Exploring Advanced eBPF Capabilities: Malicious Tracing, Network Rate Limiting, and Process Hiding

Ajitesh Kumar Singh
IIIT Bangalore
Bangalore, India
ajitesh.kumar@iiitb.ac.in

***Abstract*—Extended Berkeley Packet Filter (eBPF) enables safe, dynamic execution of custom programs within the Linux kernel without requiring kernel module loading or system restarts. This work explores three practical applications of eBPF to understand its capabilities for observation, enforcement, and stealth within modern systems. The first project demonstrates malicious tracing by intercepting the `openat` and `read` syscalls used by the `sudo` executable, modifying in-memory file contents to inject temporary privilege escalation rules into the `/etc/sudoers` file without altering data on disk. The second project implements a per-source IPv4 rate-limiting mechanism using a token-bucket algorithm attached at the TC ingress path, enabling real-time network traffic control with event reporting through a ring buffer. The third project focuses on process concealment by hooking the `getdents64` syscall and removing directory entries corresponding to a selected process ID, effectively hiding it from user-mode tools such as `ps`, `top`, and `ls /proc`. Collectively, these explorations illustrate the flexibility and power of eBPF to instrument kernel behavior, support performance and security mechanisms, and reveal both defensive and offensive implications worth consideration.**

***Index Terms*—eBPF, syscall tracing, network rate limiting, process hiding, kernel instrumentation**

I. Introduction

Extended Berkeley Packet Filter (eBPF) is a technology built into the Linux kernel that enables the safe execution of user-defined programs within kernel space. Originally introduced as a packet filtering mechanism, eBPF has evolved into a general-purpose framework for system introspection, performance monitoring, networking, and security. It allows dynamic instrumentation without requiring kernel modules or system reboot, while maintaining strong safety guarantees through verification and restricted execution environments.

Modern operating systems increasingly rely on eBPF due to its efficiency and flexibility. Observability platforms, security monitoring tools, networking frameworks, and container runtime systems integrate eBPF programs to provide deep insight and control with minimal performance overhead. At the same time, these capabilities introduce opportunities for both defence and offensive research, revealing how powerful kernel-space programmability can influence system behavior.

This work focuses on three experimental projects developed to explore the capabilities of eBPF across different functional domains. The first project demonstrates a malicious tracing technique that intercepts critical syscalls from the `sudo` program, altering in-memory file data to simulate privilege escalation without modifying persistent storage. The second project implements a per-source IPv4 rate limiter by attaching an eBPF program to the Traffic Control (TC) ingress path, enforcing a token-bucket algorithm directly in the kernel. The third project explores process hiding by hooking the `getdents64` syscall, removing directory entries to conceal selected process identifiers from user-space tools.

Together, these projects illustrate how eBPF can observe, manipulate, and enforce kernel-level behavior, highlighting both its potential for legitimate system enhancement and its implications for adversarial misuse.

II. Background

eBPF extends the capabilities of the original Berkeley Packet Filter by allowing verified programs to execute safely inside the Linux kernel. These programs can be dynamically attached to various kernel hooks, such as tracepoints, kprobes, uprobes, network data paths, and control plane components. The eBPF verifier ensures that loaded programs do not compromise kernel stability by enforcing strict constraints on memory access, loops, and execution flow.

A fundamental advantage of eBPF is the ability to observe or modify kernel-level operations without resorting to kernel modules, which traditionally posed risks due to direct access to kernel memory and the possibility of system crashes. By contrast, eBPF programs operate within a restricted environment and interact with user-space through shared maps or ring buffers for data exchange.

Several subsystems inside Linux now integrate eBPF as a core component. Examples include:

- **BPF tracing** for performance profiling through tracepoints, kprobes, and uprobes.
- **BPF networking** for packet filtering and load balancing through XDP and TC.
- **Security monitoring** through tools such as Falco and Cilium.
- **Observability frameworks** such as bpftrace and perf.

eBPF programs can attach to system calls at both entry and exit points, enabling capture of input arguments and returned values. Additionally, helper functions such as `bpf_probe_read_user` and `bpf_probe_write_user` allow reading from and writing to user-space memory locations when pointer arguments reference them. This introduces opportunities to implement advanced behavior that is traditionally difficult without kernel modification.

Maps such as `BPF_MAP_TYPE_HASH`, `BPF_MAP_TYPE_RINGBUF`, and `BPF_MAP_TYPE_PROG_ARRAY` support persistent state tracking, asynchronous event forwarding, and tail-call execution chaining. Combined with the auto-generated skeletons produced by `libbpf`, eBPF provides a complete environment for kernel-space experimentation and research.

The projects presented in this work leverage these mechanisms to investigate three different capabilities: syscall interception and data modification, real-time network traffic enforcement, and concealment of kernel process metadata. Each demonstrates distinct aspects of eBPF functionality and highlights the balance between legitimate system control and potential malicious exploitation.

III. System Design and Implementation

This section describes the design, objectives, and implementation details of the three eBPF projects developed as part of this work. Each project showcases a different aspect of kernel-space instrumentation, focusing on syscall tracing, network enforcement, and process concealment.

A. *Malicious Tracing and In-Memory sudoers Manipulation*

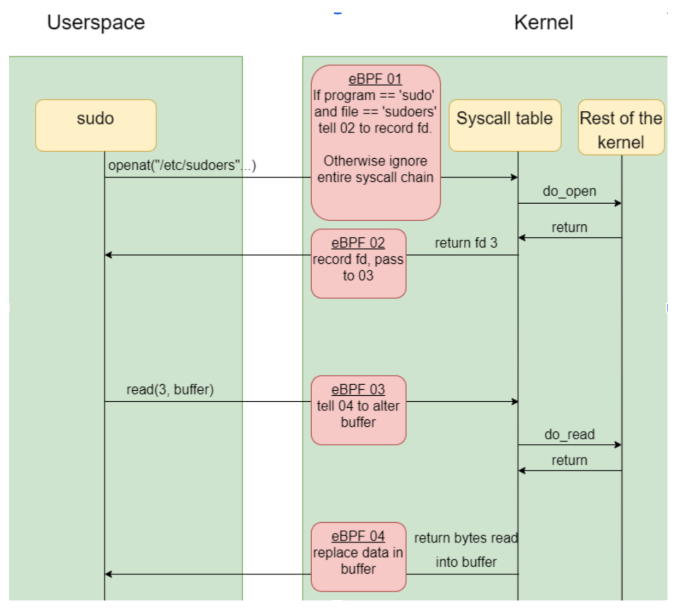

Fig. 1. Syscall flow showing eBPF interception of `openat` and `read` to modify sudoers data in memory.

The first project investigates the use of eBPF for syscall-level tracing and runtime data modification. The program attaches eBPF tracepoint handlers to the `sys_enter_openat`, `sys_exit_openat`, `sys_enter_read`, and `sys_exit_read` system calls invoked by the `sudo` binary. By filtering based on the process name and the filename `/etc/sudoers`, the program selectively captures the file descriptor and user memory buffer associated with read operations.

Once a matching read call is detected, the eBPF program uses `bpf_probe_write_user` to overwrite the returned buffer, inserting a fabricated privilege rule such as:

```
apache ALL=(ALL:ALL) NOPASSWD:ALL
```

This modification occurs only in memory and never touches the on-disk file, making the escalation temporarily visible only to the running `sudo` process. Events are communicated back to userspace through a ring buffer, allowing real-time monitoring of success or failure. This project demonstrates the risk potential of syscall interception and nondestructive privilege manipulation.

B. *Per-Source IPv4 Rate Limiter Using TC Ingress*

The second project implements a network packet rate limiter at the ingress path using the Linux Traffic Control (TC) subsystem. An eBPF program is attached to the TC hook via the `bpf_tc_attach` API, enabling packet inspection and control at the earliest stage of processing.

The implementation uses a token-bucket algorithm stored in a `BPF_MAP_TYPE_HASH` map keyed by source IPv4 address. For each packet, tokens are replenished based on elapsed time using `bpf_ktime_get_ns()` to compute packet rate against the configured limit. Packets arriving without available tokens are dropped by returning `TC_ACT_SHOT`, and an event containing the source IP and drop counter is forwarded to userspace through a ring buffer.

The user-mode controller allows runtime configuration via parameters for allowed packets-per-second, burst size, and network interface selection. This project demonstrates high-performance, real-time packet enforcement directly inside the kernel without requiring iptables, nftables, or external modules.

C. *Process Hiding via getdents64 Syscall Hook*

The third project explores stealth behavior by hiding a process ID from user applications. The program attaches to the `getdents64` syscall, which is used by tools such as `ps`, `top`, and `ls /proc` to enumerate directory entries representing processes.

Using tail calls and a `BPF_MAP_TYPE_PROG_ARRAY`, the implementation splits logic across multiple eBPF programs. After intercepting the syscall exit path, the program scans the returned directory buffer and removes entries matching the configured PID. This modification removes visibility of the selected process only from userspace, while keeping kernel state intact. Status events are again sent through a ring buffer for feedback.

This experiment highlights the potential misuse of eBPF for implementing stealth and evasion techniques traditionally associated with kernel rootkits, while maintaining verifier constraints and without requiring kernel module injection.

IV. Results and Observations

The implementation and testing of the three eBPF projects provided insight into both the power and the limitations of kernel-space instrumentation. Each project was evaluated on a Linux environment with eBPF and `libbpf` support, using real system utilities and network traffic to validate behavior.

A. Malicious Tracing and sudo Privilege Injection

The sudo manipulation experiment successfully demonstrated that eBPF programs attached to syscall tracepoints can modify user memory buffers returned from the kernel. When executing `sudo -l` as a non-privileged user, the in-memory modification caused the program to interpret an injected entry as part of the `/etc/sudoers` file, granting apparent root access without modifying the actual file on disk. Standard administrative tools such as `cat /etc/sudoers` and audit frameworks did not detect any change, confirming that the attack surface exists only at runtime. The ring buffer events reliably reported whether the payload was successfully inserted and which process instance was affected.

B. Network Rate Limiting with TC Ingress

The rate limiter demonstrated accurate enforcement of per-source packet limits under traffic load. During testing with packet generators such as `ping -f` and `iperf`, packets exceeding the configured threshold were dropped consistently according to the token-bucket policy. Real-time drop notifications were streamed to userspace, showing increasing counters for aggressive sources while legitimate traffic continued unaffected. Adjusting rate and burst parameters at startup provided predictable scaling behavior.

C. Process Hiding Using getdents64

The PID hiding project successfully prevented selected process IDs from appearing in directory listings under `/proc`. Tools including `ps`, `top`, `htop`, and `ls /proc` did not display the hidden PID, confirming the syscall data filter operated transparently. The process continued to run normally and remained observable via kernel-level interfaces such as `dmesg` or `strace`, indicating that only user-mode views were impacted. Tail-call chaining enabled modular handling of patching logic without exceeding verifier constraints. The ring buffer confirmed which processes attempted reads and whether the hide operation succeeded.

Overall, the results demonstrate that eBPF can be used effectively for both defensive and offensive manipulation of kernel-exposed information. The experiments highlight the dual-use nature of eBPF: offering valuable capabilities for security enforcement and monitoring, while also enabling techniques commonly associated with rootkit functionality if misused.

V. Conclusion

This work presented three practical explorations of eBPF demonstrating its ability to instrument kernel behavior for tracing, enforcement, and concealment. Across the malicious tracing project, the per-source IPv4 rate limiter, and the PID hiding syscall hook, the experiments show how eBPF provides fine-grained control and visibility inside the Linux kernel without the need for kernel modules or persistent system modification.

The sudo privilege manipulation experiment illustrated how eBPF can intercept system calls and alter user-space memory buffers, enabling temporary privilege escalation that is invisible to disk-based integrity checks. The network rate limiter demonstrated high-performance packet control using a token-bucket model at the TC ingress stage, offering efficient enforcement with real-time feedback. The PID hiding project revealed how selectively patching `getdents64` results can conceal processes from user-space enumeration tools while leaving kernel state intact.

These outcomes highlight both the defensive and offensive potential of eBPF. While it can enable powerful monitoring, security enforcement, and performance optimization, it can also be misused to create stealthy and difficult-to-detect behaviors. Understanding these dual implications is increasingly important as eBPF adoption expands across modern software stacks. Future work may involve eBPF-based detection of malicious BPF usage, integration with LSM hooks for access control, or extending network control through XDP for higher throughput. Overall, the projects reinforce eBPF as a flexible and influential technology for kernel-level experimentation and security research.