GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

System Call Filtering with eBPF

Anila Ghazanfar

MatrNr: 11351586

Supervisor: Hendrik Nolte

Georg-August-Universität Göttingen
Institute of Computer Science

April 1, 2025

Abstract

This report presents an eBPF-based security framework for Linux system call filtering, demonstrating significant advantages over traditional seccomp mechanisms. An eBPF-based implementation is developed that successfully enforces process-aware security policies, with specific capability to block write operations from bash while permitting other processes. The solution effectively utilizes BPF CO-RE (Compile Once - Run Everywhere) technology to overcome critical portability challenges in security tooling across kernel versions. Experimental results reveal important operational characteristics and limitations of the approach, particularly concerning synchronous hook implementations. The findings establish eBPF as a superior solution for dynamic, context-sensitive system call filtering compared to static alternatives.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 10% of the entire text
- ☐ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☑ Further, namely: - Initial research about eBPF and understanding code segments

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computing

OS Operating System

syscall system call

1 Introduction

Modern Operating System (OS) run numerous untrusted applications while relying on a trusted OS kernel. Every application interacts with the OS kernel via the syscall interface. As a result, securing syscall is crucial for safeguarding a shared kernel from untrusted user processes. Syscall filtering serves as a widely adopted security mechanism for syscall. The core concept involves limiting the syscalls a process can execute based on predefined security policies, thereby minimizing the attack surface. This filtering occurs at the entry point of each syscall to determine whether it should be permitted or denied.

1.1 Early syscall filtering mechanisms

Early syscall filtering mechanisms, such as Janus [Wag99] and Ostia [GPR+04], relied on trusted userspace agents to enforce security policies for syscalls. However, these approaches introduced substantial performance overhead due to frequent context switches between user space and the kernel for every syscall. Additionally, they were vulnerable to race conditions, such as time-of-check-to-time-of-use (TOCTTOU) attacks, which could lead to security breaches [Gar03; PG12; BTP22].

1.2 Seccomp (SECure COMPuting)

To mitigate these issues, modern techniques like Linux Seccomp (SECure COMPuting) were developed, running entirely within the kernel to eliminate context switch overhead. Seccomp is widely adopted across various applications, including Android app isolation [Law17], systemd user process sandboxing [Cor12], and lightweight virtualization technologies like Docker [Doc], gVisor [gVi], and Kubernetes [Kub]. Despite its effectiveness, Seccomp's programmability remains a significant limitation. The classic BPF (cBPF) language used in Seccomp's filter mode allows only static allow lists and lacks *stateful* processing capabilities. This is mainly because cBPF lacks the ability to store states; hence, its filters remain *stateless*. Additionally, cBPF cannot interact with other kernel utilities or invoke additional BPF programs, making it unsuitable for enforcing more *complex security policies* without extensive kernel modifications.

1.3 eBPF (extended Berkley Packet Filters)

To address these limitations, eBPF (extended BPF) offers a powerful alternative for syscall filtering which will be discussed throuhout the report. Unlike cBPF, eBPF provides enhanced programmability, supporting stateful filtering, dynamic policy enforcement, and seamless interaction with other kernel components. With eBPF, syscall filtering can be performed with fine-grained control, enabling sophisticated security mechanisms without requiring modifications to the kernel. This makes eBPF an attractive solution for modern security frameworks, providing both flexibility and efficiency in syscall security enforcement.

1.4 Contributions

This report makes the following key contributions:

- **eBPF-Based Security Framework:** Developed a dynamic syscall filtering system using eBPF, surpassing static seccomp by enabling process-aware policies (e.g., blocking bash writes).
- **BPF CO-RE Portability:** Leveraged BPF CO-RE for cross-kernel compatibility, resolving dependency issues in security tooling.
- **Practical Insights:** Identified limitations (e.g., synchronous hooks).

This report is organized into six main sections to systematically explore eBPF-based system call security. Section 1 introduces the evolution of syscall filtering, from early mechanisms to modern eBPF, and outlines the report's contributions. Section 2 establishes the background, emphasizing the critical role of syscall security in Linux. Section 3 delves into eBPF programming, covering maps, hooks, and the trace pipe, while Section 4 details the eBPF virtual machine and instruction set. Section 5 explains the workflow of an eBPF program, bridging theory and implementation. Finally, Section 6 presents two experimental setups: the first demonstrates blocking bash writes using eBPF, and the second recreates another variant of the experiment 1 using BPF LSM.

2 Background

2.1 Importance of Syscall security

Syscall security is important for several reasons:

- **Enhanced Security:** Syscall filtering reduces the attack surface by restricting the syscalls that applications can invoke, making it more difficult for malicious actors to exploit kernel vulnerabilities.
- **Improved Performance:** By filtering unnecessary syscalls, the overall communication between user applications and the kernel is optimized, leading to reduced processing overhead and improved efficiency.
- **Fine grained Control:** Syscall filtering allows enforcement at different levels, including individual processes, users, or applications, enabling more precise security policies.
- **Resource Management:** By efficiently handling and prioritizing syscall requests at the kernel level, filtering mechanisms help mitigate resource exhaustion attacks and improve system stability.
- **Application specific restrictions:** Custom security policies can be defined per application, ensuring that each application has access only to the syscalls necessary for its functionality while restricting unnecessary or potentially harmful interactions.

2.2 Activities that require Syscall security

From a security perspective, several key activities require monitoring and control. These activities can be broadly categorized into four main areas:

- **Network access:** It is essential to detect suspicious network activity, such as unexpected outbound traffic to a remote server, which may indicate malware communication or data exfiltration.
- **File access:** Applications should only access files that are within their designated permissions, preventing unauthorized data access or modification.
- **Memory access:** Executable integrity must be maintained by ensuring that applications run only approved binaries, reducing the risk of executing malicious code.
- **Process or privileges** Detecting unauthorized privilege escalation attempts is critical, as attackers often exploit system vulnerabilities to gain elevated privileges and execute malicious operations.

All these security-critical activities depend on kernel support. While applications operate in user space, any interaction with hardware or privileged system functions requires assistance from the kernel, which is facilitated through the syscall interface (see Figure 1). By implementing syscall filtering, it is possible to enforce strict security policies, thereby strengthening system integrity and reducing the risk of exploitation.

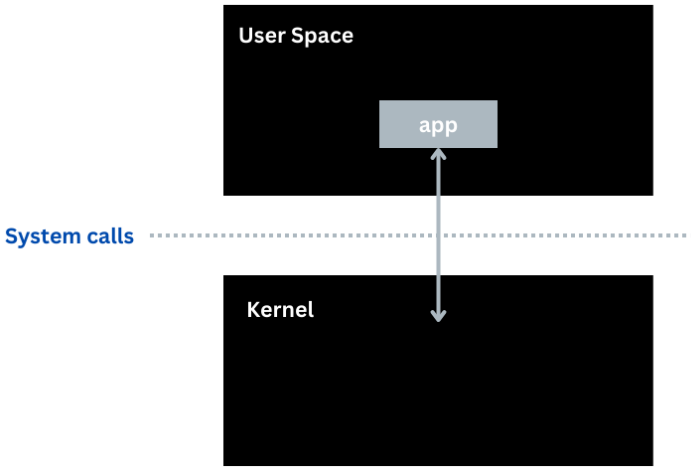

Figure 1: System call

3 eBPF

eBPF is a technology that allows to run sandboxed programs in the Linux kernel without modifying kernel source code or loading kernel modules. It is widely used for networking, observability, and security. eBPF programs are typically written in a restricted subset of C or Rust. eBPF introduces a set of fundamental concepts which include:

- **eBPF Programs:** Small, event-driven programs that execute within the kernel in response to predefined triggers, such as system calls, network events, or performance monitoring hooks. These programs run in a restricted environment, ensuring security and stability while extending kernel functionality.

- **Maps:** Efficient key-value data structures that facilitate communication between user space and kernel space. Maps allow eBPF programs to store and retrieve data persistently, enabling dynamic and stateful processing of events.
- **Helpers:** Predefined kernel-provided functions that assist eBPF programs in interacting with kernel data structures, managing memory, and performing specialized tasks. These helper functions ensure that eBPF programs can operate safely without directly modifying kernel code.
- **BPF CO-RE (Compile Once – Run Everywhere):** A mechanism that enhances the portability of eBPF programs across different Linux kernel versions. By abstracting kernel-specific details, BPF CO-RE allows eBPF programs to be compiled once and executed on multiple systems without requiring kernel version-specific modifications.

These core concepts make eBPF a versatile and powerful framework for extending kernel capabilities while maintaining performance, security, and portability.

3.1 eBPF Programming

To get grasp of eBPF programming, a basic 'Hello World' example is explained in this section. Listing 1 is the source code of the hello_world.py written using BCC's Python library.

```
#!/usr/bin/python
from bcc import BPF
program = r"""
int hello(void *ctx) {
bpf_trace_printk("Hello World!");
return 0;
}
"""
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")
b.trace_print()
```

Listing 1: "Hello, world!" in eBPF

This code is composed of two main components:

1. **the eBPF program**, which executes within the kernel. As illustrated in Figure 2, hello() is the eBPF program running in the kernel.
2. **the user space code** responsible for loading the eBPF program into the kernel and retrieving the generated trace data. As illustrated in Figure 2, hello_world.py represents the user space component of the application.

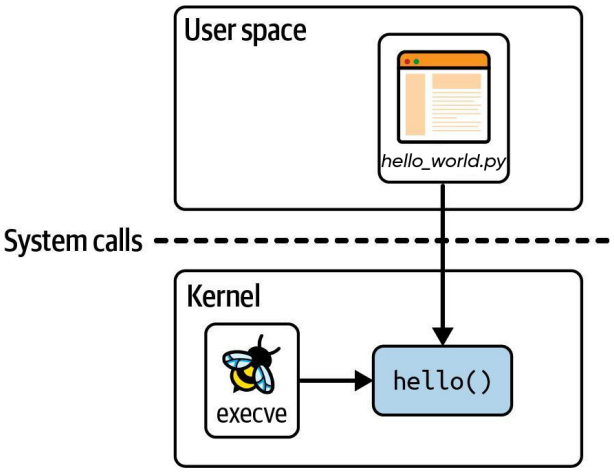

Figure 2: Components of eBPF Program **Source:** [Ric23]

Here is the brief explanation of the code. The first line indicates that this is a Python script and specifies the Python interpreter (/usr/bin/python) as the program responsible for executing it.

Lines 3 to 8 in the Listing 1, are part of eBPF program. The complete eBPF program is represented as a string named program. The eBPF program simply utilizes the helper function bpf_trace_printk() to output a message 'Hello World!'. This C program must be compiled before execution; however, BCC handles the compilation process automatically.

To use this string in the BPF framework, simply pass it as a parameter when creating a BPF object, as shown in the line 9.

eBPF programs must be linked to a specific **event**. In this example at line 10, the program is attached to the *execve* system call, which is responsible for executing a program. In this case, *syscall* refers to the name of the kernel function to which ePBF program (hello() function) will be attached using a kprobe.

At line 11, hello function is attached to that event using kprobe. At this stage, the eBPF program has been loaded into the kernel and attached to an event, meaning it will be triggered each time a new executable is launched on the machine.

At line 12, The trace_print() function will run in an infinite loop (until the program is stopped, potentially with Ctrl+C), continuously displaying any trace information. The bpf_trace_printk() helper function in the kernel always directs its output to a specific predefined pseudofile location: /sys/kernel/debug/tracing/trace_pipe.

3.1.1 Single trace pipe location

A single trace pipe is sufficient for simple examples or basic debugging, but it's limited in flexibility and only supports string output, making it unsuitable for structured data. Additionally, multiple eBPF programs writing to the same trace pipe can create confusion. A more effective way to retrieve information from an eBPF program is by using an eBPF map.

3.2 Maps

An eBPF map is a high-level data structure, often referred to as "BPF maps". These data structures enable data exchange between multiple eBPF programs or between user

space applications and kernel code. eBPF programs running within the virtual machine can access one or more maps using platform-specific load instructions.

Maps are key-value stores and come in various types, such as arrays (with a 4-byte index as the key) and hash tables (which can use arbitrary data types as keys). There are also specialized types optimized for operations like FIFO queues, LRU storage, longest-prefix matching, and Bloom filters.

3.3 Hooks

The execution of an eBPF program is event-driven, triggered when the kernel or an application reaches specific hook points. These predefined hooks are strategically placed throughout the kernel, covering various events such as system calls, function entry and exit, network activity, and tracepoints. Depending on its attach type or program type, an eBPF program is linked to the appropriate hook for execution. When predefined hooks are insufficient, developers can define custom attachment points using **kernel probes (kprobes)** or **user probes (uprobes)**, allowing eBPF programs to be attached to nearly any location within the kernel or user applications.

In the following section, we will delve into the anatomy of an eBPF program, examining its components, structure, and the way it is constructed to achieve its intended tasks within the kernel environment. The program's internal architecture plays a crucial role in defining its behavior and how it interfaces with the kernel.

4 Anatomy of an eBPF program

An eBPF program consists of a series of eBPF bytecode instructions. While it is possible to write eBPF code directly in this bytecode, similar to programming in assembly language, most developers find higher-level programming languages more manageable. Conceptually, this bytecode is executed within an eBPF virtual machine running inside the kernel. Figure 3 illustrates the stages an eBPF program undergoes, from source code to execution.

Figure 3: eBPF program C (or Rust) source code is compiled into eBPF bytecode, which is subsequently either JIT-compiled or interpreted into native machine code instructions

Source: [Ric23]

4.1 eBPF Virtual Machine

The eBPF virtual machine is a software-based execution environment that processes eBPF bytecode instructions. Initially, these instructions were interpreted within the kernel, but for better performance and security, JIT (just-in-time) compilation is now commonly used. This ensures that bytecode is converted into native machine instructions only once at load time, reducing execution overhead. The eBPF instruction set and register model

are designed to align closely with common CPU architectures, simplifying the translation from bytecode to machine code.

4.2 eBPF Registers and Instructions

The eBPF instruction set is defined within the eBPF virtual machine and supports both 64-bit and 128-bit instruction encoding. It includes general-purpose instructions for arithmetic operations, jumps, calls, loads, and stores. The instruction set utilizes 11 dedicated 64-bit registers (r0–r10) with a structured calling convention, where r10 is read-only and points to the top of the stack. For a comprehensive formal specification, refer to the eBPF Instruction Set Specification [Ker].

A core design principle of the eBPF instruction set is its close alignment with hardware instruction set architectures (ISA). This design choice simplifies the implementation of interpreters and JIT compilers while enabling optimizing compiler backends to generate eBPF assembly code with performance comparable to natively compiled programs. Because of this near-equivalence, JITs can efficiently translate eBPF instructions to native machine instructions with minimal overhead, often using a direct one-to-one mapping.

5 Workflow of an eBPF Program

In this section, the high level work flow of an eBPF program is presented. Figure 4 illustrates the sequence involved in the workflow of executing an eBPF program which are described below:

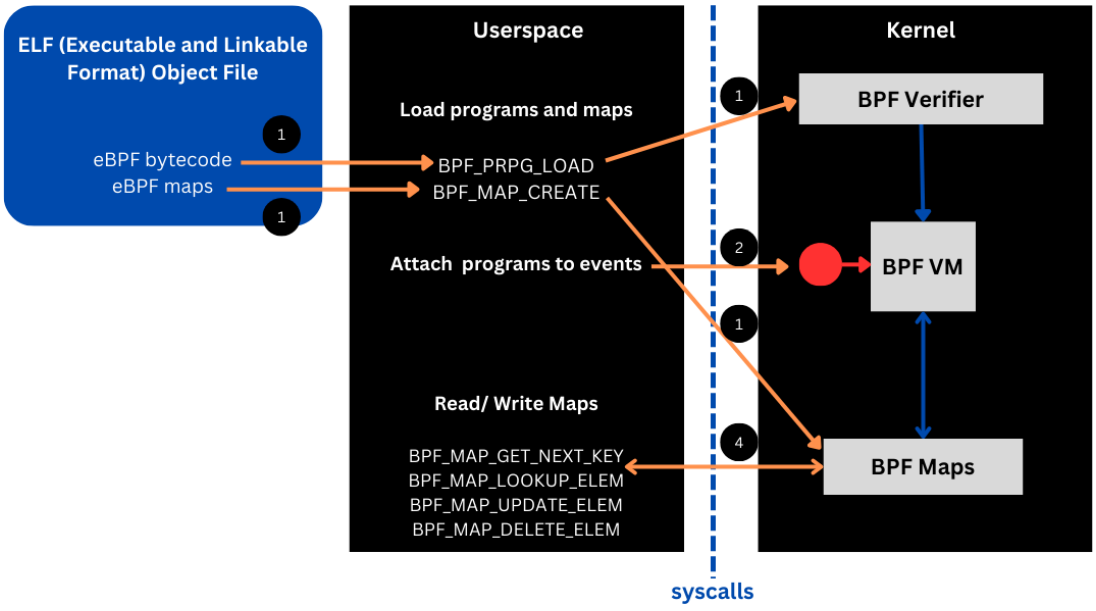

Figure 4: Workflow of an eBPF program

- **Compilation (Step 0):** The eBPF program starts as a C source file, which is compiled using the LLVM toolchain's clang compiler. The target architecture is set to bpf, instructing the compiler to generate eBPF bytecode. When writing eBPF programs, the C code is compiled into an **ELF object file**, which contains eBPF

bytecode and map definitions. Executable and Linkable Format (ELF) object file is a binary file format used in Unix-based operating systems (like Linux) for executables, shared libraries, and object files.

- **Loading into the Kernel (Step 1):** The compiled object file, including the program and associated maps, is submitted to the kernel via the bpf(2) system call using the BPF_PROG_LOAD command. This triggers the eBPF verifier, which checks whether the program is safe to execute within the kernel.
  - If the program fails verification, it is rejected, and an error is returned to user space.
  - If verification succeeds, the program is Just-In-Time (JIT) compiled, and a file descriptor corresponding to the eBPF program is returned to user space.
- **Attaching to an Event (Step 2):** Once the user space application receives the file descriptor, it attaches the eBPF program to a specific event. This event serves as the trigger for executing the eBPF program.
- **Execution and Data Retrieval (Step 3):** When the defined event occurs, the eBPF program runs. User space applications can retrieve information from eBPF maps through system calls. Once the eBPF link file descriptor is closed, the program is detached from its hooks, and the kernel frees the associated resources, including memory allocated for the program.

6 Experiments

For the experiments, the use case under consideration is Restricting Write Syscalls from Bash Using eBPF.

6.1 Objective:

The goal of this experiment is to use eBPF to detect and block write system calls when executed from bash. This aims to enhance security by preventing unauthorized write access from bash.

6.2 Experiment Setup 1:

- **OS:** Linux system with eBPF support (CONFIG_BPF_SYSCALL=y)
- **Kernel Version:** 5.x or newer
- **Tools Used:**
  - **BCC (BPF Compiler Collection)** for writing and attaching the eBPF program
  - **bpftool** for loading the eBPF bytecode
  - **tracepoints** (raw_syscalls:sys_enter) to monitor system calls

6.3 Implementation:

6.3.1 Writing the eBPF Program:

- The program hooks into the raw_syscalls:sys_enter tracepoint, which is triggered whenever a system call is invoked. See line 31 in Listing 2 where 'syscall_filter' function is attached to the raw_syscalls:sys_enter tracepoint.
- It captures the syscall ID and the process name (comm) using bpf_get_current_comm(). Lines 10 and 12 of the Listing 2 correspond to it.
- If the process name matches "bash" (lines 16 and 17), the program prints a message indicating that a restricted syscall was invoked.
- If the syscall ID corresponds to openat (257), write (1), or open (2), the program attempts to return -1 to block it (lines 20 to 22).

6.3.2 Loading the eBPF Program:

- The BPF program is compiled and loaded into the kernel using bcc.
- It attaches to raw_syscalls:sys_enter to intercept all system calls (line 31 in Listing 2).
- The program outputs detected system calls (see Figure 5) using bpf_trace_printk() (line 34 in Listing 2).

6.3.3 Execution:

- The experiment was run while executing echo "Hello from bash" > test.txt in bash as in Figure 6.
- The expected behavior was that write syscalls from bash would be blocked.
- However, the program **only detected the syscalls but did not block them**, since tracepoints do not allow modifying syscall behavior.

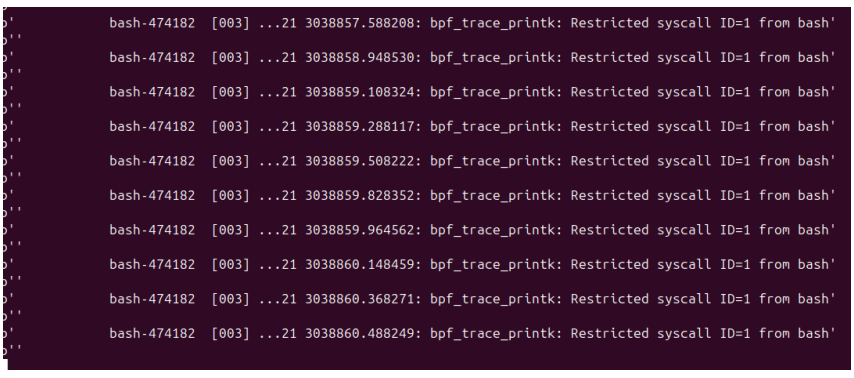

Figure 5: trace mesages

```
[cloud@anila-seminar-seccomp ~]$ echo 'Hello from bash' > test.txt
[cloud@anila-seminar-seccomp ~]$ cat test.txt
Hello from bash
```

Figure 6: Testing block write syscalls

6.3.4 Findings:

- Tracepoints (raw_syscalls:sys_enter) can only observe system calls but cannot modify their execution.
- Returning -1 inside a tracepoint has no effect on syscall execution.
- The program successfully detected bash-initiated syscalls but failed to enforce blocking.

6.3.5 Next Steps for Improvement:

For improvement, there are few other options to consider:

- Use LSM Hooks (BPF_PROG_TYPE_LSM) to properly restrict read and write.
- Use kprobes (kprobe/sys_read and kprobe/sys_write) as an alternative to tracepoints.
- Experiment with seccomp-bpf, which is explicitly designed for syscall filtering.

For improvement, another version of the code using BPF LSM is implemented. BPF LSM extends the LSM framework by allowing developers to write eBPF programs that attach to LSM hooks dynamically.

6.4 Experimental Setup 2:

- **OS:** Linux system with eBPF support (CONFIG_BPF_SYSCALL=y)
- **Linux Kernel:** >= 5.8 (for LSM BPF hooks support)
- **Tools Used:**
  - **Eunomia BPF:** A framework for writing and deploying eBPF programs with CO-RE (Compile Once, Run Everywhere).
  - **bpftool:** For managing and loading eBPF programs.
  - **clang** (>= v10.0) + **llvm** (for BPF bytecode compilation)
  - **libbpf** (for BPF CO-RE support)

6.4.1 Kernel Configuration:

Ensure the kernel supports:

- eBPF JIT compilation (CONFIG_BPF_JIT=y)
- LSM BPF hooks (CONFIG_BPF_LSM=y)

- Kernel debugging symbols (CONFIG_DEBUG_INFO=y) for BPF CO-RE
- Verify with:

```
grep -E "BPF_JIT|BPF_LSM|DEBUG_INFO" /boot/config-$(uname -r)
```

6.5 Implementation:

6.5.1 Synchronous Hook Implementation:

A hook is "synchronous" when the eBPF program executes immediately during the kernel operation it monitors (e.g., a security check like file_permission). The kernel pauses its workflow, runs the eBPF code, and resumes only after the hook completes. This eBPF program uses a synchronous LSM hook (lsm/file_permission) to:

- Check if the calling process is bash (via bpf_get_current_comm).
- Validate the file operation (mask & MAY_WRITE) for bash while allowing terminal devices (/dev/pts/*, /dev/tty*).
- Synchronously allow/deny the syscall by returning -EPERM or ret.
- Log blocked attempts via bpf_trace_printk.

6.5.2 Key Components of the Program (Listing 1):

Table 1 shows key components of the C code that restricts write syscalls in bash using BPF LSM.

Table 1: Key components of the code utilizing LSM Hooks

Component	Function	Line
LSM Hook	SEC("lsm/file_permission") attaches to kernel's file-permission checks	15
Process Filtering	bpf_strncmp(comm, "bash") ensures only bash is targeted	22
File Write Check	mask & MAY_WRITE detects write operations	26
Device Exclusion	BPF_CORE_READ(inode, i_rdev) != 0 skips terminal devices	35
Filename Extraction	BPF_CORE_READ(dentry, d_name.name) reads the filename safely	53
Logging	bpf_trace_printk("Blocked write: %s", name) logs to /sys/kernel/debug/tracing/trace_pipe	62

6.5.3 Execution:

- The experiment was run while executing echo "Hello from bash" > files/test.txt in bash as in Figure 7.
- The behavior is as expected, that is, blocking write syscalls from bash.
- But when some text is written to files using python or shell, it was allowed which is the expected behavior.

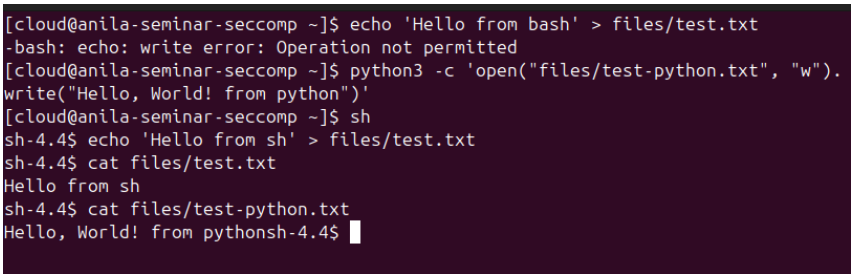

Figure 7: BPF LSM blocking write syscalls

6.6 Limitations:

There are few limitations to the BPF programs:

- **Performance Overhead:** LSM hooks run synchronously—may impact I/O-heavy workloads.
- **Kernel Dependency:** Requires BPF LSM support (Linux >= 5.8).
- **Security Bypass:** A malicious process could rename itself to bypass comm checks. This problem is specific to both variants of code used in this report.

```
#include "vmlinux.h"
#include <linux/errno.h>     // For EPERM
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

#ifndef MAY_WRITE
#define MAY_WRITE 0x2
#endif

#ifndef TASK_COMM_LEN
#define TASK_COMM_LEN 16
#endif

SEC("lsm/file_permission")
    int BPF_PROG(file_permission, struct file *file, int mask, int ret)
```

```
17      {
18          char comm[TASK_COMM_LEN];
19          bpf_get_current_comm(comm, sizeof(comm));
20
21          // Check if the process is "bash"
22          if (bpf_strncmp(comm, TASK_COMM_LEN, "bash") != 0) {
23                return ret; // Allow non-bash processes
24          }
25          // Check if the operation is a write
26          if (mask & MAY_WRITE) {
27
28              // Get the file's inode and device
29              struct inode *inode = BPF_CORE_READ(file, f_inode);
30              if (!inode) {
31                  return ret; // Skip if inode is invalid
32              }
33
34              // Exclude terminal devices (e.g., /dev/pts/*, /dev/tty*)
35              if (BPF_CORE_READ(inode, i_rdev) != 0) {
36                  return ret; // Allow writes to device files
37              }
38
39              // Log the blocked write attempt
40              char fmt[] = "Blocked write syscall from bash: %s\n";
41              char name[256] = {0};  // Buffer for filename
42
43              // Use BPF CO-RE to safely read the file name
44              const char *filename_ptr = NULL;
45
46              // Read the dentry pointer from the file struct
47              struct dentry *dentry = BPF_CORE_READ(file, f_path.dentry);
48              if (!dentry) {
49                  return ret; // Skip if dentry is invalid
50              }
51
52              // Read the filename from the dentry
53              filename_ptr = BPF_CORE_READ(dentry, d_name.name);
54              if (!filename_ptr) {
55                  return ret; // Skip if filename is invalid
56              }
57
58              // Safely read the filename into the buffer
59              bpf_probe_read_kernel_str(name, sizeof(name), filename_ptr);
60
61              // Log the blocked write attempt
62              bpf_trace_printk(fmt, sizeof(fmt), name);
63
```

```
64              return -EPERM;  // Deny write operation
65          }
66
67          return ret; // Allow other operations
68      }
69
70  char _license[] SEC("license") = "GPL";
```

Listing 1: Blocking write syscalls using LSM hooks

7 Discussion

eBPF is a powerful Kernel technology. I have worked on my use case (restricting bash sys calls) first, by attaching eBPF program to tracepoint as a hook point and another experiment by connecting to BPF LSM hook point.

The experiments show that when eBPF programs are attached to tracepoints they act as passive and do not actively block syscalls. BPF LSM hooks actively block sys calls from bash. In terms of passivity, eBPF is generally considered to be a **passive** technology, meaning that it does not actively enforce security policies or modify system behavior. Instead, eBPF programs are executed in response to specific events or requests, and they can only observe and report on system behavior.

On the other hand, BPF LSM is an **active** technology, meaning that it actively enforces security policies and modifies system behavior. BPF LSM uses eBPF programs to define security policies, which are then enforced by the kernel. This means that BPF LSM can actively prevent or allow certain system calls, network connections, or other events from occurring, based on the defined security policies.

8 Conclusion

eBPF provides a powerful and efficient way to filter system calls, making it a valuable tool for security and observability in modern Linux environments. In this report, I explored the fundamentals of eBPF, system call hooking mechanisms, and the structure of an eBPF program. Using BCC, I developed a simple eBPF program to show how a minimal eBPF program works.

As a practical hands on experiment, I worked on use case—restricting write syscalls from Bash. This use case highlights eBPF's ability to enforce security policies at the kernel level with minimal overhead. When attached to LSM hooks or kprobes, eBPF can actively restrict syscalls, providing fine-grained control over process behavior. However, despite its advantages, eBPF has a learning curve due to its intricate programming model and kernel-level constraints.

As eBPF continues to evolve, its role in system security will expand, offering new ways to enhance access control, intrusion detection, and policy enforcement with greater efficiency than traditional methods.

References

[BTP22] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. "Midas: Systematic kernel {TOCTTOU} protection". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 107–124.

[Cor12] Jonathan Corbet. *Systemd gets seccomp filter support*. `https://lwn.net/Articles/507067/`. Accessed: 2025-3-21. July 2012.

[Doc] Docker. *Seccomp security profiles for Docker*. `https://docs.docker.com/engine/security/seccomp/`. Accessed: 2025-3-24.

[Gar03] Tal Garfinkel. "Traps and pitfalls: Practical problems in system call interposition based security tools". In: *In Proc. Network and Distributed Systems Security Symposium*. 2003.

[GPR+04] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. "Ostia: A delegating architecture for secure system call interposition." In: *NDSS*. Citeseer. 2004.

[gVi] gVisor. *gVisor*. `https://docs.docker.com/engine/security/seccomp/`. Accessed: 2025-3-24.

[Ker] Kernel. *eBPF Instruction Set Specification, v1.0*. `https://docs.kernel.org/6.3/bpf/instruction-set.html`. Accessed: 2025-3-24.

[Kub] Kubernetes. *Kubernetes*. `https://kubernetes.io/docs/tasks/configure-pod-container/security-context/`. Accessed: 2025-3-24.

[Law17] Paul Lawrence. *Seccomp filter in Android O*. `https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html`. Accessed: 2025-3-21. July 2017.

[PG12] Mathias Payer and Thomas R Gross. "Protecting applications against TOCTTOU races by user-space caching of file metadata". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 2012, pp. 215–226.

[Ric23] L. Rice. *Learning EBPF*. O'Reilly Media, 2023. ISBN: 9781098135096. URL: `https://books.google.de/books?id=dW-yEAAAQBAJ`.

[Wag99] David A Wagner. "Janus: an approach for confinement of untrusted applications". MA thesis. University of California, Berkeley, 1999.

A Installing eunomia-bpf Development Tools:

Download and install eunomia-bpf using the following steps:

- Download the ecli tool for running eBPF programs, see Listing 3:
- Download the compiler toolchain for compiling eBPF kernel code into config files or WASM modules, as in Listing 4:

A.1 Compile using ecc or docker image:

- To compile and execute this program, utilize the ecc tool along with the ecli command. On Ubuntu/Debian, start by running the following command (Listing 5):
- To compile using ecc, run the following command (Listing 6):
- To compile using docker image (Listing 7):

A.2 Running the Compiled Program

- Run the compiled program using ecli (Listing 8):
- Once the program is executed, you can view the eBPF program's output by inspecting the /sys/kernel/debug/tracing/trace_pipe file, see Listing 9.

```
 from bcc import BPF

bpf_program = """
#include <linux/ptrace.h>
#include <linux/sched.h>

BPF_HASH(counter, u32, u64);

int syscall_filter(struct tracepoint__raw_syscalls__sys_enter *ctx) {
    u64 syscall_id = ctx->id; // Extract syscall ID
    char comm[16];
    bpf_get_current_comm(&comm, sizeof(comm)); // Get process name
   // u32 pid = bpf_get_current_pid_tgid(); // Get process ID

    // Check if the process is a bash process
    if (comm[0] == 'b' && comm[1] == 'a'
    && comm[2] == 's' && comm[3] == 'h') {

        // Restrict specific syscalls: open=2, write=1, openat=257
        if (syscall_id == 2 || syscall_id == 1 || syscall_id == 257) {
            bpf_trace_printk("Restricted syscall ID=%llu from bash\\n", syscall_id);
            return -1; // Block syscall
        }
    }
    return 0; // Allow syscall
}
"""

# Load and attach BPF program
b = BPF(text=bpf_program)
b.attach_tracepoint(tp="raw_syscalls:sys_enter", fn_name="syscall_filter")

print("Tracing syscalls for /bin/bash... Press Ctrl+C to stop.")
b.trace_print()
```

Listing 2: Blocking write syscalls using tracepoints

```
$ wget https://aka.pw/bpf-ecli -O ecli && chmod +x ./ecli
$ ./ecli -h
Usage: ecli [--help] [--version] [--json] [--no-cache] url-and-args
```

Listing 3: Download ecli tool

```
$ wget https://github.com/eunomia-bpf/eunomia-bpf/releases/latest/download/ecc && chm
$ ./ecc -h
eunomia-bpf compiler
Usage: ecc [OPTIONS] <SOURCE_PATH> [EXPORT_EVENT_HEADER]
....
```

Listing 4: Download ecc compiler

```
sudo apt install clang llvm
```

Listing 5: Install clang and llvm

```
$ ./ecc minimal.bpf.c
 Compiling bpf object...
 Packing ebpf object and config into package.json...

```

Listing 6: Compile using ecc

```
docker run -it -v `pwd`/:/src/ ghcr.io/eunomia-bpf/ecc-`uname -m`:latest
```

Listing 7: Compile using docker image

```
$ sudo ./ecli run package.json
Running eBPF program...
```

Listing 8: Run using ecli

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

Listing 9: Trace pipe messages