LSS EU · PRAGUE · 8 OCTOBER 2026

Detection Engineering with RSigma

Portable detections on Linux, from kernel telemetry to one higher-confidence alert.

Mostafa Moradian

Sigma · Linux telemetry · Stateful correlation

About me

Mostafa Moradian

Creator and maintainer of RSigma

- 15+ years of experience
- Head of Security at Tiger Data
- OSS developer and maintainer (@mostafa)
- Stockholm, Sweden

Why I built RSigma

- Already built similar tooling at Grafana
- Portable Sigma execution without a SIEM
- Fast, inspectable tooling written in Rust
- Reproducible detection engineering workflows

01 · OPENING

One alert


```
{
  "title": "Linux Discovery-to-Execution Sequence",
  "level": "critical",
  "host": "lss-demo",
  "user": "1001",
  "signals": 4,
  "window": "60s"
}
```


No SIEM. No vendor query language. One local process.

01 · OPENING

Four ordinary signals

1

Discovery

Identify the current account and host context.

2

Download

Fetch a payload with a command-line tool.

3

Permission

Make the staged file executable.

4

Network

Open a loopback connection with a utility.

Individually noisy. Together, ordered by one host and user, much more useful.

TALK MAP

From one alert to a repeatable system

01 The signal
Four ordinary Linux events

02 The contract
Portable rules and ordered correlation

03 The proof
auditd and eBPF in one live path

04 The engine
Pruning, bounded state, and scale

05 The test
Replay, backtests, and coverage gates

06 The boundary
Least privilege and reproducibility

02

THE COMMON THREAD

How does a Linux host reach that result locally?

From native telemetry to one portable, correlated detection.

02 · THE PROBLEM

Linux already gives us the events

Kernel and host sources

- Linux audit subsystem
- eBPF sensors
- journald
- syslog

The missing layer

- Normalize source-specific fields
- Evaluate portable behavior
- Hold bounded temporal state
- Emit actionable results

> Collection is available. Preserving meaning across schemas and time is the missing layer.

Sigma is the portable contract

```yaml
logsource:
  product: linux
  category: process_creation
detection:
  selection:
    Image|endswith: /curl
  condition: selection
level: low
```

The rule describes behavior. A pipeline adapts its fields to the event schema.

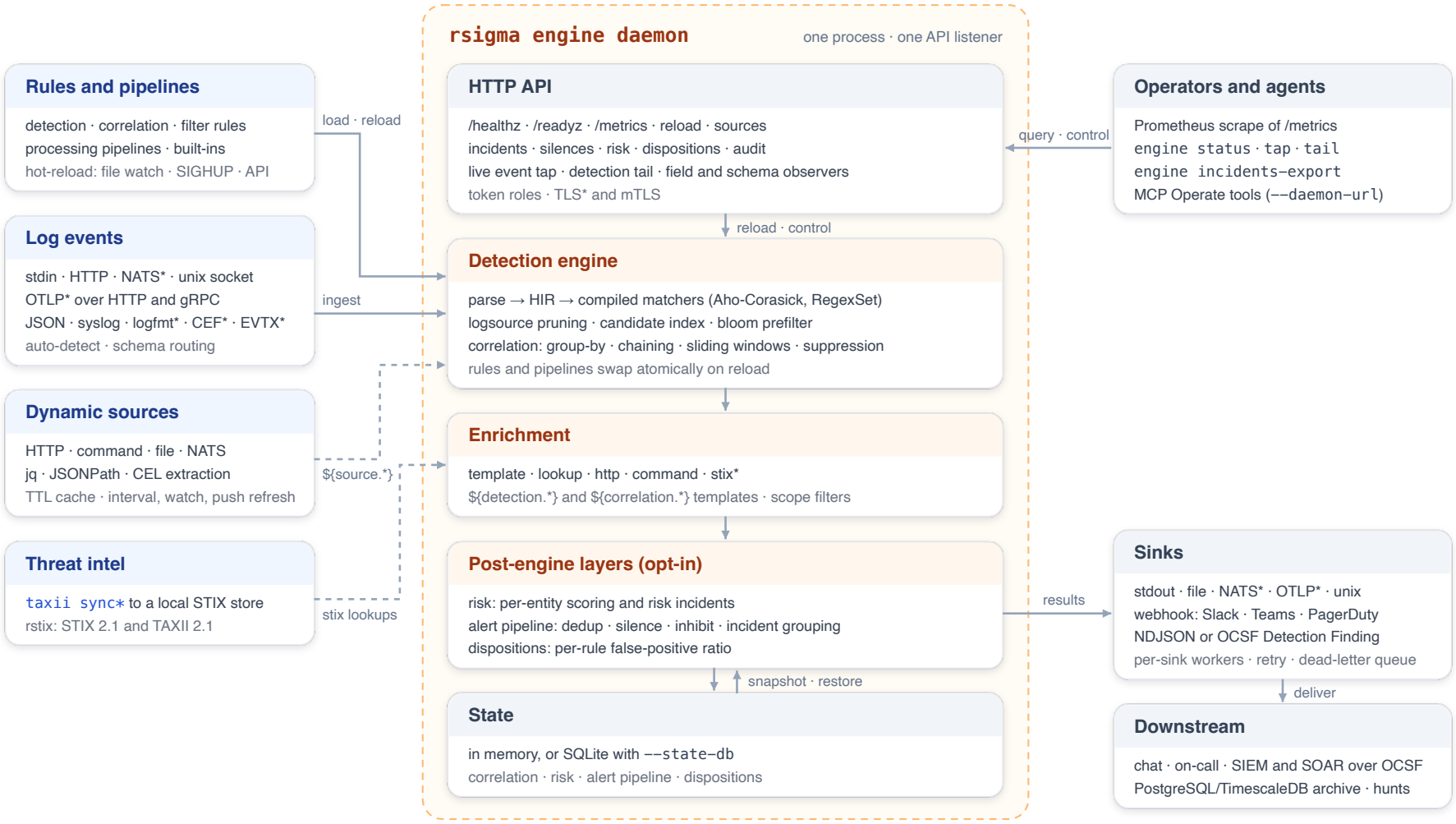
Order turns four detections into one result

```
rsigma.action: reset
correlation:
  type: temporal_ordered
  rules:
    - 9a0d8ca0-2385-4020-b6c6-cb6153ca56f3  # whoami
    - ea34fb97-e2c4-4afb-810f-785e4459b194  # curl
    - 74c01ace-0152-4094-8ae2-6fd776dd43e5  # chmod
    - 3e102cd9-a70d-4a7a-9508-403963092f31  # nc
  group-by: [Host, User]
  timespan: 60s
level: critical
```

Same host. Same user. Correct order. One bounded window.

RSigma

Streaming detection architecture



Same engine, outside the daemon

`engine eval` one-shot and batch evaluation · `backend convert` to SQL, SPL, KQL, Lucene, and more · `hunt run*` against the PostgreSQL archive
`rule lint` · `draft` · `tune` · `backtest` · `mcp serve*` Engineer tools for AI agents · `rsigma-lsp` diagnostics in editors

* opt-in Cargo feature, included in the prebuilt binaries and the Docker image

03

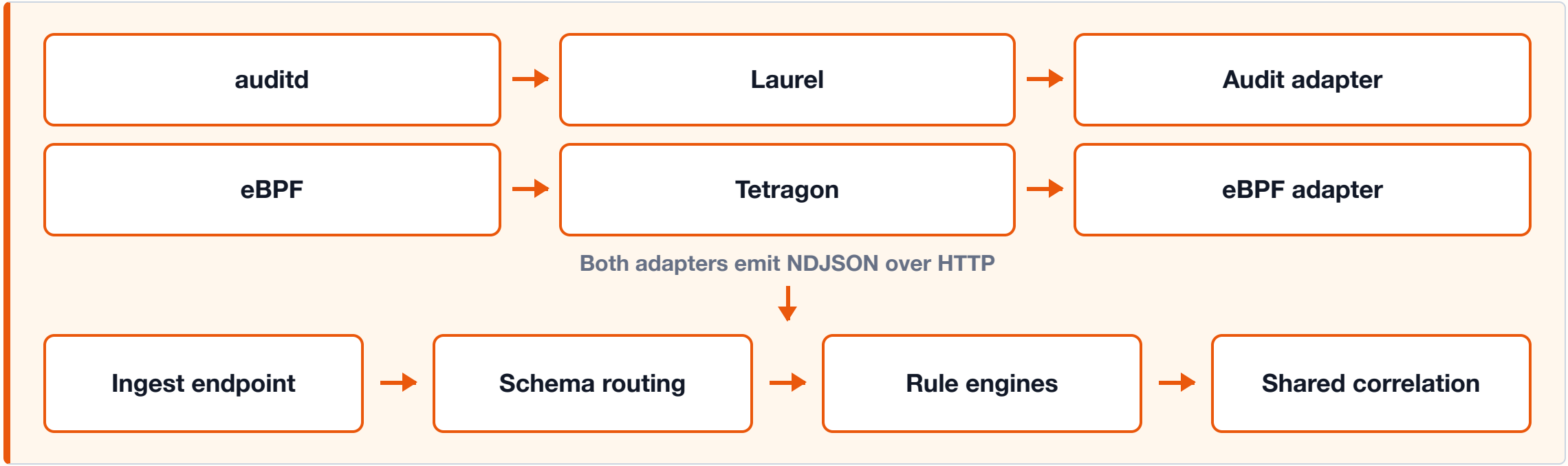
LIVE PROOF

Live: auditd + Tetragon + Sigma

Two native event shapes, four weak signals, one ordered alert.

03 · LIVE PATH

The sensors stay native



RSigma runs inside the same Ubuntu VM as the sensors.

Same behavior, different event shapes

auditd

```
{
  "source_type": "auditd",
  "Host": "rsigma-lss",
  "User": "1001",
  "type": "EXECVE",
  "a0": "whoami"
}
```

Tetragon

```
{
  "source_type": "tetragon",
  "Host": "rsigma-lss",
  "User": "1001",
  "process_exec": {
    "process": {
      "binary": "/usr/bin/curl"
    }
  }
}
```

Pipelines map portable rule fields onto each source without flattening away the native event.

03 · TRIGGER

Trigger the isolated sequence

1

whoami

Account discovery through auditd.

2

curl

Loopback download through Tetragon.

3

chmod +x

Permission change through auditd.

4

nc

Loopback connection through Tetragon.

`./demo/attack/chain.sh` · No venue network · UID 1001 · Fixed local payload

03 · RESULT

Four detections, one correlation

```
LOW       System Owner or User Discovery - Linux
LOW       Curl Usage on Linux
LOW       File or Folder Permissions Change
LOW       Linux Network Service Scanning Tools Execution
CRITICAL  Linux Discovery-to-Execution Sequence
```

Ordered within 60 seconds, grouped by `Host` and `User`, reset after firing.

03 · CLAIM BOUNDARY

What this proof establishes

It demonstrates

- Portable rules across two event shapes
- Ordered, entity-scoped correlation
- Bounded local execution
- A reproducible five-result contract

It does not establish

- Sensor completeness
- General cross-sensor deduplication
- Late-event reconciliation
- Production precision from one fixture

More context makes this alert **higher-confidence**, not infallible.

RSigma

Detection engineering loop

1 Author

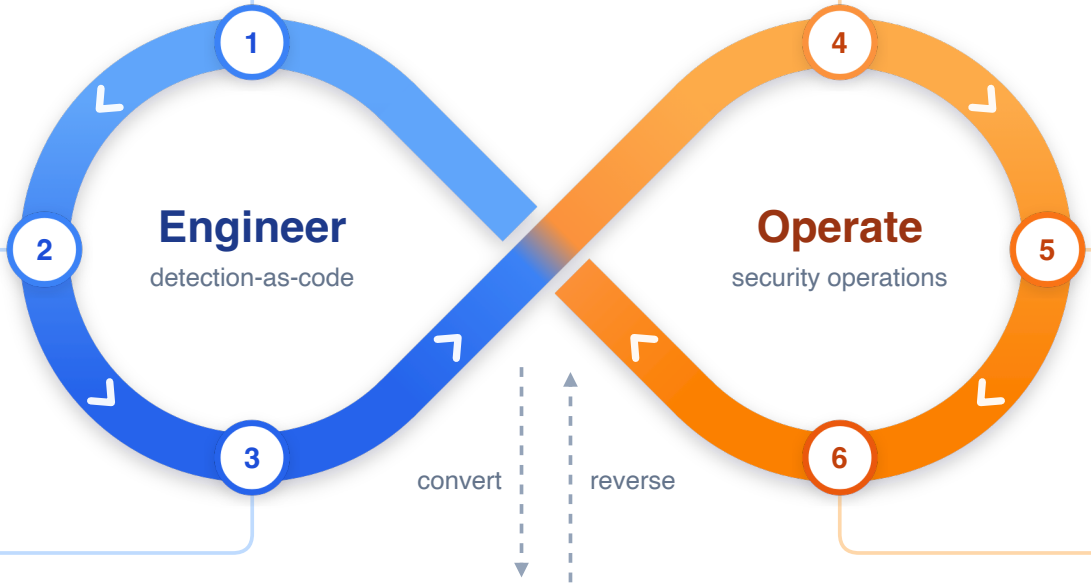
- `rule draft` rules from exemplar logs
- `rule tune` FP/TP → verified filter
- `rule reverse` Lucene query → Sigma
- lint · doc · LSP · MCP for AI agents

2 Test

- `engine explain` why (no) match
- `rule backtest` corpus regression
- `pipeline diff` transform preview
- `engine eval` one-shot evaluation

3 Deploy

- `rsigma-action` PR gate in CI
- `rsigma config` layered defaults
- **hot reload** rules + pipelines
- Docker · signed release binaries



4 Detect

- `engine daemon` streaming · RBAC API
- **correlation** windows · chains
- JSON · syslog · CEF · EVTX · OTLP
- schema routing · discovery · cloud

5 Alert & Triage

- **enrichment** GeoIP · intel · asset
- **risk-based alerting** entity score
- dedup · silence · inhibit · group
- webhooks: Slack · PagerDuty · ...

6 Measure

- `rule coverage` ATT&CK Navigator
- `rule visibility` DeTT&CT export
- **field observability** gap signals
- scorecard · FP ratio · dispositions

Hunt

- `backend convert` PostgreSQL SQL
- `sigma-cli` Splunk · 30+ backends
- `rule fields` mapping catalog
- *findings feed new rules*

04

INSIDE THE ENGINE

Why does this stay small and predictable?

Parse and compile at load time, prune per event, and bound temporal state.

Compile once, prune on every event

```rust
pub(crate) fn candidates(&self, event: &impl Event) -> Vec<usize> {
    let mut set = CandidateSet::new(self.rule_count);
    set.extend(&self.always);
    set.extend(&self.pending);
    self.collect_field_hits(event, &mut set);
    self.collect_keyword_hits(event, &mut set);
    set.finish()
}
```

crates/rsigma-eval/src/candidate_index.rs

Load time: parse, validate, and compile matchers. **Event time:** inspect only plausible candidates.

04 · CORRELATE

Keep bounded temporal state

```rust
/// Maximum state entries across all correlations and groups.
/// Default: 100_000.
pub max_state_entries: usize,

/// Maximum retained entries within a single group's window.
/// When exceeded, the oldest entries are dropped.
pub max_group_entries: Option<usize>,
```

crates/rsigma-eval/src/correlation_engine/types.rs

Group by entity, evict by time and capacity, and reset after the alert.

04 · MEASURE

Will it work at realistic scale?

3,132
SigmaHQ rules
Pinned corpus, identical matches.

0.29-3.86%
p95 candidates
Rules inspected per event shape.

87.7K-401K/s
HTTP daemon
Four deterministic event lanes.

~113 MB
Peak RSS
Full corpus load and evaluation.

State is the workload: 1M unique keys peak at 39.8 MiB with the default 100K cap, or 327.4 MiB when all 1M groups stay live

This demo: 21.5 MiB RSS after correlation · tuned detection-only raw-Windows path: ~708K/s on 8 threads

05

DETECTION AS CODE

Turn the incident into a test

Captured evidence becomes the regression suite and the CI gate.

05 · TEST

Lint, validate, backtest

```
rsigma rule lint demo/rules/
rsigma rule validate demo/rules/ -p demo/pipelines/
rsigma rule backtest \
  -r demo/rules/ \
  --corpus demo/fixtures/ \
  --expectations demo/expectations.yml
```

Known-bad must fire. Known-good must remain quiet.

05 · COVERAGE

Measure coverage, then gate the change

```
rsigma rule coverage \
  -r demo/rules/ \
  --navigator evidence/coverage.json \
  --targets demo/threat-model.txt \
  --fail-on-gaps
```

The CI decision combines syntax, semantics, behavior, and expected coverage.

Least privilege applies to security tools too

```
docker run --rm \
  --network=none \
  --read-only \
  --cap-drop=ALL \
  --security-opt=no-new-privileges:true \
  --pids-limit=64 \
  -v "${PWD}/demo/rules:/rules:ro" \
  ghcr.io/timescale/rsigma:0.23.0 \
  rule validate /rules
```

The sensor needs kernel access. The detection engine does not.

05 · REPRODUCIBILITY

Reproducible inputs and outputs

Release assets

Pinned versions and checksums.

Rules

Pinned SigmaHQ commit.

Container

Verified Sigstore signature.

Provenance

SBOM and SLSA attestation.

Fixtures

Captured malicious and benign events.

Replay

One deterministic command.

Reproducibility includes captured inputs and expected outputs, not only build metadata.

06 · CONTRIBUTING

Build in the open

- Array matching and rule-version proposals to SigmaHQ
- External pipeline sources adopted by pySigma
- A withdrawn time-window proposal after maintainer review
- Reproducible rules, fixtures, VM, evidence, and slides

Build against the standard, then contribute what you learn.

`github.com/timescale/rsigma`

06 · TAKEAWAYS

Five lessons

1

Normalize

Adapt telemetry at the boundary.

2

Compile

Move work out of the hot path.

3

Bound

Cap state by entity, time, and capacity.

4

Test

Keep malicious and benign fixtures.

5

Separate

Collection, detection, storage, and response.

Four ordinary Linux events become one explainable, reproducible alert without giving the detector kernel privilege.



THANK YOU

Questions?

A detection engine you can read, run, test, and try to evade.

GitHub
@mostafa

LinkedIn
/in/mostafa-moradian

X
@MosiMoradian

Blog
mostafa.dev

`rsigma.io` · `github.com/timescale/rsigma`