Exploring System Call Tracing using eBPF and BCC

Contents

1 Introduction to eBPF and BCC

The Extended Berkeley Packet Filter (eBPF) is a revolutionary Linux kernel technology that allows executing custom programs safely within the kernel. Unlike traditional kernel modules, eBPF programs are verified before loading, ensuring they cannot crash the system. They are also JIT compiled, which makes them highly efficient.

Why eBPF?

- Observe system calls, kernel functions, and networking events in real time.
- Perform profiling, security monitoring, and performance analysis.
- Safer than kernel modules and more flexible than static kernel tracing.

BCC (BPF Compiler Collection):

- A set of Python-based tools and bindings to make eBPF programming simpler.
- Provides both ready-made tracing tools (like execsnoop, opensnoop) and support for writing custom programs.

2 Categories of eBPF Programs

1. **System Call Tracing:** Monitor and analyze system calls invoked by processes.
2. **Performance Monitoring:** Measure latency, frequency, and resource usage of syscalls and kernel events.
3. **Network Packet Statistics:** Track and analyze packets at different layers of the networking stack.
4. **Security Monitoring:** Detect unusual system activity by combining syscall tracing and packet statistics.
5. **Observability and Debugging:** Provide detailed visibility into kernel and user-space interactions.

3 System Call Tracing and Performance Monitoring

System calls are the gateway between user applications and the kernel. Tracing system calls can help us:

- Understand process behavior (which files they access, what programs they run).
- Debug performance issues (why some syscalls are slow).
- Detect suspicious activities (like unauthorized access to /etc/shadow).

3.1 Experiment 1 – Tracing File Open System Calls

Objective: Trace every file opened by a process. This helps administrators see what files are being accessed in real time.

C Program (trace_open.c):

```
#include <uapi/linux/ptrace.h>

int trace_open(struct pt_regs *ctx, const char __user *
    filename, int flags) {
     bpf_trace_printk("File opened: %s\n", filename);
     return 0;
}
```

Python Loader (trace_open.py):

```
from bcc import BPF

prog = open("trace_open.c").read()
b = BPF(text=prog)
b.attach_kprobe(event="do_sys_open", fn_name="trace_open")
b.trace_print()
```

3.2 Experiment 2 – Measuring Syscall Latency (read())

Objective: Measure the time taken by read() system calls. This helps profile slow processes.

C Program (latency_read.c):

```
#include <uapi/linux/ptrace.h>
BPF_HASH(start, u32);

int trace_start(struct pt_regs *ctx) {
```

```
    u32 pid = bpf_get_current_pid_tgid();
    u64 ts = bpf_ktime_get_ns();
    start.update(&pid, &ts);
    return 0;
}

int trace_end(struct pt_regs *ctx) {
    u32 pid = bpf_get_current_pid_tgid();
    u64 *tsp = start.lookup(&pid);
    if (tsp) {
        u64 delta = bpf_ktime_get_ns() - *tsp;
        bpf_trace_printk("PID␣%d␣Read␣latency:%llu␣ns\n", pid
            , delta);
        start.delete(&pid);
    }
    return 0;
}
```

Python Loader (latency_read.py):

```
from bcc import BPF

prog = open("latency_read.c").read()
b = BPF(text=prog)

b.attach_tracepoint(tp="syscalls:sys_enter_read", fn_name="
    trace_start")
b.attach_kretprobe(tp="syscalls:sys_exit_read", fn_name="
    trace_end")

print("Tracing␣read()␣latency...␣Ctrl-C␣to␣stop.")
b.trace_print()
```

3.3 Experiment 1 – Counting Packets by Protocol

Objective: Track the number of packets handled per protocol (e.g., TCP vs UDP).

C Program (count_packets.c):

```
#include <uapi/linux/ptrace.h>
#include <linux/skbuff.h>

BPF_HASH(counter, u32, u64);

int count_packets(struct __sk_buff *skb) {
    u32 proto = skb->protocol;
    u64 *count = counter.lookup(&proto);
    if (count) (*count)++;
```

```
    else {
        u64 initial = 1;
        counter.update(&proto, &initial);
    }
    return 0;
}
```

Python Loader (count_packets.py):

```
from bcc import BPF
import time

prog = open("count_packets.c").read()
b = BPF(text=prog)

print("Counting packets by protocol... Ctrl-C to stop.")
while True:
    time.sleep(5)
    print("\n--- Protocol Packet Counts ---")
    for k, v in b["counter"].items():
        print("Protocol %d: %d packets" % (k.value, v.value))
```

4 Appendix

Helper Functions:

- `bpf_trace_printk()`: Prints debug messages to /sys/kernel/debug/-tracing/trace_pipe.
- `bpf_get_current_pid_tgid()`: Returns PID and TGID for identifying processes.
- `bpf_ktime_get_ns()`: Returns kernel time in nanoseconds, useful for measuring latency.

Common BCC Tools:

- `execsnoop`: Traces process execution (execve).
- `opensnoop`: Traces file open calls (openat).
- `tcpconnect`: Traces outgoing TCP connections.

Section 1 – System Call Tracing

Assignment 1: Tracing File Open Syscalls

Task:

- Write an eBPF program (C with BCC) that traces the `open()` syscall.
- Print the PID, process name, and filename whenever a process opens a file.
- Loader program must be in Python using BCC APIs.

Test using:

- `ls` → opens directories.
- `cat /etc/passwd` → should show `/etc/passwd` file access.
- `touch newfile` → should show `newfile`.

Assignment 2: Measuring Syscall Latency

Task:

- Write an eBPF program (C with BCC) that measures the latency of `read()` syscalls.
- Print the latency for each `read` call in microseconds.
- Modify the program to compute both average and maximum latency per process.

Test using:

- `cat small.txt` → small reads, low latency.
- `cat largefile.txt` → larger reads, higher latency.

Section 2 – Network Packet Monitoring

Assignment 3: Counting Packets by Protocol

Task:

- Write an eBPF program (Python with BCC) that counts packets by protocol type (TCP, UDP, ICMP).
- Maintain counters for each protocol.
- Print packet counts every 5 seconds.

Test using:

- `curl google.com` → generates TCP traffic.
- `ping 8.8.8.8` → generates ICMP traffic.
- DNS queries (`dig google.com`) → generate UDP traffic.

Assignment 4: Measuring Per-Protocol Bandwidth Usage

Task:

- Write an eBPF program (C with BCC + Python loader) to measure total bytes transferred per protocol.
- Maintain per-protocol counters for bytes.
- Print a summary every 5 seconds.

Test using:

- `wget http://example.com/file` → large TCP transfer.
- `ping -s 1000 8.8.8.8` → increases ICMP bytes.
- DNS lookups (`dig`) → small UDP byte count.