EVERTRUST

Event Forwarder

Version 1.0, 2026-07-02

Table of Contents

Chapter 1. Introduction

Event Forwarder is a small CLI that fetches events from a Service events API, such as Horizon or Stream, and emits each event to stdout as newline-delimited JSON (NDJSON). Logs and diagnostics are written to stderr, so stdout can be consumed safely by log collectors.

Event Forwarder data flow

```
flowchart LR
    service["Horizon or Stream<br/>events API"] -->|"paginated search requests"|
forwarder["event-forwarder"]
    forwarder -->|"one JSON event per line"| stdout[("stdout")]
    forwarder -->|"logs and diagnostics"| stderr[("stderr")]
    forwarder -->|"last timestamp + event IDs"| state[("state file")]
    stdout --> collector["existing log collector<br/>Vector, Alloy, Fluent Bit, etc."]
    collector --> siem["SIEM"]
    collector --> datalake["datalake"]
```

The intended use case is to let customers reuse their existing log scraping stack. Event Forwarder does not try to become a complete log pipeline. It only handles the Service-specific work:

- authenticating to the events API;
- paging through event search results;
- retrying transient HTTP 429 and 5xx responses;
- preserving each original event payload;
- writing exactly one compact JSON document per stdout line;
- persisting a cursor when a state file is configured.

This design makes the integration portable. Any collector that can execute a command, read a container log stream, or tail a file can ingest Event Forwarder output and then forward it to the customer's SIEM, datalake, message bus, or log analytics platform.

State persistence is optional but recommended for scheduled or long-running collector integrations. When `SERVICE_STATE_FILE` or `--state-file` is set, the forwarder stores the newest emitted timestamp and the event IDs seen at that same timestamp. The file is updated after each event is successfully written to stdout, allowing the next run to resume without re-emitting events that were already forwarded.

Chapter 2. Installation

Event Forwarder is released as a container image, RPM package, and standalone Linux binary for `amd64` and `arm64`.

Docker image

Release images are published to:

```
registry.evertrust.io/event-forwarder
```

Run the image directly and mount a persistent volume for cursor state:

```
docker pull registry.evertrust.io/event-forwarder:latest

docker run --rm \
  -e SERVICE_ENDPOINT="https://service.example.com/api/v1/events/search" \
  -e SERVICE_API_KEY="$SERVICE_API_KEY" \
  -e SERVICE_STATE_FILE="/var/lib/event-forwarder/state.json" \
  -v event-forwarder-state:/var/lib/event-forwarder \
  registry.evertrust.io/event-forwarder:latest \
  --daemon \
  --poll-interval 10s
```

This mode is useful when your existing logging stack already scrapes Docker or Kubernetes container stdout.

RPM package

Installing from the Evertrust repository

Create a `/etc/yum.repos.d/event-forwarder.repo` file containing the EverTrust repository info:

```
[event-forwarder]
enabled=1
name=Event Forwarder Repository
baseurl=https://repo.evertrust.io/repository/event-forwarder-rpm/
gpgcheck=1
gpgkey=https://evertrust.io/.well-known/rpm/gpg.pub
username=<username>
password=<password>
```

Replace `<username>` and `<password>` with the credentials you were provided.

Make sure the Evertrust GPG key is trusted:

```
# rpm --import https://evertrust.io/.well-known/rpm/gpg.pub
```

You can then run the following to install the latest Event Forwarder version:

```
# yum install event-forwarder
```

To prevent unattended upgrades when running `yum update`, pin the Event Forwarder version by adding the following line at the end of `/etc/yum.repos.d/event-forwarder.repo` after installing Event Forwarder:

```
exclude=event-forwarder
```

Installing from the package file

Download the latest RPM for Event Forwarder from the Official EVERTRUST repository.

Upload the file `event-forwarder-<latest>.<arch>.rpm` to the server. Replace `<arch>` with the target RPM architecture, such as `x86_64` or `aarch64`.

Access the server with an account with administrative privileges.

Install the Event Forwarder package with the following command:

```
# yum localinstall /root/event-forwarder-<latest>.<arch>.rpm
```

If you wish to verify the signature of the RPM package, the EVERTRUST key can be added to your trusted keys using the following command:

```
# rpm --import https://evertrust.io/.well-known/rpm/gpg.pub
```

The signature can then be verified using the following command:

```
# rpm -K /root/event-forwarder-<latest>.<arch>.rpm
```

The RPM installs the binary as:

```
/usr/bin/event-forwarder
```

Standalone binary

Download the raw Linux binary for your architecture from GitHub releases:

```
event-forwarder_<version>_linux_amd64
```

```
event-forwarder_<version>_linux_arm64
```

Install it in a directory on `PATH`:

```
VERSION="v0.0.0"
chmod +x "./event-forwarder_${VERSION}_linux_amd64"
sudo install -m 0755 "./event-forwarder_${VERSION}_linux_amd64"
/usr/local/bin/event-forwarder
```

Check the installed version:

```
event-forwarder --version
```

Chapter 3. Configuration

Event Forwarder can be configured with CLI flags or environment variables. CLI flags are convenient for direct testing; environment variables are usually easier for collectors, containers, and system services.

CLI flags

```
event-forwarder \
  --endpoint "https://service.example.com/api/v1/events/search" \
  --api-key "$SERVICE_API_KEY" \
  --state-file "/var/lib/event-forwarder/state.json"
```

Environment variables

```
export SERVICE_ENDPOINT="https://service.example.com/api/v1/events/search"
export SERVICE_API_KEY="replace-me"
export SERVICE_STATE_FILE="/var/lib/event-forwarder/state.json"

event-forwarder
```

Available options:

CLI flag	Environment variable	Default	Description
`--endpoint`	`SERVICE_ENDPOINT`	Required	Service events search endpoint.
`--api-key`	`SERVICE_API_KEY`	Required	Service API key sent as the `X-API-KEY` header.
`--api-id`	`SERVICE_API_ID`	`admin`	Service API ID sent as the `X-API-ID` header.
`--user-agent`	`SERVICE_USER_AGENT`	`vector-event-forwarder/1.0`	HTTP `User-Agent` header.
`--page-size`	`SERVICE_PAGE_SIZE`	`50`	Number of events requested per page.
`--start-page`	`SERVICE_START_PAGE`	`1`	First API page index requested.
`--max-pages`	`SERVICE_MAX_PAGES`	`0`	Maximum pages to fetch in one run. `0` means unlimited.
`--timeout`	`SERVICE_TIMEOUT`	`30s`	HTTP request timeout.
`--state-file`	`SERVICE_STATE_FILE`	Empty	Cursor state path. Leave empty to disable durable state.
`--start-timestamp`	`SERVICE_START_TIMESTAMP`	Empty	Initial lower bound as RFC3339/RFC3339Nano or Unix milliseconds.

CLI flag	Environment variable	Default	Description
`--retries`	`SERVICE_RETRIES`	`2`	Retry count for HTTP 429 and 5xx responses.
`--retry-backoff`	`SERVICE_RETRY_BACKOFF`	`1s`	Base retry backoff. Retry attempt `n` waits `retry-backoff * n`.
`--tls-insecure-skip-verify`	`SERVICE_TLS_INSECURE_SKIP_VERIFY`	`false`	Disable TLS certificate verification. Use only in controlled test environments.
`--log-level`	`SERVICE_LOG_LEVEL`	`info`	`trace`, `debug`, `info`, `warn`, `error`, or `disabled`.
`--trace-file`, `--trace`	`SERVICE_TRACE_FILE`	Empty	Write OpenTelemetry diagnostic traces to the given OTLP JSON Lines file. Event bodies are included.
`--daemon`	`SERVICE_DAEMON`	`false`	Keep the process alive and poll periodically for new events. Requires a state file.
`--poll-interval`	`SERVICE_POLL_INTERVAL`	`10s`	Delay between daemon-mode fetch runs.
`--version`, `-v`	-	-	Print binary and Go versions.

Show the generated CLI help:

```
event-forwarder --help
```

State file

When state persistence is enabled, the state file contains:

```
{
  "last_timestamp_ms": 1782209730123,
  "event_ids_at_last_timestamp": ["event-a", "event-b"]
}
```

Use a persistent directory for this file. For containers, mount a volume. For collectors running as system services, store it under a writable data directory such as `/var/lib/event-forwarder/state.json`.

If no state file is configured, each run starts from `--start-timestamp` when it is set, otherwise from `--start-page`.

Daemon mode

Use `--daemon` when the forwarder should stay alive and poll for new events itself instead of being restarted by a scheduler or collector:

```
event-forwarder \
  --endpoint "https://service.example.com/api/v1/events/search" \
  --api-key "$SERVICE_API_KEY" \
  --state-file "/var/lib/event-forwarder/state.json" \
  --daemon \
  --poll-interval 10s
```

Daemon mode runs one fetch immediately, then waits for `--poll-interval` before the next fetch. It keeps running until the process receives SIGINT/SIGTERM or the parent process stops it. Before each wait, the daemon logs the next scheduled poll time to stderr.

A state file is required in daemon mode. Without a cursor, the process would have no durable record of the last emitted event and could re-emit old events on each poll.

Chapter 4. Debug

Use diagnostic traces when EVERTRUST Support needs a detailed local execution trace to investigate an issue.

Diagnostic traces

Set `--trace-file` or `SERVICE_TRACE_FILE` to write a trace file:

```
event-forwarder \
  --endpoint "https://service.example.com/api/v1/events/search" \
  --api-key "$SERVICE_API_KEY" \
  --state-file "/var/lib/event-forwarder/state.json" \
  --trace-file "./event-forwarder-trace.jsonl"
```

The generated file uses the OpenTelemetry Protocol JSON file serialization format: one trace export request per line with top-level `resourceSpans`. It includes spans for CLI execution, daemon polls, state loading and saving, page fetch attempts, HTTP requests, and event processing.

Attach this trace file to the related support ticket when requested. It gives EVERTRUST Support enough local execution detail to provide a more accurate answer and debug the issue from our side.

Each processed event span includes the raw event JSON in `event.body`, so handle this file like customer log data.

Chapter 5. Integration

Event Forwarder works best as a source adapter in front of an existing collector. The collector should treat stdout as newline-delimited JSON and should ignore or separately route stderr diagnostics.

Vector from Datadog

Vector has an `exec` source that can run Event Forwarder on a schedule and parse stdout as JSON lines.

Use Vector's scheduled `exec` mode for one-shot forwarder runs. Use Event Forwarder's daemon mode when Vector, Kubernetes, Docker, or another collector is scraping a long-running process stdout stream.

Minimal Vector example:

```
data_dir: "${VECTOR_DATA_DIR:-/var/lib/vector}"

sources:
  service_events_api:
    type: exec
    command:
      - /usr/local/bin/event-forwarder
    mode: scheduled
    include_stderr: false
    environment:
      SERVICE_ENDPOINT: "${SERVICE_ENDPOINT:?SERVICE_ENDPOINT must be set}"
      SERVICE_API_KEY: "${SERVICE_API_KEY:?SERVICE_API_KEY must be set}"
      SERVICE_API_ID: "${SERVICE_API_ID:-admin}"
      SERVICE_STATE_FILE: "${SERVICE_STATE_FILE:-/var/lib/vector/event-
forwarder-state.json}"
      SERVICE_PAGE_SIZE: "${SERVICE_PAGE_SIZE:-50}"
      SERVICE_TIMEOUT: "${SERVICE_TIMEOUT:-30s}"
      SERVICE_LOG_LEVEL: "${SERVICE_LOG_LEVEL:-info}"
    framing:
      method: newline_delimited
    decoding:
      codec: json
    scheduled:
      exec_interval_secs: ${SERVICE_SCRAPE_INTERVAL_SECS:-10}

transforms:
  service_normalize:
    type: remap
    inputs:
      - service_events_api
    drop_on_error: true
    source: |
      .event_id = del(._id) ?? .event_id
```

```
      .timestamp_ms = to_int(.timestamp_ms) ?? to_int!(.timestamp)
      .timestamp = from_unix_timestamp!(.timestamp_ms, unit: "milliseconds")
      .source = "service"

sinks:
  console:
    type: console
    inputs:
      - service_normalize
    encoding:
      codec: json
```

Replace the `console` sink with your existing Vector sink, such as Datadog Logs, Splunk HEC, Kafka, S3, HTTP, or another destination.

Grafana Alloy

Grafana Alloy is usually a good fit when the forwarder runs as a container or Kubernetes workload and Alloy scrapes that workload's stdout. In this pattern, Event Forwarder keeps its state on a mounted volume, while Alloy reads the container log stream and forwards events to Loki or another configured pipeline.

Example Docker run:

```
docker run -d \
  --name event-forwarder \
  --label logs.service=event-forwarder \
  -e SERVICE_ENDPOINT="https://service.example.com/api/v1/events/search" \
  -e SERVICE_API_KEY="$SERVICE_API_KEY" \
  -e SERVICE_STATE_FILE="/var/lib/event-forwarder/state.json" \
  -v event-forwarder-state:/var/lib/event-forwarder \
  registry.evertrust.io/event-forwarder:latest \
  --daemon \
  --poll-interval 10s
```

Example Alloy pipeline that discovers the Docker container logs and parses the forwarded JSON:

```
discovery.docker "containers" {
  host = "unix:///var/run/docker.sock"
}

discovery.relabel "event_forwarder" {
  targets = discovery.docker.containers.targets

  rule {
    source_labels = ["__meta_docker_container_label_logs_service"]
    regex         = "event-forwarder"
    action        = "keep"
```

```
  }
}

loki.source.docker "event_forwarder" {
  host    = "unix:///var/run/docker.sock"
  targets = discovery.relabel.event_forwarder.output
  forward_to = [loki.process.service_events.receiver]
}

loki.process "service_events" {
  stage.json {
    expressions = {
      event_id = "_id",
      timestamp_ms = "timestamp",
    }
  }

  stage.labels {
    values = {
      event_id = "",
    }
  }

  forward_to = [loki.write.default.receiver]
}

loki.write "default" {
  endpoint {
    url = "https://loki.example.com/loki/api/v1/push"
  }
}
```

The same pattern applies in Kubernetes: run Event Forwarder as a deployment or sidecar, mount a persistent volume for `SERVICE_STATE_FILE`, and configure Alloy to collect that pod's stdout.

Fluent Bit

Fluent Bit can either run Event Forwarder with the `exec` input plugin or read the forwarder's container logs through its normal container log inputs. The `exec` input is useful for lightweight host-based deployments.

Example `fluent-bit.conf`:

```
[SERVICE]
    Flush        5
    Log_Level    info
    Parsers_File parsers.conf

[INPUT]
```

```
    Name      exec
    Tag       service.events
    Command   /usr/local/bin/event-forwarder
    Interval_Sec 10
    Parser    json

[FILTER]
    Name    modify
    Match   service.events
    Add     source service

[OUTPUT]
    Name    stdout
    Match   service.events
    Format  json_lines
```

Example `parsers.conf`:

```
[PARSER]
    Name   json
    Format json
```

Set the Event Forwarder configuration in the Fluent Bit service environment:

```
export SERVICE_ENDPOINT="https://service.example.com/api/v1/events/search"
export SERVICE_API_KEY="replace-me"
export SERVICE_STATE_FILE="/var/lib/fluent-bit/event-forwarder-state.json"
```

Then replace the `stdout` output with your production output, such as Splunk, HTTP, Kafka, S3, OpenSearch, or another Fluent Bit destination.

Generic collector pattern

For any other logging stack, use one of these patterns:

- Run `event-forwarder` directly as a scheduled command and decode stdout as newline-delimited JSON.
- Run `event-forwarder --daemon` as a long-running process and scrape stdout.
- Run `event-forwarder` as a container and scrape the container stdout stream.
- Run `event-forwarder` from a systemd timer or cron job and pipe stdout into a collector-supported input.

Keep these rules consistent across integrations:

- preserve stdout for event JSON only;
- keep stderr separate for logs and diagnostics;
- configure a persistent `SERVICE_STATE_FILE`;

- protect `SERVICE_API_KEY` with the collector or orchestrator's secret mechanism;
- use collector-side transforms only for routing, enrichment, normalization, and destination-specific schema requirements.

Official collector references:

- Vector `exec` source
- Grafana Alloy Docker log source
- Grafana Alloy processing stages
- Fluent Bit `exec` input

Chapter 6. Release notes