

Homebrew SIEM

How to build meaningful security monitoring without a SIEM product, and how to reason about what to watch.

- [Concepts](#)
- [Architecture](#)
- [Collection](#)
- [What to Monitor](#)
- [Analysis](#)
- [Alerting and Scheduling](#)
- [Operations](#)
- [Lessons Learned](#)

Concepts

Meaningful security monitoring without a SIEM product, and how to reason about what to watch.

This book is a working reference for building basic security monitoring across a self-hosted (linux) environment using tools you already have: the system logger, a few scripts, and an automation layer. It is not a guide to deploying an enterprise SIEM product. It opens with what a real SIEM does and why a small environment does not need most of it, then covers centralizing logs, deciding what is worth watching, turning logs into signals, and getting alerts to a human.

A note on what this book deliberately omits. A monitoring system's exact thresholds and detection logic are kept private, because publishing where the lines are just tells someone how to stay under them. So this book is generous with architecture and reasoning and deliberately quiet on specifics. You will find the shape of how detection works and the categories worth monitoring, but not the exact numbers, timings, or parsing rules. Those are particular to an environment anyway, and adapting the approach to your own is more useful than copying someone else's tripwires.

What a SIEM actually is

SIEM stands for Security Information and Event Management. Underneath the acronym it names its function: gather logs and events, put them in one place, analyze them for signs of trouble, and alert a human when something warrants attention. Collect, centralize, analyze, alert.

The products that carry the SIEM label are another matter. A commercial or enterprise SIEM typically offers a large set of capabilities:

- **Log aggregation at scale** from hundreds or thousands of sources, with parsing and normalization into a common schema.
- **Long-term retention and fast search** across enormous volumes of historical events, for investigation and compliance, and potentially forensic evidence.
- **Correlation engines** that tie together events from different systems to spot multi-step attacks no single log would reveal.
- **Threat intelligence integration**, matching observed activity against feeds of known-bad indicators.
- **User and entity behaviour analytics**, modelling normal behaviour and flagging deviations.
- **Prebuilt detection rules** mapped to known attack techniques, maintained by the vendor.

- **Case management and workflow**, so a security team can triage, assign, and track incidents.
- **Dashboards and compliance reporting** for auditors and management.

That is a serious toolset, and for an organization with a security team, meaningful traffic, and regulatory obligations, it earns its cost. It is built for scale, for teams, and for threat models that a small ecosystem doesn't necessarily have.

Why a small ecosystem does not need most of that

In a self-hosted environment serving a few dozen people, most of that capability is overkill, and paying for it, in money or in operational weight, would be spending on a threat model you do not have. You are not correlating attacks across thousands of endpoints. You do not have a compliance regime demanding a year of searchable retention. You do not have a team to staff a case-management workflow. You have less than a dozen hosts, a small set of services, and one person who needs to know when something is wrong.

What that person actually needs is narrow. Know when someone is trying to break in, know when a host is unhealthy, know when a security control has fired, and get told about it without having to watch logs by hand. That is the *collect, centralize, analyze, alert* function, and every piece of it can be built from tools already present on the systems. The system logger for collection, a central host to receive the logs, small scripts to analyze them, and an automation layer to send alerts.

A homebrew SIEM is an essentially free basic monitoring kit, not a full-fledged SIEM. It gives you the core function, detection and alerting appropriate to a small environment, without the product, the cost, or the operational weight. It will not do behaviour analytics or correlate a sophisticated multi-stage intrusion without a lot of planning and scripting. It will tell you that something is hammering your login endpoint, that a host is running out of disk, that a firewall is dropping a flood of traffic, or that a service is throwing critical errors, which, for a small environment, is most of what monitoring is for.

The honest limits

- **This is detection and alerting, not a security operations centre.** It watches for the categories of trouble worth watching and tells you about them. It does not investigate, correlate broadly, or hunt for threats you did not think to look for.
- **Coverage is a deliberate, bounded choice.** You monitor the things most worth monitoring and accept that you are not watching everything. That is the correct tradeoff

at this scale, but it is a tradeoff, and pretending otherwise would be dishonest.

- **The monitoring is itself infrastructure that must be secured.** The central log collector and the automation layer that sends alerts are high-value. The collector holds everyone's logs, and the automation layer can reach the environment and send mail. Monitoring infrastructure needs the same scrutiny as what it monitors, a point the operating page returns to.

What the following pages cover

- **The architecture:** the collect, centralize, analyze, alert pipeline.
- **Collection:** forwarding logs from every host to one place.
- **What to monitor, and why:** the categories worth watching, and the reasoning, without the tripwire specifics.
- **Analysis:** turning collected logs into signals, with one worked example and the rest in general terms.
- **Alerting and scheduling:** getting signals to a human, by cron or by an automation layer, and how often.
- **Operating it:** tuning out noise and watching the monitor itself.
- **What I'd tell someone starting out:** the lessons learned.

Architecture

The whole system is four stages in a line: collect, centralize, analyze, alert. Every later page is one of these stages in detail.

The pipeline

1. **Collect.** Every host generates logs already; authentication attempts, firewall decisions, service errors, system messages. The system logger on each host is the source. No new agent is needed because the logging is happening whether you use it or not. The job is to route the useful parts off the host.
2. **Centralize.** Each host forwards its logs to one central collector. From that point on there is a single place that holds a copy of what every host saw, which is the foundation everything else builds on.
3. **Analyze.** On the collector, small scripts read the gathered logs and reduce them to signals: not "here are ten thousand log lines" but "this many failed logins from one address in the last hour," "this host's disk crossed a capacity threshold," "a critical error appeared." Analysis turns volume into meaning.
4. **Alert.** When a script produces a signal worth acting on, an alerting layer delivers it to a human. Most of the time nothing fires; the point is that when something does, you are told rather than having to look.

Why centralize at all

Forwarding logs off each host, rather than analyzing them where they sit, buys three things that matter:

- **A compromised host cannot cover its tracks.** If an attacker gets onto a host, one of the first things they may do is tamper with local logs to hide what they did. If the relevant events have already been forwarded off the host as they happened, that tampering is defeated. The evidence is already somewhere the attacker does not control. This alone is a strong argument for centralization.
- **Correlation across hosts becomes possible.** Some things are only visible when you can see several hosts at once, the same source probing multiple services, a pattern that moves from host to host. Logs scattered on their originating machines cannot show you that. Logs in one place can.

- **The logs survive the host.** If a host falls over, crashes, is compromised, is rebuilt, its local logs may be lost or suspect. Forwarded logs persist independently of the host that produced them.

Where analysis lives

Analysis runs on the central collector, close to the gathered logs, as a set of small single-purpose scripts. Each script owns one category, reads that category's logs for a recent window, and emits a signal if something crosses a line. Keeping each script narrow, one concern each, is what makes the system understandable and maintainable by one person. When an alert fires, you know exactly which small script to look at.

This is deliberately not a big correlation engine. It is a handful of focused checks, which is the right amount of machinery for the job at this scale.

Where alerting lives

The final stage, turning a signal into a notification a human receives, is handled by a scheduling-and-automation layer that runs the analysis scripts on a cadence and delivers their output. This can be as simple as cron and a mail command, or a dedicated automation platform. The alerting page covers both, and the tradeoffs between them, including the fact that the automation layer is itself something you have to secure.

The whole thing in one sentence

Every host forwards its logs to one collector, small scripts on the collector reduce those logs to signals, and an automation layer delivers the signals that matter to a human. That is the entire architecture, and its simplicity is the point. It is built from standard tools, it is understandable in one sitting, and it delivers the core SIEM function without the SIEM product.

Collection

Pattern: getting logs off every host and into one place. This is the plumbing, getting this stage right is what everything else depends on.

The mechanism is the standard Linux system logger (rsyslog), configured on each host to forward, and on one central host to receive. No extra software.

The receiving side: the collector

One host is designated the collector. It runs a listener that accepts forwarded logs from the other hosts and writes them into per-host directories, so each host's logs stay separate and identifiable.

Enable the listener (here on TCP port 514):

```
# /etc/rsyslog.d/10-listen.conf  (collector only)
module(load="imtcp")
input(type="imtcp" port="514")
```

Then route incoming logs into a directory per source host. A template builds the path from the sending host's name and the program that produced the log, so logs land at predictable locations:

```
# /etc/rsyslog.d/40-remote.conf  (collector)

# Critical events (severity <= err) copied to a per-host critical log,
# in addition to their normal destination.
if ($fromhost-ip != "127.0.0.1" and $syslogseverity <= 3) then {
    action(
        type="omfile"
        File="/var/log/remote/%HOSTNAME%/critical.log"
    )
    # no "stop": the message continues to its normal per-program log too
}

# Everything else from a remote host: file it under the host's directory,
# named for the program that produced it.
template(name="RemoteLogs" type="string"
```

```

string="/var/log/remote/%HOSTNAME%/%PROGRAMNAME%.log")

if ($fromhost-ip != "127.0.0.1") then {
    action(type="omfile" DynaFile="RemoteLogs")
    stop
}

```

The result on the collector is a tidy tree: `/var/log/remote/<host>/<program>.log`, with a separate `critical.log` per host catching anything high-severity. That structure is what the analysis scripts read later; predictable paths simplify analysis.

The sending side: every host

Each host forwards the log streams worth centralizing to the collector. A named ruleset defines the forwarding action once, with a queue so that logs are not lost if the collector is briefly unreachable:

```

# /etc/rsyslog.d/60-remote.conf  (every host)

template(name="RemoteFormat" type="string"
    string="%timereported% %HOSTNAME% %syslogtag%%msg%\n")

ruleset(name="sendToFiles") {
    action(
        type="omfwd"
        target="x.x.x.x"           # placeholder: the collector's address
        port="514"
        protocol="tcp"
        template="RemoteFormat"
        action.resumeRetryCount="100" # keep retrying if the collector is down
        queue.type="linkedlist"      # queue in memory rather than dropping
        queue.size="10000"
        queue.saveonshutdown="on"    # persist the queue across a restart
    )
}

```

The queue settings matter more than they look. `action.resumeRetryCount` and the linked-list queue mean that if the collector is down for a while, the sending host holds its logs and delivers them when the collector returns, rather than silently dropping them. For a monitoring system, silently losing the events you are trying to monitor is the worst failure, so the forwarding is configured to tolerate an outage without data loss.

With the ruleset defined, each host calls it for the log streams worth forwarding. Rather than shipping every log line from every host (which would drown the collector in noise), each host forwards the categories that matter: high-severity events, authentication activity, boundary and firewall events, health snapshots, and a few service-specific streams. The next page is about which categories those are and why.

Selecting categories: the matcher shapes

Category selection is a set of rules, each of which matches one kind of log and calls the forwarding ruleset for it. The rules almost all take one of three shapes, and knowing the three is enough to express any selection you need:

```
# Shape 1: match by tag.
# For logs you emit yourself under a known tag (like the health snapshots).
if $syslogtag startswith "health" then {
    call sendToFiles
    stop
}

# Shape 2: match by severity.
# A single rule that forwards anything the system itself flagged as serious.
if $syslogseverity <= 3 then {          # emerg/alert/crit/err
    call sendToFiles
    stop
}

# Shape 3a: match by message content.
# For signals identified by text in the log line, such as firewall drops.
if ($msg contains "[UFW " or $msg contains "UFW BLOCK") then {
    call sendToFiles
    stop
}

# Shape 3b: match by program name.
# For everything a particular program emits, such as the intrusion-prevention tool.
if ($programname == "fail2ban") then {
    call sendToFiles
```

```
    stop
}
```

The three shapes cover the cases:

- **By tag** for streams you generate and label yourself, like the per-host health snapshots. You control the tag, so matching it is exact.
- **By severity** for the catch-all "something serious happened" category, which needs no per-service knowledge because the severity is already assigned.
- **By content or program name** for signals produced by other software, where you match either a distinctive string in the message (a firewall's drop marker) or everything a given program emits (the intrusion-prevention daemon). Content matching is for when one program's logs contain both interesting and uninteresting lines; program matching is for when you want the whole stream. Every category you choose to forward is one of these rules. Boundary events, authentication signals, service-specific streams, they differ only in what they match, not in shape. Once you have written the four above, adding a category is copying the closest shape and changing the match.

When a service logs to files, not syslog

The forwarding above assumes a host's programs log through the system logger, which most system services do. But plenty of applications write their own log files instead and never touch syslog: web servers, containerized apps, and many self-hosted services log to a path under their own directory. Those logs will not be collected by the rules above, because they never enter rsyslog in the first place.

The bridge is rsyslog's `imfile` module, which tails a file and injects each new line into rsyslog as though it had been logged normally, with a tag and facility you assign. Once a file-based log is tagged this way, it flows through the same forwarding rules as everything else. This is how a service that logs to files gets pulled into the pipeline without changing the application.

A per-service custom config on the host that produces the logs (kept in its own file so it is easy to find and change) does the tailing:

```
# /etc/rsyslog.d/61-custom.conf (on the host running the service)
module(load="imfile")

# A service's auth log (e.g. failed logins, MFA events).
input(
    type="imfile"
```

```

File="/opt/<service>/logs/identity/*.txt"    # the app's own log path
Tag="myservice-identity"                    # tag it so rules can match it
Severity="info"
Facility="local6"
PersistStateInterval="100"                  # remember read position periodically
freshStartTail="on"                         # on first start, read new lines only,
                                             # not the entire existing file
)

# The same service's web access log (e.g. registrations, requests).
input(
    type="imfile"
    File="/opt/<service>/logs/nginx/access.log"
    Tag="myservice-nginx"
    Severity="info"
    Facility="local6"
    PersistStateInterval="100"
)

```

Two options are worth understanding:

- **PersistStateInterval** makes rsyslog remember how far it has read into each file, so a restart resumes where it left off rather than re-reading (and re-forwarding) everything.
- **freshStartTail="on"** means that the very first time rsyslog sees the file, it starts from the end rather than ingesting the entire backlog. Without it, a first start floods the collector with the whole history of the file. Use it on high-volume logs where the backlog is noise; omit it where you genuinely want the existing contents ingested once. With the file tagged, the collector's routing files it like any other tagged stream, and the forwarding rules send it on. The tag is the seam: everything downstream, the per-host log file on the collector, the analysis scripts, treats a tailed file-based log exactly like a native syslog stream, because by the time it leaves the host it is one.

A note on what is forwarded

Not everything is forwarded, and that is deliberate. Forwarding every log line would bury the signal in volume and strain the collector. The selection, which categories get sent, is a design decision, and the subject of the next page. The collection mechanism shown here is neutral plumbing; the judgment is in what you choose to run through it.

Adapt this for...

Any environment that needs central logging, which is any environment with more than one host. The tool is rsyslog and the transport is plain syslog over TCP, but the shape is universal; one collector with a listener and per-source separation, every host forwarding a selected set of streams with a queue that tolerates the collector being briefly down. Get the paths predictable and the forwarding loss-tolerant, and the analysis stage has a solid foundation to read from.

What to Monitor

Pattern: deciding what is worth watching. This is the judgment page, and it is deliberately general. It walks the categories worth monitoring and the reasoning for each, without the thresholds or exact detection logic, because those are the part that helps an attacker if published. The transferable value is the reasoning: *why* each category earns a place, and what class of trouble it catches.

The governing principle is that you cannot watch everything, and trying to would bury the signal that matters under the volume that does not. So monitoring is a series of deliberate choices. Watch the categories most likely to reveal trouble, and accept the risk of what you are not watching. Here are the categories that earn their place, and why.

Authentication events

Why it matters: authentication is where an outsider becomes an insider, so it is the single most valuable thing to watch. A burst of failed logins, attempts against accounts that do not exist, or a sudden change in the pattern of who is authenticating from where, are the earliest visible signs of someone trying to get in.

What it catches: brute-force attempts, credential-stuffing, username enumeration (probing to discover which accounts exist), and the difference between a real user fumbling their password and an automated tool grinding through guesses. The identity provider and any service handling logins are the sources.

The reasoning: because centralized authentication funnels most logins through one system, watching that one system's events gives broad coverage cheaply. This is a direct dividend of consolidating identity; one place to watch for the most important category of event.

Boundary and perimeter events

Why it matters: these are the events from the controls that sit at the edges; the firewall, the intrusion-prevention tooling, the web application firewall, the VPN. They record what was refused at the boundary, which is a running picture of what is probing you.

What it catches: port scans and connection attempts the firewall dropped, IPs that tripped automated banning, requests the web application firewall refused, and connection attempts to the VPN. Individually these are routine background noise; in aggregate, a spike or a pattern is the

signal. A sudden surge of drops from one source or a coordinated probe across services.

The reasoning: the boundary controls are already making decisions and logging them. Watching their output turns controls that silently defend into controls you can see defending, which is both reassuring when quiet and informative when not.

System health

Why it matters: not every threat is an attacker. A host that runs out of disk, exhausts memory, or pegs its CPU can take a service down as surely as an intrusion, and a resource anomaly can also be a *symptom* of one (a sudden disk fill from logging a flood of attacks, a CPU spike from something that should not be running).

What it catches: hosts approaching resource exhaustion, unusual load, and disk filling over time. This is the category with the most operational (rather than strictly security) value.

The reasoning: availability is part of security, and health monitoring is cheap. The same pipeline that watches for attackers may as well watch for the mundane failures that are statistically far more likely to actually take something down.

Critical-severity events

Why it matters: the logging system already classifies events by severity, and the highest levels, the errors and critical conditions, are worth surfacing regardless of which service produced them. A critical event is by definition something the system itself considered serious.

What it catches: service failures, crashes, and the kind of severe conditions that a service flags on its own. Rather than needing a bespoke check per service, a single rule that surfaces anything above a severity line catches the serious problems across everything at once.

The reasoning: this is high-value and low-effort. The severity classification is free and already applied; watching the top of it is a broad net for "something is seriously wrong" that needs no per-service tuning.

Service-specific signals

Why it matters: a few services warrant their own attention beyond the generic categories, either because they are especially sensitive (a password manager, the identity provider) or because they emit signals worth watching that the generic categories would not capture (new account registrations, specific security-relevant actions).

What it catches: things particular to a service, such as a new user registering somewhere that should rarely see new users, a sensitive administrative action, a service-specific error pattern. These are narrow, deliberate additions on top of the broad categories.

The reasoning: the generic categories give broad coverage; a small number of service-specific checks add depth exactly where the environment's most sensitive services are. Add these sparingly, for the services that genuinely warrant it, not as a reflex for every service.

The coverage-versus-noise tradeoff

Every category above is a choice to watch something, and implicit in each is a choice not to watch a hundred other things. The goal is not total coverage, it is the highest-value coverage that stays quiet enough to be trusted. A monitoring system that watches everything generates so much noise that its alerts get ignored, which is worse than watching less and actually reading what it reports. Choose the categories that reveal the most trouble per unit of noise, and be disciplined about not adding checks that generate more false alarms than genuine findings.

Where exactly the lines sit within each category, the thresholds, the windows, the specific conditions, is the tuning that this book deliberately does not publish. Those values are where an attacker would look to learn what they can do without tripping an alert, and they are specific to an environment's normal traffic in any case. The categories and the reasoning transfer; the tripwire settings you work out for yourself, against your own baseline.

Adapt this for...

Any environment deciding what to monitor. The categories, authentication, boundary, health, critical severity, and a few service-specific signals, are a sound starting set for almost any small environment, because they concentrate on where trouble first becomes visible. Adapt the specifics to what you run, keep the discipline of watching high-value categories rather than everything, and set your own thresholds against your own normal. The judgment, not the numbers, is the thing worth carrying over.

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.

Alerting and Scheduling

Pattern: getting signals to a human, on a cadence. Analysis produces signals, this stage runs the analysis on a schedule and delivers whatever warrants attention. There are two ways to do it, a scheduler and a script, or a dedicated automation platform, and the right choice depends on how much orchestration you want and how much attack surface you are willing to add.

The two approaches

Cron and a mail command. The simplest possible alerting layer, cron runs each analysis script on a schedule, and the script (or a wrapper) mails its output when there is something to report. Nothing to install, nothing new to secure, universally understood. This is what most people should reach for first, and for many environments it is entirely sufficient.

An automation platform. A dedicated workflow tool runs the scripts, collects their output, and handles delivery, formatting, and routing, with a visual interface for building and maintaining the workflows. It offers more than cron, richer notification formatting, conditional routing, easy fan-out to multiple destinations, retries, and a place to see run history, at the cost of being another piece of software to run, secure, and keep patched.

This environment uses an automation platform (n8n) for the alerting workflows, so this book describes that approach more fully, but the transferable idea is the *pattern*. An automation layer runs the checks and routes their output, not the specific product. If you prefer cron and a mail command the whole system works exactly the same. Only this final delivery stage changes.

A necessary caution about the automation layer

Recommending any automation platform to a security-minded reader requires honesty about a real consideration. **An internet-adjacent automation platform is attack surface, and it has to be secured and kept patched like everything else.** These platforms are powerful precisely because they can reach across your environment and take actions, which is exactly what makes them a high-value target. N8n specifically has had security vulnerabilities disclosed, as such platforms periodically do, and running one means committing to keeping it current and to put reasonable constraints on what it can reach.

If you adopt an automation platform for alerting:

- Keep it patched, and follow its security advisories, because a tool that can reach your whole environment is one you cannot afford to run stale.
- Constrain what it can reach and what credentials it holds to the minimum its workflows actually need.
- Treat it as sensitive infrastructure, not just a convenience.

There is a broader principle underneath this, returned to on the operating page: the monitoring system is part of the environment it monitors. The automation layer that sends your alerts can reach your hosts and send mail on your behalf. That makes it one of the more sensitive things you run, and it deserves scrutiny accordingly.

Frequency: matching the cadence to the category

How often a check runs should match how quickly its category matters. The reasoning by class:

- **Security-relevant categories run frequently**, on the order of every hour or so. Authentication attacks, boundary probes, and critical errors are things you want to know about while they are happening, not the next day. A shorter interval means faster awareness, at the cost of running the checks more often, which at this scale is negligible.
- **Health and capacity categories run less frequently**, on a daily rhythm for the slower-moving ones. Disk filling over time is a trend, not an emergency measured in minutes; a daily check catches it with plenty of margin. The per-host health snapshots underneath still happen every few minutes, but the *alerting* check that reads them can run hourly for acute spikes and daily for slow trends.
- **Reports and summaries run on a long cadence**, weekly or monthly, because their value is in the trend over time rather than immediate notification.

The point is to run each check as often as its category's trouble develops, and no more often than that. Running everything every minute would add load and noise for no benefit. Running everything daily would miss a fast-moving attack. Match the cadence to how fast the thing you are watching for actually moves.

Sample cron job

In order to get email alerts, you need to have a working MTA like `postfix` or `sendmail`. They are well documented and easy to set up. <https://ubuntu.com/server/docs/how-to/mail-services/>.

```
# /etc/cron.d/fleet-health-check
# Run the fleet health check hourly, at 5 past the hour.
# Runs on the COLLECTOR (the files host), as root, since it reads
# under /var/log/remote. Output (WARN lines, or nothing) is mailed
# to the address in MAILTO when the script produces any output.
```

```
MAILTO=alerts@example.org
```

```
5 * * * * root /usr/local/bin/fleet-health-check.sh
```

You would need to set up a job like this to execute each script on the cadence that aligns to the requirements of the category.

Adapt this for...

Any environment turning signals into notifications. Start with cron and a mail command; it is the simplest thing that works and adds no attack surface. Move to an automation platform only if you genuinely want its orchestration, and if you do, run it as the sensitive, patched, minimally-privileged infrastructure it is. Either way, match each check's frequency to how fast its category moves. The delivery mechanism is the most swappable part of the whole system; the analysis and the reasoning behind it are what matter.

Operations

A monitoring system that is installed and never tended decays into either noise you ignore or silence you trust wrongly. Operating it means two things: keep the alerts meaningful, and make sure the monitor itself is still working.

Tuning out noise

The failure mode that kills a monitoring system is not too few alerts, it is too many. A system that alerts on things that turn out not to matter trains you, surprisingly quickly, to ignore its alerts and an ignored, valid alert is just as bad as no alert. So the ongoing work is keeping the signal-to-noise ratio at a reasonable level.

In practice this means:

- **When an alert fires that did not need to, tune the check that produced it.** A threshold slightly too tight, a window slightly too short, a condition that does not distinguish a transient blip from a sustained problem, each produces false alarms. The health check's distinction between a momentary spike and a sustained condition is the model: most false positives come from not making that distinction somewhere.
- **Resist the urge to monitor more.** Every new check is new potential noise. Add checks deliberately, for categories that genuinely reveal trouble, not just for awareness. You still have the full logs when you just want to see how systems are behaving. A smaller set of trusted alerts beats a larger set you have trained yourself to skim past.
- **Review what has been firing.** Periodically look at what the system has alerted on and ask whether each alert was worth it. The ones that were consistently not worth it are candidates for tuning or removal.

The conservative posture throughout, watching high-value categories, requiring conditions to persist before alerting, keeping thresholds where real problems live rather than where noise does, is all in service of one goal. When an alert arrives, you have confidence in it enough to act.

Watching the watcher

Here is the failure that a monitoring system is uniquely prone to, and it is subtle: **a monitor that has silently stopped looks exactly like a quiet, healthy environment.** No alerts arrive in both cases. If your collector stops receiving logs, if a forwarding rule breaks, if the automation layer stops running the checks, the symptom is silence, and silence is also what "everything is fine"

looks like. You cannot tell the two apart by waiting for alerts, because the broken system will never send one.

So part of operating a monitoring system is monitoring the monitoring:

- **Watch for the absence of expected logs.** Some log streams are steady, the health snapshots arrive every few minutes from every host, for instance. A stream that normally flows steadily and then stops is itself a signal, and a check that notices "I expected snapshots from this host and got none" catches a broken pipeline that would otherwise be invisible.
- **Confirm the checks are actually running.** An automation platform shows run history, a cron can log its executions. Periodically confirm the scheduled checks ran when they should have, rather than assuming that no alert means no problem.
- **Test the path end to end occasionally.** Deliberately produce something that should alert, and confirm the alert arrives. This is the monitoring equivalent of testing a smoke detector. The only way to know it still works is to set it off on purpose now and then.

Adapt this for...

Any monitoring system. The operating disciplines are universal. Keep the noise down so the alerts stay trusted, watch for the silence that means the monitor stopped rather than that all is well, test the path deliberately, and treat the monitoring infrastructure as the sensitive, high-value thing it is. The install was the easy part, maintaining its relevancy is what makes it a system you can rely on a year later.

Lessons Learned

The lessons I've learned from building basic security monitoring without a SIEM product.

- **You don't need a SIEM product to have a SIEM.** The function, collect, centralize, analyze, alert, is achievable with the system logger, a few scripts, and a scheduler. The expensive product buys scale, correlation, and team workflow that a small environment does not need. Build the function from what you have, and spend nothing.
- **Monitor categories, not everything.** Watching every log line buries the signal that matters under the volume that does not. Choose the categories that reveal the most trouble; authentication, boundary events, health, critical severity, a few service-specific signals, and accept that you are deliberately not watching the rest. High-value coverage that stays relevant beats total coverage that becomes noise.
- **Noise is the enemy.** A monitoring system dies when its alerts stop being trusted, and they stop being trusted when too many of them do not matter. Tune aggressively toward fewer, more meaningful alerts. Requiring a condition to persist before firing, keeping thresholds where real problems live, resisting the urge to add more checks, all of it serves the goal that every alert should be worth reading.
- **Watch for silence, not just alarms.** This is the failure unique to monitoring. A monitor that has silently stopped looks identical to a healthy environment. Watch for the absence of logs you expect, confirm the checks are running, and test the alerting path deliberately now and then. Never treat "no alerts" as proof that all is well without confirming the monitor is still watching.
- **Start simple.** Cron and a mail command are a complete alerting layer, and adding no attack surface is a feature. Reach for a heavier automation platform only when you genuinely want its orchestration and are willing to secure it. The simplest thing that delivers the alert is usually the right thing.