University of sousse

National School of Engineers in Sousse

المدرسة الوطنية للمهندسين بسوسة

École Nationale d'Ingénieurs de Sousse

Semestral Project Report

Subject: Real-time Kernel-Level Observability for Docker Containers using eBPF

produced by:
Fedi Ben Abdesslem
Amir Soussi

Supervised by:
Dr. Manel Fourati

December 2025

Abstract

Containerization has revolutionized software deployment, but it introduces significant security and observability challenges. Traditional monitoring tools often rely on logs or metrics that lack the granularity needed to detect sophisticated runtime attacks or subtle misconfigurations. This report presents the *Container Security Visualizer*, a system that leverages Extended Berkeley Packet Filter (eBPF) to monitor Docker containers at the Linux kernel level. By capturing system calls and network events in real-time and mapping them to specific containers, the system provides a live, interactive graph visualization of container behavior without requiring any modification to the containers themselves. The result is a powerful tool for observing internal container dynamics, detecting anomalies, and understanding the complex interactions within a containerized environment.

1 Introduction

The rapid adoption of containerization technologies like Docker has transformed modern infrastructure, enabling scalable and portable application deployment. However, this shift has also obscured the visibility into application behavior. Containers share the host kernel, but their isolation mechanisms (namespaces and cgroups) can make it difficult to monitor their internal activities using traditional host-based tools.

Standard monitoring solutions often rely on parsing logs or collecting high-level metrics (CPU, memory usage). While useful for performance tuning, these methods are insufficient for security monitoring. They fail to capture the granular execution details—such as specific system calls or unexpected network connections—that are critical for detecting runtime threats like privilege escalation, shell code execution, or lateral movement.

This project addresses the need for deep, kernel-level visibility into containerized environments. We propose a system that uses eBPF to safely and efficiently intercept system events directly from the kernel. Our objective is to build a pipeline that captures these events, correlates them with container metadata, and visualizes them in a user-friendly interface. This approach provides a "transparent" monitoring layer that requires no instrumentation within the containers, offering a robust solution for security auditing and behavioral analysis.

2 Problem Statement

Docker and similar container runtimes provide basic logging drivers and resource statistics. However, these mechanisms leave a significant gap in observability:

- **Lack of Syscall Visibility:** Standard logs do not reveal which files a process is accessing or what child processes it is spawning.
- **Opaque Network Behavior:** It is often difficult to trace which specific container initiated an outbound connection, especially when using bridge networks or overlays.
- **Delayed Detection:** Log-based alerts are typically reactive and delayed, whereas security incidents require real-time detection.

Consider a scenario where a compromised web application container begins scanning the internal network. Traditional metrics might show a slight increase in CPU usage, which is easily ignored. Logs might not show anything if the attacker suppresses output. A kernel-level monitor, however, would immediately see a burst of `connect` syscalls to internal IP addresses.

The core problem this project seeks to answer is: *"How can container behavior be observed in* *real-time, at the granularity of system calls and network flows, without modifying the containers or incurring significant performance overhead?"*

3 Background & Core Concepts

3.1 Linux Kernel Basics

The Linux kernel is the core interface between hardware and software. User-space applications interact with the kernel through **System Calls (syscalls)**, such as `read`, `write`, `execve`, and `connect`. Monitoring these syscalls provides a definitive record of what an application is doing. **Processes** are the active execution units, identified by a Process ID (PID). In Linux, threads are also treated as processes (Lightweight Processes). The kernel manages these resources and enforces permissions.

3.2 Container Internals

Containers are not real physical objects but are created using kernel features:

- **Namespaces:** Provide isolation for resources like PIDs, networks, and mount points. A process in a container sees itself as PID 1, but it has a different PID in the host's root namespace.
- **Control Groups (cgroups):** Limit and account for resource usage (CPU, memory). They are also crucial for tracking which processes belong to which container.

The key challenge in container monitoring is mapping the host-level PID (seen by the kernel) back to the container context (seen by the user).

3.3 eBPF Overview

Extended Berkeley Packet Filter (eBPF) is a revolutionary technology that allows running sandboxed programs in the Linux kernel without changing kernel source code or loading modules.

- **Safety:** The kernel verifier ensures that eBPF programs cannot crash the system or enter infinite loops.
- **Observability:** eBPF programs can be attached to tracepoints, kprobes, and perf events, allowing them to capture data at the source.
- **Efficiency:** Data is filtered and aggregated in the kernel before being sent to user space, minimizing overhead compared to traditional tracing tools like `strace`.

Unlike kernel modules, which can introduce stability risks, eBPF provides a safe and programmable way to extend kernel capabilities for monitoring and security.

4 System Architecture

The system is designed as a modular pipeline consisting of three main layers: the Monitoring Layer, the Backend/Collector Layer, and the Visualization Layer.

4.1 High-Level Architecture

1. **Monitoring Layer (eBPF):** Runs in the kernel space. It attaches to specific system calls and captures event data (timestamp, PID, arguments).

2. **Collector / Backend (User Space):** A Python-based service that loads the eBPF programs, reads the event stream, enriches the data with container metadata, and serves it via an API.
3. **Visualization Layer (Frontend):** A web-based user interface that consumes the event stream and renders a real-time graph of container interactions.

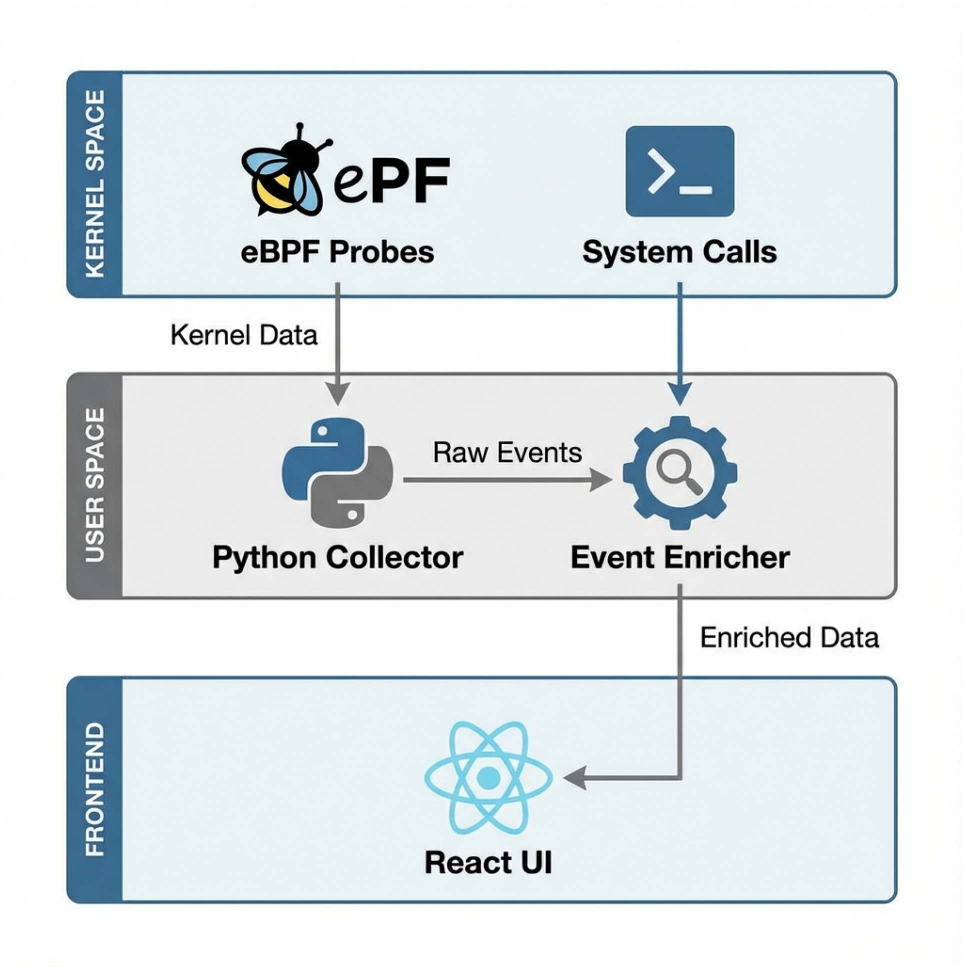

Figure 1: High-Level System Architecture

4.2 End-to-End Data Flow

The data flow proceeds as follows:

1. A process inside a container executes a syscall (e.g., `curl google.com` triggers `connect`).
2. The eBPF program attached to the `sys_enter_connect` tracepoint triggers.
3. The eBPF program captures the PID and socket arguments and pushes a struct to a shared `perf_buffer`.
4. The user-space Collector polls the buffer, retrieves the raw event, and maps the PID to a Container ID.
5. The Backend enriches the event with the container name and image, then broadcasts it via WebSocket.
6. The Frontend receives the event and updates the visual graph, drawing an edge between the container node and the external IP.

4.3 Design Decisions

- **Host-Based Monitoring:** We chose to run a single monitoring agent on the host rather than injecting sidecars into every container. This approach is less intrusive and easier to manage.
- **Read-Only Observation:** To ensure system stability, our eBPF programs are read-only. We do not block or modify system calls, minimizing the risk of disrupting application logic.
- **Real-Time Streaming:** Security events are time-sensitive. We prioritized a streaming architecture over batch processing to provide immediate visibility into ongoing activities.

5 Monitoring Layer Implementation

This section details the core technical implementation of the kernel-side monitoring using eBPF and BCC (BPF Compiler Collection).

5.1 Monitored Events

We focus on two primary categories of events that are most relevant for security and observability:

- **Process Execution (`execve`):** Captures when new programs are started. This is critical for detecting suspicious binaries or shell commands.
- **Network Connections (`connect`):** Captures outbound TCP/UDP connections. This helps identify unauthorized data exfiltration or lateral movement.

5.2 Probe Attachment

We utilize **Tracepoints** rather than Kprobes. Tracepoints provide a stable API (ABI) that is less likely to change between kernel versions compared to internal kernel function names used by Kprobes.

- `tracepoint:syscalls:sys_enter_execve`
- `tracepoint:syscalls:sys_enter_connect`

5.3 Data Capture & PID Resolution

When a tracepoint is hit, our eBPF C program executes:

1. **Context Retrieval:** We call `bpf_get_current_pid_tgid()` to get the host-level PID and Thread Group ID.
2. **Argument Reading:**
   - For `execve`, we read the filename pointer using `bpf_probe_read_user_str()`.
   - For `connect`, we read the `sockaddr` structure to extract the destination IP and port.
3. **Data Submission:** The captured data is packed into a custom C struct and submitted to user space via `perf_submit()`.

A critical challenge is mapping the host PID to a container. Since eBPF runs in the kernel, it sees the global PID. We resolve this in the user-space collector by inspecting `/proc/<pid>/cgroup`. This file contains the cgroup paths, which include the Container ID (e.g., `/docker/<container_id>`). This allows us to attribute every system call to a specific container.

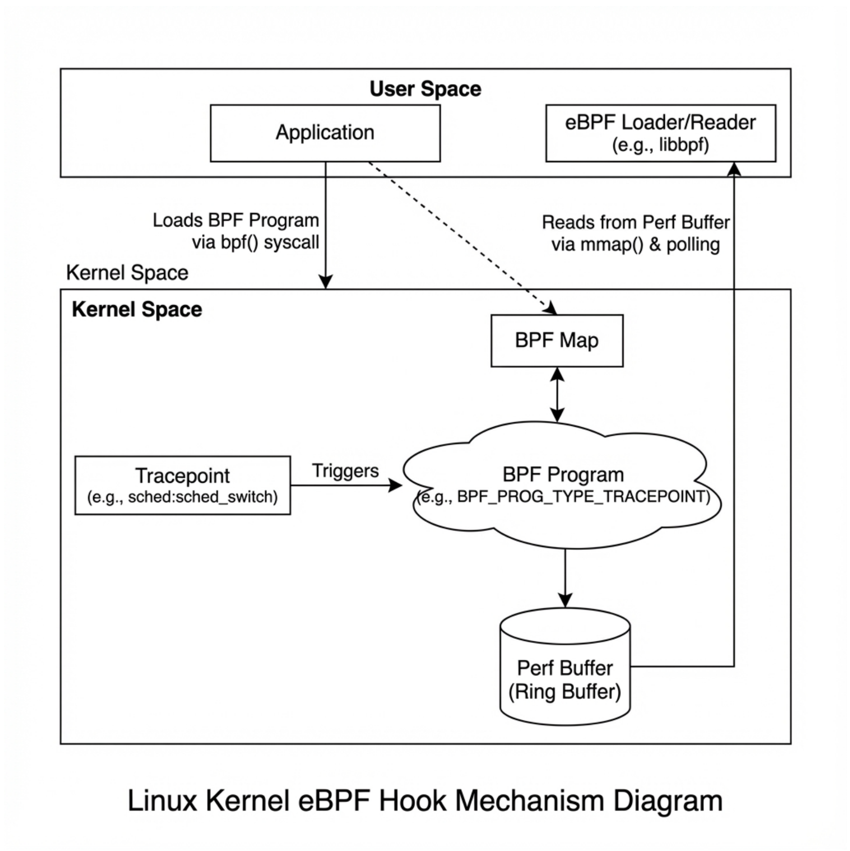
User Space
Application
eBPF Loader/Reader (e.g., libbpf)
Loads BPF Program via bpf() syscall
Reads from Perf Buffer via mmap() & polling
Kernel Space
Kernel Space
BPF Map
Tracepoint (e.g., sched:sched_switch)
Triggers
BPF Program (e.g., BPF_PROG_TYPE_TRACEPOINT)
Perf Buffer (Ring Buffer)
Linux Kernel eBPF Hook Mechanism Diagram

Figure 2: eBPF Tracepoint and Data Capture Flow

6 Backend / Event Processing Layer

The backend acts as the bridge between the raw kernel events and the user interface.

6.1 Event Collection and Normalization

The Collector uses the BCC Python library to load the eBPF programs and poll the perf buffers. Raw binary data from the kernel is unpacked into Python dictionaries. Timestamps are converted from kernel monotonic time to standard epoch time for correlation.

6.2 Enrichment

Raw events contain only PIDs and IPs. The `EventEnricher` component adds context:

- **Container Metadata:** Using the Container ID derived from the PID, we query the Docker API (or cache) to get the Container Name, Image, and Status.
- **Risk Scoring:** We assign a basic risk score to events. for example, connections to sensitive ports (like SSH/22) or execution of sensitive binaries (like `nc` or `nmap`) receive higher scores.

6.3 Filtering and Aggregation

To prevent overwhelming the visualization, we implement basic filtering. For instance, repeated connections to the same endpoint within a short window can be aggregated. We also filter out internal system processes that are not part of the containerized workload.

6.4 Data Preparation

The processed events are formatted into a JSON structure suitable for the frontend graph model. Network events are transformed into "edges" (Source Container $\rightarrow$ Destination IP), while process events are treated as attributes or alerts on the "node" (Container).

7 Visualization Layer

The frontend provides the human-readable interface for the system.

7.1 Goals

The primary goal is to transform abstract log data into an intuitive visual model. This allows operators to quickly identify relationships and anomalies that would be difficult to spot in text logs.

7.2 Graph Model

We utilize a force-directed graph layout where:

- **Nodes** represent Containers and External IPs.
- **Edges** represent active network connections or recent communication.

The graph updates in real-time as new events arrive via WebSockets.

7.3 Visual Cues

To aid rapid decision-making, we use color-coding:

- **Green:** Normal behavior and trusted containers.
- **Red:** High-risk events (e.g., connection to a known malicious IP or execution of a suspicious binary).
- **Yellow:** Warning or unknown behavior.

This immediate visual feedback helps human operators focus their attention on potential threats.

8 Security & Performance Considerations

8.1 Safety

The use of eBPF provides strong safety guarantees. The kernel verifier analyzes every eBPF program before loading it to ensure it terminates and does not access invalid memory. This makes our solution significantly safer than traditional kernel modules.

8.2 Performance

eBPF is highly efficient. The overhead of capturing a syscall is measured in nanoseconds. By filtering events in the kernel (e.g., only tracing specific PIDs or syscalls if needed), we minimize the volume of data sent to user space, ensuring that the monitoring system does not degrade the performance of the applications it monitors.

8.3 Read-Only Monitoring

Our system is strictly observational. It does not block or intercept syscalls in a way that could alter application behavior. This ”passive” monitoring approach is crucial for production environments where stability is paramount.

9 Experimental Setup Demo Scenario

9.1 Environment

The system was developed and tested on a Linux environment with:

- **Kernel:** Linux 5.x+ (required for modern eBPF features).
- **Runtime:** Docker Engine.
- **Languages:** C (eBPF), Python (Collector/Backend), JavaScript/React (Frontend).

9.2 Demo Scenario

To validate the system, we created a controlled test environment with two containers:

1. **Target Container:** A standard Nginx web server.
2. **Attacker Container:** A container simulating a compromised service.

We simulated an attack sequence where the ”Attacker” container:

- Scans the network using `nmap` (generating network events).
- Attempts to access sensitive files (generating `open`/`execve` events).
- Establishes a reverse shell connection (generating outbound `connect` events).

10 Results Observations

During the demo scenario, the system successfully captured and visualized the events:

- **Real-Time Visibility:** As soon as the ”Attacker” container initiated network scans, new edges appeared on the graph connecting the container to various IP addresses.
- **Anomaly Detection:** The execution of `nmap` and the reverse shell command were flagged with high risk scores and displayed in red on the UI.
- **Granularity:** Unlike standard Docker logs, which only showed ”container started”, our tool showed the exact commands executed inside the container, proving the value of kernel-level monitoring.

11 Limitations

While effective, the current implementation has limitations:

- **Host-Level Visibility Only:** We cannot inspect the payload of encrypted network traffic (HTTPS) without more invasive techniques (e.g., uprobes on SSL libraries).
- **Scalability:** Visualizing hundreds of containers in a single graph can become cluttered. Grouping and filtering mechanisms would be needed for large-scale deployments.

- **Ephemeral Containers:** Extremely short-lived containers might exit before their metadata can be fully enriched, leading to gaps in attribution.

12 What We Learned

This project provided deep insights into:

- **Kernel Internals:** Understanding how the kernel handles syscalls and how namespaces isolate resources.
- **eBPF Programming:** The constraints of the eBPF verifier and the power of the BCC framework.
- **Container Plumbing:** How Docker uses cgroups and namespaces, and how to map between kernel and user-space views of a process.

13 Future Work

Future enhancements could include:

- **Kubernetes Support:** Integrating with K8s API to visualize Pods and Services instead of just raw containers.
- **Advanced Anomaly Detection:** Using Machine Learning to learn ”normal” behavior baselines and alert on deviations.
- **Policy Enforcement:** Extending the eBPF programs to `LSM` (Linux Security Modules) hooks to actively block malicious syscalls, moving from detection to prevention.

14 Conclusion

The Container Security Visualizer demonstrates the power of eBPF for modern cloud-native observability. By looking below the container abstraction layer directly into the kernel, we achieved a level of visibility that traditional tools cannot match. The system successfully answers the problem statement, providing real-time, granular, and safe monitoring of container behavior. This project highlights that while containers provide isolation, the kernel remains the ultimate source of truth for system activity.