism involves utilizing LKRG (Linux Kernel Runtime Guard), which provides runtime integrity verification for the kernel. It can identify and prevent numerous attack types, and specifically, LKRG monitors modifications to the credential structure and can identify unauthorized alterations that might indicate a privilege escalation attempt, preventing the exploit's success.

An additional compelling approach to address the security flaw, representing the central contribution of this thesis, involves implementing a BPF LSM program. This program utilizes the eBPF infrastructure to enforce detailed security policies directly via the Linux Security Module (LSM) framework. By implementing this approach, it enables the operating system to restrict or mediate the loading and execution of eBPF programs according to customizable policies, thereby addressing attack categories similar to the one exploited in CVE-2022-23222.

BPF LSM Program Implementation

The deployed BPF LSM program delivers a multi-layered security mechanism utilizing eBPF to enforce comprehensive policy controls within the Linux Security Module framework. The fundamental concept behind its architecture involves monitoring and controlling exclusively those processes that invoke BPF syscalls, as these represent the processes relevant to the security flaw exploited in CVE-2022-23222.

Fundamentally, an eBPF map named `tracked_processes` is utilized to maintain a snapshot of the initial credentials of each monitored process. This map accommodates up to 1024 entries, where each entry encompasses multiple user and group identifiers including UID, EUID, GID, FSUID, FSGID, SUID, and SGID. These data are retrieved directly from the kernel task's credential structure utilizing CO-RE (Compile Once - Run Everywhere) safe operations, enabling the program to function reliably across different kernel versions

without recompilation.

The selection of LSM hooks represents the central element of the program's effectiveness and is founded on the Linux kernel's extensive hook infrastructure, as specified in `lsm_hooks.h`. These hooks deliver interception points at critical kernel security decision junctures, enabling the BPF LSM program to enforce its policy in real time.

The initial hook, `lsm/bpf`, functions as the program's entry point: it intercepts every BPF syscall issued by any process. When activated, this hook verifies if the issuing process is already monitored. If not, it retrieves the current credentials from the kernel's task structure utilizing `bpf_core_read()` and stores them in the `tracked_processes` map. This ensures that every process utilizing eBPF features is monitored from the moment it interacts with the BPF subsystem.

Subsequently, the `lsm/task_alloc` hook intercepts process creation events. When the kernel allocates a new task (i.e., a new process), this hook verifies if the parent process is monitored. If so, it duplicates the parent's credential snapshot to the child process, ensuring that the monitoring state is inherited across typical process control flows including `fork()` or `clone()`. This mechanism prevents adversaries from bypassing controls by spawning new processes.

To identify unauthorized modifications in process privileges, the `lsm/cred_prepare` hook is employed. This hook executes whenever the kernel prepares new credentials for a process, commonly during system calls including `setuid()` or `setgid()`, or during execve transitions. The program retrieves again the current credentials from the kernel and compares them against the stored baseline for the monitored process. Any discrepancy indicates a potential privilege escalation attempt. In such instances, the operation is logged and blocked immediately by returning `-EPERM`, halting the escalation early.

Additional protection is delivered by the `lsm/inode_permission` hook, which is invoked before accessing an inode in the filesystem. This hook enables verification that the monitored process has not gained unauthorized privileges allowing it to perform file operations it was not permitted initially. It retrieves the current process credentials and compares them against the stored snapshot, denying permission if a mismatch is identified. This ensures that monitored processes cannot exploit privilege escalations to access sensitive filesystem resources.

The `lsm/task_kill` hook monitors signaling operations where one process sends a signal to another. Sending signals to another process generally requires

appropriate privileges, so the hook verifies whether a monitored BPF process attempting to send a signal maintains the original authorized credentials. Any unauthorized modification results in denial of the signal send operation. This prevents misuse of signaling to interfere with other processes or escalate privileges.

Finally, the `lsm/task_free` hook manages cleanup by removing entries from the `tracked_processes` map when a process terminates. This prevents resource leaks and maintains the monitoring map consistent with the current active process set.

In summary, this BPF LSM program implements a fail-secure design layered deeply into the kernel's security infrastructure. By selectively monitoring only BPF-related processes, it minimizes performance overhead while ensuring that any attempt to illegitimately modify credentials or misuse privileges is promptly identified and blocked. The pervasive logging via `bpf_printk()` facilitates auditing and forensic analysis of security-relevant events. This approach effectively addresses exploits similar to CVE-2022-23222, adding a strong, kernel-integrated defense layer to protect the system.

To validate the BPF LSM program, a test environment was established using a virtual machine configured with Ubuntu 20.04 and kernel version `5.13.0-27-generic`, which is vulnerable to CVE-2022-23222. The kernel was compiled with `CONFIG_BPF_LSM` enabled, and the `bpf` LSM was activated by adding it to the `lsm` kernel parameter at boot time.

The BPF LSM program was loaded using a custom C loader that invokes the `bpf` syscall to load the program and attach it to the appropriate LSM hooks. Upon executing the exploit, the BPF LSM program successfully detected and blocked the privilege escalation attempt in real time. The enforcement action was confirmed through log messages generated by `bpf_printk()`.

The complete C implementation of both the BPF LSM program and its loader, along with screenshots demonstrating the exploit being blocked and the corresponding kernel log entries, are provided in Appendix A.2.

6.1.4 Comparison and Evaluation of Mitigation Strategies

While upgrading the kernel to a patched release represents the most straightforward and effective solution to address CVE-2022-23222, it may not always be practical due to compatibility or stability concerns. In such instances, implementing the patch manually and recompiling the kernel represents a viable alternative, though it requires technical expertise, maintainability, and

sometimes downtime that may not be acceptable in production environments. Also disabling unprivileged BPF utilization represents a strong mitigation, but it can restrict the functionality of applications relying on eBPF, potentially impacting performance and features.

To provide supplementary security layers, runtime integrity verification tools including LKRG can be employed. LKRG attempts to identify and promptly respond to unauthorized modifications in the running Linux kernel and process credentials. Specifically, while it performs *post-detection* for kernel integrity violations (i.e., reacting after unauthorized changes occur), LKRG acts in a *proactive* manner when it comes to process credentials: it attempts to identify exploits and intervene before the kernel grants access based on unauthorized credentials (such as opening files).

Empirical studies, including the Master's Thesis of Juho Junnila [50], identify LKRG as the most effective rootkit detector among solutions evaluated. LKRG defeats many pre-existing exploits of Linux kernel vulnerabilities, and will likely defeat many future exploits (including of yet unknown vulnerabilities) that do not specifically attempt to bypass LKRG. While LKRG can be bypassed by design, such bypasses tend to require more complicated and less reliable exploits.

An additional relevant property of LKRG involves providing security through diversity, much like running an uncommon Operating System (OS) kernel would, yet without the usability drawbacks of actually running a less-supported operating system. Nevertheless, if LKRG becomes popular and begins to be deliberately bypassed by adversaries, developers may introduce new versions as a means both to increase diversity and to sustain the project, offering specialized features and potentially distribution-specific builds.

However, LKRG is not free from constraints. Like any additional kernel component, it may introduce bugs or vulnerabilities of its own. Being an out-of-tree module, it inevitably carries some risk of incompatibility with particular kernel builds. Although the developers test LKRG against a wide range of kernel releases, changes in future versions may require extra maintenance effort. This creates a trade-off: administrators need to weigh the benefit of additional protection against the potential for instability and higher operational overhead, especially in contexts where live patching or scheduled reboots are challenging. Finally, as an experimental project, LKRG still suffers from occasional false positives, particularly on kernel versions or configurations not explicitly targeted during testing.

As a complementary approach, BPF LSM programs deliver a highly flexible framework for implementing security policies within the kernel. Leveraging

eBPF, these modules enable administrators to monitor and control fine-grained operations in a proactive fashion, intervening before the corresponding actions are executed. The CO-RE (Compile Once – Run Everywhere) approach further minimizes maintenance complexity by enabling the same code to operate across multiple kernel versions without modification. Since eBPF represents an established component of the Linux kernel, its long-term availability is practically guaranteed, making this approach sustainable.

However, the effectiveness of BPF LSM depends critically on the completeness and frequent updating of the underlying policy. For instance, the current implementation effectively blocks privilege escalation attempts but still fails to prevent socket creation or the loading of additional eBPF programs unless explicitly addressed by new policy hooks. Moreover, BPF LSM requires a kernel compiled with `CONFIG_BPF_LSM` support and the activation of the `bpf` option in the `lsm` kernel parameter at boot time. Availability is also limited to Linux 5.7 and newer. In this particular scenario, the affected kernel versions (all greater than 5.8) are compatible with BPF LSM, yet this requirement should be considered for the mitigation of future vulnerabilities.

In summary, LKRG and BPF LSM represent two complementary approaches: while LKRG offers broad and generic protection against entire categories of exploits, even unforeseen ones, at the cost of potential instability and compatibility issues, BPF LSM delivers a highly customizable and policy-based mechanism that enables fine-grained control but whose effectiveness ultimately depends on the completeness and continuous maintenance of the defined policies. A combined deployment of both solutions can therefore deliver a more comprehensive defense strategy, achieving the principle of security in depth by layering generic exploit mitigation with tailored, policy-driven protections.

6.2 Use Case 2 – Kernel Panic and Denial of Service

6.2.1 Vulnerability Description

The second case study focuses on two distinct security flaws: CVE-2024-56614 and CVE-2024-56615. These affect AF_XDP (Address Family eXpress Data Path) socket implementation alongside DEVMAP (device maps) functionality.

Specifically, the `xsk_map_delete_elem` function within AF_XDP socket maps (XSK maps) suffers from CVE-2024-56614, while CVE-2024-56615 targets the `dev_map_delete_elem` function in device maps. Identical root causes plague both vulnerabilities: implicit type conversion problems occur when user-controlled signed integer indices face comparison with unsigned integers representing maximum `map_entries`. Negative indices get misinterpreted as large positive values through this flaw, causing out-of-bounds memory access that potentially triggers denial of service and kernel panics.

Both flaws earned CVSS v3.1 base scores of 7.8 (HIGH severity classification), yet they impact different Linux kernel versions:

- CVE-2024-56614:
  - From (including) 4.18 up to (excluding) 5.15.174
  - From (including) 5.16 up to (excluding) 6.1.120
  - From (including) 6.2 up to (excluding) 6.6.66
  - From (including) 6.7 up to (excluding) 6.12.5
- CVE-2024-56615:
  - From (including) 4.14 up to (excluding) 5.4.287
  - From (including) 5.5 up to (excluding) 5.10.231
  - From (including) 5.11 up to (excluding) 5.15.174
  - From (including) 5.16 up to (excluding) 6.1.120
  - From (including) 6.2 up to (excluding) 6.6.66
  - From (including) 6.7 up to (excluding) 6.12.5

6.2.2 Exploitation Process

The exploitation of CVE-2024-56614 and CVE-2024-56615 involves crafting eBPF programs that interact with the vulnerable map types (XSK maps for CVE-2024-56614 and DEVMAPs for CVE-2024-56615) in a way that triggers the out-of-bounds write condition. Both flawed implementations of `xsk_map_delete_elem` and `dev_map_delete_elem` share the same root cause, which can be illustrated through the vulnerable code snippet for the DEVMAP implementation:

```
1  static int dev_map_delete_elem(struct bpf_map *map, void *key
       )
2  {
3      struct bpf_dtab *dtab = container_of(map, struct bpf_dtab
       , map);
4      struct bpf_dtab_netdev *old_dev;
5      int k = *(u32 *)key;  // Unsigned-to-signed assignment
6
7      if (k >= map->max_entries)  // Vulnerable comparison
8          return -EINVAL;
9
10     old_dev = unrcu_pointer(xchg(&dtab->netdev_map[k], NULL))
       ;
11     if (old_dev) {
12         call_rcu(&old_dev->rcu, __dev_map_entry_free);
13         atomic_dec((atomic_t *)&dtab->items);
14     }
15     return 0;
16 }
```

Listing 6.1. Defective dev_map_delete_elem function, Linux kernel v5.13 [1]

This signed/unsigned integer confusion manifests through multi-stage exploitation where user-supplied map keys start as 32-bit unsigned integers (`u32 *key`), then get assigned to signed integer variable `k` via `int k = *(u32 *)key`. Values exceeding `0x80000000` (2,147,483,648) become negative in the signed `k` variable due to two's complement representation. The subsequent bounds checking `if (k >= map->max_entries)` becomes vulnerable through C language implicit type conversion rules, where signed-to-unsigned comparison requires promoting signed values to unsigned before comparison. Consequently, negative `k` values convert to large positive values during comparison, with a `k` value of `-1` becoming `0xFFFFFFFF` (4,294,967,295) when promoted, potentially passing or failing bounds checks depending on `max_entries` configuration.

Successfully bypassing bounds checking allows negative `k` values as direct

array indices in `dtab->netdev_map[k]`, accessing memory locations preceding the allocated array in kernel memory space. This creates powerful primitives for arbitrary kernel memory access at predictable negative offsets from map base addresses, enabling writes to kernel data structures located before map arrays in memory.

Attackers orchestrate exploitation through carefully sequenced operations, initially creating DEVMAP or XSK maps with `max_entries` values strategically exceeding `0x80000000` to ensure certain negative indices bypass bounds checking while remaining valid during unsigned comparison phases. Malicious keys within range `[0x80000000, 0x80000000 + (max_entries - 0x80000000)]` are selected, which become negative values when assigned to signed integer `k`, yet pass unsigned comparisons through integer promotion rules. BPF syscalls with `BPF_MAP_DELETE_ELEM` commands then trigger vulnerable delete functions using these carefully constructed keys, with subsequent `xchg` operations accessing memory at negative offsets from map arrays and modifying critical kernel data structures located in those memory regions.

Publicly available proof-of-concept exploits demonstrate both CVE-2024-56614 and CVE-2024-56615, proving practical feasibility of triggering out-of-bounds access conditions [51, 52]. These create XSKMAP/DEVMAP with `max_entries` set to `0x80000000 + 2`, using malicious key values of `0x80000000 + 1` which, when interpreted as signed integers, translates to `-2147483647`, successfully bypassing bounds checking and triggering out-of-bounds memory access.

Exploitation typically requires `CAP_NET_ADMIN` capabilities, granting privileges for creating and managing BPF maps related to network devices. This capability is required instead of `CAP_BPF` because these types are classified as networking resources that directly interact with network device structures (`struct net_device`) and control packet redirection between interfaces, thus falling under network administration privileges rather than general BPF program management.

Despite providing out-of-bounds memory access primitives, practical exploitation faces significant constraints as attackers cannot directly control kernel objects residing at negative offsets from map arrays. Reliable exploitation heavily depends on kernel memory layouts, SLAB allocator behavior, and Address Space Layout Randomization (ASLR) configurations, where unpredictable kernel memory organization means most exploitation attempts result in kernel panics rather than controlled modifications. Sophisticated attackers possessing deep kernel internals knowledge might achieve arbitrary code execution by targeting specific kernel data structures like credential

objects or function pointers located in accessible memory regions, though such exploitation requires extensive reconnaissance and memory layout analysis. More commonly, after successfully triggering out-of-bounds writes, kernel panics leading to Denial of Service (DoS) conditions represent the most frequent outcome, as achieved and tested under kernel version `5.13.0-27 generic`.

Another significant limitation is that target systems require at least 18 gigabytes of Random Access Memory (RAM) for creating vulnerable maps with sufficiently large `max_entries` values, potentially limiting attack applicability in resource-constrained environments. In the Appendix B.1, the execution of the exploit is demonstrated with screenshots, showcasing the successful privilege escalation to root.

6.2.3 Hardening Plan

Multiple approaches exist for addressing vulnerabilities CVE-2024-56614 and CVE-2024-56615, each offering different levels of system protection enhancement.

Like the previous case, kernel updates remain the most direct solution. These particular flaws got patches in versions 5.15.174, 5.4.287, 5.10.231, 6.1.120, 6.6.66, and 6.12.5. Systems should be upgraded to these versions at minimum. When upgrades aren't possible, manual patch application with kernel recompilation works as backup.

Unlike the first use case, restricting eBPF access through the `kernel.unprivileged_bpf_disabled` sysctl parameter provides no protection against these vulnerabilities. This parameter only controls BPF access for unprivileged users, but both CVE-2024-56614 and CVE-2024-56615 exploitation requires `CAP_NET_ADMIN` capabilities for creating XSKMAP and DEVMAP respectively. Since `unprivileged_bpf_disabled` does not restrict operations performed by users possessing `CAP_NET_ADMIN`, attackers with the necessary privileges can exploit these vulnerabilities regardless of this setting.

When basic measures prove insufficient, advanced security mechanisms become necessary. LKRG focuses on kernel structure modification detection but doesn't specifically target these CVE types. Tetragon policies offer a more targeted approach instead. Hook selection matters enormously for effective vulnerability detection, requiring careful attack surface analysis for these specific CVEs. The designed policy targets three strategic hook points providing comprehensive vulnerable code path coverage.

Map allocation functions `xsk_map_alloc` and `dev_map_alloc` form the first hook set. These create XSK and DEVMAP maps respectively and

represent critical chokepoints since attackers must create vulnerable map types with specific configurations for exploitation success. Monitoring these allocation functions logs all XSK and DEVMAP creation events, providing visibility into potentially vulnerable map type allocations.

`dev_map_alloc` and `xsk_map_alloc` selection proves strategically sound because these functions trigger exclusively during vulnerable map type creation. This ensures high specificity with minimal false positives. Unlike generic BPF allocation functions called for various map types, these specialized allocators provide precise attack vector targeting.

The third hook attaches to generic `sys_bpf` system calls, recording every `BPF_MAP_DELETE_ELEM` command invocation. This hook captures exact delete element operation execution moments, corresponding to vulnerable `dev_map_delete_elem` and `xsk_map_delete_elem` function entry.

These three hooks form comprehensive monitoring frameworks spanning complete attack lifecycles from initial map creation through exploitation attempts. Allocation hooks provide early warnings about potentially vulnerable configurations. Syscall hooks detect actual exploitation attempts. However, significant filtering capability limitations exist, as documented in the following section. Consequently, more sophisticated context-aware hardening solutions become necessary for effectively mitigating these vulnerability risks. BPF LSM programs represent the preferred approach and core thesis contribution, providing finer-grained context-sensitive security enforcement.

BPF LSM Program Implementation

The BPF LSM program combines LSM hooks and tracepoints to monitor and enforce policy on DEVMAP and XSKMAP operations. The base idea of the program is to track only the vulnerable maps, that is eBPF maps of type `BPF_MAP_TYPE_XSKMAP` or `BPF_MAP_TYPE_DEVMAP` created with `max_entries` $\geq 0x80000000$. Then, if a delete element is called on one of the suspicious maps, it is a good indicator of potential exploitation, and so the program logs and blocks the operation. To achieve this, two BPF hash maps are required: `suspicious_map_fds` tracks map file descriptors (FDs) with their related large `max_entries` value, and `pending_map_creation` temporarily holds creation parameters (`max_entries`, timestamp and map type) with their associated process ids (PIDs).

First of all, the `lsm/bpf` hook is used. This hook is invoked at the start of every `bpf(2)` syscall, before any kernel-side allocation or setup is performed. When a `BPF_MAP_CREATE` command is detected in the syscall arguments, the

hook retrieves the map type and `max_entries` fields from the `bpf_attr` union using CO-RE helper macros. If `max_entries` exceeds `0x80000000`, an alert is logged, indicating a suspicious map that will require later monitoring. The hook then records the creation parameters in `pending_map_creation` under the current PID so they can be correlated once the syscall returns.

Because the map FD is not available until the syscall completes, a tracepoint on `sys_exit_bpf`, the kernel tracepoint for `syscalls:sys_exit_bpf`, is employed to perform the correlation. This tracepoint fires when the `bpf(2)` syscall returns from kernel space back to user space. The handler checks if the return value is a nonnegative FD (indicating an error), then looks up the `pending_map_creation` map for the current PID. If an entry is found, the program updates `suspicious_map_fds` map with the new FD and its `max_entries`.

Once suspicious map FDs are registered, enforcement occurs via the same `lsm/bpf` LSM hook. When the `BPF_MAP_DELETE_ELEM` command is issued, the hook is again invoked at syscall entry. At this point, the hook reads the map FD and the user-space pointer to the key. It then uses an eBPF probe-read helper to copy the key into kernel space. If the key is $\geq 0x80000000$ and the FD exists in `suspicious_map_fds`, the hook computes the precise exploitable key range from the stored `max_entries`. Keys within this range result in an immediate return of `-EPERM`, blocking the operation, instead keys outside the range are allowed to proceed.

To handle lifecycle cleanup, an `lsm/task_free` hook on the LSM hook point for task termination is installed. This hook triggers whenever a process is freed, ensuring that any pending creation records keyed by PID are deleted. In parallel, a tracepoint on `sys_enter_close` fires at syscall entry for the `close(2)` syscall. This handler checks if the FD being closed exists in `suspicious_map_fds` and removes it if present, preventing stale entries after file descriptors are released.

All hooks leverage CO-RE for safe structure field reading across kernel versions, emitting diagnostic messages via `bpf_printk()` for real-time auditing. This multi-stage instrumentation yields precise, context-aware enforcement mechanisms that only deny operations when signed/unsigned confusion exploit conditions truly exist.

The complete C implementation of both the BPF LSM program and its loader, along with screenshots demonstrating the exploit being blocked and the corresponding kernel log entries, are provided in Appendix B.2.

6.2.4 Comparison and Evaluation of Mitigation Strategies

Basic mitigation approaches for CVE-2024-56614 and CVE-2024-56615 closely resemble CVE-2022-23222 proposals, with primary recommendations focusing on kernel updates incorporating necessary patches or direct patch application. Remember that kernel updating isn't always practical due to compatibility concerns, system stability issues, or operational downtime, and manual patching demands advanced technical skills plus careful management.

This use case benefits from Tetragon alternative, a tool offering eBPF-based observability through declarative Yet Another Markup Language (YAML) policy languages. Compared to BPF LSM programs, Tetragon policy development proves generally more accessible, requiring less deep kernel knowledge. Moreover, Tetragon supports kernels from version 4.19 onward, covering large portions of deployed systems. Some vulnerable kernel versions remain uncovered though, like version 4.18. However, utility in this specific context stays limited. Tetragon logs generic creation events for `BPF_MAP_TYPE_XSKMAP` and `BPF_MAP_TYPE_DEVMAP` maps but lacks crucial parameter filtering abilities like `max_entries`. Similarly, it records all `BPF_MAP_DELETE_ELEM` command invocations but can't filter based on map type, deletion keys or associated map file descriptors.

These limitations stem from Tetragon's inability to dereference pointers or inspect complex data structures like unions or nested structs composing BPF syscall arguments. It only filters primitive argument types such as integers or strings, as happens with command arguments in `bpf(2)` syscalls. This produces high logged data volumes including numerous false positives potentially overwhelming monitoring systems, obscuring actual incidents, and prohibiting rapid incident response without extensive manual analysis or external tool integration.

BPF LSM program deployment addresses these shortcomings. This solution offers significantly more granular context-sensitive enforcement mechanisms by integrating both LSM hooks and kernel tracepoints. It enables targeted monitoring of kernel functions and syscalls associated with DEVMAP and XSKMAP object lifecycles. Thanks to CO-RE, programs maintain compatibility across various kernel versions, simplifying maintenance. Crucially, BPF LSM approaches allow immediate intervention by blocking suspicious operations real-time, rather than restricting themselves to passive logging. This proactive enforcement substantially improves system security postures.

Nevertheless, recognize that BPF LSM usage requires kernels compiled with

`CONFIG_BPF_LSM` enabled and `bpf` options set in kernel LSM boot parameters. Additionally, BPF LSM availability starts with Linux kernel version 5.7. Consequently, several vulnerable kernels, particularly those older than 5.7, remain unsupported by this mitigation strategy.

Chapter 7

Conclusions and Future Works

This research explored the security landscape of eBPF through comprehensive vulnerability analysis and the development of practical defensive mechanisms. Two representative vulnerabilities exemplified distinct attack vectors within the eBPF ecosystem: CVE-2022-23222 exploits verifier logic flaws to manipulate credential structures, achieving complete privilege escalation from unprivileged contexts, while CVE-2024-56614/56615 leverage integer overflow in map operations to trigger kernel panics through out-of-bounds memory access. The examination of these vulnerabilities in Chapter 6 underscored the necessity for context-aware, vulnerability-specific hardening approaches rather than generic security solutions.

The comparative evaluation of defensive strategies revealed distinct trade-offs between protection effectiveness, operational feasibility, and implementation complexity. Kernel patches provide definitive remediation but require system downtime and face deployment challenges in production environments where continuous availability is critical. LKRG offers broad kernel integrity protection yet operates as an out-of-tree module with potential compatibility limitations and may not specifically target eBPF-centric attack vectors with sufficient granularity. Tetragon presents an accessible YAML-based monitoring framework with observability capabilities, though its enforcement mechanisms suffer from limited filtering precision and elevated false positive rates due to constraints in inspecting complex kernel data structures. In contrast, BPF LSM implementations demonstrated superior precision through programmatic access to kernel internals, enabling context-aware filtering that distinguishes malicious exploitation from legitimate system operations, as

demonstrated in Appendix **??**. However, this precision comes at the cost of increased technical complexity, requiring kernel-specific configuration and deeper eBPF programming expertise.

Future research should investigate the performance implications and overhead characteristics of deploying these hardening solutions, particularly when multiple concurrent eBPF-based security mechanisms coexist within a single system. Understanding the interaction dynamics between layered eBPF security tools, their cumulative resource consumption, and potential conflicts or synergies would inform best practices for designing defense-in-depth architectures.

Based on the findings emerged from the comprehensive analysis and research conducted with the fellow university colleague, different best practices emerge for securing eBPF-enabled environments. Organizations should implement comprehensive verification procedures for every loaded eBPF program, ensuring programs are trusted and limiting their access to only necessary maps and resources. Proper management of Linux capabilities, particularly `CAP_BPF`, represents a fundamental security control that should be strictly enforced through principle of least privilege, granting eBPF permissions only to processes that explicitly require them. Given the security risks demonstrated in the analyzed vulnerabilities, disabling unprivileged eBPF by design (sysctl parameter `kernel.unprivileged_bpf_disabled=1`) should be considered standard practice in production environments, as unprivileged access significantly expands the attack surface by allowing non-root users to load potentially malicious programs. Continuous monitoring of eBPF map creation and usage patterns enables detection of anomalous behavior, including unusually large map configurations or suspicious key ranges that could facilitate exploitation. Deploying solutions that track credential identifiers of processes utilizing eBPF capabilities provides essential visibility for detecting privilege escalation attempts across various vulnerability categories, with particular attention to capability transitions that might indicate unauthorized privilege acquisition. Finally, staying informed about emerging CVEs and evolving attack methods, particularly those threats that new Linux kernel features might introduce, since this knowledge remains fundamental for proactive security management.

These patterns provide a foundation for developing proactive detection mechanisms that can identify suspicious eBPF operations before exploitation occurs. This research has identified promising directions for future work. One particularly compelling approach involves analyzing eBPF program instructions at load time using kprobe-based LSM BPF programs to identify known

attack patterns or track register values for anomalous behavior. This pre-execution analysis could provide an additional security layer by intercepting malicious programs before they gain kernel-level access.

```
<...>-77036  [001] d...  6201.385613: bpf_trace_printk: BPF syscall cmd=5, size=120
<...>-77036  [001] d...  6201.385628: bpf_trace_printk:  insn_cnt=37, insn_ptr=000000000045ce1

<...>-77036  [001] d...  6201.385629: bpf_trace_printk: Total read: 37 insns

<...>-77036  [001] d...  6201.385629: bpf_trace_printk: Chunks read: 2

<...>-77036  [001] d...  6201.385630: bpf_trace_printk: Insn: code=b7, dst_reg=0, src_reg=0
<...>-77036  [001] d...  6201.385630: bpf_trace_printk: imm=0, offset=0

<...>-77036  [001] d...  6201.385630: bpf_trace_printk: Insn: code=63, dst_reg=10, src_reg=0
<...>-77036  [001] d...  6201.385630: bpf_trace_printk: imm=0, offset=-4

<...>-77036  [001] d...  6201.385631: bpf_trace_printk: Insn: code=bf, dst_reg=2, src_reg=10
<...>-77036  [001] d...  6201.385631: bpf_trace_printk: imm=0, offset=0

<...>-77036  [001] d...  6201.385631: bpf_trace_printk: Insn: code=7, dst_reg=2, src_reg=0
<...>-77036  [001] d...  6201.385631: bpf_trace_printk: imm=-4, offset=0

<...>-77036  [001] d...  6201.385632: bpf_trace_printk: Insn: code=18, dst_reg=1, src_reg=1
<...>-77036  [001] d...  6201.385632: bpf_trace_printk: imm=3, offset=0

<...>-77036  [001] d...  6201.385632: bpf_trace_printk: Insn: code=0, dst_reg=0, src_reg=0
<...>-77036  [001] d...  6201.385632: bpf_trace_printk: imm=0, offset=0

<...>-77036  [001] d...  6201.385632: bpf_trace_printk: Insn: code=85, dst_reg=0, src_reg=0
<...>-77036  [001] d...  6201.385632: bpf_trace_printk: imm=1, offset=0

<...>-77036  [001] d...  6201.385633: bpf_trace_printk: Insn: code=55, dst_reg=0, src_reg=0
<...>-77036  [001] d...  6201.385633: bpf_trace_printk: imm=0, offset=1

<...>-77036  [001] d...  6201.385633: bpf_trace_printk: Insn: code=95, dst_reg=0, src_reg=0
```

```
33 <...>-77036  [001] d...  6201.385633: bpf_trace_printk: imm=0, offset=0
```

Listing 7.1. Real-time eBPF instruction analysis

The listing 7.1 demonstrates a prototype implementation of this concept, showing real-time eBPF instruction analysis captured via `bpf_trace_printk`. The output displays detailed instruction-level information including operation codes, register assignments, immediate values, and memory offsets for each eBPF instruction as it is processed during program loading. This granular visibility into program structure enables pattern matching against known exploit signatures, detection of suspicious register manipulation sequences, and identification of potentially malicious memory access patterns that could indicate privilege escalation or out-of-bounds operations.

Such an approach would complement existing verifier checks, offering the possibility of implementing policy-based program inspection that goes beyond traditional safety verification.

Extending this approach further, ongoing artificial intelligence technology evolution creates substantial opportunities for enhancing these eBPF security analysis capabilities. AI-assisted analysis might significantly improve instruction-level program inspection accuracy and scope, potentially spotting novel attack patterns and zero-day exploitation techniques through behavioral analysis and pattern recognition approaches. Machine learning models trained on known eBPF vulnerabilities and attack signatures could deliver real-time threat assessment capabilities, enabling dynamic security policies that adapt to emerging threats. This technological advancement represents a natural evolution from the static pattern matching described above toward more sophisticated, adaptive security mechanisms.

This research shows that while eBPF vulnerabilities create substantial security challenges, focused hardening strategies can effectively counter these threats when properly designed and deployed. The combination of systematic vulnerability analysis, practical defensive implementations, and collaborative research approaches delivers solid foundations for advancing eBPF security research and establishing more secure kernel programmability frameworks for future systems.

Appendix A

Use Case 1 – Privilege Escalation

A.1 Exploit Execution

The exploit for CVE-2022-23222 was executed in a controlled environment to validate its privilege escalation capabilities. As described in Section 6.1, the exploit implementation includes enhanced functionalities beyond the original proof-of-concept, enabling multiple attack variants targeting different credential manipulation strategies.

The following screenshots demonstrate the exploit's effectiveness when executed by a non-privileged user. Figure A.1 shows the successful escalation to a root shell, with all credential identifiers modified to zero, granting unrestricted system access. Figure A.2 demonstrates an alternative attack variant that modifies only the filesystem credential identifiers (FSUID, FSGID), enabling unauthorized access to privileged files such as `/etc/shadow` while maintaining the appearance of running as a regular user. These demonstrations confirm that the vulnerability can be reliably exploited to achieve complete privilege escalation from an unprivileged user context.

A.2 BPF LSM Program Implementation

This appendix presents the complete implementation of the BPF LSM-based hardening solution for CVE-2022-23222, consisting of three components: the kernel-space BPF program that enforces security policies, the userspace loader responsible for program deployment, and the automated build script that

```
vc@ubuntu2004:~/sh/usecase3/CVE_2022_23222_poc$ ./exploit 1
[+] Starting exploit in mode 1
[+] eBPF enabled, ringbuf created!
[!] staring to create new maps until we get two consecutive maps
[+] created map 1
[+] generated random value: 3830d16dc100f4f5
[+] created map 2
[+] generated random value: 2560aea7f6f615ab
[+] two new maps created!
[+] value read from slab: 500000500
[+] value read from slab: 67f969a4
[+] created map 3
[+] generated random value: f327563fdc571fcc
[+] created map 4
[+] generated random value: 7511d15b2f82c1c7
[+] eBPF enabled, maps created!
 [+] value read from slab: 7511d15b2f82c1c7
 [+] aligned map found in map 3
 1: 998c723d98aa8268
 2: 7511d15b2f82c1c7
[+] closing unnecessary_maps
[+] found map address: 0xffff888105e4b400
[+] overriding map_ops
[+] detected kernel slide 0
[+] setting spin_lock = 0
[+] setting max_entries = 0xffffffff
[+] setting map_type = BPF_MAP_TYPE_STACK
[+] getting root
[+] iterating over task_struct list to find out process
[+] got it!
[+] Mode 1: Full privilege escalation
[+] setting uid to 0x0 at 0xffff8881031d9244
[+] setting gid to 0x0 at 0xffff8881031d9248
[+] setting suid to 0x0 at 0xffff8881031d924c
[+] setting sgid to 0x0 at 0xffff8881031d9250
[+] setting euid to 0x0 at 0xffff8881031d9254
[+] setting egid to 0x0 at 0xffff8881031d9258
[+] setting fsuid to 0x0 at 0xffff8881031d925c
[+] setting fsgid to 0x0 at 0xffff8881031d9260
[+] Full privilege escalation successful!
[+] uid=0 euid=0 gid=0 egid=0
[+] cleaning up
[+] getting shell!
root@ubuntu2004:~/sh/usecase3/CVE_2022_23222_poc# whoami
root
root@ubuntu2004:~/sh/usecase3/CVE_2022_23222_poc# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),116(lxd),1000(vc)
```

Figure A.1. Exploit execution resulting in a root shell.

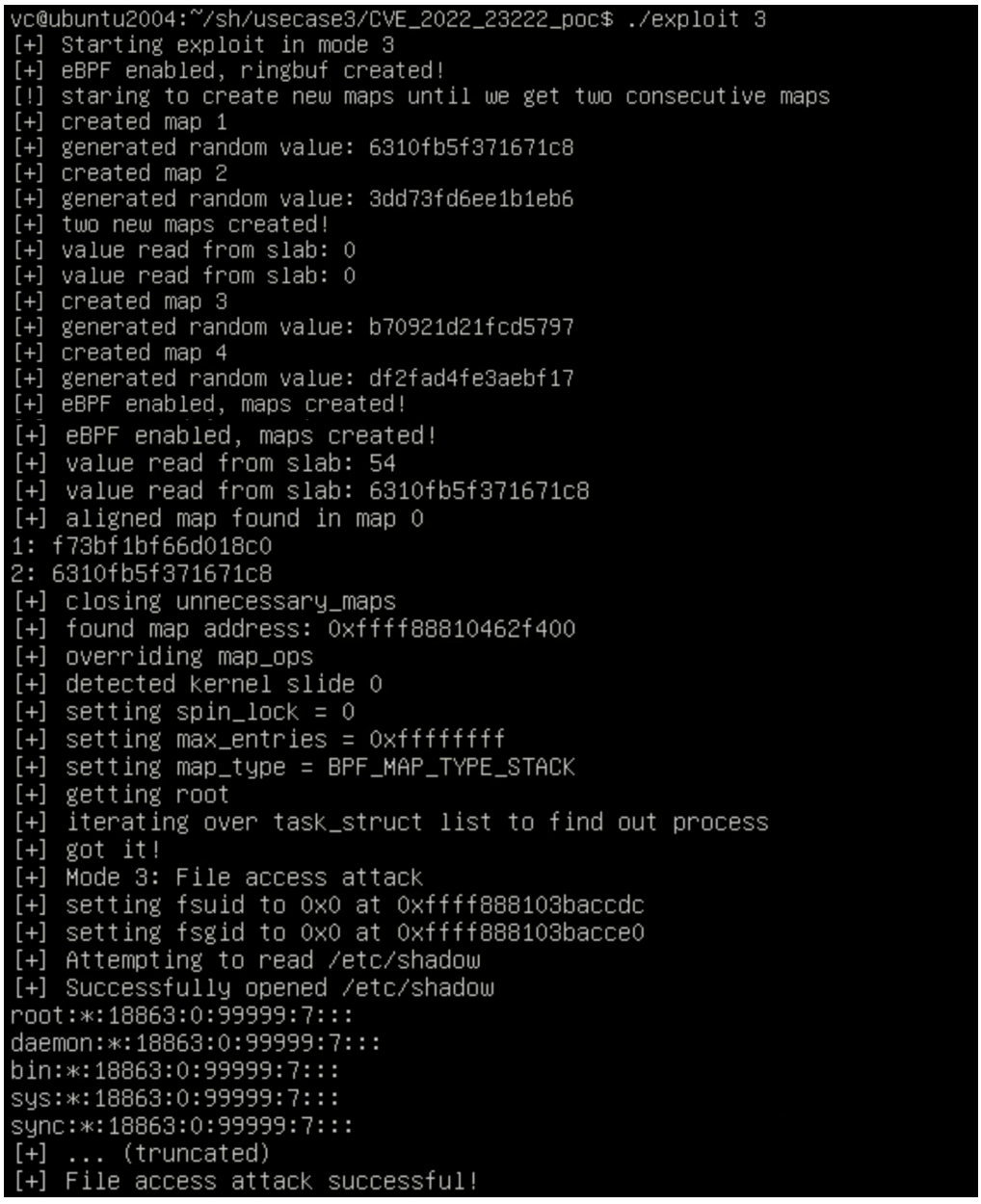

Figure A.2. Exploit execution resulting in reading the `/etc/shadow` file.

orchestrates the compilation and execution workflow.

A.2.1 Kernel-Space BPF LSM Program

The BPF LSM program implements the core security enforcement logic by attaching to multiple LSM hooks within the Linux kernel.

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

char LICENSE[] SEC("license") = "GPL";

#define EPERM 1
#define MAX_TRACKED_PROCESSES 1024

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, MAX_TRACKED_PROCESSES);
    __type(key, __u32);
    __type(value, struct tracked_entry);
} tracked_processes SEC(".maps");

struct tracked_entry {
    __u32 uid;
    __u32 gid;
    __u32 euid;
    __u32 fsuid;
    __u32 fsgid;
    __u32 suid;
    __u32 sgid;
};

/* Hook for BPF syscalls to track processes using BPF */
SEC("lsm/bpf")
int BPF_PROG(bpf_syscall, int cmd, union bpf_attr *attr, unsigned int size)
{
    pid_t pid = bpf_get_current_pid_tgid() >> 32;

    // Track any process that uses BPF syscalls
    struct tracked_entry *entry = bpf_map_lookup_elem(&tracked_processes, &pid);
    if (entry) {
        // Already tracking this process
        return 0;
    }

    // Get current task credentials
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    if (!task)
        return 0; // Don't block, just don't track

    const struct cred *cred = NULL;
    bpf_core_read(&cred, sizeof(cred), &task->cred);
    if (!cred)
        return 0; // Don't block, just don't track

    __u32 uid_val = 0, euid_val = 0, gid_val = 0, fsuid_val = 0, fsgid_val = 0, suid_val
    =0, sgid_val =0;
    bpf_core_read(&uid_val, sizeof(uid_val), &cred->uid.val);
    bpf_core_read(&euid_val, sizeof(euid_val), &cred->euid.val);
    bpf_core_read(&gid_val, sizeof(gid_val), &cred->gid.val);
```

```
55      bpf_core_read(&fsuid_val, sizeof(fsuid_val), &cred->fsuid.val);
56      bpf_core_read(&fsgid_val, sizeof(fsgid_val), &cred->fsgid.val);
57      bpf_core_read(&suid_val, sizeof(suid_val), &cred->suid.val);
58      bpf_core_read(&sgid_val, sizeof(sgid_val), &cred->sgid.val);
59
60      struct tracked_entry new_entry = {
61          .uid = uid_val,
62          .euid = euid_val,
63          .gid = gid_val,
64          .fsuid = fsuid_val,
65          .fsgid = fsgid_val,
66          .suid = suid_val,
67          .sgid = sgid_val
68      };
69
70      bpf_printk("BPF syscall tracking process %d (cmd=%d)\n", pid, cmd);
71      bpf_printk("Initial credentials uid=%d, euid=%d, gid=%d\n", uid_val, euid_val,
        gid_val);
72      bpf_printk("Initial credentials fsuid=%d, fsgid=%d\n", fsuid_val, fsgid_val);
73
74      int ret = bpf_map_update_elem(&tracked_processes, &pid, &new_entry, BPF_ANY);
75      if (ret < 0) {
76          bpf_printk("Failed to track BPF process %d\n", pid);
77      } else {
78          bpf_printk("Successfully tracking BPF process %d\n", pid);
79      }
80
81      return 0;
82  }
83
84  SEC("lsm/task_alloc")
85  int BPF_PROG(task_alloc, struct task_struct *task, unsigned long clone_flags)
86  {
87      pid_t parent_pid = bpf_get_current_pid_tgid() >> 32;
88      pid_t child_pid;
89
90      // Read the child process PID
91      int ret = bpf_probe_read_kernel(&child_pid, sizeof(child_pid), &task->pid);
92      if (ret < 0) {
93          return 0; // Don't block on read failure
94      }
95
96      // Check if parent is tracked
97      struct tracked_entry *parent_entry = bpf_map_lookup_elem(&tracked_processes, &
        parent_pid);
98      if (parent_entry) {
99          // Inherit tracking from parent
100         struct tracked_entry child_entry = {
101             .uid = parent_entry->uid,
102             .euid = parent_entry->euid,
103             .gid = parent_entry->gid,
104             .fsuid = parent_entry->fsuid,
105             .fsgid = parent_entry->fsgid,
106             .suid = parent_entry->suid,
107             .sgid = parent_entry->sgid
108         };
109
110         ret = bpf_map_update_elem(&tracked_processes, &child_pid, &child_entry, BPF_ANY);
111         if (ret == 0) {
112             bpf_printk("Inherited tracking: child %d from parent %d\n", child_pid,
        parent_pid);
113         }
114     }
115
```

```
116     return 0;
117 }
118
119
120 SEC("lsm/cred_prepare")
121 int BPF_PROG(cred_prepare, struct cred *new, const struct cred *old, gfp_t gfp)
122 {
123     pid_t pid = bpf_get_current_pid_tgid() >> 32;
124     // Check if this process is tracked (BPF-related)
125     struct tracked_entry *entry = bpf_map_lookup_elem(&tracked_processes, &pid);
126     if (!entry) {
127         // Not tracked - allow privilege escalation (sudo, etc.)
128         return 0;
129     }
130
131     // Read new credentials
132     __u32 new_uid = 0, new_euid = 0, new_gid = 0, new_fsuid = 0, new_fsgid = 0, new_sgid
    =0, new_suid = 0;
133     bpf_core_read(&new_uid, sizeof(new_uid), &new->uid.val);
134     bpf_core_read(&new_euid, sizeof(new_euid), &new->euid.val);
135     bpf_core_read(&new_gid, sizeof(new_gid), &new->gid.val);
136     bpf_core_read(&new_fsuid, sizeof(new_fsuid), &new->fsuid.val);
137     bpf_core_read(&new_fsgid, sizeof(new_fsgid), &new->fsgid.val);
138     bpf_core_read(&new_suid, sizeof(new_suid), &new->suid.val);
139     bpf_core_read(&new_sgid, sizeof(new_sgid), &new->sgid.val);
140
141     bpf_printk("LSM cred_prepare: tracked BPF pid=%d\n", pid);
142     bpf_printk("Expected uid=%d, euid=%d, gid=%d\n", entry->uid, entry->euid, entry->gid)
    ;
143     bpf_printk("Expected fsuid=%d, fsgid=%d\n", entry->fsuid, entry->fsgid);
144     bpf_printk("New cred uid=%d, euid=%d, gid=%d\n", new_uid, new_euid, new_gid);
145     bpf_printk("New cred fsuid=%d, fsgid=%d\n", new_fsuid, new_fsgid);
146
147
148     // Only block privilege escalation to root for BPF-related processes
149     if ((new_uid != entry->uid ) ||
150         (new_euid != entry->euid) ||
151         (new_gid != entry->gid) ||
152         (new_fsuid != entry->fsuid ) ||
153         (new_fsgid != entry->fsgid ) ||
154         (new_sgid != entry->sgid ) ||
155         (new_suid != entry->suid )
156     ){
157         bpf_printk("[!LSM ALERT] BPF process privilege escalation in cred_prepare!\n");
158         bpf_printk("[!LSM ALERT] UID: %d->%d\n", entry->uid, new_uid);
159         bpf_printk("[!LSM ALERT] EUID: %d->%d\n", entry->euid, new_euid);
160         bpf_printk("[!LSM ALERT] GID: %d->%d\n", entry->gid, new_gid);
161         bpf_printk("[!LSM ALERT] FSUID: %d->%d\n", entry->fsuid, new_fsuid);
162         bpf_printk("[!LSM ALERT] FSGID: %d->%d\n", entry->fsgid, new_fsgid);
163         return -EPERM;
164     }
165
166     return 0;
167 }
168
169 SEC("lsm/inode_permission")
170 int BPF_PROG(inode_permission, struct inode *inode, int mask){
171
172     pid_t pid = bpf_get_current_pid_tgid() >> 32;
173     // Check if this process is tracked (BPF-related)
174     struct tracked_entry *entry = bpf_map_lookup_elem(&tracked_processes, &pid);
175     if (!entry) {
176         // Not tracked - allow legitimate privilege escalation (sudo, etc.)
177         return 0;
```

```
178     }
179
180
181     struct task_struct *task = (struct task_struct *)bpf_get_current_task();
182
183     const struct cred *cred = NULL;
184     bpf_core_read(&cred, sizeof(cred), &task->cred);
185     if (!cred)
186         return -EPERM;
187
188     __u32 new_uid = 0, new_euid = 0, new_gid = 0, new_fsuid = 0, new_fsgid = 0, new_suid
        =0, new_sgid=0;
189     bpf_core_read(&new_uid, sizeof(new_uid), &cred->uid.val);
190     bpf_core_read(&new_euid, sizeof(new_euid), &cred->euid.val);
191     bpf_core_read(&new_gid, sizeof(new_gid), &cred->gid.val);
192     bpf_core_read(&new_fsuid, sizeof(new_fsuid), &cred->fsuid.val);
193     bpf_core_read(&new_fsgid, sizeof(new_fsgid), &cred->fsgid.val);
194     bpf_core_read(&new_suid, sizeof(new_suid), &cred->suid.val);
195     bpf_core_read(&new_sgid, sizeof(new_sgid), &cred->sgid.val);
196
197     bpf_printk("LSM inode_permission: tracked BPF pid=%d\n", pid);
198     bpf_printk("Expected uid=%d, euid=%d, gid=%d\n", entry->uid, entry->euid, entry->gid)
        ;
199     bpf_printk("Expected fsuid=%d, fsgid=%d\n", entry->fsuid, entry->fsgid);
200     bpf_printk("New cred uid=%d, euid=%d, gid=%d\n", new_uid, new_euid, new_gid);
201     bpf_printk("New cred fsuid=%d, fsgid=%d\n", new_fsuid, new_fsgid);
202     // Only block privilege escalation to root for BPF-related processes
203
204     if ((new_uid != entry->uid ) ||
205         (new_euid != entry->euid) ||
206         (new_gid != entry->gid) ||
207         (new_fsuid != entry->fsuid ) ||
208         (new_fsgid != entry->fsgid ) ||
209         (new_sgid != entry->sgid ) ||
210         (new_suid != entry->suid )
211         ){
212         bpf_printk("[!LSM ALERT] BPF process privilege escalation in inode_permission!\n"
        );
213         bpf_printk("[!LSM ALERT] UID: %d->%d\n", entry->uid, new_uid);
214         bpf_printk("[!LSM ALERT] EUID: %d->%d\n", entry->euid, new_euid);
215         bpf_printk("[!LSM ALERT] GID: %d->%d\n", entry->gid, new_gid);
216         bpf_printk("[!LSM ALERT] FSUID: %d->%d\n", entry->fsuid, new_fsuid);
217         bpf_printk("[!LSM ALERT] FSGID: %d->%d\n", entry->fsgid, new_fsgid);
218
219         return -EPERM;
220     }
221
222     return 0;
223 }
224
225 SEC("lsm/task_kill")
226 int BPF_PROG(task_kill, struct task_struct *p, struct kernel_siginfo *info,
227    int sig, const struct cred *cred)
228 {
229     pid_t pid = bpf_get_current_pid_tgid() >> 32;
230     // Check if this process is tracked (BPF-related)
231     struct tracked_entry *entry = bpf_map_lookup_elem(&tracked_processes, &pid);
232     if (!entry) {
233         // Not tracked - allow legitimate privilege escalation (sudo, etc.)
234         return 0;
235     }
236
237     __u32 new_uid = 0, new_euid = 0, new_gid = 0, new_fsuid = 0, new_fsgid = 0, new_suid
        =0, new_sgid=0;
```

```
238     bpf_core_read(&new_uid, sizeof(new_uid), &cred->uid.val);
239     bpf_core_read(&new_euid, sizeof(new_euid), &cred->euid.val);
240     bpf_core_read(&new_gid, sizeof(new_gid), &cred->gid.val);
241     bpf_core_read(&new_fsuid, sizeof(new_fsuid), &cred->fsuid.val);
242     bpf_core_read(&new_fsgid, sizeof(new_fsgid), &cred->fsgid.val);
243     bpf_core_read(&new_suid, sizeof(new_suid), &cred->suid.val);
244     bpf_core_read(&new_sgid, sizeof(new_sgid), &cred->sgid.val);
245 
246     bpf_printk("LSM task_kill: tracked BPF pid=%d\n", pid);
247     bpf_printk("Expected uid=%d, euid=%d, gid=%d\n", entry->uid, entry->euid, entry->gid)
        ;
248     bpf_printk("New cred uid=%d, euid=%d, gid=%d\n", new_uid, new_euid, new_gid);
249 
250     // Only block privilege escalation to root for BPF-related processes
251     if ((new_uid != entry->uid ) ||
252         (new_euid != entry->euid) ||
253         (new_gid != entry->gid) ||
254         (new_fsuid != entry->fsuid ) ||
255         (new_fsgid != entry->fsgid ) ||
256         (new_sgid != entry->sgid ) ||
257         (new_suid != entry->suid )
258         ){
259         bpf_printk("[!LSM ALERT] BPF process privilege escalation in task_kill!\n");
260         bpf_printk("[!LSM ALERT] UID: %d->%d\n", entry->uid, new_uid);
261         bpf_printk("[!LSM ALERT] EUID: %d->%d\n", entry->euid, new_euid);
262         bpf_printk("[!LSM ALERT] GID: %d->%d\n", entry->gid, new_gid);
263         bpf_printk("[!LSM ALERT] FSUID: %d->%d\n", entry->fsuid, new_fsuid);
264         bpf_printk("[!LSM ALERT] FSGID: %d->%d\n", entry->fsgid, new_fsgid);
265 
266         return -EPERM;
267     }
268 
269     return 0;
270 }
271 
272 SEC("lsm/task_free")
273 int BPF_PROG(task_free, struct task_struct *task)
274 {
275     pid_t pid;
276     int ret = 0;
277     ret = bpf_probe_read_kernel(&pid, sizeof(pid), &task->pid);
278 
279     if(ret<0){
280         bpf_printk("LSM failed to read pid of dying process\n");
281         return 0; // Don't block, just return
282     }
283 
284     // Check if this process is actually tracked before trying to remove it
285     struct tracked_entry *entry = bpf_map_lookup_elem(&tracked_processes, &pid);
286     if (!entry) {
287         // Process not tracked, nothing to do
288         return 0;
289     }
290 
291     // Process is tracked, remove it from the map
292     ret = bpf_map_delete_elem(&tracked_processes, &pid);
293     if (ret == 0) {
294         bpf_printk("Removed tracked entry for BPF process pid=%d\n", pid);
295     } else {
296         bpf_printk("Failed to remove tracked entry for pid=%d, ret=%d\n", pid, ret);
297     }
298     return 0;
299 }
```

Listing A.1. BPF LSM program implementation

A.2.2 Userspace Loader Program

The userspace loader serves as the deployment mechanism for the BPF LSM program, utilizing the libbpf library to interface with the kernel's BPF subsystem. It handles the complete lifecycle of BPF program management: opening and loading the compiled BPF object through the auto-generated skeleton interface, attaching the program to the appropriate LSM hooks, and maintaining execution until explicitly terminated. The loader implements signal handling for graceful shutdown and provides runtime feedback through libbpf's diagnostic callbacks.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include <signal.h>
#include "lsm_hardening.skel.h"

/* Notice: Ensure your kernel version is 5.7 or higher, BTF (BPF Type Format) is enabled,
 * and the file '/sys/kernel/security/lsm' includes 'bpf'.
 */

static int libbpf_print_fn(enum libbpf_print_level level, const char *format, va_list
    args)
{
  return vfprintf(stderr, format, args);
}

static volatile bool exiting = false;

static void handle_signal(int sig)
{
  exiting = true;
}

int main(int argc, char **argv)
{
  struct lsm_hardening_bpf *skel;
  int err=0;
  /* Set up libbpf errors and debug info callback */
  libbpf_set_print(libbpf_print_fn);
        signal(SIGINT, handle_signal);
  signal(SIGTERM, handle_signal);

  /* Open, load, and verify BPF application */
  skel = lsm_hardening_bpf__open_and_load();
  if (!skel) {
      fprintf(stderr, "Failed to open and load\n");
      return 1;
  }

  /* Attach lsm handler */
  err = lsm_hardening_bpf__attach(skel);
  if (err) {
    fprintf(stderr, "Failed to attach BPF skeleton\n");
    goto cleanup;
  }

  printf("Successfully started! Please run `sudo cat /sys/kernel/debug/tracing/trace_pipe
    ` "
```

```
48          "to see output of the BPF programs.\n");
49
50    for (;!exiting;) {
51      /* trigger our BPF program */
52      fprintf(stderr, ".");
53      sleep(1);
54    }
55    printf("Exiting...\n");
56  cleanup:
57    lsm_hardening_bpf__destroy(skel);
58    return -err;
59  }
```

Listing A.2. BPF LSM loader implementation

A.2.3 Automated Build and Deployment Script

The build script orchestrates the complete compilation and deployment pipeline for the BPF LSM solution. It begins by conditionally generating the `vmlinux.h` header containing BTF type information extracted from the running kernel, which is essential for CO-RE (Compile Once – Run Everywhere) functionality. The script then compiles the BPF program to bytecode using Clang's BPF target, generates a skeleton header that provides type-safe userspace access to BPF maps and programs, compiles the userspace loader with appropriate library linkage, and finally executes the loader with elevated privileges. This automated workflow eliminates manual compilation steps and ensures consistent deployment across different development and testing environments.

```
1  #!/bin/bash
2  set -e
3
4  # Configuration
5  BPF_PROG="lsm_hardening"
6  BPF_SOURCE="${BPF_PROG}.bpf.c"
7  BPF_OBJECT="${BPF_PROG}.bpf.o"
8  SKEL_HEADER="${BPF_PROG}.skel.h"
9  LOADER_SOURCE="${BPF_PROG}.c"
10 LOADER_BINARY="lsm_loader"
11 VMLINUX_HEADER="vmlinux.h"
12
13 # Generate BTF header from running kernel (if not exists)
14 if [ ! -f "${VMLINUX_HEADER}" ]; then
15     echo "[1/5] Generating BTF type information..."
16     bpftool btf dump file /sys/kernel/btf/vmlinux format c > ${VMLINUX_HEADER}
17 else
18     echo "[1/5] BTF header already exists, skipping generation"
19 fi
20
21 # Compile BPF program to object file
22 echo "[2/5] Compiling BPF program..."
23 clang -O2 -Wall -target bpf -g -c ${BPF_SOURCE} -o ${BPF_OBJECT}
24
25 # Generate BPF skeleton header
26 echo "[3/5] Generating BPF skeleton..."
27 bpftool gen skeleton ${BPF_OBJECT} > ${SKEL_HEADER}
```

```
28
29 # Compile userspace loader
30 echo "[4/5] Compiling loader program..."
31 gcc -g -O2 -Wall -I. -I/usr/include/bpf \
32     -o ${LOADER_BINARY} ${LOADER_SOURCE} ${SKEL_HEADER} \
33     -lbpf -lelf -lz
34
35 # Run loader with elevated privileges
36 echo "[5/5] Running BPF LSM loader..."
37 sudo ./${LOADER_BINARY}
```

Listing A.3. Build script for BPF LSM program and loader

A.3 BPF LSM program Validation

To validate the BPF LSM program's effectiveness against CVE-2022-23222, exploit variants were executed with active enforcement. Figures A.3 and A.4 show exploit attempts under BPF LSM enforcement. Each screenshot's upper half displays the loader confirming successful attachment, while the lower half shows the exploit execution.

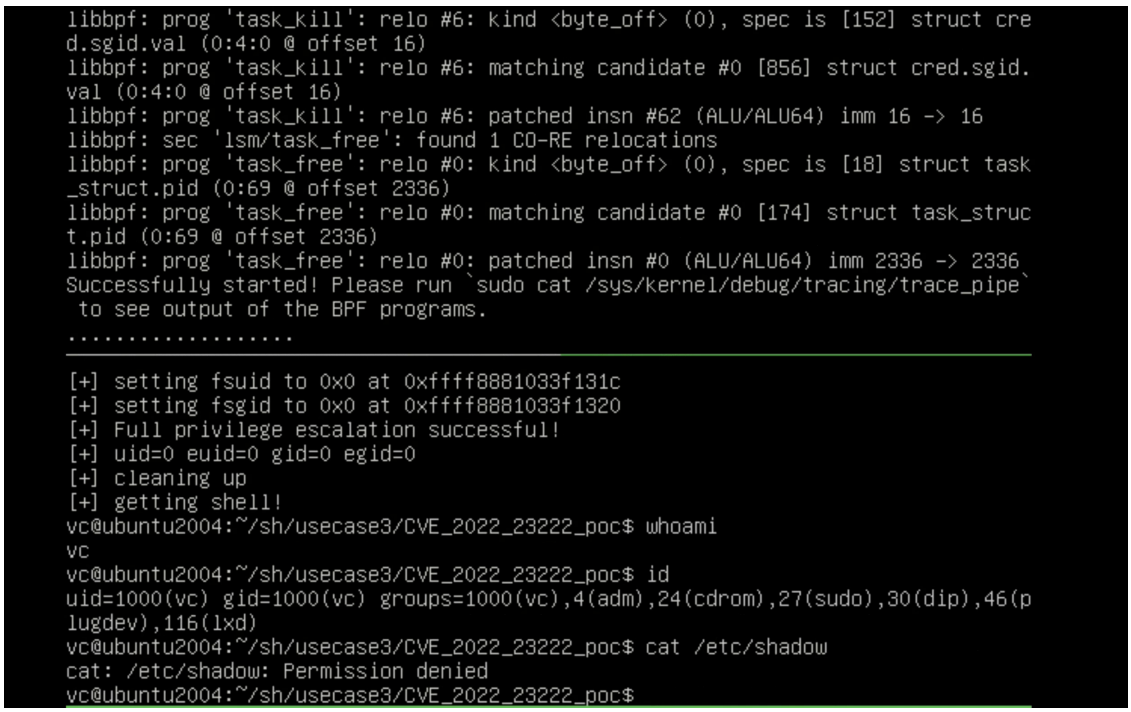

Figure A.3. Complete privilege escalation attempt blocked by BPF LSM

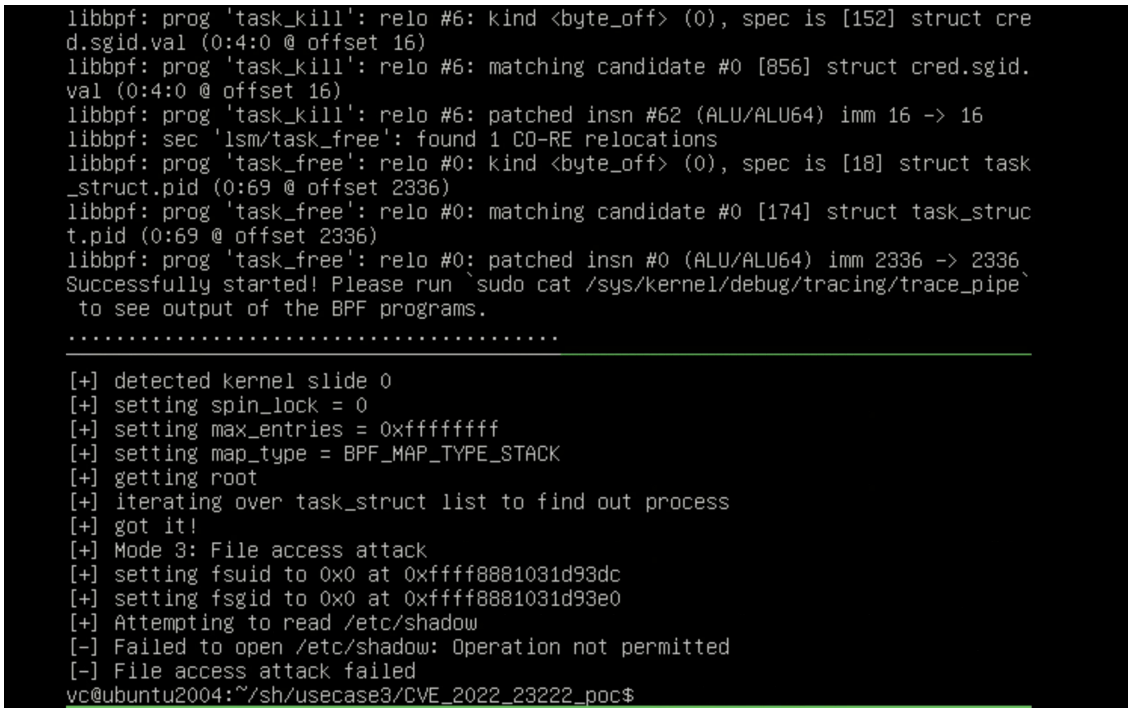

Figure A.4. Privileged file access attempt blocked by BPF LSM

Figure A.3 shows the exploit completing without visible errors, yet the BPF LSM successfully blocks privilege escalation. This occurs because the exploit does not verify credential modification success, it simply attempts memory manipulation and spawns a shell. The BPF LSM intercepts credential changes, preventing them from taking effect. In fact, the spawned shell retains unprivileged credentials, as confirmed by subsequent `id` and `whoami` commands showing unchanged UIDs and GIDs.

In addition, figures A.5 and A.6 present diagnostic logs from `/sys/kernel/debug/tracing/trace_pipe`, confirming the blocking of the exploit attempts.

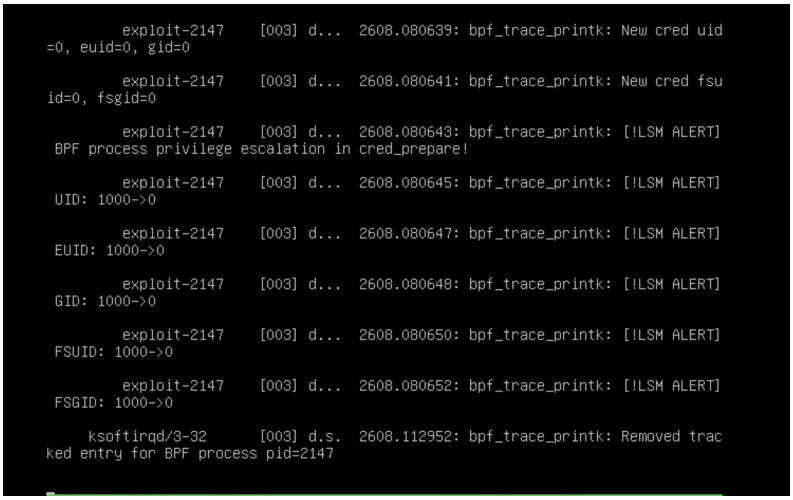

Figure A.5. BPF LSM logs showing blocked privilege escalation

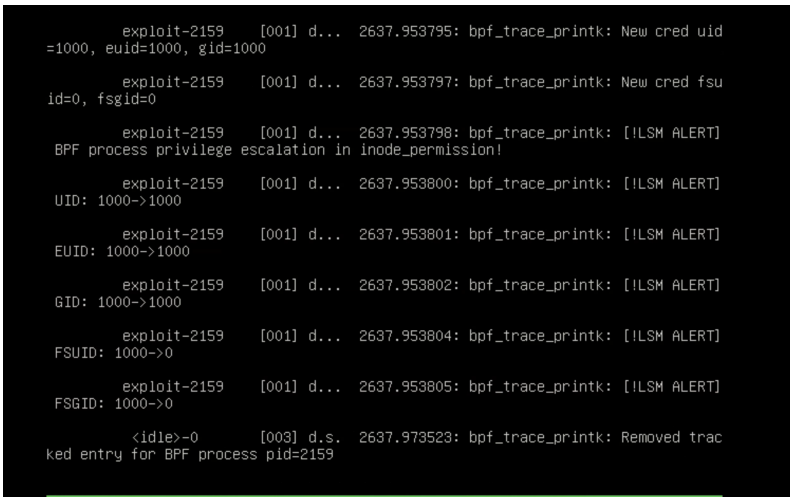

Figure A.6. BPF LSM logs showing blocked file access

Appendix B

Use Case 2 – Kernel Panic and Denial of Service

B.1 Exploit Execution

The exploit for CVE-2024-56614/56615 was executed in a controlled environment to validate its denial-of-service capabilities.

Figure B.1 shows the exploit's output immediately before kernel crash, displaying the successful creation of a `DEVMAP` with malicious parameters and the identification of two exploitable keys that trigger arbitrary memory access at calculated `netdev_map` array indices. Figure B.2 captures the resulting kernel panic triggered by the exploit out-of-bounds access.

```
vc@ebpf-svt2:~/cve_2024_56615$ ./cve_2024-56615
=== CVE-2024-56615 ===

[+] Creating DEVMAP with max_entries=0x80000002
[+] Created DEVMAP with fd=3
[+] Range of arbitrary memory access (2 exploitable keys):
  Key 0x80000000 -> array index -2147483648: SUCCESS - Accessed memory at netdev_map[-2147483648]
  Key 0x80000001 -> array index -2147483647: SUCCESS - Accessed memory at netdev_map[-2147483647]
```

Figure B.1. Exploit output showing successful DEVMAP creation and arbitrary memory access

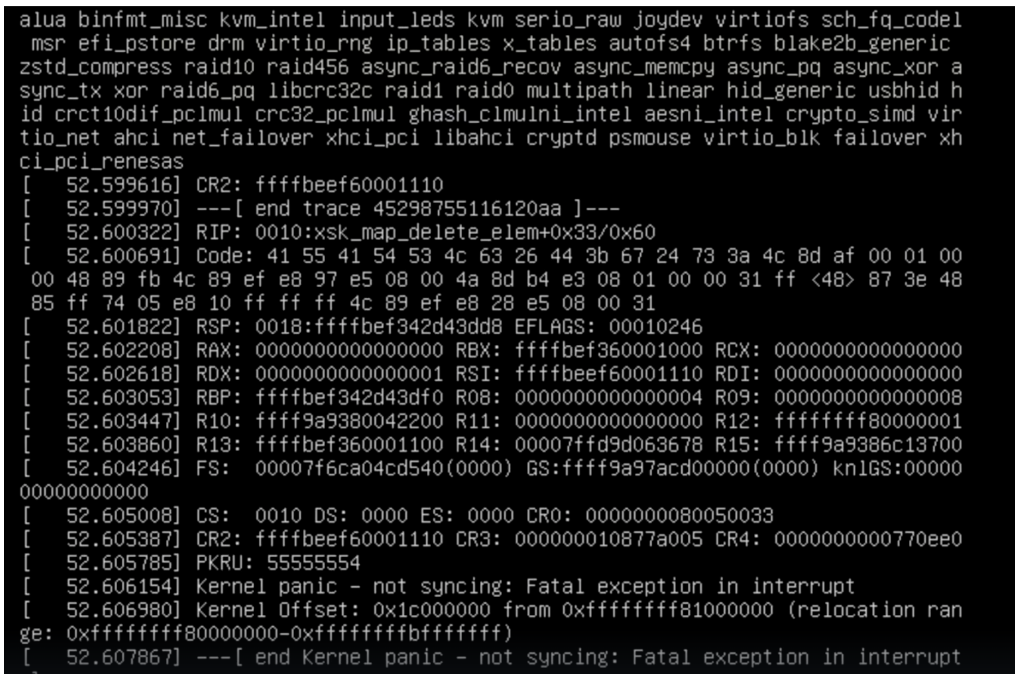

Figure B.2. Kernel panic triggered by out-of-bounds memory access

B.2 BPF LSM Program Implementation

This appendix presents the complete implementation of the BPF LSM-based hardening solution for CVE-2024-56614 and CVE-2024-56615, consisting of: the kernel-space BPF program that enforces security policies, the userspace loader responsible for program deployment, and the automated build script that orchestrates the compilation and execution workflow.

B.2.1 Kernel-Space BPF LSM Program

The BPF LSM program implements the core security enforcement logic by attaching to multiple LSM hooks within the Linux kernel.

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

#define EPERM 1
#define BPF_MAP_TYPE_DEVMAP 14
#define BPF_MAP_TYPE_XSKMAP 17
#define MAX_TRACKED_MAPS 1024

// For syscalls:sys_enter_close
struct sys_enter_close_args {
    u64 __unused;          // common fields handled by BTF
    long __syscall_nr;
    long fd;
};

// For syscalls:sys_exit_bpf
struct sys_exit_bpf_args {
    u64 __unused;
    long __syscall_nr;
    long ret;              // return value of bpf(2)
};

// Track specific suspicious DEVMAPs/XSKMAPs by their file descriptor
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, MAX_TRACKED_MAPS);
    __type(key, u32);     // map_fd
    __type(value, u32);   // max_entries of the suspicious DEVMAP/XSKMAP
} suspicious_map_fds SEC(".maps");

// Track map creation by PID to correlate with map_fd later
struct pending_info {
    u32 max_entries;
    u64 timestamp;
    u32 map_type;
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, MAX_TRACKED_MAPS);
    __type(key, u32);     // PID
    __type(value, struct pending_info);
} pending_map_creation SEC(".maps");
```

```
46
47 // --- 1) Correlate pending creation -> map_fd on bpf() return ---
48 SEC("tracepoint/syscalls/sys_exit_bpf")
49 int on_sys_exit_bpf(struct sys_exit_bpf_args *ctx)
50 {
51     long ret = ctx->ret;
52     if (ret < 0)
53         return 0;  // bpf() failed; no fd returned
54
55     u32 pid = bpf_get_current_pid_tgid() >> 32;
56
57     // Do we have a pending DEVMAP/XSKMAP creation recorded by the LSM hook?
58     struct pending_info *pending = bpf_map_lookup_elem(&pending_map_creation, &pid);
59     if (!pending)
60         return 0;
61
62     // ret is the newly created object fd (map fd here)
63     u32 map_fd = (u32)ret;
64
65     // IMMEDIATE correlation - store ONLY suspicious DEVMAPs/XSKMAPs
66     if ((pending->map_type == BPF_MAP_TYPE_DEVMAP || pending->map_type ==
       BPF_MAP_TYPE_XSKMAP) &&
67         pending->max_entries > 0x80000000) {
68         bpf_map_update_elem(&suspicious_map_fds, &map_fd, &pending->max_entries, BPF_ANY)
       ;
69         bpf_printk("[LSM+TP] tracked %s fd=%d max_entries=0x%x\n",
70                    pending->map_type == BPF_MAP_TYPE_DEVMAP ? "DEVMAP" : "XSKMAP",
71                    map_fd, pending->max_entries);
72     }
73
74     // Always clear pending (we've processed this create)
75     bpf_map_delete_elem(&pending_map_creation, &pid);
76     return 0;
77 }
78
79 // --- 2) Drop tracking when the fd is closed ---
80 SEC("tracepoint/syscalls/sys_enter_close")
81 int on_sys_enter_close(struct sys_enter_close_args *ctx)
82 {
83     int fd = (int)ctx->fd;
84     if (fd < 0)
85         return 0;
86
87     // If we tracked this fd as suspicious, remove it now
88     u32 key = (u32)fd;
89     int err = bpf_map_delete_elem(&suspicious_map_fds, &key);
90     if (err == 0) {
91         bpf_printk("[LSM+TP] cleanup: fd=%d removed from suspicious_map_fds", fd);
92     }
93     return 0;
94 }
95
96 SEC("lsm/bpf")
97 int BPF_PROG(bpf_security_check, int cmd, union bpf_attr *attr, unsigned int size)
98 {
99     if (!attr) {
100         return 0;
101     }
102
103     u32 pid = bpf_get_current_pid_tgid() >> 32;
104
105     // Track DEVMAP/XSKMAP creation
106     if (cmd == BPF_MAP_CREATE) {
107         u32 map_type = BPF_CORE_READ(attr, map_type);
```

```
108         if (map_type == BPF_MAP_TYPE_DEVMAP || map_type == BPF_MAP_TYPE_XSKMAP) {
109             u32 max_entries = BPF_CORE_READ(attr, max_entries);
110             bpf_printk("[LSM] %s creation: max_entries=0x%x",
111                        map_type == BPF_MAP_TYPE_DEVMAP ? "DEVMAP" : "XSKMAP", max_entries
    );
112 
113             // Always store creation info for correlation
114             struct pending_info info = {
115                 .max_entries = max_entries,
116                 .timestamp = bpf_ktime_get_ns(),
117                 .map_type = map_type
118             };
119             bpf_map_update_elem(&pending_map_creation, &pid, &info, BPF_ANY);
120 
121             // Alert for suspicious creation but ALWAYS allow
122             if (max_entries > 0x80000000) {
123                 u32 exploitable_keys = max_entries - 0x80000000;
124                 bpf_printk("[LSM] ALERT: SUSPICIOUS %s CREATION!",
125                            map_type == BPF_MAP_TYPE_DEVMAP ? "DEVMAP" : "XSKMAP");
126                 bpf_printk("[LSM] max_entries=0x%x enables %u exploitable keys",
127                            max_entries, exploitable_keys);
128                 bpf_printk("[LSM] This %s will be monitored for exploitation attempts\n",
129                            map_type == BPF_MAP_TYPE_DEVMAP ? "DEVMAP" : "XSKMAP");
130             }
131         } else {
132             // Clean up any stale pending creation to avoid false correlation
133             bpf_map_delete_elem(&pending_map_creation, &pid);
134         }
135         return 0;
136     }
137 
138     // Handle MAP_DELETE_ELEM operations
139     if (cmd == BPF_MAP_DELETE_ELEM) {
140         u32 map_fd = BPF_CORE_READ(attr, map_fd);
141         u64 key_ptr = BPF_CORE_READ(attr, key);
142 
143         if (map_fd == 0 || key_ptr == 0) {
144             return 0;
145         }
146 
147         u32 user_key;
148         int ret = bpf_probe_read_user(&user_key, sizeof(user_key), (void *)key_ptr);
149         if (ret != 0) {
150             return 0;
151         }
152 
153         bpf_printk("[LSM] DELETE_ELEM: map_fd=%d, key=0x%x", map_fd, user_key);
154 
155         // Only check large keys that could be exploitable
156         if (user_key >= 0x80000000) {
157             // Check if this map_fd is a known suspicious DEVMAP/XSKMAP
158             u32 *stored_max_entries = bpf_map_lookup_elem(&suspicious_map_fds, &map_fd);
159             if (stored_max_entries) {
160                 u32 max_entries = *stored_max_entries;
161 
162                 // Calculate exact exploitable range like the exploit does
163                 u32 exploitable_keys = max_entries - 0x80000000;
164                 u32 max_exploitable_key = 0x80000000 + exploitable_keys - 1;
165 
166                 bpf_printk("[LSM] Checking suspicious DEVMAP/XSKMAP fd=%d, max_entries=0x
    %x", map_fd, max_entries);
167                 bpf_printk("[LSM] Exploitable range: [0x80000000, 0x%x] (%u keys)",
168                            max_exploitable_key, exploitable_keys);
169 
```

```
170                 // Block only keys in the exact exploitable range
171                 if (user_key >= 0x80000000 && user_key <= max_exploitable_key) {
172                     bpf_printk("[LSM] BLOCKED: Key 0x%x is in exploitable range!",
    user_key);
173                     return -EPERM;
174                 } else {
175                     bpf_printk("[LSM] Key 0x%x outside exploitable range, allowed",
    user_key);
176                 }
177             } else {
178                 // Not a tracked suspicious DEVMAP/XSKMAP - allow
179                 bpf_printk("[LSM] Large key 0x%x allowed (not on suspicious DEVMAP/XSKMAP
    )", user_key);
180             }
181         }
182
183         return 0;
184     }
185
186     return 0;
187 }
188
189 SEC("lsm/task_free")
190 int BPF_PROG(task_free_security_check, struct task_struct *task)
191 {
192     if (!task)
193         return 0;
194
195     u32 pid = BPF_CORE_READ(task, pid);
196
197     // Clean up any pending creation for this process
198     int res = bpf_map_delete_elem(&pending_map_creation, &pid);
199     if (res == 0) {
200         bpf_printk("[LSM] Cleaned up pending creation for exiting PID %u", pid);
201     }
202
203     return 0;
204 }
205
206 char _license[] SEC("license") = "GPL";
```

Listing B.1. BPF LSM program implementation

B.2.2 Userspace Loader Program

The userspace loader serves as the deployment mechanism for the BPF LSM program, utilizing the libbpf library to interface with the kernel's BPF subsystem. It handles the complete lifecycle of BPF program management: opening and loading the compiled BPF object through the auto-generated skeleton interface, attaching the program to the appropriate LSM hooks, and maintaining execution until explicitly terminated. The loader implements signal handling for graceful shutdown and provides runtime feedback through libbpf's diagnostic callbacks. It's possible to use the same userspace loader program A.2 used for the previous use case, with the only difference being the BPF program and skeleton header filenames.

B.2.3 Automated Build and Deployment Script

As an alternative to the bash script presented in the previous use case, a Makefile-based build system provides a more robust and efficient compilation workflow. Unlike the sequential bash script, the Makefile leverages dependency tracking to perform incremental builds, recompiling only components that have changed or whose dependencies have been modified. This approach significantly reduces rebuild times during iterative development cycles.

```
# Name of your program (without extensions)
prog_name := lsm

# Compiler and flags
CLANG := clang
CC := gcc
CFLAGS := -g -O2 -Wall
BPF_CFLAGS := -O2 -Wall -target bpf -g -D__TARGET_ARCH_x86

# Includes
INCLUDES := -I. -I/usr/include/bpf
# Libraries
LIBS := -lbpf -lelf -lz

# Files
BPF_SRC := $(prog_name).bpf.c
BPF_OBJ := $(prog_name).bpf.o
SKEL_HDR := $(prog_name).skel.h
USER_SRC := $(prog_name).c
USER_BIN := $(prog_name)

.PHONY: all clean

all: $(USER_BIN)

# Generate vmlinux.h if not present
vmlinux.h:
  bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h

# Compile the BPF program and generate skeleton
$(SKEL_HDR): $(BPF_OBJ)
  bpftool gen skeleton $< > $@

$(BPF_OBJ): $(BPF_SRC) vmlinux.h
  $(CLANG) $(BPF_CFLAGS) -c $< -o $@

# Build user-space binary
$(USER_BIN): $(USER_SRC) $(SKEL_HDR)
  $(CC) $(CFLAGS) $(INCLUDES) -o $@ $(USER_SRC) $(LIBS)

clean:
  rm -f $(BPF_OBJ) $(SKEL_HDR) $(USER_BIN)
```

Listing B.2. Makefile for BPF LSM program and loader

In this case, once the Makefile is executed, the resulting binary can be run with elevated privileges to load and attach the BPF LSM program running:

```
sudo ./$(USER_BIN)
```

B.3 BPF LSM program Validation

To validate the BPF LSM program's effectiveness against CVE-2024-56614 and CVE-2024-56615, the exploits were executed with active enforcement. The validation demonstrates the program's ability to prevent denial-of-service attacks while maintaining system stability.

Figure B.3 shows the exploit execution under BPF LSM enforcement. The exploit successfully creates the malicious DEVMAP, however, when attempting to trigger the out-of-bounds memory access, the BPF LSM intercepts the operation at the `bpf_map_delete_elem` hook and blocks it with "Operation not permitted", preventing the kernel panic.

```
vc@ebpf-svt2:~/cve_2024_56615$ ./cve_2024-56615
=== CVE-2024-56615 ===

[+] Creating DEVMAP with max_entries=0x80000002
[+] Created DEVMAP with fd=3
[+] Range of arbitrary memory access (2 exploitable keys):
  Key 0x80000000 -> array index -2147483648: BLOCKED - Operation not permitted
```

Figure B.3. Denial-of-service attempt blocked by BPF LSM

Figure B.4 presents the diagnostic logs from `/sys/kernel/debug/tracing/trace_pipe`. The logs show the complete enforcement cycle: detection of suspicious DEVMAP creation with `max_entries=0x80000002`, identification of the exploitable key range, tracking of the suspicious map in the monitoring system, and ultimately the blocking of delete operations with `[LSM] BLOCKED` messages.

```
          <...>-1498918 [008] ...11 3715877.215670: bpf_trace_printk: [LSM] DEVMAP creation: max_entries=0x80000002
          <...>-1498918 [008] ...11 3715877.215729: bpf_trace_printk: [LSM] ALERT: SUSPICIOUS DEVMAP CREATION!
          <...>-1498918 [008] ...11 3715877.215732: bpf_trace_printk: [LSM] max_entries=0x80000002 enables 2 exploitable keys
          <...>-1498918 [008] ...11 3715877.215734: bpf_trace_printk: [LSM] This DEVMAP will be monitored for exploitation attemp
ts

  cve_2024-56615-1498918 [008] ...21 3715902.301564: bpf_trace_printk: [LSM+TP] tracked DEVMAP fd=3 max_entries=0x80000002

  cve_2024-56615-1498918 [008] ...11 3715902.301674: bpf_trace_printk: [LSM] DELETE_ELEM: map_fd=3, key=0x80000000
  cve_2024-56615-1498918 [008] ...11 3715902.301675: bpf_trace_printk: [LSM] Checking suspicious DEVMAP/XSKMAP fd=3, max_entries=0
x80000002
  cve_2024-56615-1498918 [008] ...11 3715902.301676: bpf_trace_printk: [LSM] Exploitable range: [0x80000000, 0x80000001] (2 keys)
  cve_2024-56615-1498918 [008] ...11 3715902.301676: bpf_trace_printk: [LSM] BLOCKED: Key 0x80000000 is in exploitable range!
  cve_2024-56615-1498918 [008] ...21 3715902.301713: bpf_trace_printk: [LSM+TP] cleanup: fd=3 removed from suspicious_map_fds
```

Figure B.4. BPF LSM logs showing blocked denial-of-service attempts

Bibliography

[1] bootlin Elixir Cross Referencer. Defective dev_map_delete_elem function, Linux kernel v5.13. https://elixir.bootlin.com/linux/v5.13/source/kernel/bpf/devmap.c#L532. Accessed: 2025/09/22.

[2] Linux Kernel Documentation. BPF Documentation. https://docs.kernel.org/bpf/index.html. Accessed: 2025/08/31.

[3] Adrian Ratiu. An eBPF overview, part 1, Introduction. https://www.collabora.com/news-and-blog/blog/2019/04/05/an-ebpf-overview-part-1-introduction/. Accessed: 2025/09/03.

[4] Alexei Starovoitov. bpf: introduce bounded loops. https://github.com/torvalds/linux/commit/2589726d12a1b12eaaa93c7f1ea64287e383c7a5. Accessed: 2025/09/03.

[5] eBPF Documentation. Verifier. https://docs.ebpf.io/linux/concepts/verifier/. Accessed: 2025/09/01.

[6] Cilium. JIT. https://docs.cilium.io/en/latest/reference-guides/bpf/architecture/#jit. Accessed: 2025/09/04.

[7] eBPF Documentation. Maps. https://docs.ebpf.io/linux/concepts/maps/. Accessed: 2025/09/01.

[8] Linux Kernel Documentation. BPF Maps. https://docs.kernel.org/bpf/maps.html. Accessed: 2025/09/01.

[9] cilium. XDP. https://docs.cilium.io/en/latest/reference-guides/bpf/progtypes/#xdp. Accessed: 2025/09/02.

[10] Linux manual page. bpf-helpers. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html. Accessed: 2025/09/02.

[11] eunomia. Categorization of eBPF Hooks. https://eunomia.dev/zh/others/miscellaneous/ebpf-usecases/. Accessed: 2025/08/31.

[12] eBPF Documentation. Program types (Linux). https://docs.ebpf.io/linux/program-type/. Accessed: 2025/09/02.
[13] eBPF Documentation. BTF. https://docs.ebpf.io/linux/concepts/btf/. Accessed: 2025/09/02.
[14] libbpf. libbpf. https://github.com/libbpf/libbpf. Accessed: 2025/09/02.
[15] Jack Kelly, James Callaghan, Andrew Martin. eBPF Security Thread Model. https://www.linuxfoundation.org/hubfs/eBPF/ControlPlane%20âĂŤ%20eBPF%20Security%20Threat%20Model.pdf. Accessed: 2025/09/02.
[16] 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 *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5971–5988. USENIX Association, August 2023. ISBN: 978-1-939133-37-3. Accessed: 2025/09/04. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/he.
[17] Rex Guo and Junyuan Zeng. Phantom Attack: Evading System Call Monitoring. In *DEF CON 29*, pages 16–25. DEF CON, August 2021. Accessed: 2025/09/04, DOI: 10.5446/54228. 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, doi:10.5446/54228.
[18] The Trail of Bits Blog. Pitfalls of relying on eBPF for security monitoring (and some solutions). https://blog.trailofbits.com/2023/09/25/pitfalls-of-relying-on-ebpf-for-security-monitoring-and-some-solutions/. Accessed: 2025/09/04.
[19] Colson Wilhoit Jake King. A peek behind the BPFDoor. https://www.elastic.co/security-labs/a-peek-behind-the-bpfdoor. Accessed: 2025/09/04.
[20] Cilium. Tetragon. https://github.com/cilium/tetragon. Accessed: 2025/09/02.
[21] Datadog. Cloud Monitoring as a Service. https://www.datadoghq.com/. Accessed: 2025/09/04.
[22] National Vulnerability Database. CVE-2022-0500. https://nvd.nist.gov/vuln/detail/CVE-2022-0500. Accessed:

2025/09/04.
[23] National Vulnerability Database. CVE-2024-50164. https://nvd.nist.gov/vuln/detail/CVE-2024-50164, 2024. Accessed: 2025/09/04.
[24] National Vulnerability Database. CVE-2020-8835. https://nvd.nist.gov/vuln/detail/CVE-2020-8835. Accessed: 2025/09/04.
[25] Manfred Paul. CVE-2020-8835: Linux Kernel Privilege Escalation via Improper eBPF Program Verification. https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification. Accessed: 2025/09/02.
[26] National Vulnerability Database. CVE-2023-2163. https://nvd.nist.gov/vuln/detail/CVE-2023-2163. Accessed: 2025/09/04.
[27] Juan José López Jaimez, Meador Inge. A deep dive into CVE-2023-2163: How we found and fixed an eBPF Linux Kernel Vulnerability. https://bughunters.google.com/blog/6303226026131456/a-deep-dive-into-cve-2023-2163-how-we-found-and-fixed-an-ebpf-linux-kernel-vulnerability. Accessed: 2025/09/02.
[28] National Vulnerability Database. CVE-2021-3490. https://nvd.nist.gov/vuln/detail/CVE-2021-3490. Accessed: 2025/09/04.
[29] National Vulnerability Database. CVE-2017-16995. https://nvd.nist.gov/vuln/detail/CVE-2017-16995. Accessed: 2025/09/04.
[30] National Vulnerability Database. CVE-2024-50203. https://nvd.nist.gov/vuln/detail/CVE-2024-50203. Accessed: 2025/09/04.
[31] National Vulnerability Database. CVE-2021-29154. https://nvd.nist.gov/vuln/detail/CVE-2021-29154. Accessed: 2025/09/04.
[32] iovisor. kretprobes are mysteriously missed. https://github.com/iovisor/bcc/issues/2825, 2020. Accessed: 2025/10/08.
[33] h3xduck. TripleCross GitHub repository. https://github.com/h3xduck/triplecross, 2022. Accessed: 2025/10/08.
[34] Serge E. Hallyn and Andrew G. Morgan. Linux capabilities: making

them work. In *Linux Symposium 2008*, page 10. Linux Symposium Inc., July 2008. URL: https://www.kernel.org/doc/ols/2008/ols2008v1-pages-163-172.pdf.

[35] Alexei Starovoitov. bpf, capability: Introduce CAP BPF. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a17b53c4a4b55ec322c132b6670743612229ee9c. Accessed: 2025/09/01.

[36] Prasme Systems Vandana Salve. Inside the Linux Security Module (LSM). In *Embedded Linux Conference (ELC)*. The Linux Foundation, September 2020. Accessed: 2025-09-09. URL: https://elinux.org/images/0/0a/ELC_Inside_LSM.pdf.

[37] SELinuxProject. selinux. https://github.com/SELinuxProject/selinux. Accessed: 2025/09/09.

[38] Canonical Ltd. AppArmor. https://apparmor.net/. Accessed: 2025/09/09.

[39] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *11th USENIX Security Symposium (USENIX Security 02)*, San Francisco, CA, August 2002. USENIX Association. Accessed: 2025/09/09. URL: https://www.usenix.org/conference/11th-usenix-security-symposium/linux-security-modules-general-security-support-linux.

[40] eBPF Documentation. Program type BPF_PROG_TYPE_LSM. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_LSM/. Accessed: 2025/09/09.

[41] KP Singh. Kernel Runtime Security Instrumentation (KRSI). In *Linux Security Summit North America (LSS-NA 2019)*, San Diego, CA, August 2019. Linux Foundation.

[42] Elixir Bootlin Cross Reference. lsm_hooks.h, v5.13. https://elixir.bootlin.com/linux/v5.13/source/include/linux/lsm_hooks.h. Accessed: 2025/09/10.

[43] Elixir Bootlin Cross Reference. lsm_hook_defs.h, v5.13. https://elixir.bootlin.com/linux/v5.13/source/include/linux/lsm_hook_defs.h. Accessed: 2025/09/10.

[44] lkrg org. Linux Kernel Runtime Guard (LKRG). https://github.com/lkrg-org/lkrg. Accessed: 2025/09/08.

[45] H. Xu, W. Du, and S.J. Chapin. Detecting exploit code execution in loadable kernel modules. In *20th Annual Computer Security Applications*

Conference, pages 101–110, 2004. doi:10.1109/CSAC.2004.18.

[46] NIST National Vulnerability Database. Vulnerability metrics. https://nvd.nist.gov/vuln-metrics/cvss. Accessed: 2025/09/19.

[47] National Vulnerability Database. CVE-2022-23222. https://nvd.nist.gov/vuln/detail/CVE-2022-23222. Accessed: 2025/09/15.

[48] PenteraIO. CVE-2022-23222-POC. https://github.com/PenteraIO/CVE-2022-23222-POC. Accessed: 2025-09-15.

[49] Matan Liber. The good, bad and compromisable aspects of linux ebpf. Technical report, Pentera Labs Research Series, 2022. Accessed: 2025-09-15. URL: https://www.pentera.io/pentera-labs-the-good-bad-and-compromisable-aspects-of-linux-ebpf.pdf.

[50] Juho Junnila. Effectiveness of Linux Rootkit Detection Tools. Master's thesis, University of Oulu, 2020.

[51] rcorrea35. Linux Kernel: Integer Overflow in eBPF XSK map_delete_elem Leads to Out of Bounds. https://github.com/google/security-research/security/advisories/GHSA-cqc2-6j63-6qrx, 2024. Accessed: 2025/09/17.

[52] rcorrea35. Integer Overflow in eBPF DEVMAP map_delete_elem Leads to Out of Bounds. https://github.com/google/security-research/security/advisories/GHSA-fphp-6498-x998, 2024. Accessed: 2025/09/17.