IronClad Kernel Guardian

Comprehensive Operations Manual v2.0

Document Classification: Technical / Operational
Target Audience: System Administrators, Security Operations Centers (SOC), Kernel Engineers **Date:** November 2025

Table of Contents

Chapter 1: The IronClad Philosophy

1.1 The Active Defense Paradigm

Traditional security tools (Antivirus, EDR) operate on a "Detection" basis. They scan files on disk or monitor logs after an event has occurred. This introduces a **latency gap**—the time between an attacker executing a payload and the defender noticing it.
IronClad operates on an **Active Defense** paradigm. It assumes that the perimeter has already been breached and focuses on **denial of capability**. Instead of asking "Is this file malicious?", IronClad asks "Is this process allowed to execute new code?"
By sitting directly on the execution path in the kernel, IronClad transforms the OS from a passive victim into an active combatant.

1.2 Why eBPF? The Kernel Advantage

eBPF (Extended Berkeley Packet Filter) allows us to run sandboxed programs inside the Linux kernel without changing kernel source code or loading risky kernel modules.

- **Safety**: The eBPF Verifier ensures our code cannot crash the kernel.
- **Visibility**: We can see every system call, every network packet, and every process spawn.
- **Speed**: eBPF programs are JIT-compiled into native machine code, running at CPU speed.

1.3 The "Zero-Downtime" Imperative

Patching a critical vulnerability (like Log4Shell or a glibc exploit) usually requires restarting services or rebooting servers. In high-availability environments, this downtime is costly.
IronClad provides **Virtual Patching**. By blocking the specific exploit triggers or the vulnerable process's ability to fork/exec, you can secure the system immediately without restarting anything, buying time for a proper patch window.

Chapter 2: System Architecture Deep Dive

IronClad uses a "Split-Brain" architecture to balance flexibility (User Space) with raw power (Kernel Space).

2.1 The Split-Brain Model

The application consists of two distinct components that run asynchronously but share memory maps.

1. **ironclad (Go Binary)**: Runs in user space. Handles UI, parsing CSVs, resolving PIDs, and high-level logic.
2. vuln_detector.o **(eBPF Object)**: Runs in kernel space. Hooks system calls and enforces decisions made by the Go binary.

2.2 User Space Controller (The "Brain")

Located in main.go, scanner/, and ui/.

- **Scanner Engine**: Queries dpkg (Debian/Ubuntu) or rpm (RHEL/CentOS) databases to build a software inventory.
- **Exploit Correlator**: Compares the inventory against the local exploitdb CSV using fuzzy string matching (e.g., matching "OpenSSH 8.2p1" against "OpenSSH < 8.5").
- **Map Manager**: Writes to the block_map eBPF map to update the "Kill List."

2.3 Kernel Space Enforcer (The "Sentry")

Located in ebpf_src/vuln_detector.bpf.c.

- **LSM Hook**: We attach to lsm/bprm_check_security. This Linux Security Module hook is triggered whenever a process calls execve() (tries to run a program).
- **Atomic Lookup**: It performs a specific O(1) lookup: Does the current PID exist in the Block Map?
- **Decision**:
    - **Yes**: Return -EPERM (Operation Not Permitted). The kernel kills the request immediately.
    - **No**: Return 0 (Allow). The process continues normally.

2.4 The Communications Bus

- **block_map (BPF_MAP_TYPE_HASH)**:
    - **Key**: u32 (PID)
    - **Value**: u8 (Status Flag)
    - **Flow**: User Space writes -> Kernel Space reads.
- events **(BPF_MAP_TYPE_RINGBUF)**:
    - **Data**: struct event_t { pid, type, comm }
    - **Flow**: Kernel Space writes -> User Space reads.
    - **Advantage**: Ring Buffers are shared memory regions that allow the kernel to push data to user space without the overhead of system calls or polling.

Chapter 3: Installation & Deployment Strategy

3.1 Kernel Prerequisites & BTF

IronClad requires **BTF (BPF Type Format)** information to be exposed by the kernel. This allows our C code to be "Portable" (CO-RE: Compile Once, Run Everywhere) by understanding the memory layout of kernel structures dynamically.

- **Check Support**: Run ls /sys/kernel/btf/vmlinux. If this file exists, you are ready.
- **Missing BTF**: On older kernels (pre-5.8) or stripped kernels, you may need to install linux-tools-generic and bpftool to generate a header file manually (handled by our installer).

3.2 The Universal Installer

The provided install.sh automates the dependency hell often associated with eBPF.

1. **Header Resolution**: It effectively runs apt-get install linux-headers-$(uname -r). This is critical because eBPF needs the *exact* headers for your running kernel version.
2. **Toolchain**: It installs clang (the compiler used for BPF) and llvm.
3. **Compilation**: It compiles vuln_detector.bpf.c on the *target machine*. This ensures perfect compatibility with the host kernel.

3.3 Systemd Persistence

To function as a true guardian, IronClad installs a systemd unit at /etc/systemd/system/ironclad.service.

- **Restart Policy**: Restart=always ensures that if the process crashes, it is immediately revived.
- **User**: Runs as root (Required for BPF map access).
- **Logs**: Output is redirected to journalctl. View with journalctl -u ironclad -f.

Chapter 4: The Interface (TUI) Reference

4.1 The Heads-Up Display (HUD)

The top bar displays critical system health:

- **CPU Load**: High load might indicate an active crypto-mining infection or scanning activity.
- **RAM Usage**: High RAM usage might indicate a memory leak or buffer overflow attempt.
- **Root Status**: A warning appears if the dashboard is merely "viewing" (non-root) and cannot "act" (mitigate).

4.2 Tab 1: Vulns (Vulnerabilities)

This is the strategic view.

- **Data Source**: Cross-reference of scanner.ScanSystem() (installed pkgs) vs scanner.LoadExploits() (CSV).
- **Visuals**:
  - **Red**: Critical exploits (Remote Code Execution, Root Privilege Escalation).
  - **Yellow**: Medium risk (DoS, Local Info Disclosure).
- **Workflow**: Identify a threat here -> Press M to activate defenses.

4.3 Tab 2: Logs & Forensics

This is the tactical view.

- **Initialization Logs**: Confirm that BPF maps are loaded and linked.
- **Mitigation Logs**: "MATCH: Exploit X mapped to PID Y". This confirms the logic engine is working.
- **ALARM Logs**: "⚠️ ALARM: Blocked execution attempt...". This is the most important line. It means **an attack was stopped**. It tells you *who* tried to execute (PID) and *what* they tried to run (Comm).

4.4 Tab 3: Process Manager

This is the operational view.

- **Real-time**: Updates every second.
- **Controls**:
  - K (Kill): Standard syscall.Kill(pid, syscall.SIGKILL). Use for non-critical processes behaving badly.
  - B (Block): The "IronClad Special". Adds the PID to the BPF map. Use this for critical services (like a web server) that you suspect are compromised but cannot shut down. It freezes their ability to spawn shells while keeping the main process alive to serve requests.

4.5 Tab 4: Network Firewall

This is the perimeter view.

- **Visibility**: Shows all active sockets.
- **Correlation**: If you see a connection to a suspicious foreign IP, you can immediately identify the PID owning it and hit B to block that PID from doing further harm.

Chapter 5: Operational Mechanics (Under the Hood)

5.1 The Scan Cycle

1. **Trigger**: On startup and every 6 hours (or manual U press).
2. **Inventory**: The Go binary executes dpkg-query -W. It parses the text output into struct Package { Name, Version }.
3. **Database**: It reads /opt/exploitdb/files_exploits.csv. This file contains ~45,000 rows.
4. **Fuzzy Matcher**: It iterates through the CSV. It checks if Exploit.Description contains Package.Name AND Package.Version.
    - *Optimization*: It skips exploits marked for "Windows", "MacOS", or "Hardware".

5.2 The Kill Chain (LSM Hook)

When you press M (Mitigate):

1. **Resolver**: The Go app takes the list of vulnerable *package names* (e.g., "openssh").
2. **PID Finder**: It scans /proc to find PIDs whose comm (command name) matches the package (e.g., processes named "sshd").
3. **Map Update**: It performs a bpf_map_update_elem syscall.
    - Key: PID (1234)
    - Value: 1
4. **Kernel Check**:
    - Process 1234 calls execve("/bin/bash", ...).
    - The CPU switches to Kernel Mode.
    - The lsm/bprm_check_security hook fires.
    - The BPF program hashes PID 1234 and looks it up in the map.
    - It finds 1.
    - It executes return -1.
    - The kernel sees the error and aborts the system call.
    - Process 1234 receives an "Operation not permitted" error.

5.3 The Ring Buffer: Nanosecond Telemetry

When the block happens:

1. The BPF program reserves space in the events Ring Buffer.
2. It writes the metadata (PID, Command Name, Timestamp).
3. It submits the event.
4. The Go app (running ebpf.StartEventLoop) wakes up instantly.
5. It decodes the binary data into a Go struct.
6. It prints the "ALARM" line to the UI.

- *Note*: This happens in microseconds, far faster than reading log files.

Chapter 6: Operational Playbooks

6.1 Scenario A: Active Exploitation Attempt

Situation: You see high CPU usage on your web server (nginx). You suspect a shell injection attack.

1. **Open IronClad**: sudo ironclad.
2. **Check Tab 3 (Procs)**: Find the nginx worker process with high CPU.
3. **Action**: Highlight the process and press B (Block).
4. **Monitor Tab 2 (Logs)**: Watch for "ALARM: Blocked execution attempt".
    - *Result*: If the attacker tries to spawn a reverse shell or a crypto-miner, IronClad blocks it. The web server continues serving legitimate HTTP requests (which don't require execve), but the RCE is neutralized.

6.2 Scenario B: Zero-Day Mitigation

Situation: A new vulnerability is announced for sudo (like PwnKit), but no patch is available yet.

1. **Open IronClad**.
2. **Tab 3 (Procs)**: Locate any running suspicious shells or services.
3. **Pre-emptive Block**: You can proactively block non-essential services that might be vectors.
4. **Update**: Press U to pull the latest ExploitDB. If the exploit has been added to the DB, IronClad will now flag your sudo version in Tab 1.
5. **Mitigate**: Press M. IronClad will automatically lock down the vulnerable processes.

6.3 Scenario C: Handling False Positives

Situation: You blocked a process, but now it can't do its legitimate job (e.g., a build server that *needs* to compile code).

1. **Identify**: The logs show "ALARM: Blocked..." for a legitimate action.
2. **Resolve**: Currently, IronClad v2.0 requires a restart to clear the BPF maps (systemctl restart ironclad).
    - *Future v2.1 Feature*: An "Unblock" key (U in Tab 3) is planned to remove PIDs from the map dynamically.

Chapter 7: Troubleshooting & Diagnostics

7.1 Kernel Headers & BTF Issues

Symptom: fatal error: 'vmlinux.h' file not found.
Cause: The compiler cannot find the type definitions for your running kernel.
Fix:

1. Install bpftool and linux-headers-$(uname -r).
2. Re-run the installer. It will attempt to dump the BTF data again.
3. If all else fails, the installer generates a "Stub" header that works for 90% of cases but may lack deep inspection features.

7.2 BPF Verifier Errors

Symptom: program check_exec: load program: permission denied or invalid access to map.
Cause: The Linux Kernel BPF Verifier rejected the code because it deemed it "unsafe" (e.g., potential infinite loop or invalid memory access).
Fix: This usually indicates a bug in the C code or an incompatibility with a very old kernel (< 5.7). Upgrade your kernel.

7.3 Performance Tuning

IronClad is designed to be lightweight. However, the Ring Buffer size (256 * 1024 bytes) is fixed.