

Initial Architecture Document

Linux Kernel Security Monitor (LKSM)

Team Number: Group 32

Team Members: Brett Balquist, Kaden Huber, Jamie King,
Dustin Le, Max Biundo, Hart Nurnberg

Project Name: LKSM

Date: February 2026

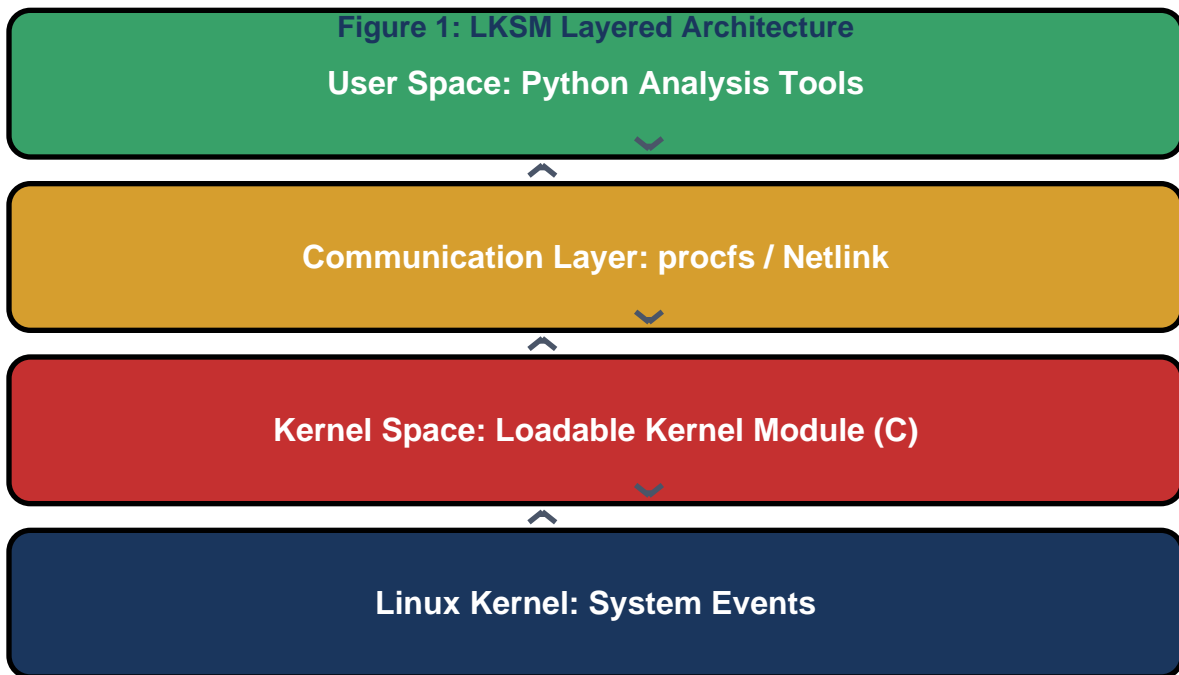
Project Synopsis

A loadable kernel module (C) and Python toolkit that monitors Linux kernel events (process creation, file access, network connections, module loading) in real-time, providing anomaly detection, alerting, and security reporting capabilities.

1. Architecture Overview

The Linux Kernel Security Monitor (LKSM) implements a defense-in-depth security monitoring solution using a two-tier architecture. The system operates across the kernel-user space boundary, with a Loadable Kernel Module (LKM) written in C handling low-level event capture, and a Python-based analysis suite providing intelligent processing, visualization, and alerting capabilities.

The architecture follows a producer-consumer pattern where kernel hooks act as event producers, feeding data through a buffered communication channel to user-space consumers. This separation ensures system stability while enabling sophisticated analysis without kernel modification risks.



The four-layer architecture ensures clean separation of concerns and secure event propagation.

2. Core Components

2.1 Kernel Module (LKM)

The Loadable Kernel Module forms the foundation of LKSM's monitoring capabilities. Written in C and compiled against the target kernel headers, it leverages kprobes and tracepoints to intercept critical system events without modifying kernel source code. The module implements four primary hook categories: process lifecycle events (fork/exec), file system access on sensitive paths, kernel module load/unload operations, and network connection establishment. Each hook extracts relevant metadata including PIDs, UIDs, timestamps, and contextual information before forwarding to the communication layer.

Key features include self-protection mechanisms to prevent unauthorized unloading, a rate-limiting buffer to handle event storms, and an optional eBPF-based syscall filtering engine for per-process security policies. The module maintains minimal kernel footprint while providing comprehensive visibility into system activity.

2.2 Communication Channel

The communication layer bridges kernel and user space through either a procfs interface (`/proc/lkasm`) or netlink sockets. This design choice enables efficient, low-latency event delivery while maintaining security boundaries. The procfs approach provides simplicity and easy debugging, while netlink offers

higher throughput for production deployments. Events are serialized in a compact binary format before transmission, with the Python layer handling deserialization and JSON conversion.

2.3 Python Analysis Suite

The user-space component comprises multiple interconnected Python modules. The Event Reader continuously polls the communication channel, deserializing incoming events and feeding them to the Log Parser. Parsed events are written to structured JSON log files for persistence and later analysis. The Rule Engine applies configurable detection logic based on YAML-defined allowlists and denylists, flagging policy violations for further processing.

The Anomaly Detector implements behavioral analysis, identifying suspicious patterns such as shells spawned by web servers, connections to unusual ports, or high-frequency activity from single processes. The Network Correlator enriches connection data by mapping socket operations to originating processes. A real-time terminal dashboard (using Rich or ncurses) provides operators with live visibility, while the Alert System dispatches notifications via syslog and configurable webhooks.

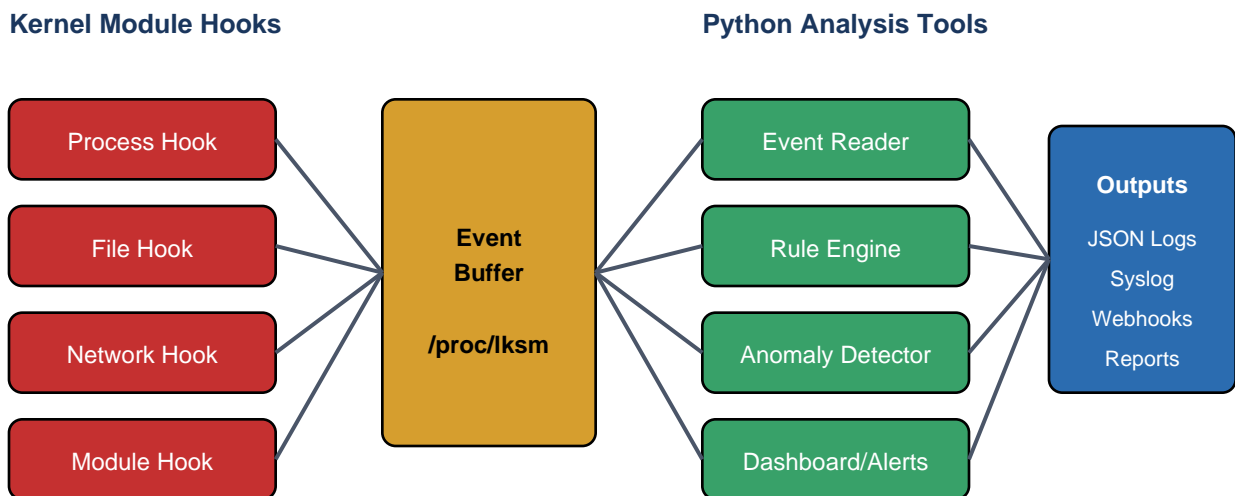


Figure 2: Component Interaction Flow

3. Data Flow Architecture

Events flow through a well-defined pipeline optimized for both performance and analytical depth. When a monitored kernel event occurs, the appropriate hook captures essential metadata and queues it in an internal ring buffer. This buffering mechanism, combined with rate limiting, prevents event loss during burst activity while protecting kernel stability. The communication layer transports events to user space where they undergo parsing, enrichment, and storage.

The analysis stage applies rule-based detection and statistical anomaly identification in parallel. Alerts generated by either mechanism feed into the unified notification system. The Forensic Timeline Reconstructor provides post-incident analysis by correlating events across all categories into chronological attack-chain visualizations, enabling security teams to understand complex intrusion patterns.



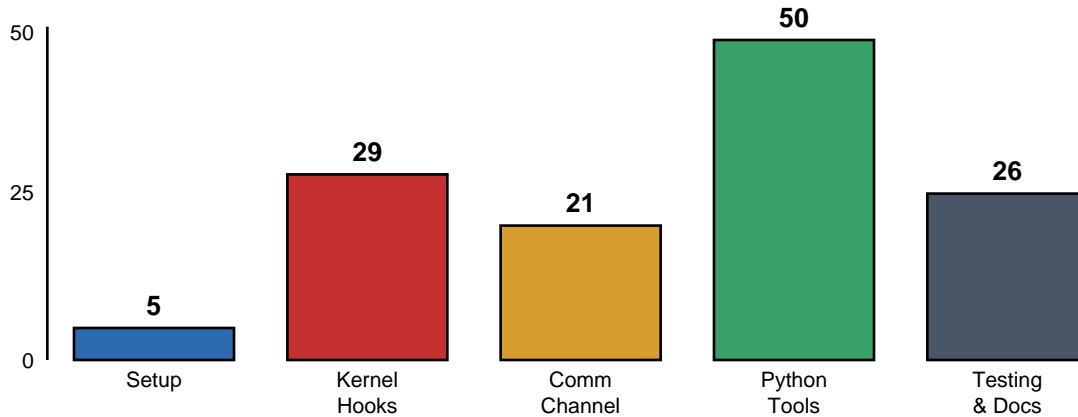
Figure 3: Event Processing Pipeline

Events traverse six processing stages from kernel capture to final reporting.

4. Technology Stack

Layer	Technology	Purpose
Kernel Module	C, kprobes, tracepoints	Event capture and filtering
Communication	procfs, netlink	Kernel-user data transfer
Analysis Engine	Python 3.x	Event processing and detection
Configuration	YAML	Rule and policy definitions
Dashboard	Rich / ncurses	Real-time visualization
Storage	JSON files	Event logging and persistence
Testing	pytest, bash scripts	Unit and integration testing

Figure 4: Story Points by Component Category



Distribution of 131 story points across component categories (excluding 3 advanced requirements).

5. Development Plan

The project spans 10 weeks organized into 6 sprints, following an incremental delivery model. Early sprints establish foundational infrastructure including the development environment, skeleton LKM, and communication channel. Middle sprints focus on expanding hook coverage and building the Python analysis pipeline. Final sprints address testing, documentation, performance optimization, and demonstration preparation. The 28 requirements total 154 story points, with three advanced requirements (eBPF filtering, forensic timeline, and high-throughput buffering) accounting for 39 points of enhanced functionality.

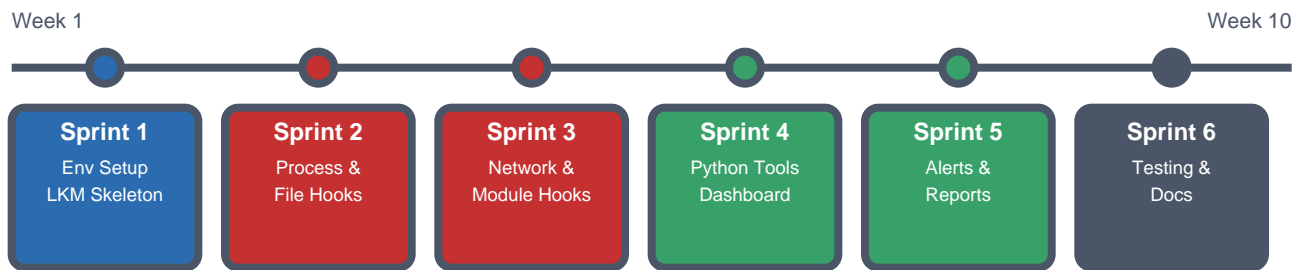


Figure 5: Sprint Timeline (10 Weeks)

6. Security Considerations

LKSM implements defense-in-depth principles in its own design. The kernel module includes self-protection against unauthorized removal, preventing attackers from disabling monitoring after initial compromise. The eBPF syscall filtering capability enables proactive threat mitigation beyond passive monitoring. Configuration files support both allowlist and denylist approaches, enabling

flexible security policies tailored to specific environments. All inter-component communication remains local to the host, minimizing network attack surface.

7. Conclusion

The LKSM architecture provides comprehensive Linux security monitoring through a modular, extensible design. By separating kernel-level event capture from user-space analysis, the system achieves both the deep visibility required for security monitoring and the flexibility needed for sophisticated threat detection. The layered approach enables independent evolution of components while maintaining stable interfaces, supporting long-term maintainability and feature expansion.