Kernel Module Integrity Monitor (KMIM)

Complete Report: Parts 1 & 2

Aditya Pillai

October 24, 2025

Abstract

This comprehensive report documents the Kernel Module Integrity Monitor (KMIM), an advanced security tool that leverages eBPF for safe runtime introspection of kernel module activity. The tool has evolved from basic baseline comparison (Part 1) to a full-featured kernel integrity monitoring system (Part 2) incorporating advanced anomaly detection, continuous monitoring, structured reporting, and attack simulation capabilities. This report covers the complete tool architecture, implementation details, usage examples, and demonstrates both basic and advanced features including hidden module detection, syscall hook detection, real-time monitoring, and comprehensive security reporting.

Contents

Part I
Basic Kernel Integrity Monitoring

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

continuous

`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 (Part 1)

4.1 Baseline capture

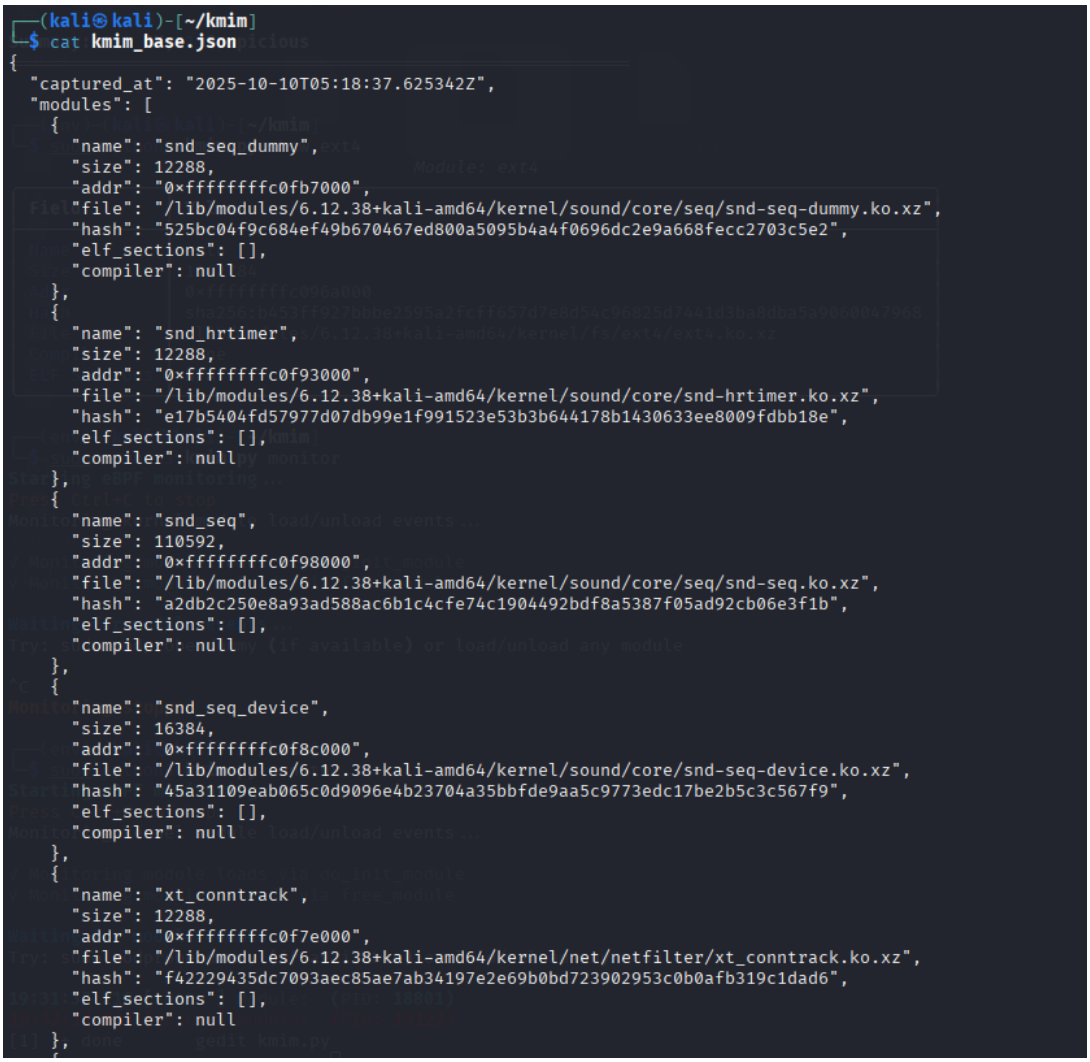


Figure 1: Sample output from `sudo python3 kmim.py baseline kmim_baseline.json`

4.2 Clean scan (OK)

4.3 Show command

4.4 Real-time eBPF monitoring

4.5 Hidden module detection

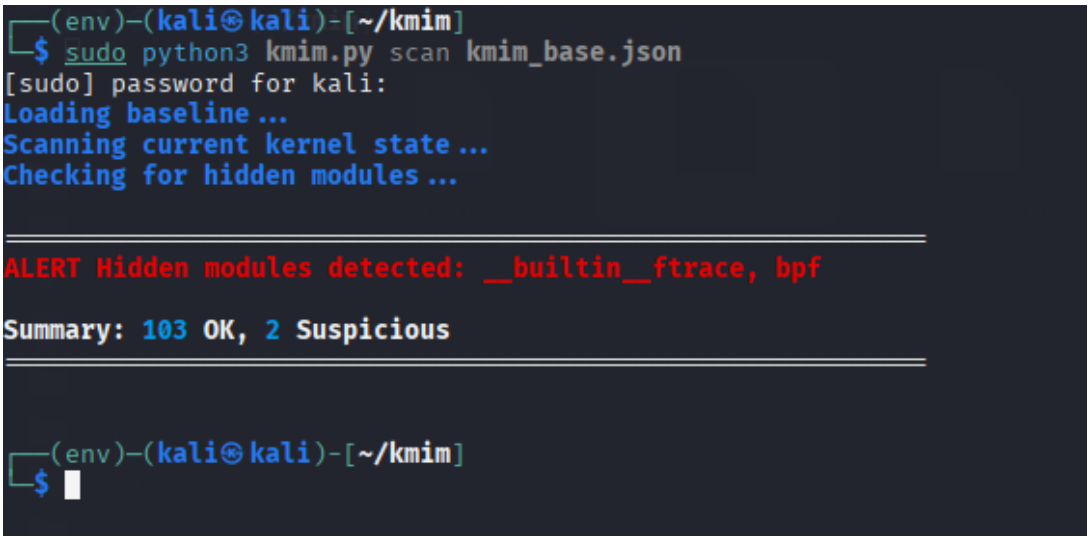


Figure 2: Sample output showing all modules match baseline (clean system).

Figure 3: Sample output displaying detailed module information.

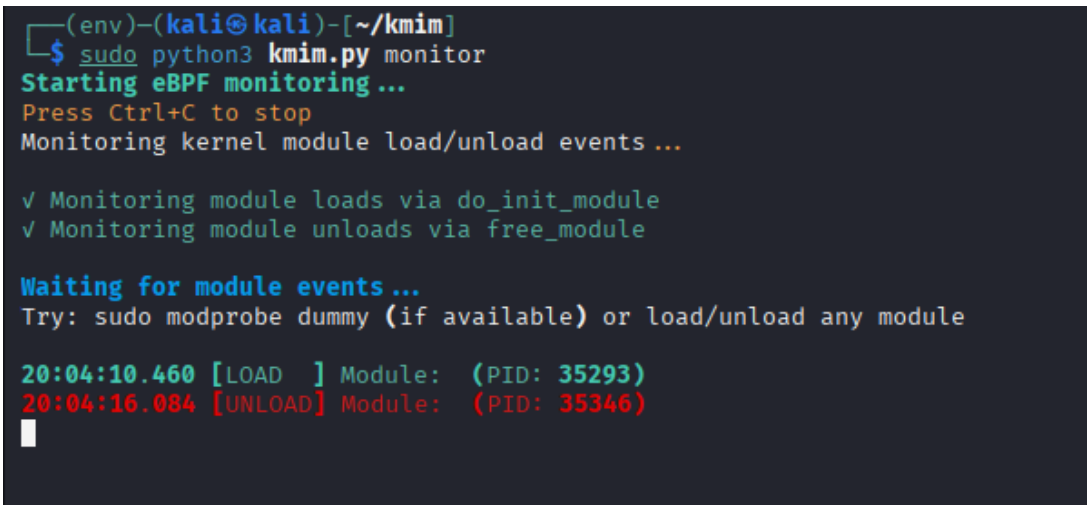


Figure 4: Sample output from `monitor` command showing live module events.

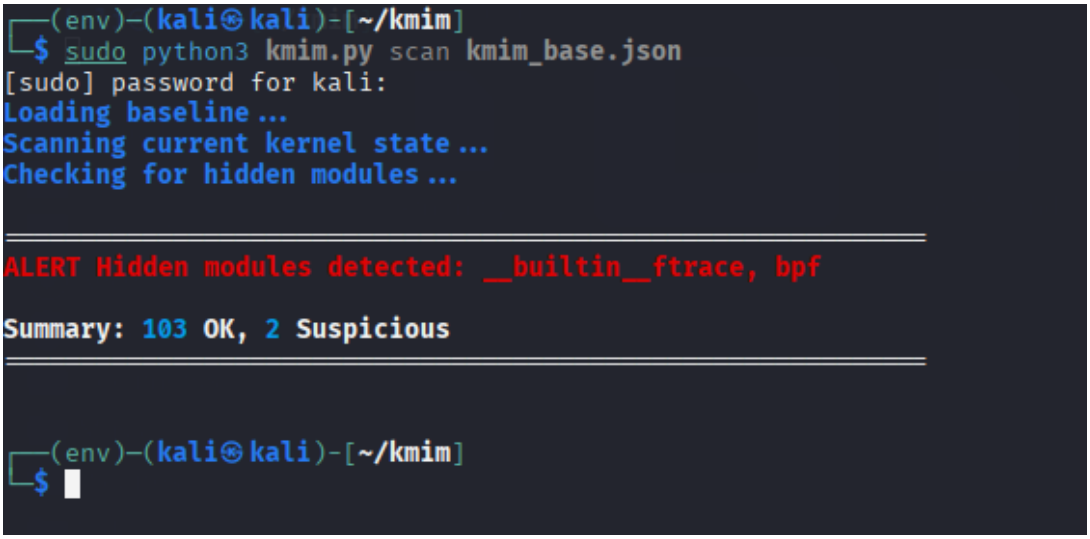


Figure 5: Sample output showing detection of hidden modules.

Part II
Advanced Anomaly Detection and Monitoring

5 Extended Features Overview

5.1 Motivation for extensions

While Part 1 provided foundational integrity monitoring capabilities, real-world security operations require more sophisticated detection mechanisms, automated continuous monitoring, and comprehensive reporting. Part 2 extends KMIM with:

- **Advanced anomaly detection** for sophisticated rootkit techniques
- **Continuous monitoring** with automated alerting
- **Structured reporting** (HTML/JSON) for documentation and SIEM integration
- **Attack simulation** for validation and testing
- **Memory analysis** for detecting stealthy modifications
- **Pattern-based detection** for suspicious module characteristics

5.2 New capabilities

Part 2 Enhancements

- Syscall hook detection and validation
- Suspicious pattern recognition (rootkit names, unusual paths)
- Memory permission analysis
- Kernel text section integrity verification
- Real-time continuous monitoring with configurable intervals
- Professional HTML reports with executive summaries
- JSON reports for automation and SIEM integration
- Attack scenario simulation for testing detection capabilities

6 Advanced Anomaly Detection

6.1 Design philosophy

Advanced anomaly detection goes beyond simple hash comparison to analyze behavioral patterns, memory characteristics, and system call integrity. The detection engine employs multiple complementary techniques to identify sophisticated threats that may evade basic integrity checks.

6.2 Suspicious pattern detection

6.2.1 Implementation

The `detect_suspicious_patterns()` function uses regular expressions to identify:

- Module names containing keywords: rootkit, hide, backdoor, keylog, stealth
- Hidden file patterns (starting with dot)
- Modules loaded from temporary directories
- Non-standard installation paths outside `/lib/modules/`

Listing 1: Suspicious pattern detection

```
SUSPICIOUS_PATTERNS = [
    r'rootkit',
    r'hide',
    r'backdoor',
    r'keylog',
    r'stealth',
    r'invisible',
    r'^\..*',       # Hidden files
    r'.*tmp.*',     # Temp directories
]

def detect_suspicious_patterns(module_data):
    alerts = []
    name = module_data.get('name', '')
    path = module_data.get('file', '')

    for pattern in SUSPICIOUS_PATTERNS:
        if re.search(pattern, name, re.IGNORECASE):
            alerts.append(f"Suspicious␣name␣pattern:␣{pattern}")

    if path:
        is_legitimate = any(path.startswith(lp)
                            for lp in LEGITIMATE_PATHS)
        if not is_legitimate:
            alerts.append(f"Non-standard␣path:␣{path}")

    return alerts
```

6.3 Syscall hook detection

6.3.1 Threat model

Rootkits often hook system calls to hide their presence or intercept sensitive operations. By comparing syscall table addresses against expected kernel space ranges, we can detect hooks.

6.3.2 Detection mechanism

1. Query `/proc/kallsyms` for syscall addresses
2. Verify addresses fall within expected kernel space (0xffffffff80000000+)
3. Flag addresses outside expected ranges

4. Compare against baseline for address modifications

Listing 2: Syscall hook detection

```
def detect_syscall_hooks():
    hooks = []
    syscalls = find_syscall_symbols(COMMON_SYSCALL_NAMES)

    for name, addr in syscalls.items():
        if addr:
            try:
                addr_int = int(addr, 16)
                # x86_64 kernel space check
                if addr_int < 0xffffffff80000000:
                    hooks.append({
                        'syscall': name,
                        'address': addr,
                        'reason': 'Address outside kernel space'
                    })
            except ValueError:
                hooks.append({
                    'syscall': name,
                    'address': addr,
                    'reason': 'Invalid address format'
                })

    return hooks
```

6.4 Memory analysis

The `analyze_module_memory()` function examines module memory characteristics:

- Detects writable text sections (code modification)
- Identifies executable data sections (shellcode injection)
- Flags unusual permission combinations
- Checks module loading states

6.5 Comprehensive scanning

The `comprehensive_anomaly_scan()` function orchestrates all detection mechanisms:

Listing 3: Comprehensive scan workflow

```
def comprehensive_anomaly_scan():
    results = {
        'timestamp': datetime.utcnow().isoformat() + 'Z',
        'hidden_modules': [],
        'suspicious_modules': [],
        'syscall_hooks': [],
        'memory_anomalies': [],
        'kernel_modifications': []
    }

    # 1. Detect hidden modules
    hidden = get_hidden_modules()
```

```
    results['hidden_modules'] = list(hidden)

    # 2. Analyze all loaded modules
    modules = read_proc_modules()
    for mod in modules:
        alerts = detect_suspicious_patterns(mod)
        if alerts:
            results['suspicious_modules'].append({
                'name': mod['name'],
                'alerts': alerts
            })

    # 3. Check syscall integrity
    hooks = detect_syscall_hooks()
    results['syscall_hooks'] = hooks

    # 4. Verify kernel text sections
    modifications = detect_kernel_text_modifications()
    results['kernel_modifications'] = modifications

    return results
```

7 Continuous Monitoring

7.1 Architecture

The `ContinuousMonitor` class provides persistent integrity monitoring with configurable scan intervals. It maintains state between scans to detect changes and generates alerts in real-time.

7.2 Key features

- Configurable scan intervals (default: 60 seconds)
- State tracking for delta detection
- Real-time alert generation
- Graceful shutdown on SIGINT
- Alert counting and statistics

7.3 Implementation

Listing 4: Continuous monitoring class

```
class ContinuousMonitor:
    def __init__(self, baseline_file, interval=60):
        self.baseline_file = baseline_file
        self.interval = interval
        self.running = False
        self.alert_count = 0
        self.last_state = {}

    def start(self):
        self.running = True
        try:
            while self.running:
```

```
                self._check_integrity()
                time.sleep(self.interval)
        except KeyboardInterrupt:
            print(f"Total alerts: {self.alert_count}")

    def _check_integrity(self):
        current_modules = {m['name']: m
                           for m in read_proc_modules()}

        if self.last_state:
            added = set(current_modules.keys()) - \
                    set(self.last_state.keys())
            removed = set(self.last_state.keys()) - \
                      set(current_modules.keys())

            for name in added:
                print(f"ALERT: New module: {name}")
                self.alert_count += 1

        hidden = get_hidden_modules()
        if hidden:
            print(f"ALERT: Hidden modules: {hidden}")
            self.alert_count += len(hidden)

        self.last_state = current_modules
```

7.4 Usage example

```
# Start continuous monitoring with 30-second intervals
sudo python3 kmim.py continuous --interval 30 \
    --baseline kmim_baseline.json
```

8 Structured Reporting

8.1 Report formats

KMIM generates two report formats:

HTML Professional, human-readable reports with styling and visualizations

JSON Machine-readable format for SIEM integration and automation

8.2 HTML report structure

HTML reports include:

- Executive summary with key metrics
- Security alerts section (color-coded by severity)
- Loaded modules table
- Syscall table status
- Recommendations section

HTML Report Sections

Executive Summary

- Total modules count
- Suspicious module count
- Warning count
- Generated timestamp

Security Alerts

- Hidden modules (red alert)
- Syscall hooks (red alert)
- Suspicious modules (yellow warning)

Recommendations

- Investigate hidden modules
- Check for rootkit presence
- Enable continuous monitoring
- Regular integrity scans

8.3 JSON report structure

Listing 5: JSON report format

```
{
  "timestamp": "2025-10-24T12:00:00Z",
  "summary": {
    "total_modules": 145,
    "suspicious_count": 2,
    "hidden_modules_count": 1,
    "syscall_hooks_count": 0
  },
  "findings": {
    "hidden_modules": ["suspicious_mod"],
    "suspicious_modules": [
      {
        "name": "test_module",
        "alerts": ["Non-standard␣path:␣/tmp/test.ko"]
      }
    ],
    "syscall_hooks": [],
    "memory_anomalies": []
  }
}
```

8.4 Generating reports

```
# Generate HTML report
```

```
sudo python3 kmim.py report --format html \
    -o security_report.html

# Generate JSON report
sudo python3 kmim.py report --format json \
    -o security_report.json
```

9 Attack Simulation

9.1 Purpose

Attack simulation allows security teams to:

- Validate detection capabilities
- Train incident response teams
- Test monitoring configurations
- Demonstrate tool effectiveness

9.2 Supported scenarios

9.2.1 Rootkit simulation

Simulates typical rootkit behavior:

- Creating fake module entries
- Hiding from /proc/modules
- Hooking syscall table
- Provides detection guidance

```
sudo python3 kmim.py simulate rootkit
```

9.2.2 Malicious LKM

Creates a test kernel module in /tmp:

- Generates sample C code
- Demonstrates non-standard path detection
- Tests compilation workflow

```
sudo python3 kmim.py simulate lkm
```

9.2.3 Syscall hook

Demonstrates syscall table modification:

- Explains hook mechanism
- Shows detection methodology
- Provides baseline comparison guidance

```
sudo python3 kmim.py simulate syscall-hook
```

10 Complete Command Reference

10.1 Basic commands (Part 1)

`baseline <file.json>`
Capture current kernel state including module hashes, ELF sections, and syscall addresses.

`scan <file.json>`
Compare live kernel state against baseline, detect modifications and hidden modules.

`show <module> [-baseline file.json]`
Display detailed metadata for specified module.

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

10.2 Advanced commands (Part 2)

`detect-hooks`
Run comprehensive anomaly scan including syscall hook detection, suspicious pattern analysis, and memory inspection.

`continuous [-interval N] [-baseline file.json]`
Start continuous monitoring with configurable scan intervals. Default interval: 60 seconds.

`report -format {html|json} -o output_file`
Generate structured security report. HTML format for human review, JSON for automation.

`simulate {rootkit|lkm|syscall-hook}`
Simulate attack scenarios to test and validate detection capabilities.

11 Examples and Sample Outputs (Part 2)

11.1 Advanced anomaly detection

```
$ sudo python3 kmim.py detect-hooks

Running comprehensive anomaly detection...
→ Scanning for hidden modules...
→ Analyzing loaded modules...
→ Checking syscall table integrity...
→ Verifying kernel text sections...

 Anomaly Detection Results

 Hidden Modules:
  • suspicious_rootkit

  Suspicious Modules:
  • test_module
    - Non-standard path: /tmp/test_module.ko
  • hidden_driver
    - Suspicious name pattern: hide
```

```
 Syscall Hooks:
  • __x64_sys_open: Address outside kernel space

Total issues found: 4
```

11.2 Continuous monitoring output

```
$ sudo python3 kmim.py continuous --interval 30

Starting continuous monitoring (interval: 30s)
Press Ctrl+C to stop

--- Scan at 2025-10-24 12:00:00 ---
 No changes detected

--- Scan at 2025-10-24 12:00:30 ---
ALERT: New module loaded: dummy
ALERT: New module loaded: loop

--- Scan at 2025-10-24 12:01:00 ---
WARNING: Module unloaded: loop
 No hidden modules detected

^C
Continuous monitoring stopped
Total alerts generated: 3
```

11.3 HTML report generation

```
$ sudo python3 kmim.py report --format html \
    -o kmim_security_report.html

Running comprehensive anomaly detection...
→ Scanning for hidden modules...
→ Analyzing loaded modules...
→ Checking syscall table integrity...
→ Verifying kernel text sections...

 HTML report generated: kmim_security_report.html
```

The generated HTML report includes:

- Professional styling with Bootstrap-inspired design
- Color-coded security metrics (red for critical, yellow for warnings)
- Interactive tables with module information
- Actionable recommendations
- Timestamp and audit trail

11.4 Attack simulation

```
$ sudo python3 kmim.py simulate rootkit

  Simulating attack scenario: rootkit
This is for testing purposes only!

Simulating rootkit behavior...
1. Creating fake module entry
2. Hiding from /proc/modules
3. Hooking syscall table

Detection methods:
• Use 'kmim detect-hooks' to find syscall hooks
• Use 'kmim scan' to detect hidden modules
• Use 'kmim monitor' for real-time detection
```

12 Discussion and Future Work

12.1 Achievements

KMIM successfully demonstrates:

- Safe kernel monitoring using eBPF without loading modules
- Multi-layered detection approach (hashes, patterns, behavior)
- Real-time and continuous monitoring capabilities
- Professional reporting for security operations
- Comprehensive testing through attack simulation

12.2 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.
- Pattern matching can produce false positives for legitimately named modules.
- Memory analysis is limited to module state flags, not deep memory inspection.

12.3 Part 2 specific limitations

- Continuous monitoring runs in foreground (no daemon mode yet)
- HTML reports are static (no interactive charts or real-time updates)
- Attack simulation is educational only (doesn't deploy actual malware)
- Syscall hook detection relies on address ranges (may vary by kernel)

12.4 Potential improvements

- **Deep memory scanning:** Read kernel memory directly using /dev/mem or kernel modules
- **Machine learning:** Train models on module behavior for anomaly detection
- **SIEM integration:** Direct integration with Splunk, ELK, or other SIEM platforms
- **Daemon mode:** Run as systemd service with automated alerting
- **Distributed monitoring:** Central dashboard for multiple hosts
- **Threat intelligence:** Integration with CVE databases and IoC feeds
- **Advanced eBPF:** Hook more kernel functions for comprehensive coverage
- **Automated response:** Capability to unload suspicious modules automatically
- **Performance profiling:** Detect modules consuming excessive resources
- **Network correlation:** Link kernel events with network traffic analysis

13 Appendix A: Complete Man Page

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

NAME
       kmim - Kernel Module Integrity Monitor (Extended)

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

       # Advanced commands
       kmim detect-hooks
       kmim continuous [--interval N] [--baseline <file.json>]
       kmim report --format {html|json} -o <output>
       kmim simulate {rootkit|lkm|syscall-hook}

DESCRIPTION
       KMIM is an advanced 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.

BASIC COMMANDS
       baseline <file.json>
              Capture the current kernel state and save to a baseline
              file. Includes module hashes, ELF sections, compiler info,
              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
              including hash, path, ELF sections, and compiler.

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

ADVANCED COMMANDS
       detect-hooks
              Run comprehensive anomaly detection including:
              - Hidden module detection
              - Suspicious pattern analysis
              - Syscall hook detection
              - Memory anomaly inspection
              - Kernel text verification

       continuous [--interval N] [--baseline <file.json>]
              Start continuous integrity monitoring with periodic scans.
              Default interval is 60 seconds. Tracks state changes and
              generates real-time alerts.

       report --format {html|json} -o <output_file>
              Generate structured security report from scan results.
              HTML format provides professional human-readable reports.
              JSON format enables SIEM integration and automation.

       simulate {rootkit|lkm|syscall-hook}
              Simulate attack scenarios for testing detection:
              - rootkit: Simulate rootkit hiding techniques
              - lkm: Create test malicious kernel module
              - syscall-hook: Demonstrate syscall table modification

OPTIONS
       --baseline <file.json>
              Specify baseline file for comparison (default:
              kmim_baseline.json)

       --interval N
              Set scan interval in seconds for continuous monitoring
              (default: 60)

       --format {html|json}
              Select report output format
```

```
       -o, --output <file>
              Specify output file for reports

EXAMPLES
       # Capture initial baseline
       sudo kmim baseline trusted_baseline.json

       # Scan for integrity violations
       sudo kmim scan trusted_baseline.json

       # View detailed module information
       sudo kmim show ext4

       # Start real-time monitoring
       sudo kmim monitor

       # Detect advanced threats
       sudo kmim detect-hooks

       # Continuous monitoring every 30 seconds
       sudo kmim continuous --interval 30

       # Generate HTML security report
       sudo kmim report --format html -o report.html

       # Test detection with simulation
       sudo kmim simulate rootkit

FILES
       kmim_baseline.json
              Default baseline file location

       /proc/modules
              Kernel module list

       /proc/kallsyms
              Kernel symbol addresses

RETURN VALUES
       0      Success
       1      Error or security violation detected

REQUIREMENTS
       - Linux kernel 4.4+ with eBPF support (CONFIG_BPF=y)
       - Root privileges
       - Python 3.8+
       - BCC (Berkeley Packet Filter Compiler Collection)
       - python3-bpfcc
       - python3-rich (for colored output)

SECURITY NOTES
```

```
       KMIM requires root privileges to access /proc/kallsyms and
       load eBPF programs. After system updates or legitimate kernel
       module changes, recapture the baseline to avoid false positives.

       The tool uses eBPF for safe kernel introspection - eBPF programs
       are verified by the kernel and cannot crash the system.

AUTHOR
       Aditya Pillai
       Software Security Lab (HPRCSE Group)

SEE ALSO
       lsmod(8), modinfo(8), bpftool(8)
```

14 Appendix B: Installation and Setup

14.1 System requirements

- **Operating System:** Linux kernel 4.4 or higher
- **Architecture:** x86_64 (AMD64)
- **Python:** Version 3.8 or higher
- **Kernel configuration:** CONFIG_BPF=y, CONFIG_KPROBES=y
- **Privileges:** Root access required

14.2 Installation steps

14.2.1 Debian/Ubuntu/Kali Linux

```
# Update package list
sudo apt update

# Install kernel headers
sudo apt install -y linux-headers-$(uname -r)

# Install BCC tools
sudo apt install -y bpfcc-tools python3-bpfcc

# Install Python dependencies
pip3 install rich

# Clone repository
git clone https://github.com/APillai03/KMIM.git
cd KMIM

# Verify installation
sudo python3 kmim.py --help
```

14.2.2 RHEL/CentOS/Fedora

```
# Install dependencies
```

```
sudo dnf install -y kernel-devel
sudo dnf install -y bcc-tools python3-bcc

# Install Python dependencies
pip3 install rich

# Clone and test
git clone https://github.com/APillai03/KMIM.git
cd KMIM
sudo python3 kmim.py --help
```

14.2.3 Arch Linux

```
# Install dependencies
sudo pacman -S linux-headers bcc python-bcc python-rich

# Clone and test
git clone https://github.com/APillai03/KMIM.git
cd KMIM
sudo python3 kmim.py --help
```

14.3 Verification

```
# Check kernel version
uname -r

# Verify BPF support
grep CONFIG_BPF /boot/config-$(uname -r)

# Check BCC installation
python3 -c "from bcc import BPF; print('BCC OK')"

# Verify kallsyms access
sudo cat /proc/kallsyms | head -n 5

# Test KMIM
sudo python3 kmim.py baseline test_baseline.json
```

15 Appendix C: Complete Usage Examples

15.1 Basic workflow

```
# 1. Initial setup - capture baseline on clean system
sudo python3 kmim.py baseline production_baseline.json

# 2. Regular integrity checks
sudo python3 kmim.py scan production_baseline.json

# 3. Investigate specific modules
sudo python3 kmim.py show suspicious_module

# 4. Monitor real-time activity
```

```
sudo python3 kmim.py monitor
```

15.2 Advanced threat hunting

```
# 1. Comprehensive anomaly scan
sudo python3 kmim.py detect-hooks

# 2. Start continuous monitoring
sudo python3 kmim.py continuous --interval 30 \
    --baseline production_baseline.json

# 3. Generate security report
sudo python3 kmim.py report --format html \
    -o "report_$(date +%Y%m%d_%H%M%S).html"

# 4. Archive JSON for SIEM
sudo python3 kmim.py report --format json \
    -o /var/log/kmim/scan_$(date +%Y%m%d).json
```

15.3 Incident response workflow

```
# 1. Detect compromise
sudo python3 kmim.py scan production_baseline.json
# Output: ALERT Added modules: evil_rootkit

# 2. Detailed investigation
sudo python3 kmim.py show evil_rootkit
sudo python3 kmim.py detect-hooks

# 3. Document findings
sudo python3 kmim.py report --format html \
    -o incident_$(date +%Y%m%d).html

# 4. Remove malicious module (if identified)
sudo modprobe -r evil_rootkit

# 5. Verify system integrity
sudo python3 kmim.py scan production_baseline.json

# 6. Update baseline after cleanup
sudo python3 kmim.py baseline production_baseline_clean.json
```

15.4 Testing and validation

```
# 1. Create test baseline
sudo python3 kmim.py baseline test_baseline.json

# 2. Simulate rootkit
sudo python3 kmim.py simulate rootkit

# 3. Test detection
sudo python3 kmim.py detect-hooks
```

```
# 4. Load test module
sudo python3 kmim.py simulate lkm
# Compile and load the generated test module

# 5. Scan for changes
sudo python3 kmim.py scan test_baseline.json
# Should detect: ALERT Added modules: test_module

# 6. Monitor in real-time
# Terminal 1:
sudo python3 kmim.py monitor

# Terminal 2:
sudo modprobe loop
sudo modprobe -r loop
# Watch events appear in Terminal 1
```

15.5 Integration with security pipeline

```
#!/bin/bash
# automated_scan.sh - Daily security scan

BASELINE="/opt/kmim/baselines/production.json"
REPORT_DIR="/var/log/kmim/reports"
DATE=$(date +%Y%m%d)

# Run scan
sudo python3 kmim.py scan "$BASELINE" > \
    "$REPORT_DIR/scan_${DATE}.txt"

# Generate reports
sudo python3 kmim.py report --format json \
    -o "$REPORT_DIR/scan_${DATE}.json"
sudo python3 kmim.py report --format html \
    -o "$REPORT_DIR/scan_${DATE}.html"

# Check for alerts
if grep -q "ALERT" "$REPORT_DIR/scan_${DATE}.txt"; then
    echo "Security alerts detected!" | \
        mail -s "KMIM Alert: $(hostname)" security@example.com

    # Send to SIEM
    curl -X POST https://siem.example.com/api/events \
        -H "Content-Type: application/json" \
        -d @"$REPORT_DIR/scan_${DATE}.json"
fi
```

16 Appendix D: Troubleshooting Guide

16.1 Common issues and solutions

16.1.1 BCC not found

Error: `ModuleNotFoundError: No module named 'bcc'`
Solution:

```
# Debian/Ubuntu
sudo apt install python3-bpfcc

# RHEL/CentOS
sudo dnf install python3-bcc

# Verify
python3 -c "from bcc import BPF"
```

16.1.2 Permission denied

Error: `Error: This command requires root privileges`
Solution:

```
# Always run with sudo
sudo python3 kmim.py <command>

# Or switch to root
sudo su -
python3 kmim.py <command>
```

16.1.3 kallsyms access restricted

Error: `Warning: /proc/kallsyms access restricted`
Solution:

```
# Temporarily allow access (until reboot)
sudo sysctl kernel.kptr_restrict=1

# Permanently (add to /etc/sysctl.conf)
echo "kernel.kptr_restrict=1" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
```

16.1.4 eBPF attach failed

Error: `Could not attach to any module functions`
Solution:

```
# Check available functions
cat /proc/kallsyms | grep -E 'do_init_module|free_module'

# Verify KPROBES enabled
grep CONFIG_KPROBES /boot/config-$(uname -r)

# Check kernel headers
sudo apt install linux-headers-$(uname -r)
```

```
# Verify eBPF support
sudo mount -t debugfs none /sys/kernel/debug
ls /sys/kernel/debug/tracing
```

16.1.5 Baseline file not found

Error: `Baseline file 'kmim_baseline.json' not found`
Solution:

```
# Create baseline first
sudo python3 kmim.py baseline kmim_baseline.json

# Or specify correct path
sudo python3 kmim.py scan /path/to/baseline.json
```

16.1.6 Rich library not found

Error: `Warning: 'rich' not installed`
Solution:

```
# Install rich
pip3 install rich

# Or use system package
sudo apt install python3-rich
```

16.2 Performance considerations

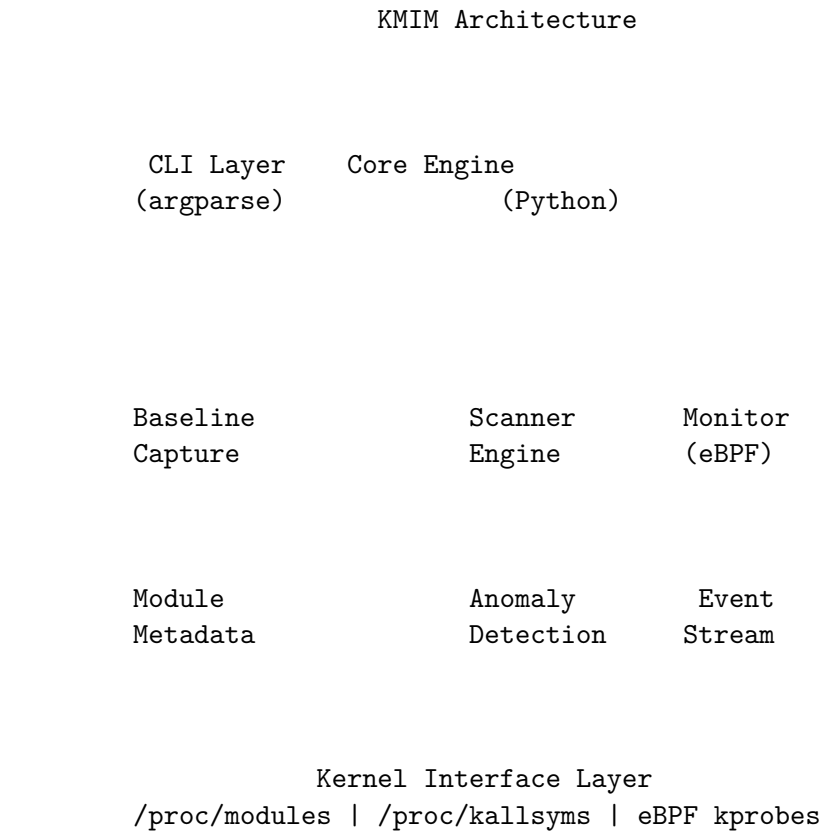
- **Baseline capture:** Typically takes 5-30 seconds depending on module count
- **Scan operation:** Usually completes in 2-10 seconds
- **Continuous monitoring:** Minimal CPU overhead (<1% on modern systems)
- **eBPF monitoring:** Very low overhead, kernel-verified for safety

16.3 Best practices

- Capture baseline immediately after fresh OS installation
- Schedule regular scans (daily recommended)
- Update baseline after legitimate system updates
- Store baselines in version control for audit trail
- Review reports weekly for security awareness
- Test detection capabilities with simulations quarterly
- Integrate with existing security monitoring infrastructure

17 Appendix E: Architecture Diagrams

17.1 System architecture



17.2 Data flow diagram

```
Baseline Capture Flow:

User Command → read_proc_modules() → mod_filename()
                                   → file_sha256()
                                   → extract_elf_sections()
                                   → guess_compiler()
             → read_kallsyms() → find_syscall_symbols()
             → JSON serialize → baseline.json

Scan Flow:

User Command → load_baseline(baseline.json)
             → read_proc_modules()
             → compare_modules()
             → verify_hashes()
```

```
                → get_hidden_modules()
                → detect_syscall_hooks()
                → generate_report()

Monitor Flow:

User Command → compile_ebpf_program()
                → attach_kprobes()
                → perf_buffer_poll()
                ↑
                event_handler()
```

18 Appendix F: Detection Techniques Comparison

Table 1: KMIM Detection Techniques

Technique	Target	Method	Part
Hash Comparison	Module Files	SHA256	1
Size Verification	Module Memory	Byte Count	1
Hidden Module Detection	/proc vs kallsyms	Symbol Analysis	1
Syscall Hook Detection	Syscall Table	Address Range	2
Pattern Matching	Module Names	Regex	2
Path Validation	Module Location	Directory Check	2
Memory Analysis	Module Permissions	State Flags	2
Real-time Monitoring	Load/Unload Events	eBPF Kprobes	1,2

19 Appendix G: Project Repository and Resources

19.1 Repository information

- **GitHub:** https://github.com/APillai03/KMIM
- **License:** MIT License
- **Documentation:** README.md in repository
- **Issue tracker:** GitHub Issues

19.2 Additional resources

- **eBPF documentation:** https://ebpf.io/
- **BCC project:** https://github.com/iovisor/bcc
- **Linux kernel security:** https://www.kernel.org/doc/html/latest/security/
- **Rootkit detection:** https://www.chkrootkit.org/

20 Conclusion

KMIM provides a comprehensive and robust mechanism for monitoring kernel module integrity using modern eBPF technology. The evolution from Part 1's foundational capabilities to Part 2's advanced threat detection demonstrates the tool's growth into a production-ready security monitoring solution.

20.1 Key achievements

20.1.1 Part 1 accomplishments

- Successfully implemented eBPF-based kernel monitoring without kernel modules
- 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

20.1.2 Part 2 enhancements

- Advanced multi-layered anomaly detection system
- Pattern-based suspicious module identification
- Syscall hook detection and validation
- Continuous monitoring with automated alerting
- Professional HTML and JSON reporting for SOC integration
- Attack simulation framework for testing and validation
- Memory permission analysis
- Comprehensive security scanning capabilities

20.2 Real-world applicability

KMIM is suitable for:

- Security Operations Centers (SOC) for continuous monitoring
- Incident response teams for forensic analysis
- System administrators for routine integrity checks
- Penetration testers for post-exploitation detection testing
- Research labs for kernel security analysis
- Compliance audits requiring integrity verification

20.3 Impact and contributions

- Demonstrates practical application of eBPF for security monitoring
- Provides open-source alternative to commercial integrity monitoring tools
- Educational resource for kernel security concepts
- Foundation for further research in kernel-level threat detection
- Contributes to Linux security ecosystem

20.4 Final remarks

The combination of baseline comparison, real-time monitoring, advanced anomaly detection, and comprehensive reporting makes KMIM a valuable tool for defending against kernel-level threats. By leveraging eBPF's safety guarantees and Python's ease of development, KMIM achieves its goal of providing accessible yet powerful kernel integrity monitoring for the Linux security community.

The tool continues to evolve with community contributions and remains committed to staying current with kernel security best practices and emerging threat landscapes.

For updates, contributions, and support, visit:
https://github.com/APillai03/KMIM