IronClad Kernel Guardian

Technical Specification & Logic Reference v2.1

Document Type: Architecture Specification
Component: Core Logic Engine
Date: November 2025

1. System Architecture Overview

IronClad operates as a hybrid userspace-kernel system. The architecture is designed to minimize kernel-side complexity (for safety) while maximizing userspace intelligence.

1.1 High-Level Data Flow

1. **Ingest**: scanner module parses local package databases (dpkg/rpm) and the exploitdb CSV.
2. **Correlate**: The matcher algorithm identifies intersection points (CVEs matching installed versions).
3. **Target**: The resolver maps vulnerability names (e.g., "openssh") to active PIDs via /proc scanning.
4. **Enforce**: The loader pushes PIDs into a BPF Hash Map.
5. **Intercept**: The BPF program hooks execve syscalls, checks the map, and returns -EPERM.
6. **Telemetry**: The BPF program pushes event structs to a Ring Buffer, consumed by the GUI.

1.2 Operational Strategy: Universal Containment

IronClad employs a **Universal Containment** strategy rather than specific signature matching.

- **Logic**: It does not analyze the exploit payload (e.g., buffer overflow byte patterns). Instead, it assumes that if a software version is vulnerable, the running process is compromised or at risk.
- **Mechanism**: The defense is behavioral. It quarantines the process by revoking its privilege to spawn child processes (execve).
- **Rationale**: 99% of Remote Code Execution (RCE) attacks rely on spawning a shell (/bin/sh) or dropping a second-stage payload. Blocking execve neutralizes the kill chain effectively without needing to understand the specific vulnerability mechanics.

2. Kernel-Space Logic (The eBPF Agent)

The kernel agent is a compiled ELF object (vuln_detector.o) loaded into the Linux kernel. It is subjected to the BPF Verifier to ensure memory safety and finite execution time.

2.1 Hooks & Attachments

- **Program Type**: BPF_PROG_TYPE_LSM (Linux Security Module).
- **Attach Point**: lsm/bprm_check_security.
    - *Why this hook?* It executes *before* the kernel commits to running a new binary (during the execve syscall). This allows for pre-execution prevention.

2.2 Data Structures (Maps)

Map 1: block_map

- **Type**: BPF_MAP_TYPE_HASH
- **Key**: u32 (PID)
- **Value**: u8 (Status Flag: 1=Block)
- **Logic**: Serves as an O(1) lookup table for the "Kill List". Userspace writes, Kernel reads.

Map 2: events

- **Type**: BPF_MAP_TYPE_RINGBUF
- **Size**: 256KB
- **Logic**: A high-performance shared memory region. Kernel writes alerts here; userspace consumes them asynchronously. This avoids the overhead of perf_event polling.

2.3 Execution Flow (Pseudo-C)

```
int check_exec(struct linux_binprm *bprm) {
   // 1. Get Context
   u32 pid = bpf_get_current_pid_tgid() >> 32;

   // 2. Lookup Policy
   u8 *status = bpf_map_lookup_elem(&block_map, &pid);

   // 3. Decision Engine
   if (status && *status == 1) {
      // 3a. Prepare Alert
      struct event_t *e = bpf_ringbuf_reserve(&events, sizeof(*e), 0);
      if (e) {
         e->pid = pid;
         e->type = EVENT_BLOCKED;
```

```
        bpf_get_current_comm(&e->comm, 16); // Capture process name
        bpf_ringbuf_submit(e, 0);
    }

    // 3b. Log to Kernel Trace Pipe (Backup debug)
    bpf_printk("IronClad: DENY PID %d", pid);

    // 3c. Enforce Block
    // CRITICAL: This stops *any* new process execution from this PID.
    // It does not inspect arguments. It is a blanket denial.
    return -EPERM; // -1
  }

  // 4. Default Allow
  return 0;
}
```

3. User-Space Logic (The Go Runtime)

The Go application acts as the "Control Plane." It is responsible for business logic, UI rendering, and map management.

3.1 The Scanner Algorithm (scanner/packages.go)

1. **Detection**:
    - Executes dpkg-query -W -f='${Package},${Version}\n' to get raw inventory.
    - Parses output into []Package struct.
2. **Normalization**:
    - Converts version strings (e.g., 1:8.2p1-4ubuntu0.5) into comparable semantic versions.

3.2 The Matcher Algorithm (scanner/exploitdb.go)

This is the core intelligence loop. It performs an O(N*M) comparison between installed packages and the 45,000+ entry ExploitDB.

Optimization Logic:

- **Pre-Filter**: Discards exploits where Platform != "linux".
- **String Containment**: Checks if Exploit.Description contains Package.Name (case-insensitive).
- **Version Check**: If a name match occurs, it checks if Exploit.Description also contains the specific version string.
    - *Note*: This is a heuristic match. It identifies *potential* risks based on metadata, not code analysis.

3.3 The Enforcer Bridge (ebpf/loader.go)

This module manages the lifecycle of the BPF program.

1. **JIT Compilation Check**:
    - On startup, checks for vuln_detector.o.
    - If missing, invokes clang to compile vuln_detector.bpf.c against the locally generated vmlinux.h.
2. **Loading**:
    - Uses cilium/ebpf to load the ELF object into kernel memory.
    - Calls link.AttachLSM to bind the program to the security hook.
3. **Targeting (The "Lock" Logic)**:
    - When the user targets a vulnerability, FindPidsByName() resolves the process name to a list of PIDs.
    - It iterates through PIDs and calls block_map.Put(pid, 1).

3.4 Event Loop (StartEventLoop)

To ensure the UI is responsive, event consumption runs in a dedicated Goroutine.

1. Opens a reader on the events Ring Buffer.
2. Blocks on reader.Read() until the kernel pushes data.
3. Decodes the binary C-struct into a Go struct using binary.LittleEndian.
4. Formats a log message and sends it via a Go Channel (chan string) to the Bubble Tea UI loop.

4. Security & Safety Model

4.1 Stability Guarantees

- **No Kernel Modules**: IronClad does not load .ko modules, eliminating the risk of kernel panics caused by pointer errors. If the BPF program crashes (which the Verifier prevents), the kernel simply unloads it.
- **Fail-Open**: If the IronClad user-space daemon crashes, the BPF map remains in memory but no new PIDs are added. The existing blocks persist until reboot or explicit unload.

4.2 Performance Impact

- **Overhead**: The bprm_check_security hook adds approximately **20-50 nanoseconds** to each execve call.
- **Memory**: The BPF maps perform hash lookups, which are O(1). The impact on system throughput is statistically negligible (< 0.1%).

4.3 Privilege Requirements

- **CAP_BPF**: Required to load programs.
- **CAP_SYS_ADMIN**: Required for certain map operations and LSM attachments.
- *Conclusion*: The daemon must run as root.

4.4 Limitations (The "Gap")

Since IronClad blocks execution (execve) rather than analyzing memory or network traffic payloads:

- **Stopped**: Remote Code Execution (RCE), Shell Spawning, Binary Droppers.
- **Not Stopped**: Pure data exfiltration (reading files without spawning processes), Denial of Service (crashing the app), or Logic Bugs within the application itself.