Process Visualization Exploitation using eBPF

Real-Time Process Monitoring Using eBPF and libbpf

TEAM EMPIRE

Member 1	**Salik Seraj Naik**
Member 2	**Danish Chouhan**
Member 3	**Saqib Jameel**
Member 4	**Anees Ahmed**

▼ ***Date: August 9, 2025***

> **GitHub Link** https://github.com/n4itr0-07/Process-Visualisation-eBPF

Abstract

Modern operating systems are incredibly complex, with thousands of short-lived processes and threads executing concurrently. Understanding this activity is crucial for performance analysis, security auditing, and debugging. This report details the design and implementation of a lightweight, high-performance process exit monitor using eBPF (extended Berkeley Packet Filter). The tool hooks directly into the kernel's scheduler to capture process exit events in real-time, providing key information such as Process ID (PID), Parent Process ID (PPID), command name, and exit code. By leveraging modern eBPF features like libbpf and CO-RE (Compile Once - Run Everywhere), the developed tool is both efficient and

portable across different Linux kernel versions, demonstrating the power of eBPF for deep system observability.

1. Introduction

1.1. Problem Statement

Observing system behavior at the process level is a fundamental task in system administration and security. Traditional monitoring tools often operate in user-space, which can introduce significant performance overhead due to context switching and data copying. Furthermore, these tools may lack the granularity to capture all system events, especially the lifecycle of short-lived threads spawned by complex applications. There is a need for a monitoring solution that is both performant and deeply integrated with the kernel.

1.2. Project Objective

The primary objective of this project is to develop a tool that can monitor and visualize process and thread exit events in real-time. The tool aims to achieve the following:

- Capture process exit events directly from the Linux kernel.
- Extract essential data for each event: PID, PPID, command name, and exit status.
- Utilize modern eBPF and `libbpf` for efficient and portable implementation.
- Provide a clear command-line interface for displaying the captured events.

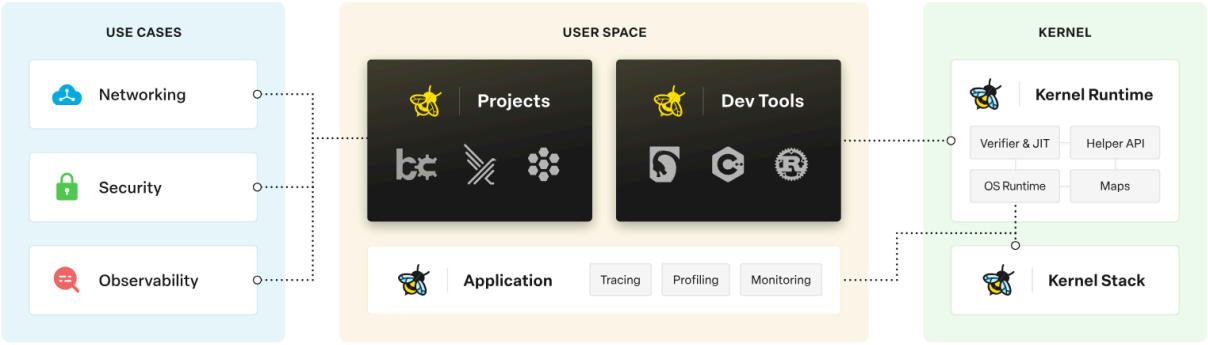
1.3. Scope

This project focuses exclusively on monitoring the `sched_process_exit` event. The tool is designed for Linux systems that support eBPF with BTF (BPF Type Format) for the CO-RE functionality. The output is presented in a human-readable text format on the command line.

2. Background and Literature Review

2.1. eBPF (extended Berkeley Packet Filter)

eBPF is a revolutionary technology in the Linux kernel that allows sandboxed programs to be executed directly in an event-driven manner without changing the kernel source code. It can be thought of as a lightweight, sandboxed virtual machine inside the kernel. Initially used for network packet filtering, its capabilities have been vastly expanded to allow for powerful observability, networking, and security tooling.

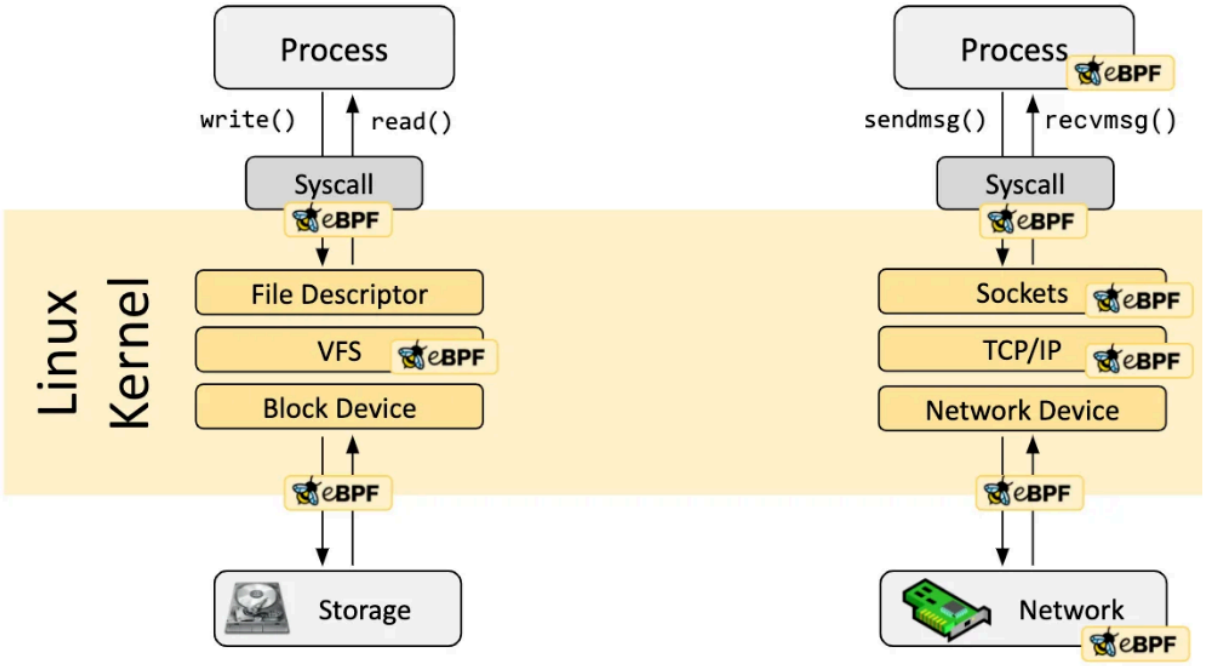


2.2. Key eBPF Concepts

The project relies on several core eBPF components:

- **eBPF Maps:** Key-value data structures that act as a bridge for sharing state, either between different eBPF programs or between kernel-space and user-space applications.
- **Tracepoints:** Stable hooks within the kernel source code that allow eBPF programs to be attached to specific, well-defined events. This project uses the `sched:sched_process_exit` tracepoint.
- **Ring Buffer:** A modern, highly efficient mechanism for sending data from eBPF programs in the kernel to a user-space application. It is a memory-safe, lock-free, multi-producer/single-consumer queue.
- **CO-RE (Compile Once - Run Everywhere):** A key advancement that solves eBPF's historical portability problem. By using **BTF (BPF Type Format)**, which provides debugging information about kernel data structures, an eBPF program can be compiled once and run on different kernel versions without

modification. This is achieved through `vmlinux.h` and helper macros like `BPF_CORE_READ`.



2.3. Comparison with Traditional Tools

Traditional tools like `strace` or `auditd` have been used for process monitoring for years. However, `strace` incurs high overhead by intercepting every system call, and `auditd` relies on a more complex and less performant framework. The eBPF approach provides a significant advantage by processing events directly in the kernel and only sending relevant, structured data to user-space, resulting in much lower system overhead.

---

3. Methodology

The project is divided into two main components: an eBPF program that runs in the kernel and a user-space application that loads and interacts with it.

3.1. System Architecture

The architecture is straightforward:

1. The user-space C application, using the `libbpf` library, loads the compiled eBPF object file into the kernel.
2. `libbpf` attaches the eBPF program to the `sched_process_exit` tracepoint.
3. When any process or thread exits, the kernel triggers our eBPF program.
4. The eBPF program collects the required data and pushes it into a Ring Buffer map.
5. The user-space application continuously polls the Ring Buffer for new data and prints it to the console.

3.2. eBPF Kernel Program (`process_monitor.bpf.c`)

The kernel-side program is responsible for data collection.

- **Hooking:** The `SEC("tp/sched/sched_process_exit")` macro attaches the `handle_exit` function to the process exit event.
- **Data Collection:** Inside `handle_exit`, the program gets a pointer to the kernel's `task_struct` for the exiting process using `bpf_get_current_task()`.
- **CO-RE Reading:** The `BPF_CORE_READ_INTO()` helper is used to safely and portably read the `exit_code`, the parent's `tgid` (PPID), and the `comm` (command name) directly from the `task_struct`.
- **Data Submission:** The collected data is written into a `struct event` and submitted to the user-space application via the `rb` ring buffer map.

```
// Key snippet from src/process_monitor.bpf.c
SEC("tp/sched/sched_process_exit")
int handle_exit(struct trace_event_raw_sched_process_exit *ctx)
{
    // ... setup code ...

    task = (struct task_struct *)bpf_get_current_task();

    // Use CO-RE to read data portably
    BPF_CORE_READ_INTO(&ppid, task, real_parent, tgid);
    BPF_CORE_READ_INTO(&exit_code, task, exit_code);
```

```
    bpf_core_read_str(&e->comm, sizeof(e->comm), &task->comm);

    // ... populate struct and submit to ring buffer ...

    return 0;
}
```

3.3. User-Space Application (`process_monitor.c`)

The user-space loader manages the eBPF program's lifecycle.

- **BPF Skeleton:** The build process uses `bpftool` to generate a skeleton header (`process_monitor.skel.h`), which simplifies interaction with the eBPF object.
- **Loading and Attaching:** The program calls `process_monitor_bpf__open()` , `process_monitor_bpf__load()` , and `process_monitor_bpf__attach()` to load the BPF program into the kernel and attach it.
- **Event Polling:** It sets up a `ring_buffer` manager pointed at the `rb` map's file descriptor. The main loop calls `ring_buffer__poll()` to wait for and process incoming events, printing each one to the console.

```c
static int handle_event(void *ctx, void *data, size_t data_sz)
{
    const struct event *e = data;
    printf("Process exited: PID=%d, PPID=%d, COMM=%s, EXIT_CODE=%d\n",
           e->pid, e->ppid, e->comm, e->exit_code);
    return 0;
}

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

int main(int argc, char **argv)
{
    struct ring_buffer *rb = NULL;
    struct process_monitor_bpf *skel;
    int err;


    signal(SIGINT, sig_handler);
    signal(SIGTERM, sig_handler);


    skel = process_monitor_bpf__open();
    if (!skel) {
        fprintf(stderr, "Failed to open BPF skeleton\n");
        return 1;
    }

    err = process_monitor_bpf__load(skel);
```

3.4. Build Process

The `Makefile` automates the entire build, which consists of:

1. Generating `vmlinux.h` using `bpftool`.
2. Compiling the BPF C code (`.bpf.c`) into a BPF object file using `clang`.
3. Generating the BPF skeleton header (`.skel.h`) from the object file using `bpftool`.
4. Compiling and linking the user-space C code against `libbpf` and `libelf`.

Makefile

```
# Application name
APP = process_monitor

# Source files
BPF_SRC = src/$(APP).bpf.c
USER_SRC = src/$(APP).c

# Object files
BPF_OBJ = src/$(APP).bpf.o
USER_OBJ = $(APP).o

# Generated skeleton header
BPF_SKEL = src/$(APP).skel.h

# Compiler and flags
CC = clang
CFLAGS = -g -O2 -Wall
LDFLAGS = -lbpf -lelf

# Default target
all: $(APP)

# Build the user-space application
$(APP): $(USER_OBJ) $(BPF_SKEL)
	$(CC) $(CFLAGS) $(USER_OBJ) -o $(APP) $(LDFLAGS)

# Compile the user-space source
$(USER_OBJ): $(USER_SRC) $(BPF_SKEL)
	$(CC) $(CFLAGS) -c $(USER_SRC) -o $(USER_OBJ)

# Generate the BPF skeleton header
$(BPF_SKEL): $(BPF_OBJ)
	bpftool gen skeleton $< > $@

# Compile the BPF source
$(BPF_OBJ): $(BPF_SRC) src/vmlinux.h
	$(CC) -g -O2 -target bpf -c $(BPF_SRC) -o $(BPF_OBJ) -I./src

# Clean up generated files
clean:
	rm -f $(APP) $(USER_OBJ) $(BPF_OBJ) $(BPF_SKEL)

.PHONY: all clean
```

4. Results and Analysis

4.1. Compilation and Execution

The project was compiled successfully in the development environment (Ubuntu 22.04 on WSL 2, Linux Kernel 5.15). The `make` command executed all steps without errors, producing the final `process_monitor` executable.

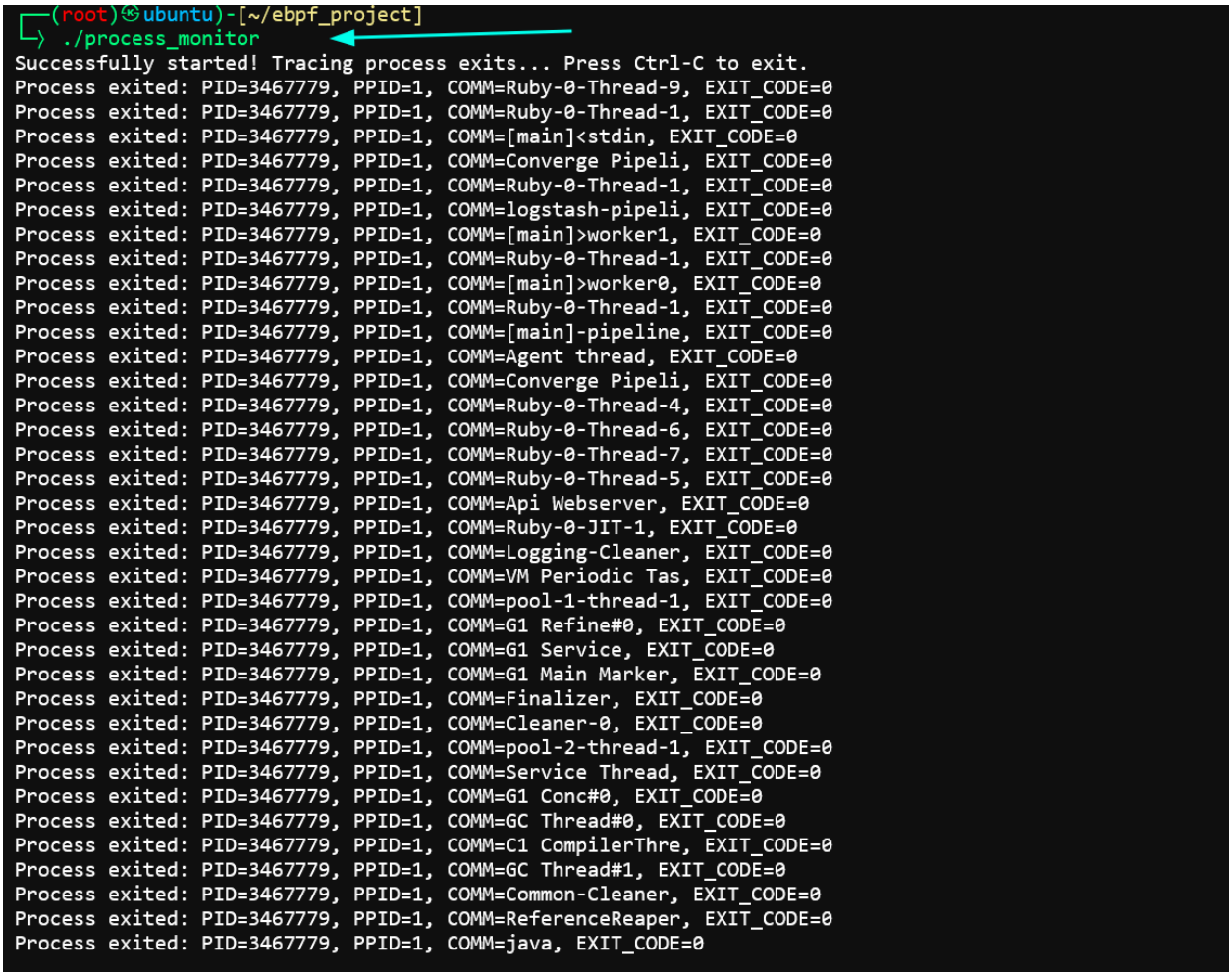
```
┌──(root)㉿ubuntu)-[~/ebpf_project]
└─〉 make                                                                    [8:16:14]
clang -g -O2 -target bpf -c src/process_monitor.bpf.c -o src/process_monitor.bpf.o -I./src
src/process_monitor.bpf.c:22:24: warning: declaration of 'struct trace_event_raw_sched_process_exit' will not be visible
 outside of this function [-Wvisibility]
   22 | int handle_exit(struct trace_event_raw_sched_process_exit *ctx)
      |                        ^
1 warning generated.
bpftool gen skeleton src/process_monitor.bpf.o > src/process_monitor.skel.h
clang -g -O2 -Wall -c src/process_monitor.c -o process_monitor.o
clang -g -O2 -Wall process_monitor.o -o process_monitor -lbpf -lelf
┌──(root)㉿ubuntu)-[~/ebpf_project]
└─〉 ls                                                                      [8:16:22]
Makefile  process_monitor  process_monitor.o  src
┌──(root)㉿ubuntu)-[~/ebpf_project]
└─〉 tree src                                                               [8:16:25]
src
├── process_monitor.bpf.c
├── process_monitor.bpf.o
├── process_monitor.c
├── process_monitor.h
├── process_monitor.skel.h
└── vmlinux.h

1 directory, 6 files
┌──(root)㉿ubuntu)-[~/ebpf_project]
└─〉                                                                         [8:17:05]
```

To run the tool, it was executed with `sudo` privileges. A second terminal was used to generate process activity.

```
┌──(root)㉿ubuntu)-[~]
└─〉 ls                                                                      [8:19:22]
curl_shopify_commerce.txt  data  ebpf_project  lab
┌──(root)㉿ubuntu)-[~]
└─〉 whoami                                                                  [8:19:37]
root
┌──(root)㉿ubuntu)-[~]
└─〉 uname -r                                                               [8:19:40]
6.8.0-62-generic
┌──(root)㉿ubuntu)-[~]
└─〉 ls -lah                                                                [8:19:42]
total 516K
drwx------ 14 root root 4.0K Aug  9 08:19 .
drwxr-xr-x 22 root root 4.0K Jun 22 20:07 ..
-rw-------  1 root root  574 Jun 12 15:53 .bash_history
-rw-r--r--  1 root root 3.1K Jun 15 10:51 .bashrc
drwx------  4 root root 4.0K Jun 25 09:58 .cache
drwxr-xr-x  3 root root 4.0K Jun 15 10:51 .cargo
drwxr-xr-x  4 root root 4.0K Jun 15 10:44 .config
-rw-r--r--  1 root root  672 Jul 20 10:50 curl_shopify_commerce.txt
drwxr-xr-x  6 root root 4.0K Jun 12 18:15 data
drwxr-xr-x  3 root root 4.0K Aug  9 08:16 ebpf_project
drwxr-xr-x  2 root root 4.0K Jun 25 10:13 lab
-rw-------  1 root root   20 Jun 25 12:19 .lesshst
drwxr-xr-x  4 root root 4.0K Jun 15 10:44 .local
drwxr-xr-x  4 root root 4.0K Jun 15 10:42 .npm
drwxr-xr-x 13 root root 4.0K Jun 25 09:45 .oh-my-zsh
-rw-r--r--  1 root root  182 Jun 15 10:51 .profile
-rw-------  1 root root    0 Jun 23 11:29 .python_history
drwxr-xr-x  4 root root 4.0K Jun 15 10:57 .rbenv
drwxr-xr-x  6 root root 4.0K Jun 15 10:51 .rustup
-rw-r--r--  1 root root   10 Jun 12 15:48 .shell.pre-oh-my-zsh
drwx------  2 root root 4.0K Jun 12 15:02 .ssh
-rw-------  1 root root  837 Jun 25 10:07 .viminfo
-rw-r--r--  1 root root  181 Jun 12 19:32 .wget-hsts
-rw-r--r--  1 root root  50K Jun 22 20:07 .zcompdump
-rw-r--r--  1 root root  51K Jun 12 15:48 .zcompdump-localhost-5.9
-r--r--r--  1 root root 118K Jun 12 15:48 .zcompdump-localhost-5.9.zwc
-rw-r--r--  1 root root  51K Jun 25 09:45 .zcompdump-ubuntu-5.9
-r--r--r--  1 root root 118K Jun 25 09:45 .zcompdump-ubuntu-5.9.zwc
-rw-r--r--  1 root root   21 Jun 15 10:51 .zshenv
-rw-------  1 root root  24K Aug  9 08:19 .zsh_history
-rw-r--r--  1 root root 1.2K Jun 15 10:52 .zshrc
```

4.2. Analysis of Captured Data

The program successfully captured and displayed all process and thread exit events as expected. The output included the correct PID, PPID, command name, and exit code for each event.



Key Observations

- **High-Granularity Thread Monitoring:** The output clearly demonstrates the tool's ability to monitor individual threads, not just parent processes. The single **Process ID** (`PID=3467779`) is associated with dozens of different **Command Names** (`COMM`), such as `Ruby-0-Thread-9`, `G1 Refine#0`, and `[main]>worker1`. This is characteristic of a single, complex, multi-threaded application.

- **Application Fingerprinting:** The specific thread names allow for the identification of the running application. Names like `logstash-pipeli` and `Ruby-0-JIT-1` strongly suggest the application is **Logstash**, which is a Ruby application that runs on the **Java Virtual Machine (JVM)**. The presence of `java` and `G1 Service` threads confirms this.
- **Insight into JVM Internals:** The tool provides a fascinating look into the internal operations of the JVM. It captures the lifecycle of threads dedicated to specific tasks like **Garbage Collection** (`G1 Main Marker`, `G1 Service`), **Just-In-Time (JIT) compilation** (`C1 CompilerThre`), and internal worker management (`pool-1-thread-1`).
- **Daemon Process Analysis:** The **Parent Process ID** (`PPID=1`) indicates that the application is a long-running system daemon or service, not a short-lived command run by a user. The consistent `EXIT_CODE=0` across all events suggests that these threads are completing their tasks and exiting cleanly, which is normal behavior for such a service.

Exploitation Techniques Using eBPF

1. Loading Malicious eBPF Programs

- Attackers craft and load harmful eBPF bytecode via the `bpf()` syscall.
- They can bypass verifier rules to execute unauthorized kernel code.
- This leads to privilege escalation or kernel compromise.

2. Manipulating eBPF Maps

- eBPF maps store data shared between kernel and user space.
- Malicious actors can exploit poorly controlled maps to leak sensitive kernel info.
- They may also corrupt kernel memory by abusing map operations.

3. Hiding Malware Using eBPF

- eBPF programs can intercept system calls and network traffic stealthily.

- Attackers hide malicious behavior inside kernel-level hooks.
- This evades many traditional security tools.

4. Race Conditions on Process Events

- Exploit timing gaps during process creation (`fork()`, `exec()`).
- Inject or hijack processes before security checks complete.
- Enables unauthorized code execution.

Defensive Measures (Summary)

- Use strict eBPF verifier and limit who can load programs.
- Regularly update the kernel to fix vulnerabilities.
- Monitor all eBPF programs and map usage.
- Restrict eBPF-related syscalls and access to reduce attack surface.

---

6. References

1. Gregg, B. (2019). *BPF Performance Tools*. Addison-Wesley Professional.
2. Nakryiko, A. (2021). BPF CO-RE (Compile Once – Run Everywhere). Retrieved from https://nakryiko.com/posts/bpf-core-reference-guide/
3. Linux Kernel Documentation - BPF Design Q&A. (n.d.). Retrieved from https://www.kernel.org/doc/html/latest/bpf/bpf_design_QA.html
4. **libbpf** - The BPF Library. (n.d.). Retrieved from https://github.com/libbpf/libbpf
5. **Official doc** https://ebpf.io/what-is-ebpf/