

Analysis

Pattern: turning collected logs into signals. Collection gathers raw logs, analysis reduces them to the handful of statements worth acting on. This page shows one category in full, health monitoring, chosen because it is operational rather than a detection tripwire and therefore safe to show in its entirety, then continues to describe the shape the other categories follow without their sensitive specifics.

The shape of an analysis script

Every analysis script has the same skeleton, regardless of category:

1. Read the relevant logs for a recent time window.
2. Reduce them to a small number of facts (a count, a maximum, a change).
3. Compare those facts against a threshold.
4. Emit a signal (exit status, output, a log line) if the threshold is crossed.

The alerting layer, covered on the next page, runs these scripts on a schedule and delivers whatever they emit. Keeping each script to this narrow shape, one category, one window, one comparison, is what makes the whole system understandable. When something fires or fails to fire, you know exactly which small script to read.

The worked example: health monitoring

Health monitoring is a good worked example because knowing how it works helps no attacker, disk and CPU thresholds are not a security tripwire, and the pattern it demonstrates is exactly the one every other category follows. It comes in two halves: a collector on each host that emits health snapshots, and a check on the central host that reads them.

Prerequisite: the per-host health snapshot

This half is a prerequisite for the fleet check and worth setting up first. Each host runs a small script every ten minutes that measures its own resource usage and logs a single tagged line. That line is forwarded to the collector like any other log (via the `health` tag rule shown on the collection

page).

```
#!/bin/bash
# /usr/local/bin/sysstat-health.sh
# Runs on EVERY host, every 10 minutes via cron.
# Emits one tagged syslog line with current resource usage,
# which rsyslog forwards to the central collector.

set -e

HOST=$(hostname -s)
TS=$(date -Is)

# CPU usage as a percentage (user + system), sampled over 1 second.
CPU_USED=$(LC_ALL=C sar -u 1 1 | awk '/Average:/ { printf "%.1f", 100-$8 }')

# Memory usage as a percentage of total.
MEM_USED=$(LC_ALL=C sar -r 1 1 | awk '/Average:/ { printf "%.1f", ($3+$5)/$2*100 }')

# 1-minute load average.
LOAD1=$(awk '{print $1}' /proc/loadavg)

# Root filesystem usage as a percentage.
DISK_ROOT=$(df -P / | awk 'NR==2 {gsub("%","", $5); print $5}')

# Emit one structured line under the "health" tag at info severity.
# rsyslog matches the tag and forwards it to the collector as health.log.
logger -p local5.info -t health \
    "ts=${TS} host=${HOST} cpu=${CPU_USED}% mem=${MEM_USED}% load1=${LOAD1}
disk_root=${DISK_ROOT}%"
```

Scheduled on each host with a cron entry:

```
# /etc/cron.d/sysstat-health
*/10 * * * * root /usr/local/bin/sysstat-health.sh
```

The host does not decide whether its own numbers are a problem. It just reports them, every ten minutes, in a structured line. All the judgment lives in one place on the collector, which keeps the per-host piece trivial and means you tune the logic in exactly one location rather than on every host.

The check: reading the fleet's health on the collector

On the collector, a script runs on a schedule and reads the forwarded health snapshots for all hosts over the recent window. Because each host reports every ten minutes, a window of the last hour gives several data points per host, enough to tell a momentary spike from a sustained problem. The script's logic, in shape:

- For each host, read its health snapshots from the recent window.
- Determine whether any resource crossed a concern threshold, and whether a transient reading (one spike) should be treated differently from a sustained one (crossing the line repeatedly across the window).
- Emit a signal naming any host that warrants attention.

```
#!/bin/bash
# /usr/local/bin/fleet-health-check.sh
# Runs on the COLLECTOR on a schedule (see the alerting page).
# Reads the forwarded health snapshots for every host over a recent
# window and emits one WARN line per host/metric that crosses a line.
# Empty output means everything is OK.
set -euo pipefail

BASE_DIR="/var/log/remote"
WINDOW=6          # recent samples to consider (~6 x 10min = 1h)
CPU_WARN=80
MEM_WARN=80
DISK_WARN=80
LOAD_FACTOR_WARN=1.0  # reserved: warn if load1 > cores * this (raw for now)

# Hosts to exempt from the memory check. Left commented until needed.
# See "A note on the commented-out exception" below.
#MEM_SKIP_HOSTS=("examplehost")  # remove after migration

alerts=()

# One health.log per host, filed by the collection stage.
for health_file in "$BASE_DIR"/*/health.log; do
    [ -f "$health_file" ] || continue
    host="$(basename "$(dirname "$health_file")")"
```

```

# Last WINDOW health lines for this host.
chunk="$(grep ' health: ' "$health_file" | tail -n "$WINDOW")"
[ -n "$chunk" ] || continue

# Parse the window into: max CPU, max mem, last disk, last timestamp,
# last load, and how many samples met/exceeded the CPU threshold.
# Tracking cpu_hits (not just the max) is what lets us require a
# SUSTAINED condition rather than firing on a single spike.
read -r max_cpu max_mem last_disk last_ts last_load cpu_hits <<<"$(printf '%s\n' "$chunk"
| awk -v cpu_warn="$CPU_WARN" '
    $3 == "health:" {
        ts = cpu = mem = disk = load = ""
        for (i = 4; i <= NF; i++) {
            if ($i ~ /^ts=/)          ts = substr($i, 4)
            else if ($i ~ /^cpu=/)    { v=$i; sub(/^cpu=/,"",v); sub(/%$/,"",v); cpu=v+0 }
            else if ($i ~ /^mem=/)    { v=$i; sub(/^mem=/,"",v); sub(/%$/,"",v); mem=v+0 }
            else if ($i ~ /^disk_root=/) { v=$i; sub(/^disk_root=/,"",v); sub(/%$/,"",v);
disk=v+0 }
            else if ($i ~ /^load1=/)  { v=$i; sub(/^load1=/,"",v); load=v+0 }
        }
        if (cpu > max_cpu) max_cpu = cpu      # track worst CPU in window
        if (mem > max_mem) max_mem = mem      # track worst mem in window
        last_disk = disk                    # disk: current value is what matters
        last_ts = ts
        last_load = load
        if (cpu >= cpu_warn) cpu_hits++      # count threshold breaches
    }
END {
    if (last_ts == "") {
        print "0 0 0 - 0 0"                # no valid samples
    } else {
        printf "%.1f %.1f %.0f %s %.2f %d\n",
            max_cpu+0, max_mem+0, last_disk+0, last_ts, last_load+0, cpu_hits+0
    }
}
')"
```

```

# Skip a host whose parse produced no usable timestamp.
[ "$last_ts" != "-" ] || continue

```

```

msg_prefix="host=${host} ts=${last_ts}"

# CPU: require the threshold to be met at least TWICE in the window,
# so a single transient spike does not alert. This is the spike-vs-
# sustained distinction, in code.
cpu_int=${max_cpu%.*}
if (( cpu_hits >= 2 )); then
    alerts+=("WARN CPU ${msg_prefix} cpu=${max_cpu}% (>=${CPU_WARN}% in
${cpu_hits}/${WINDOW} samples)")
fi

# Memory: alert if the window's peak crossed the line.
mem_int=${max_mem%.*}
if (( mem_int >= MEM_WARN )); then
    alerts+=("WARN MEM ${msg_prefix} mem=${max_mem}% (max in last ${WINDOW} samples)")
fi

# Disk: alert on the current value, since disk pressure is a state,
# not a spike.
if (( last_disk >= DISK_WARN )); then
    alerts+=("WARN DISK ${msg_prefix} disk_root=${last_disk}%")
fi
done

# Emit all alerts. Empty output is the OK signal the alerting layer keys on.
if (($#alerts[@] > 0)); then
    for a in "${alerts[@]"; do
        echo "$a"
    done
fi

```

The distinction between a single spike and a sustained condition is the useful part of the logic. A host momentarily busy is normal, a host busy across the whole window is a problem, and treating them the same produces either false alarms or missed real issues. Requiring a condition to persist across multiple readings before alerting is what keeps health monitoring quiet enough to trust. The exact thresholds and how many readings count as "sustained" are tuning particular to an environment, and are the kind of specific this book leaves you to set against your own normal.

The other categories, in shape

Every other analysis category, authentication, boundary events, critical severity, service-specific signals, follows the same skeleton as the health check, differing only in which logs it reads and what condition it looks for. Described in general, without the sensitive specifics:

- **Authentication analysis** reads the forwarded authentication logs for a recent window, counts failure patterns by source, and emits a signal when a source's behaviour looks like an attack rather than a fumble. The precise counts and the exact patterns that distinguish "attack" from "bad day" are the sensitive tuning, omitted deliberately.
- **Boundary analysis** reads the firewall and intrusion-prevention logs, aggregates recent drops and bans, and emits a signal when the volume or pattern from a source suggests a probe or flood rather than background noise.
- **Critical-severity analysis** reads the per-host critical logs (which the collection stage already separates out) for the recent window and emits a signal if anything appeared at all, since by definition these are events the systems themselves flagged as serious.
- **Service-specific analysis** reads a particular service's logs for the specific signal that service warrants, a new registration, a sensitive action, a known error pattern, and emits accordingly.

In every case the skeleton is identical. Read a window, reduce to facts, compare to a line, emit a signal. Once you have written one, the rest are variations, which is what makes this approach maintainable by one person.

Why scripts rather than a query language

An enterprise SIEM would express these as saved queries or correlation rules in its own language. Small single-purpose shell scripts do the same job here with no product, and they have a real advantage at this scale. They are transparent and debuggable with tools you already know. When a check misbehaves, you read a short script and run it by hand, rather than reverse-engineering a rule engine's behaviour. The tradeoff is that scripts do not scale to thousands of sources or complex cross-source correlation, at which point you would want a real SIEM.

Revision #3

Created 2026-07-31 20:25:28 UTC by Chris Landis

Updated 2026-07-31 22:18:19 UTC by Chris Landis