Kernel Module Integrity Monitor (KMIM) — Part 1 Report

Aditya Pillai

October 10, 2025

Abstract

This report documents the Kernel Module Integrity Monitor (KMIM), a security tool that leverages eBPF for safe runtime introspection of kernel module activity. The tool captures a baseline of trusted kernel modules and syscall table addresses, and compares the live kernel state against this baseline to detect anomalies. The report covers tool introduction, design overview, implementation details, usage examples, sample results, and includes the tool's man page as an appendix.

Contents

1 Tool Introduction

1.1 Problem statement

Rootkits and malicious kernel modules pose significant threats to system security by operating at the kernel level where they can hide their presence and compromise system integrity. Traditional integrity checkers often lack the capability to safely monitor kernel activity in real-time. KMIM (Kernel Module Integrity Monitor) is a CLI utility designed to detect rootkits, malicious kernel modules, and syscall table tampering using eBPF technology for safe kernel introspection without loading kernel modules.

1.2 Goals

The main goals of KMIM are:

- Create a JSON baseline of all loaded kernel modules with cryptographic hashes.
- Capture syscall table addresses to detect syscall hooking attacks.
- Extract ELF sections and compiler metadata from kernel modules.
- Detect hidden modules that hide from `/proc/modules`.
- Provide real-time eBPF-based monitoring of module load/unload events.
- Produce clear, color-coded CLI output with detailed scan results.
- Keep the tool lightweight and portable across Linux distributions.

2 Tool design overview

2.1 High-level architecture

KMIM is implemented as a single-file Python CLI that exposes subcommands: `baseline`, `scan`, `show`, and `monitor`. The baseline is stored as a JSON file (default: `kmim_baseline.json`) capturing the trusted state of kernel modules and syscall addresses.

2.2 Data model (baseline)

Each baseline entry is keyed by module name and stores:

- **name**: kernel module name
- **size**: module size in bytes
- **addr**: module load address in kernel memory
- **file**: path to module file (from `modinfo`)
- **hash**: SHA256 digest of the module file
- **elf_sections**: list of ELF section names
- **compiler**: compiler information (GCC/Clang version)

Additionally, the baseline captures syscall table addresses for integrity monitoring.

2.3 Detection logic

KMIM performs checks in the following order:

1. Read `/proc/modules` to get currently loaded modules.
2. Compare against baseline to detect added, removed, or modified modules.
3. Verify module file hashes using SHA256 to detect tampering.
4. Cross-reference `/proc/modules` with `/proc/kallsyms` to detect hidden modules.
5. Compare syscall table addresses to detect syscall hooking.
6. Report results with severity: OK (green), WARNING (yellow), ALERT (red).

2.4 eBPF monitoring

KMIM uses eBPF kprobes to monitor kernel activity in real-time:

- Attaches to `do_init_module` and `free_module` kernel functions.
- Captures module name, timestamp, PID, and process name.
- Forwards events to userspace via perf ring buffer.
- Displays events in real-time with color-coded output.

2.5 User-facing commands

scan-dir

`baseline <file.json>` Capture current kernel state as baseline.

`scan <file.json>` Compare live kernel state against baseline.

`show <module>` Display detailed metadata for a specific module.

`monitor` Start real-time eBPF monitoring of module events.

3 Implementation details

3.1 Language and libraries

The implementation is in **Python 3**. It uses the following libraries:

- `bcc` — eBPF/BCC for kernel monitoring
- `rich` — colored CLI output with progress indicators
- `hashlib` — SHA256 computation (standard library)
- `subprocess` — for `modinfo`, `readelf`, `strings`
- `argparse`, `json`, `os` — Python stdlib

3.2 Key modules / functions

- `read_proc_modules()`: parse `/proc/modules` to get loaded modules.
- `mod_filename(module)`: use `modinfo -n` to find module file path.
- `file_sha256(path)`: compute SHA256 hash incrementally.
- `extract_elf_sections(path)`: parse ELF sections using `readelf -S`.
- `guess_compiler(path)`: extract compiler info using `strings`.
- `read_kallsyms()`: parse `/proc/kallsyms` for kernel symbols.
- `get_hidden_modules()`: detect modules in kallsyms but not in /proc/modules.
- `run_monitor()`: load and execute eBPF program for real-time monitoring.

3.3 Baseline storage and location

By default, the baseline file is `kmim_baseline.json` in the current directory. Users can specify any path when running baseline/scan commands. The baseline contains timestamped captures for audit trail purposes.

3.4 Security and operational notes

- **Root privileges:** Required for reading `/proc/kallsyms` and loading eBPF programs.
- **Kernel compatibility:** Requires Linux kernel 4.4+ with eBPF support (CONFIG_BPF=y).
- **Updates:** After legitimate kernel updates, re-capture the baseline to avoid false positives.
- **kptr_restrict:** May need to be set to 0 or 1 for kallsyms access.
- **eBPF safety:** eBPF programs are verified by kernel and cannot crash the system.

3.5 How to run

Full usage, dependencies and setup are documented in the project's README. Typical workflow:

```
# Install dependencies
sudo apt install bpfcc-tools python3-bpfcc
pip3 install rich

# Capture baseline
sudo python3 kmim.py baseline kmim_baseline.json

# Scan for changes
sudo python3 kmim.py scan kmim_baseline.json

# Show module details
sudo python3 kmim.py show ext4

# Start real-time monitoring
sudo python3 kmim.py monitor
```

3.6 Source code

The source code for the project (single-file Python CLI) is available at:

https://github.com/APillai03/KMIM

4 Examples and sample outputs

4.1 Baseline capture

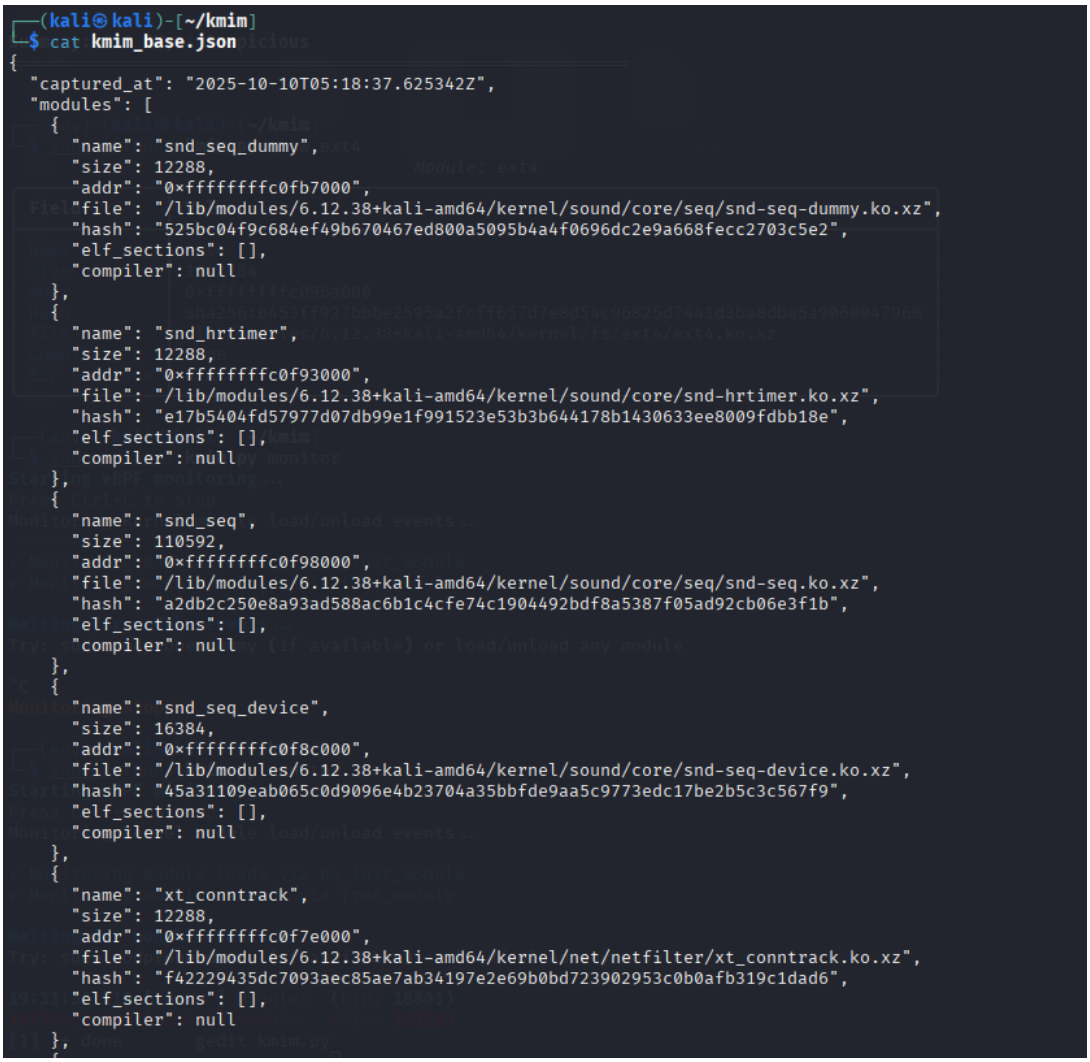

Figure 1: Sample output from `sudo python3 kmim.py baseline kmim_baseline.json`

4.2 Clean scan (OK)

4.3 Show command

4.4 Real-time eBPF monitoring

4.5 Hidden module detection

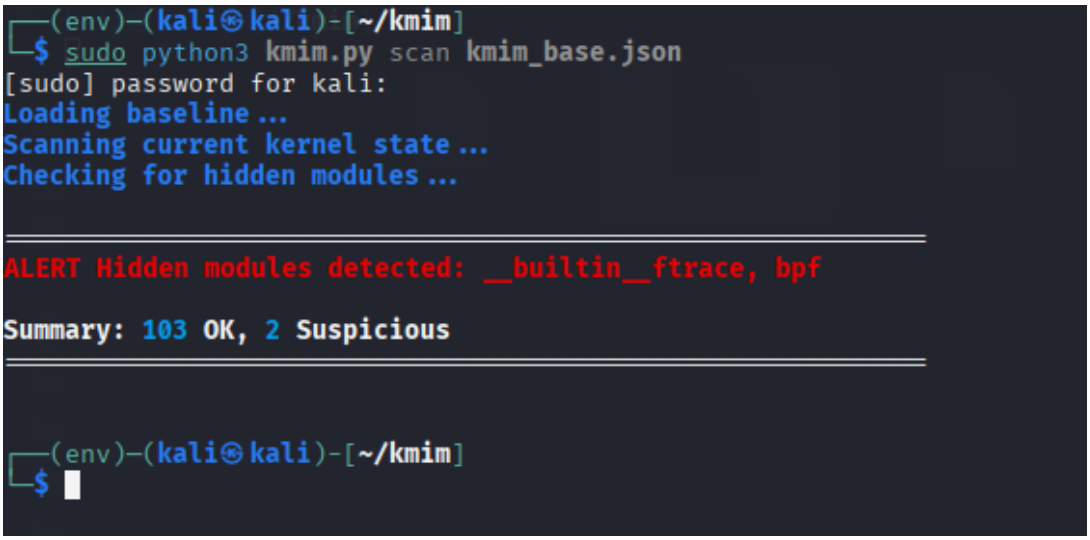

Figure 2: Sample output showing all modules match baseline (clean system).

Figure 3: Sample output displaying detailed module information.

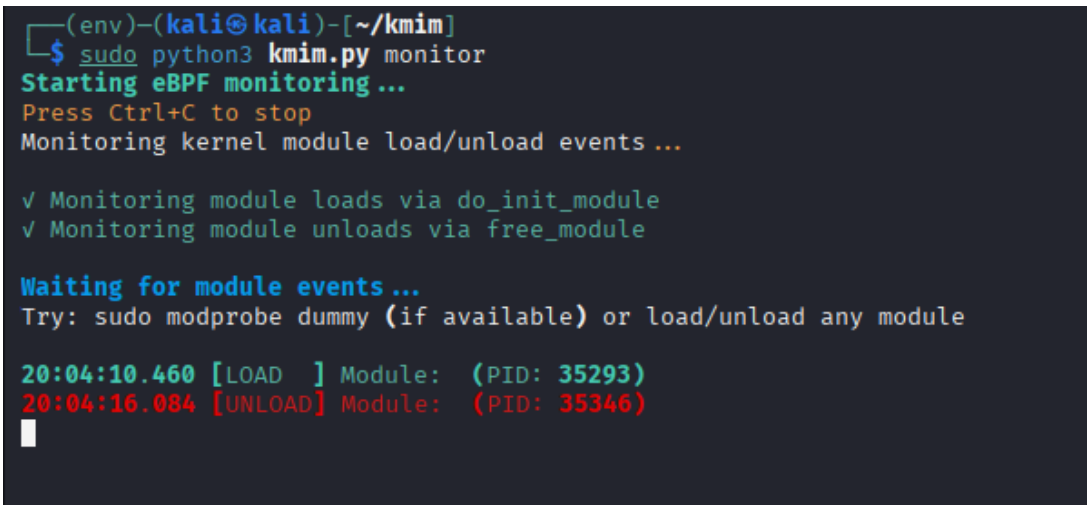

Figure 4: Sample output from `monitor` command showing live module events.

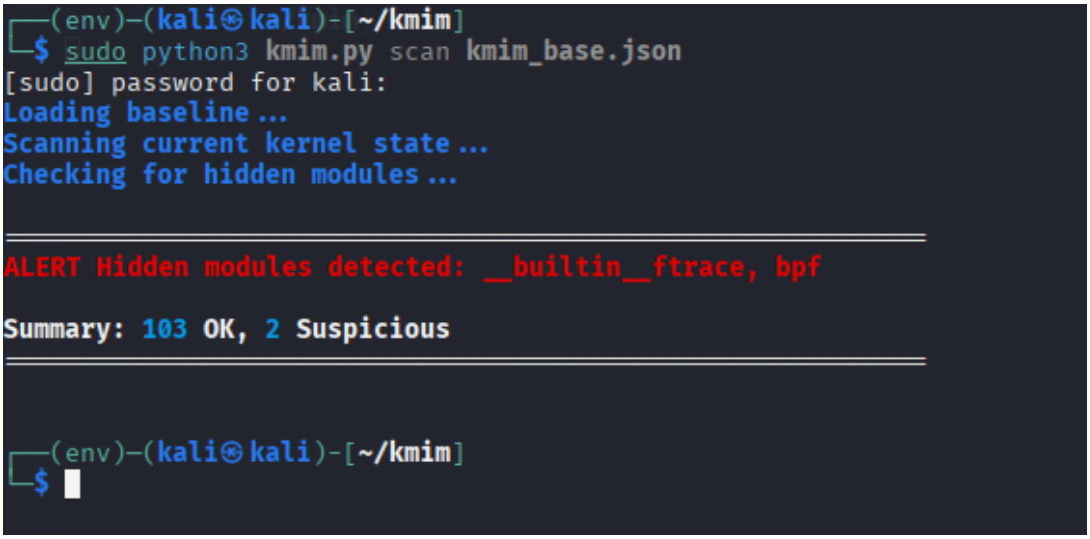

Figure 5: Sample output showing detection of hidden modules.

5 Discussion and future work

5.1 Limitations

- The current implementation relies on `/proc/kallsyms` which can be restricted or hooked by advanced rootkits.
- Hidden module detection only works if modules leave traces in kallsyms.
- The tool requires root privileges limiting its use in restricted environments.
- eBPF programs may not attach if kernel functions are not exposed or renamed in custom kernels.

5.2 Potential improvements

- Per-section cryptographic hashes for more granular integrity checking.
- Memory scanning to detect modules that hide from both /proc/modules and kallsyms.
- Integration with SIEM systems (Splunk, ELK) for centralized monitoring.
- Automated alerting via email, Slack, or PagerDuty.
- Support for more kernel versions with automatic function detection.
- Machine learning-based anomaly detection for unusual module behavior.

6 Appendix A: Man page reference

Below is the project's man page excerpt. Full manual available in repository.

```
KMIM(1)                    User Commands                    KMIM(1)

NAME
       kmim - Kernel Module Integrity Monitor

SYNOPSIS
       kmim baseline <file.json>
       kmim scan <file.json>
       kmim show <module> [--baseline <file.json>]
       kmim monitor

DESCRIPTION
       KMIM  is  a  kernel integrity monitoring tool built with eBPF.
       It captures metadata about kernel modules and  syscalls,  builds
       a trusted baseline, and compares live state against the baseline
       to detect tampering, hidden modules, or runtime anomalies.

COMMANDS
       baseline <file.json>
              Capture the current kernel state and save to a  baseline
              file. Includes module hashes, ELF sections, and syscall
              addresses.
```

```
       scan <file.json>
              Compare  the  live  kernel  modules  and  syscalls  with
              the baseline file. Detects added, removed, or  mismatched
              modules and hidden modules.

       show <module> [--baseline <file.json>]
              Display  detailed  metadata of the specified kernel module.

       monitor
              Start  real-time  eBPF monitoring of module load and unload
              events. Press Ctrl+C to stop.

AUTHOR
       Software Security Lab (HPRCSE Group)
```

7 Appendix B: README link and repository

- Project README: https://github.com/APillai03/KMIM
- Project source: https://github.com/APillai03/KMIM

8 Appendix C: Example commands run

```
# Install dependencies (Debian/Ubuntu/Kali)
sudo apt update
sudo apt install -y bpfcc-tools python3-bpfcc linux-headers-$(uname -r)
pip3 install rich

# Capture baseline
sudo python3 kmim.py baseline kmim_baseline.json

# Scan for changes
sudo python3 kmim.py scan kmim_baseline.json

# Show module details
sudo python3 kmim.py show ext4
sudo python3 kmim.py show usbcore

# Start monitoring (Ctrl+C to stop)
sudo python3 kmim.py monitor

# In another terminal, load/unload modules to test
sudo modprobe loop
sudo modprobe -r loop
sudo modprobe fuse
sudo modprobe -r fuse

# Test detection of new module
sudo python3 kmim.py baseline test_baseline.json
sudo modprobe dummy
sudo python3 kmim.py scan test_baseline.json
```

```
# Should show: ALERT Added modules: dummy
sudo modprobe -r dummy
```

9 Conclusion

KMIM provides a robust and safe mechanism for monitoring kernel module integrity using modern eBPF technology. By combining baseline comparison, real-time monitoring, and hidden module detection, KMIM helps security teams detect rootkits and unauthorized kernel modifications. The tool demonstrates how eBPF can be leveraged for security monitoring without the risks associated with traditional kernel module-based approaches.

Key achievements:

- Successfully implemented eBPF-based kernel monitoring
- Baseline and scanning functionality with cryptographic verification
- Hidden module detection through kallsyms cross-referencing
- Real-time event monitoring with minimal performance overhead
- Professional CLI interface with clear, actionable output