LOGZILLA DOCUMENTATION

Rule Test Fixtures

Format of the .tests.yaml fixtures that logzilla rules test and rules add run against rewrite and Lua rules: cases, input events, expectations, drops

Data Transforms · Generated September 11, 2026 · logzilla.ai/docs/data-transforms/rule-test-fixtures

Rule test fixtures

A rule test fixture is a YAML file that sits next to a rule file and lists input events together with the result each one must produce. `logzilla rules test --path <rule>` runs those tests against the rule. `logzilla rules add` runs the tests of the fixture named after the rule before installing it and refuses the rule when a case fails; a Lua rule cannot be added without one. Rewrite rules and Lua rules use the same fixture format.

File name and location

The fixture is found by the rule's name: the file stem for `logzilla rules test --path`, or the value given to `--name` for `logzilla rules add`. It lives in the same directory as the rule file:

```
noise-reduction.yaml
noise-reduction.tests.yaml
500-webhook-parser.lua
500-webhook-parser.tests.yaml
```

A rule may have more than one fixture file. Every file named as the rule name, a dot, an optional label, and `tests.yaml` is picked up, for example `500-webhook-parser.security.tests.yaml`.

When `logzilla rules add` is given `--name`, that name replaces the file stem, so the fixture must carry it: `rules add noise-reduction.yaml --name "Noise Reduction"` looks for `Noise Reduction.tests.yaml`. Either name the fixture after the `--name` value or omit `--name` and let the stem be the rule name.

Structure

The fixture below tests the rewrite rule from the Rewrite Rule Walkthrough (https://www.logzilla.ai/docs/data-transforms/rewrite-rule-walkthrough), which drops keepalive noise and tags `netd` events:

```
TEST_CASES:
  - name: keepalive noise is dropped
    event:
      program: netd
      message: "keepalive: OK"
    expect:
      type: DROPPED
  - name: interface event keeps its message and gets the site tags
    event:
      program: netd
      message: interface ge-0/0/1 up
    expect:
      message: !SAME
```

```
      user_tags:
        site: west-dc
        device_role: edge
  - name: other programs are left alone
    event:
      program: sshd
      message: interface ge-0/0/1 up
    expect:
      user_tags: {}
```

`TEST_CASES` is a list. Each case has an optional `name`, shown in the test output (unnamed cases are numbered), an `event` block, and an `expect` block.

The `event` block

The fields of the input event. Any field not listed keeps its default: the message `Some random message`; empty `host`, `program`, and `cisco_mnemonic`; `severity` and `facility` of `0`; no user tags; no extra fields. Common keys:

- `message`, `host`, `program`, `cisco_mnemonic`
- `severity` (0 to 7) and `facility` (0 to 23)
- `user_tags`: a mapping of tags already present on the event
- `extra_fields`: a mapping of extra fields, for example the `_source_type` stamp that a dedicated application port adds

The `expect` block

Each key names a field and the value the rule must leave in it. Only the listed keys are checked; anything else may change freely.

- `message`, `host`, `program`, `severity`, `facility`: compared for equality.
- `user_tags` or `extra_fields` given as a mapping: the whole mapping must match. `user_tags: {}` asserts that the event ends with no tags (an input that already carried tags fails it too), and a tag the rule set but the fixture did not list fails the case as an extra key. (The `Compliance Framework` tag is exempt from the extra-key check.)
- A single tag or field with a dotted key, such as `user_tags.SrcIP: 192.0.2.10` or `extra_fields._source_type: unifi`, checks that one entry and ignores the rest.
- `!SAME`: the field must equal its value in the input event.
- `type: DROPPED`: the rule dropped the event, either a Lua rule returning `Result.DROP` or a rewrite rule with `drop: true` whose match succeeded. Events that were not dropped have `type: UNKNOWN`.

A failing case prints the fixture location, the event before and after the rule ran, and the mismatch.

Dedicated application ports

Lua rules that declare `SOURCE_FILTER` receive, in production, only events stamped with that source type when the matching application port is enabled. Fixtures skip that engine filter by default and feed every case to the rule. To simulate the filter, set `DEDICATED_PORT: true` at the top of the file, or `dedicated_port: true` on a single case. Under the simulation only events whose `extra_fields._source_type` equals the rule's `SOURCE_FILTER` reach the rule; every other event passes through untouched.

```
DEDICATED_PORT: true
TEST_CASES:
  - name: stamped event is parsed
    event:
      message: "<vendor log line>"
      extra_fields:
        _source_type: vendorname
    expect:
      user_tags:
        Vendor: Vendor
        Product: Product
  - name: unstamped event never reaches the rule
    event:
      message: "<vendor log line>"
    expect:
      user_tags: {}
  - name: this case runs without the engine filter
    dedicated_port: false
    event:
      message: "<vendor log line>"
    expect:
      user_tags:
        Vendor: Vendor
        Product: Product
```

Using the simulation on a rule without `SOURCE_FILTER` is reported as an error.

Rule configuration files

Lua rules that read configuration files can be tested with different configurations. A `DEFAULT_CONFIGS` mapping at the top of the file applies to every case; a `configs` mapping on a case overrides it for that case. Each key becomes a file named `<key>.yaml` in the rule's configuration directory for the duration of the case, with the mapping as its content. The example below assumes a rule that reads `lookup_field` from `config.yaml` and sets the `Site` tag from a lookup on that field.

```
DEFAULT_CONFIGS:
  config:
    lookup_field: host
TEST_CASES:
  - name: default configuration
```

```
    event:
      host: host1
      message: "text"
    expect:
      user_tags.Site: dc1
  - name: alternative configuration
    configs:
      config:
        lookup_field: program
    event:
      program: host1
      message: "text"
    expect:
      user_tags.Site: dc1
```

Running the tests

```
# A local rule file with its fixture next to it
logzilla rules test --path /path/to/noise-reduction.yaml

# The tests of installed rules, by name filter or all
logzilla rules test "Noise Reduction"
logzilla rules test --all

# Stop at the first failing case
logzilla rules test --path /path/to/noise-reduction.yaml --exitfirst
```

`logzilla rules add` runs the tests of the fixture named after the rule before installing it. For a Lua rule the fixture file is also copied next to the installed rule, so `logzilla rules test "My Lua Rule"` keeps working after installation. For a rewrite rule the fixture is not installed: keep it with the rule source and run its tests with `logzilla rules test --path`.

Related reading

- Rewrite rules (https://www.logzilla.ai/docs/data-transforms/rewrite-rules)
- Lua rules (https://www.logzilla.ai/docs/data-transforms/lua-rules)
- Testing and verification (https://www.logzilla.ai/docs/data-transforms/testing-and-verification)
- Command Line Tools —- Data Commands (https://www.logzilla.ai/docs/command-line-tools/data-commands)