

3 keywords: [Security, Privacy, Data Processing, Data Storage]

Data Processing & Storage FAQs

Is my data accessible by anyone outside my organization?

Your data belongs ONLY to you, and is ONLY accessible by members of your organization.

Employee's of Levo DO NOT have access to your data.

What kind of customer data does Levo process and store?

There are two distinct workflows where customer data is processed and stored.

Levo does not process or store PII or PSI data of any kind. Any exceptions will be specifically called out in the workflow descriptions below.

1. API Catalog & Schema

Levo's API Catalog can be populated either via auto discovery or via manual import of OpenAPI Schemas.

In either case no actual customer data, including PII/PSI data is ingested or stored. Only API metadata is ingested and stored.

The below diagram describes the two methods of populating the API Catalog.

Manually Imported APIs

In this case OpenAPI specifications are imported by the customer and stored in Levo's API Catalog.

This data is ONLY accessible by members of your organization.

As a best practice, please ensure that your OpenAPI specifications do not contain secrets, tokens, or PSI/PII data.

Auto Discovered APIs

Levo can auto discover all your APIs and their schema in your live environments (using an eBPF Sensor).

Even though the eBPF Sensor captures real API traffic (that could potentially have PII/PSI data), the Satellite anonymises all the API traces, and only sends API metadata to Levo SaaS.

The below diagram shows how PII/PSI data from the raw API trace is anonymized and sent to Levo SaaS as API metadata.

2. API Security Testing

This workflow has 4 distinct phases.

The Test Runner (CLI) pulls Test Plans from Levo SaaS. Test Plans have NO real customer data, and only have information on API endpoints (metadata specified in the API catalog).

Any user credentials or tokens required to test API endpoints stays within the customer premises, and are supplied locally to the CLI. This information is NEVER transmitted to Levo SaaS.

The CLI executes the test plan against your API endpoints (similar to integration tests), and sends the results to Levo SaaS.

Any authentication/authorization tokens are stripped from the test results before transmission to Levo SaaS. Test results may contain the detailed HTTP request/response of the API endpoint being tested.

Even though test results are ONLY accessible by members of your organization, it is good practice to ensure, the API endpoints being tested DO NOT return real PII/PSI data.

Test results are viewed from Levo SaaS. Test results do not contain any authentication/authorization tokens, and are ONLY accessible by members of your organization. Employee's of Levo, cannot access your test results or other data.

Does Levo store authentication tokens or secrets?

Levo does not ingest or store authentication credentials, tokens or other secrets. All of this remains within your premises. Please see section above for more details.

Is TLS used for all data transmissions?

Yes, TLS is used wherever there is data in motion.

Does Levo SaaS require inbound network connectivity to my datacenter/VPC?

NO. Levo's CLI runs within your datacenter/VPC, and makes outbound network connections to Levo SaaS.

FAQs

General (general.md)

Data Processing & Storage (data-processing-storage.md)

Satellite-Sensor FAQs (satellite-sensor-faqs.md)

Sample Applications (sample-apps.md)

General FAQs

Why do you need Levo?

Automated security testing of microservices, which uncovers sophisticated business logic (<https://www.hackerone.com/company-news/rise-idor>) and access control-based attacks, is a significant gap today. Continuous Security Assurance from Levo.ai provides fully automated and effortless (runtime) security testing for Microservices in CI/CD.

How is Levo different from other Application Security Testing tools?

Modern attacks target business logic flaws

(<https://www.hackerone.com/company-news/rise-idor>) that arise from sub-optimal authentication and authorization across API endpoints.

AST tools like SCA

(<https://www.synopsys.com/glossary/what-is-software-composition-analysis.html>) & SAST (<https://www.microfocus.com/en-us/what-is/sast>) statically analyze source code for security defects, but are unaware of authentication & authorization flaws. DAST (<https://www.microfocus.com/en-us/what-is/dast>) tools focus on the runtime but lack adoption due to the significant manual heavy lifting required. Moreover, they are “business logic blind” (<https://engineeringblog.yelp.com/2020/01/automated-idor-discovery-through-stateful-swagger-fuzzing.html>) as they are unable to uncover sophisticated business logic and access control violation attacks. IAST (<https://snyk.io/learn/application-security/iastr-interactive-application-security-testing/>) tools require comprehensive unit test coverage written by developers, and are also “business logic blind”. Levo is the only purpose-built security solution for APIs & microservices that provides comprehensive detection of both business logic, and OWASP Top 10 vulnerabilities.

What CI/CD environments are supported?

Levo supports all popular CI/CD environments. Please refer to Integrations (</integrations/integrations.md>) for more information.

Sample Applications FAQs

crAPI

Why are several test cases in the onboarding test plan for crAPI disabled?

The objective of onboarding is to let new users experience Levo's security testing capabilities in a quick and stress free manner. To keep the test plan execution time for onboarding under 5 minutes, we disable some tests.

We ensure that there is at least one test case enabled, for each vulnerability type that we test for.

Satellite-Sensor FAQs

FAQs

Table of Contents

Getting Help (<satellite-sensor-faqs.md#getting-help>)

Sensor (<satellite-sensor-faqs.md#sensor>)

Satellite (<satellite-sensor-faqs.md#satellite>)

API Catalog (<satellite-sensor-faqs.md#api-catalog>)

Getting Help

Please email support@levo.ai for assistance with installation, product questions, roadmap, etc. Please provide as much details as possible in your support request.

Sensor

What OS platforms are supported?

Please see OS Platforms
(/guides/general/supported-platforms.mdwhat-os-platforms-are-supported).

What Kubernetes platforms are supported?
Please see K8s Platforms
(/guides/general/supported-platforms.mdwhat-kubernetes-platforms-are-supported).

Is Docker Desktop or minikube on Mac OSX, supported?
Support for Docker Desktop, Docker Desktop based Kubernetes, and minikube on MacOS is on the roadmap.
Developers can evaluate API Observability on their macOS Laptops, via a proxy based Sensor. Please refer to Quickstart for macOS/Windows (/quickstart/quickstart-mitm.md).

Is Windows OS supported?
Microsoft is currently building support for eBPF in Windows
(<https://github.com/microsoft/ebpf-for-windows>). Windows support will be added subsequent to the completion of that effort.
Developers can evaluate API Observability on their Windows Laptops, via a proxy based Sensor. Please refer to Quickstart for macOS/Windows (/quickstart/quickstartmitm.md).

Is there a script that can assess if my OS platform is compatible?
Yes. Please see install guide (/guides/install-guide).

How do I install the Sensor?
Please see install guide (/guides/install-guide).

What kind of API traffic is discovered?
Currently REST APIs only. Support for gRPC, and GraphQL are on the roadmap.

What is an API Trace?
An API Trace is the representation of an API invocation, generated from captured traffic. It contains the full HTTP request & response, including the request URL, request headers, request body, response code, response headers, and response body.

Can APIs running over TLS (HTTPs) be discovered?
Yes, for applications that use OpenSSL (<https://www.openssl.org/>). Support for Java TLS (<https://www.ateam-oracle.com/post/transport-level-security-tls-and-java>), and Boring SSL (<https://boringssl.googlesource.com/boringssl/>) are on the roadmap.

Are REST APIs over HTTP/2 (<https://en.wikipedia.org/wiki/HTTP/2>) supported?
Currently only REST APIs over HTTP/1.x are supported. Vast majority of REST APIs still use HTTP/1.x. Support for HTTP/2 is on the roadmap.

Does the sensor need special privileges?

Yes, like all other vendor solutions in the market that use eBPF. The sensor requires root privileges to capture all API traffic and associated metadata.

Will the sensor impact my application workloads?

The sensor is passive and not inline with your application. It uses eBPF probes (<https://epbf.io>) to make passive copies of API traffic (HTTP), similar to network traffic mirroring (<https://docs.aws.amazon.com/vpc/latest/mirroring/what-is-traffic-mirroring.html>). The Sensor will not impact your application's function or performance unlike other inline solutions (sidecar proxies, in-app agents, and SDKs).

What is the CPU impact of the sensor?

Less than 5%, as the sensor can sample API traffic.

Can API traffic be sampled?

Yes. API traffic can be sampled in high traffic environments to optimize CPU consumption of the Sensor. Unlike vendors building API security solutions that are anomaly based (where every single API call has to be captured), Levo can aggressively sample API traffic. Sampled API traffic is used to discover API endpoints and their underlying schema.

Can API traffic be filtered?

Yes. Please see [Filtering API Traffic \(/install-traffic-capture-sensors/common-tasks/filter-traffic.md\)](#).

Can I consume captured API Traces from the sensor?

Yes. The Sensor exports captured API Traces in industry standard OpenTelemetry (<https://opentelemetry.io/docs/concepts/what-is-opentelemetry/>) format. These traces can be visualized using tools like Jaeger (<https://www.jaegertracing.io/>), etc.

What is BPF Type Format Info (BTF)?

Levo's eBPF sensor uses probes (<https://ebpf.io/what-is-ebpf/hook-overview>) to copy API data from socket system calls. BTF (<https://www.kernel.org/doc/html/latest/bpf/btf.html>) provides syscall function symbol information, that is used by the Sensor to attach probes.

Do Linux distributions ship with pre-built BTF info?

Most modern Linux distributions contain pre-built BTF info. If BTF is missing for your specific version of Linux, Levo Support can build a Sensor that contains a custom BTF for your version of Linux.

Satellite

I don't care about data privacy in pre-production. Can Levo host the Satellite for me?

Yes, Levo certainly can host the Satellite for you. Please contact support@levo.ai for

assistance.

Can multiple Sensors send API Traces to the same Satellite?

Yes. Multiple sensors from different hosts/clusters can be configured to send API Traces to the same Satellite.

Can the Satellite be deployed in a different host/cluster?

Yes. Please see [Install Satellite \(/install-satellite\)](#).

How does the Satellite detect sensitive data in API Traffic?

The Satellite has a pre-trained ML model that is used to detect sensitive data such as PII, PSI, etc.

Can the Satellite be upgraded?

Levo provides updated Docker images for the Satellite. You can upgrade the Satellite at your own convenience, to take advantage of new features, and/or bug fixes.

What is the resource consumption of the Satellite?

It depends on your traffic patterns. This is usually not a concern, as the Satellite can be run on a dedicated host/cluster and will not contend with your production workloads.

API Catalog

What OpenAPI Specification version for the auto discovered APIs?

Version 3.0.x

This section describes tasks that are common when using Levo.

[OS-Compatibility-Check \(/guides/general/os-compat-check\)](#)

[Private Registry \(/guides/general/private-registry\)](#)

[Supported Platforms \(/guides/general/supported-platforms\)](#)

Use a Private Docker Registry for

Kubernetes Installations

To use private images while installing Levo's services in your environment, you need to follow 3 steps:

1. Copy Levo's public Docker images into your registry.
2. Create a secret in your Kubernetes cluster with the credentials to access your private registry.
3. Specify a values file to the Levo Helm chart to use your private registry.

Copy Levo's public Docker images into your registry

An example bash script to do this for AWS ECR has been provided below. Please modify this script to suit your needs.

```

!/usr/bin/env bash
Dependencies: yq, helm, awscli, docker
set -e
trap "exit" INT
region="us-west-2"
registry="your.registry"
helm repo add levoai https://charts.levo.ai || true
helm repo update
images=$(helm template levoai/levoai-satellite | yq -N '..|.image? | select(.) | sort -u)
images+=$(helm template levoai/levoai-ebpf-sensor | yq -N '..|.image? | select(.) | sort -u)
for image in "${images[@]}"; do
src_image=${image#"docker.io/" }
dest_image="$registry/$src_image"
repo_name=${src_image%:*}
aws ecr describe-repositories --repository-names $repo_name --region $region || aws ecr
create-repository --repository-name $repo_n
echo "Copying $src_image to $dest_image"
docker buildx imagetools create --tag $dest_image $src_image
done

```

Create a secret in your Kubernetes cluster with the credentials to access your private registry

Adapt the following command for your private registry:

```
kubectl create secret docker-registry ecr-auth --docker-server=your.registry
```

```
--docker-username=AWS --docker-password=$(aws ecr get-ic
```

Specify a values file to the Levo Helm chart to use your private registry

eBPF Sensor

sensor:

imageRepo: your.registry/levoai/ebpf_sensor

Satellite

global:

levoai_config_override:

onprem-api:

org-id: <id>

refresh-token: <token>

imageRegistry: your.registry

imagePullSecrets:

- name: ecr-auth

Supported Platforms

What OS platforms are supported?

x86-64 Processors only.

Linux running on bare metal, virtual machine, and container formats.

Linux Kernel versions 4.14 and above.

Debian, Fedora, OpenSUSE, and Amazon Linux based distributions

macOS & Windows Laptops are supported via a proxy based Sensor. Please refer to Quickstart for macOS/Windows (/quickstart/quickstart-mitm.md).

What Kubernetes platforms are supported?

minikube on Linux (<https://minikube.sigs.k8s.io/docs/>)

AKS (<https://azure.microsoft.com/en-us/services/kubernetes-service/overview>)

GKE (Debian Nodes Only. No Container-Optimized OS)

(<https://cloud.google.com/kubernetes-engine>)

EKS (<https://aws.amazon.com/eks/>)

Support for Docker Desktop, Docker Desktop based Kubernetes, and minikube on MacOS is on the roadmap.

What is the minimum Kubernetes version supported?

Minimum Kubernetes version: 1.18.0.

Kubernetes Node's Linux Kernel version ≥ 4.14 .

Install Guide

This guide provides comprehensive instructions for installing the Satellite, and Sensor on a supported platform of your choice (Kubernetes, Docker, or Linux Virtual Machine).

Platform specific instructions are described in the steps below.

Your estimated completion time is 10 minutes.

Install Steps

Signup with your enterprise email (<https://app.levo.ai/signup>)

OS Platform Compatibility Check (/guides/general/os-compat-check.mdx)

Install Satellite (/install-satellite)

Install Sensor (/install-traffic-capture-sensors)

Direct API Integrations

Direct API integrations with Levo is enabled via a GraphQL (<https://graphql.org/>) API endpoint.

All operations present in the levo UI & CLI are also accessible via the GraphQL API.

Please contact support@levo.ai for full GraphQL schema details with respect to the operations.

JUnit Format Test Results

JUnit XML format

(<https://www.ibm.com/docs/en/developer-for-zos/14.1.0?topic=formats-junit-xml-format>) is a really popular industry format that is used for test result reporting.

Levo CLI can produce JUnit/XUnit format output for all test results (security/schema

conformance). Please refer to the CLI Command Reference (/security-testing/test-laptop/levo-clicommand-reference.md) for more details on command line switches (--export-junit-xml) that activate this output.

There are many tools in the industry that consume this format, and produce HTML, and PDF reports of test results. A few examples are Jenkins JUnit Plugin (<https://plugins.jenkins.io/junit/>), Jenkins XUnit Plugin (<https://plugins.jenkins.io/xunit/>), junit2html (<https://gitlab.com/inorton/junit2html>), Ant JUnitReport (<https://ant.apache.org/manual/Tasks/junitreport.html>), xunit-viewer (<https://github.com/lukejpreston/xunit-viewer>), etc.

Below are some examples of JUnit format test results rendered in Jenkins via the JUnit plugin (<https://plugins.jenkins.io/junit/>).

JUnit Build Summary

JUnit Build Results

JUnit Build Detail-1

JUnit Build Detail-2

Providing RBAC Information for APIs

Associating RBAC Information with APIs (api-rbac-apis.md)

Associating RBAC Information Using Pattern Matching (Glob) (api-rbac-glob.md)

Associating RBAC Information with APIs

APIs specified in the API Catalog can be associated with RBAC information using the metadata.yml file. The association is made by importing a properly constructed metadata.yml into the specific API Catalog (Application or Service).

Please refer to the API Catalog screens in the UI to import a metadata.yml file.

What is the structure of the metadata.yml file?

Consider a scenario where you have the API endpoints mentioned below, and have implemented role based access controls (RBAC) for your APIs.

GET /

GET /identity/api/v1/admin/users/find

GET /identity/api/v2/vehicle/{vehicleId}/location

GET /workshop/api/shop/orders/{order_id}

Say, that there are two roles: ROLE_USER and ROLE_ADMIN associated with your API endpoints. These roles provide certain entitlements (capability to access specific API operations after authentication) to regular users and administrators.

The table below represents the RBAC entitlements:

API Endpoint

Roles Allowed to Access Endpoint

Comments

GET /

ROLE_USER, ROLE_ADMIN

Available to all roles

GET /identity/api/v1/admin/users/find ROLE_ADMIN

Endpoint has elevated privileges

GET /identity/api/v2/vehicle/{vehicleId}/location ROLE_USER

N/A for Admins

GET /workshop/api/shop/orders/

ROLE_USER

N/A for Admins

For the above scenario, YAML file (shown below), provides a mapping between the API endpoints and their associated roles.

This is an example metadata.yml file

roles:

This section captures the set of roles that are available to be associated with your API endpoints

- role: ROLE_USER

description: USER role that provides certain entitlements for regular users

- role: ROLE_ADMIN

description: ADMIN role that provides elevated privileges/entitlements for administrator

api:

This section defines actual associations between API endpoints and roles at the global level

This can be overridden at the individual API endpoint level

roles:

- ROLE_USER

- ROLE_ADMIN

The default role (and an associated user) that should be used to access all endpoints

This can be overridden at the individual API endpoint level

default_role: ROLE_USER

endpoints:

This section defines API endpoint specific overrides

- endpoint: GET /identity/api/v1/admin/users/find

roles:

- ROLE_ADMIN `admin/users/find` should only be accessed by ROLE_ADMIN

default_role: ROLE_ADMIN override the default role for this endpoint

- endpoint: GET /identity/api/v2/vehicle/{vehicleId}/location

roles:

- ROLE_USER `vehicle/{vehicleId}/location` should only be accessed by ROLE_USER

No need to override the default_role here, as it is already ROLE_USER

- endpoint: GET /workshop/api/shop/orders/{order_id}

roles:

- ROLE_USER `/shop/orders/{order_id}` should only be accessed by ROLE_USER

No need to override the default_role here, as it is already ROLE_USER

If you have many API endpoints and have complex requirements for associating RBAC information, the next section will help simplify the association, via the usage of pattern matching glob (<https://github.com/begin/globbingwhat-is-globbing>), etc.

Associating RBAC Information Using Pattern

Matching (Glob)

What problem does this solve?

Some applications contain a numerous API endpoints and associated RBAC roles/scopes.

This feature supports rapid, and efficient mapping of roles/scopes with numerous API endpoints at scale.

The feature uses a technique called Globbing, which is described below.

What is Globbing?

The term "globbing", also referred to as "glob matching" or "URL path expansion", is a programming concept that describes the process of using wildcards, referred to as "glob patterns" or "globs", for URL paths or other similar sets of strings.

You can read more about it here (<https://github.com/begin/globbingwhat-is-globbing>).

What are some use cases?

Consider a scenario where your application has numerous API endpoints, and three roles (shown below) for RBAC.

RBAC Role Name

Role Description

ROLE_USER

A consumer of the application

ROLE_MECHANIC Skilled at working with machines

ROLE_ADMIN

Administrator. Oversees everything

Below are example use cases, and the corresponding metadata.yml file structure.

1. Allow ROLE_USER access to all POST endpoints

roles:

This section captures the set of roles that are available to be associated with your API endpoints

- role: ROLE_USER

description: "A consumer of the application."

- role: ROLE_MECHANIC

description: "Skilled at working with machines."

- role: ROLE_ADMIN

description: "Administrator. Oversees everything."

api:

This section defines actual associations between API endpoints and roles at the global level

This can be overridden at the individual API endpoint level

roles:

- ROLE_USER

- ROLE_MECHANIC

- ROLE_ADMIN

The default role (and an associated user) that should be used to access all endpoints

This can be overridden at the individual API endpoint level

default_role: ROLE_USER

endpoint_groups:

This section allows Method and URL path based mapping between endpoints and roles

This specifies that ROLE_USER should have access to all POST endpoints

- methods:

- "POST"

roles:

- ROLE_USER

2. Allow ROLE_MECHANIC & ROLE_ADMIN access to GET endpoints beginning with

/workshop/

For example the API endpoints under consideration could be:

GET /workshop/shop

GET /workshop/shop/products

GET /workshop/mechanic

GET /workshop/mechanic/service_requests

roles:

This section captures the set of roles that are available to be associated with your API endpoints

- role: ROLE_USER

description: "A consumer of the application."

- role: ROLE_MECHANIC

description: "Skilled at working with machines."

- role: ROLE_ADMIN

description: "Administrator. Oversees everything."

api:

This section defines actual associations between API endpoints and roles at the global level

This can be overridden at the individual API endpoint level

roles:

- ROLE_USER
- ROLE_MECHANIC
- ROLE_ADMIN

The default role (and an associated user) that should be used to access all endpoints

This can be overridden at the individual API endpoint level

default_role: ROLE_USER

endpoint_groups:

This section allows Method and URL path based mapping between endpoints and roles

This specifies that ROLE_MECHANIC & ROLE_ADMIN should have access to all GET endpoints beginning with /workshop/

- methods:

- "GET"

patterns:

This describes the URI path to match as a glob string. This can be a list of URI paths.

- "/workshop/**"

roles:

- ROLE_MECHANIC
- ROLE_ADMIN

3. Allow ROLE_MECHANIC & ROLE_ADMIN access to GET endpoints with URI pattern /workshop/<URI path segment>

For example the API endpoints under consideration could be:

GET /workshop/shop

GET /workshop/mechanic

However the below API is not under consideration as it has more than one path segments:

GET /workshop/mechanic/service_requests

roles:

This section captures the set of roles that are available to be associated with your API endpoints

- role: ROLE_USER

description: "A consumer of the application."

- role: ROLE_MECHANIC

description: "Skilled at working with machines."

- role: ROLE_ADMIN

description: "Administrator. Oversees everything."

api:

This section defines actual associations between API endpoints and roles at the global level

This can be overridden at the individual API endpoint level

roles:

- ROLE_USER

- ROLE_MECHANIC

- ROLE_ADMIN

The default role (and an associated user) that should be used to access all endpoints

This can be overridden at the individual API endpoint level

default_role: ROLE_USER

endpoint_groups:

This section allows Method and URL path based mapping between endpoints and roles

This specifies that ROLE_MECHANIC & ROLE_ADMIN should have access to all GET endpoints with URI pattern `/workshop/<path segment>`

- methods:

- "GET"

patterns:

This describes the URI path to match as a glob string. This can be a list of URI paths.

- "/workshop/*"

roles:

- ROLE_MECHANIC

- ROLE_ADMIN

FAQs

What happens when an endpoint is part of a group and also is listed explicitly under the endpoints section?

In such case the explicit listing will take precedence over the group pattern matching. Please see example below.

roles:

This section captures the set of roles that are available to be associated with your API endpoints

- role: ROLE_USER

description: "A consumer of the application."

- role: ROLE_MECHANIC

description: "Skilled at working with machines."

- role: ROLE_ADMIN

description: "Administrator. Oversees everything."

api:

This section defines actual associations between API endpoints and roles at the global level

This can be overridden at the individual API endpoint level

roles:

- ROLE_USER

- ROLE_MECHANIC

- ROLE_ADMIN

The default role (and an associated user) that should be used to access all endpoints

This can be overridden at the individual API endpoint level

default_role: ROLE_USER

endpoint_groups:

This section allows Method and URL path based mapping between endpoints and roles

This specifies that ROLE_MECHANIC & ROLE_ADMIN should have access to all GET endpoints with URI pattern ``/workshop/<path segment>`

- methods:

- "GET"

patterns:

This describes the URI path to match as a glob string. This can be a list of URI paths.

- `"/workshop/*"`

roles:

- ROLE_MECHANIC

- ROLE_ADMIN

endpoints:

This section defines API endpoint specific overrides

- endpoint: GET `/workshop/list`

roles:

This overrides the mapping specified in the ``endpoint_groups`` section above

- ROLE_ADMIN ``/workshop/list`` should only be accessed by ROLE_ADMIN

default_role: ROLE_ADMIN override the default role for this endpoint

What methods are supported in the endpoint_groups?

All RESTful Methods are supported. In case no method is specified, the glob string will be matched against all RESTful methods.

What happens when the patterns list is absent?

In case the patterns list is absent, all endpoints matching the specified methods will be matched.

What matching operators are allowed for the patterns glob string?

Segments and Separators (/)

The separator is always the / character. A segment is everything that comes between the two separators. This includes path parameters.

Example:

`/workshop/api`

Here workshop and api are the segments and / is the separator.

Single Asterisk (*)

Single Asterisk (*) matches zero or more characters within one segment. It is used for globbing the URI path within one URI path segment.

Example:

The glob `/workshop/api/shop/*` will match endpoints such as:

`/workshop/api/shop/return_qr_code`

but not endpoints like:

`/workshop/api/shop/orders/all` or `/workshop/api/shop/orders/{order_id}`

Double Asterisk (**)

Double Asterisk (**) matches zero or more characters across multiple URI path segments.

Example:

The glob `/workshop/api/shop/**` will match the endpoints such as:

`/workshop/api/shop/return_qr_code` `/workshop/api/shop/orders/all`

`/workshop/api/shop/orders/{order_id}`

Question Mark(?)

Question mark(?) matches a single character within one URI path segment. When some URIs differ just one character, you can use the ?.

Example:

Glob string `/community/api/v?/coupon/*` will match:

`/community/api/v2/coupon/new-coupon` `/community/api/v2/coupon/validate-coupon`

`/community/api/v1/coupon/validate-coupon`

Providing Authentication for Tests

Most API endpoints require some form of user/client authentication. Effective security testing requires providing valid authentication credentials to Levo's autogenerated Test Plans.

This information can be provided in a secure, and structured manner via an `environment.yml` file.

How do I use `environment.yml` file?

The `environment.yml` file is autogenerated per Test Plan, and needs to be completed with appropriate authentication information, prior to the execution of the Test Plan.

The completed file is provided as an argument to the CLI. The CLI uses the credentials to access the target APIs and evaluate their security posture.

Are my secrets sent to Levo SaaS?

The `environment.yml` file contains secrets and is never sent to, or stored in Levo SaaS. This file is solely consumed by the CLI, and Levo SaaS does not have access to your secrets.

Please treat this file securely, and take all precautions necessary for handling secrets.

What is the structure of this autogenerated file?

This section covers authentication for standard security tests. For test plans that involve Horizontal Authorization Abuse

(https://en.wikipedia.org/wiki/Privilege_escalationHorizontal), and Vertical Authorization Abuse

(https://en.wikipedia.org/wiki/Privilege_escalationVertical) test cases,

please refer to the Providing Authorization Info (authz.md) section.

If the API endpoints you are testing have no role/scope information (used for granular authorization), and/or not susceptible to Horizontal Authorization Abuse (https://en.wikipedia.org/wiki/Privilege_escalationHorizontal), then autogenerated file will have the below structure.

Environment file that contains users, roles and their Authentication mechanisms that will be used by the API endpoints.

iam:

users:

- name: user_1

default: true This user's credentials will be used to access all API endpoints requiring AuthN

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

The default authentication mechanism used by Levo is Bearer Authentication (<https://swagger.io/docs/specification/authentication/bearer-authentication/>). You are required to provide

valid bearer tokens for the user above (user_1).

The default: true for user_1 specifies that this user's credentials will be used to access all API endpoints that require authentication.

Does Levo support other authentication methods?

The next section describes support for various standard and custom authentication methods.

Providing Authentication / Authorization for Tests

Most API endpoints require some form of user/client authentication, and authorization. Effective security testing requires providing valid authentication credentials to Levo's autogenerated Test Plans.

Below sections cover the provisioning of authentication and authorization information for test plans.

Providing Authentication Information (</guides/security-testing/common-tasks/authnauthz/authn>)
Supported Authentication Methods

(</guides/security-testing/common-tasks/authnauthz/supported-auth-methods>)

Providing Authorization Information for Authorization Abuse Tests

(</guides/securitytesting/common-tasks/authn-authz/authz>)

Providing Authorization Information for Authorization Abuse Tests

Levo's autogenerated Test Plans, evaluate API vulnerabilities related to Horizontal Authorization

Abuse (https://en.wikipedia.org/wiki/Privilege_escalationHorizontal), and Vertical Authorization Abuse (https://en.wikipedia.org/wiki/Privilege_escalationVertical). In order to effectively access and test APIs for these vulnerabilities, Levo may require authentication credentials for additional users, and their associated role information.

Provision Horizontal AuthZ Abuse Test Plans ([horizontal-authz.md](#))

Provision Vertical AuthZ Abuse Test Plans ([vertical-authz.md](#))

Provision Horizontal/Vertical AuthZ Abuse Test Plans ([horizontal-n-vertical-authz.md](#))

Providing Authorization Information for

Horizontal Authorization Abuse Test

Cases

Testing for Horizontal Authorization Abuse

(https://en.wikipedia.org/wiki/Privilege_escalationHorizontal), requires credentials for additional users.

Since horizontal authorization abuse (BOLA ([/vulnerabilities/v1/OWASP-API-10/A1-BOLA](#))) is about violating resource ownership constraints among users, these tests operate with a notion of users owning specific RESTful resources.

The tests first access the API's RESTful resource via the user who owns the resource, and then attempt to do the same with an additional user, who does not have resource ownership.

The autogenerated environment.yml file will have the below structure (assuming the default Bearer AuthN mechanism is being used).

Environment file that contains users, roles and their Authentication mechanisms that will be used by the API endpoints.

iam:

users:

- name: user_1

default: true This user should own RESTful resources subject to horizontal abuse testing

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

- name: user_2

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

You are required to provide valid bearer tokens for two users above (user_1 and user_2).

If you are wondering why bearer tokens for two users are required, user_1 is the default user, that is used in most of the API testing.

user_2 specifies credentials for another user (at the same role level as user_1), and is used in horizontal privilege escalation ([/vulnerabilities/v1/OWASP-API-10/A1BOLA](#)) tests.

If using an authentication mechanism other than Bearer AuthN, please modify the auto

generated YAML appropriately.

Providing Authorization Information for Test

Plans - Horizontal & Vertical

Authorization Abuse Test Cases

Often you will want to test APIs for both Horizontal Authorization Abuse (https://en.wikipedia.org/wiki/Privilege_escalationHorizontal), and Vertical Authorization Abuse (https://en.wikipedia.org/wiki/Privilege_escalationVertical), in a single test plan.

This requires credentials for additional users, and their associated role information.

User role information for API endpoints is provided in the API catalog via the metadata.yml file (/guides/security-testing/concepts/api-catalog/metadata.yml.md). The metadata file specifies the various roles used by the API, and specific roles that apply to specific API endpoints.

While the metadata file is used to specify role information, the environment.yml file requires the provisioning of one or more users per role (as specified in the metadata file), and their respective authentication credentials.

For example if the metadata file has specified two roles (ROLE_USER, and ROLE_ADMIN), the autogenerated environment.yml file will have the below structure (assuming the default Bearer AuthN mechanism is being used).

Environment file that contains users, roles and their Authentication mechanisms that will be used by the API endpoints.

iam:

users:

- name: user_1

Default user for `ROLE_USER` that is used in general, unless overridden by a specific test case.

This user is the primary user (victim) in horizontal authZ abuse test cases involving `ROLE_USER`.

default: true

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

roles:

- ROLE_USER

- name: user_2

This additional user at role `ROLE_USER` is used in horizontal authZ abuse test cases.

This user is the secondary user (attacker) in horizontal authZ abuse test cases.

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

roles:

- ROLE_USER

- name: user_3

Default user for `ROLE_ADMIN` that is used in general, unless overridden by a specific test case.

This user is the primary user (victim) in horizontal authZ abuse test cases involving `ROLE_ADMIN`.

default: true

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

roles:

- ROLE_ADMIN

- name: user_4

This additional user at role `ROLE_ADMIN` is used in horizontal authZ abuse test cases.

This user is the secondary user (attacker) in horizontal authZ abuse test cases involving `ROLE_ADMIN`.

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

roles:

- ROLE_ADMIN

Since the test plan has test cases for both horizontal and vertical authZ abuse, we have to provide credentials for 4 users. Two users for ROLE_USER, two users for ROLE_ADMIN. The two users at each role level will be used for the horizontal authorization abuse test cases (Victim user and Attacker user).

If using an authentication mechanism other than Bearer AuthN, please modify the auto generated YAML appropriately.

Supported Authentication Methods

The default authentication mechanism is Bearer Authentication

(<https://swagger.io/docs/specification/authentication/bearer-authentication/>), and the environment.yml file is autogenerated to use this method.

You can customize the authentication method to suit your needs.

Below are various authentication methods supported by levo, and the corresponding structure of the environment.yml file, to properly activate the authentication method.

Please customize the auto generated environment.yml file accordingly.

Bearer Tokens

The default authentication mechanism is Bearer Authentication

(<https://swagger.io/docs/specification/authentication/bearer-authentication/>). You are required to provide valid bearer

tokens for user_1 in the example below.

iam:

users:

- name: user_1

default: true This user's credentials will be used for all authn-authz

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

Login API / form based login (aka http_call)

If you use an API (or HTTP JSON forms) to acquire a authentication token (bearer token) in exchange for user credentials, you can use the http_call method.

This method requires you to provide the following:

The login URL

The HTTP method to use when fetching the URL. Only POST & GET supported. If unspecified will use POST.

The key names for both the username and password values that are sent in the login request's POST (JSON) body.

The location in the login URL's JSON response, where the authentication token is present.

The username and base64 encoded password values for the user_1 used in security tests.

Below is the syntax to enable http_call based login.

iam:

This section specifies how to extract an authn-authz token

authenticators:

- name:

<your friendly name for this authenticator. E.g. my_authenticator>

type: http_call Makes a HTTP request using the specified method

method: <POST | GET> Defaults to POST if unspecified

login_url: <Enter URL value. E.g. /identity/api/auth/login> URL for HTTP request

username_key: <JSON key for username> Key in HTTP request's JSON body that specifies the user value

password_key: <JSON key for password> Key in HTTP request's JSON body that specifies the password value

This section specifies how to extract a token in the HTTP response

session_credential_extractors:

- name: access_token

type: bearer_token

location: <header | body> Specifies the location to extract the token. Header or Body.

path: <JSON path expression> In case the location is `body`, a JSON path expression to the token in the response body

This section specifies actual user information the test plan will use

users:

- name: user_1

default: true This user's credentials will be used to access all API endpoints requiring AuthN

username: <user_id>

password_base64: <base64 password> Passwords need to be base64 encoded

authenticator: <friendly name of the authenticator specified above. E.g. my_authenticator>

Basic Authentication

Below are format examples for Basic Authentication with and without role information.

Basic Authentication (no roles)

iam:

authenticators:

- name: <your friendly name for this authenticator>

type: basic_auth Use Basic Authentication for API calls

users:

This section defines users and their respective credentials

The credentials will be used in the Basic Authentication scheme

- name: user_1

username: <username for an actual user in your API's backend>

password_base64: <password for the specified user> Passwords need to be base64 encoded

Below defines which authn-authz mechanism to use

authenticator: <friendly name of the authenticator specified above>

Basic Authentication (with roles)

iam:

authenticators:

- name: <your friendly name for this authenticator>

type: basic_auth Use Basic Authentication for API calls

users:

This section defines users and their respective credentials

The credentials will be used in the Basic Authentication scheme

`user_1` with role ROLE_USER

- name: user_1

default: true Default user for `ROLE_USER`

username: <username for an actual user in your API's backend>

password_base64: <password for the specified user> Passwords need to be base64 encoded

Below defines which authn-authz mechanism to use

authenticator: <friendly name of the authenticator specified above>

roles:

- ROLE_USER

`user_2` with role ROLE_USER

- name: user_2

username: <username for an actual user in your API's backend>

password_base64: <password for the specified user> Passwords need to be base64 encoded

Below defines which authn-authz mechanism to use

authenticator: <friendly name of the authenticator specified above>

roles:

- ROLE_USER

`admin_1` with role ROLE_ADMIN

- name: admin_1

default: true Default user for `ROLE_ADMIN`

username: <username for an actual user in your API's backend>

password_base64: <password for the specified user> Passwords need to be base64 encoded

Below defines which authn-authz mechanism to use

authenticator: <friendly name of the authenticator specified above>

roles:

- ROLE_ADMIN

`admin_2` with role ROLE_ADMIN

- name: admin_2

username: <username for an actual user in your API's backend>

password_base64: <password for the specified user> Passwords need to be base64 encoded

Below defines which authn-authz mechanism to use

authenticator: <friendly name of the authenticator specified above>

roles:

- ROLE_ADMIN

API key based authentication

The OpenAPI specification file (in the API catalog), specifies if the API uses API keys for authentication, and the exact location of the API key (query parameter, header, etc).

The environment.yml file provides specific values for the API key and can be specific for each user. Below is the format when using API keys.

iam:

authenticators:

- name: <your friendly name for this authenticator>

type: api_key Use API key authn-authz for API calls

users:

This section defines users and their respective API keys

- name: user_1

api_keys:

- name: <friendly name for your API key for user_1>

value: <your api key value>

Below defines which authn-authz mechanism to use

authenticator: <friendly name of the authenticator specified above>

If you using roles, the format is similar to the Basic Authentication example. You just need to use API key instead of username and password.

Cookie based authentication

Below examples are applicable, when cookies are being used for user authentication.

Use existing cookie values

Below example is applicable, when the cookie values are known a priori.

iam:

users:

- name: user_1

default: true This user's credentials will be used for all authn-authz

cookies:

- name: <Enter exact cookie name. E.g. JSESSIONID> Cookie is case sensitive

value: <Enter the cookie value>

Extract cookie via API call (aka http_call)

If you use an API (or HTTP JSON forms) to acquire a authentication cookie in exchange for user credentials, you can use the http_call method.

This method requires you to provide the following:

The HTTP URL of the API endpoint that returns the authentication cookie.

The HTTP method to use when fetching the URL. Only POST & GET supported. If unspecified will use POST.

The name/location of the username field that is sent in the login request's (JSON) body. This needs to be a JSON path expression.

The name/location of the password field that is sent in the login request's (JSON) body. This needs to be a JSON path expression.

The name of the cookie header in the JSON response, where the cookie is present.

The username and base64 encoded password values for the user_1 used in security tests.

Below is the syntax to extract cookies using a http_call.

iam:

authenticators:

- name: <your friendly name for this authenticator. E.g. my_auth_cookie_extractor>

type: http_call

method: <POST | GET> Defaults to POST if unspecified

login_url: <Enter URL value. E.g. /login> URL for HTTP request

request_params:

- name: username
value: <JSON path expression. E.g. \$\$.user.username> JSON path of username field.
- name: password
value: <JSON path expression. E.g. \$\$.user.password> JSON path of password field

JSON path expressions in the example shown above is representative of the below JSON

```
{  
  
"user": {  
  
"username": "<value>",  
  
"password": "<value>"  
  
}  
}
```

This section specifies how to extract the cookie from the HTTP response
session_credential_extractors:
- name: <your friendly name for this cookie extractor. E.g. my-cookies>
type: cookies Use cookie based authn-authz
location: headers Location of the cookie is in the response headers
header_name: Set-Cookie Case sensitive name of the header. All cookies in the Set-Cookie header are extracted

This section specifies actual user information the test plan will use
users:
- name: user_1
default: true This user's credentials will be used to access all API endpoints requiring AuthN
username: <user_id> Specify the actual user id
password_base64: <base64 password> Specify the user's base64 encoded password.
Below defines which authn-authz mechanism to use
authenticator: <friendly name of the authenticator specified above. E.g. my_auth_cookie_extractor>
Use existing cookie values (usage with roles)
Below example is applicable when you are running tests that involve multiple users belonging to different roles.
There are two roles and three users in the below example, that require cookie values to be specified.

iam:

users:

This section defines users and their respective cookie based credentials

`admin_1` with role ROLE_ADMIN

- name: admin_1

default: true Default user for `ROLE_ADMIN`

roles:

- ROLE_ADMIN

Use the below cookie header for authn-authz

cookies:

- name: <Enter exact cookie name. E.g. JSESSIONID> Cookie is case sensitive

value: <Enter the cookie value>

`user_1` with role ROLE_USER

- name: user_1

default: true Default user for `ROLE_USER`

roles:

- ROLE_USER

cookies:

- name: <Enter exact cookie name. E.g. JSESSIONID> Cookie is case sensitive

value: <Enter the cookie value>

`user_2` with role ROLE_USER

- name: user_2

roles:

- ROLE_USER

cookies:

- name: <Enter exact cookie name. E.g. JSESSIONID> Cookie is case sensitive

value: <Enter the cookie value>

Extract cookie via API call (usage with roles)

Below example is applicable when you are running tests that involve multiple users belonging to different roles, and you want to extract authentication cookies for them

There are two roles and three users in the below example, that require cookie values to be extracted using the http_call.

iam:

authenticators:

- name: <your friendly name for this authenticator. E.g. my_auth_cookie_extractor>

type: http_call

method: <POST | GET> Defaults to POST if unspecified

login_url: <Enter URL value. E.g. /login> URL for HTTP request

request_params:

- name: username
value: <JSON path expression. E.g. \$\$.user.username> JSON path of username field.
- name: password
value: <JSON path expression. E.g. \$\$.user.password> JSON path of password field

This section specifies how to extract the cookie from the HTTP response
session_credential_extractors:
- name: <your friendly name for this cookie extractor. E.g. my-cookies>
type: cookies Use cookie based authn-authz
location: headers Location of the cookie is in the response headers
header_name: Set-Cookie Case sensitive name of the header. All cookies in the Set-Cookie header are extracted.

This section specifies actual user information the test plan will use
users:

`admin_1` with role ROLE_ADMIN
- name: admin_1
default: true Default user for `ROLE_ADMIN`
username: <user_id> Specify the actual user id
password_base64: <base64 password> Specify the user's base64 encoded password.
Below defines which authn-authz mechanism to use
authenticator: <friendly name of the authenticator specified above. E.g.
my_auth_cookie_extractor>

`user_1` with role ROLE_USER
- name: user_1
default: true Default user for `ROLE_USER`
username: <user_id> Specify the actual user id
password_base64: <base64 password> Specify the user's base64 encoded password.
Below defines which authn-authz mechanism to use
authenticator: <friendly name of the authenticator specified above. E.g.
my_auth_cookie_extractor>

`user_2` with role ROLE_USER
- name: user_2
username: <user_id> Specify the actual user id
password_base64: <base64 password> Specify the user's base64 encoded password.
Below defines which authn-authz mechanism to use
authenticator: <friendly name of the authenticator specified above. E.g.
my_auth_cookie_extractor>

OAuth
Below methods describe how access tokens

(<https://www.oauth.com/oauth2-servers/access-tokens/>) can be extracted using the OAuth protocol.

Password Grant

The Password grant (<https://www.oauth.com/oauth2-servers/access-tokens/password-grant/>) enables Levo's Test Plans to exchange the user's username and password for an access token.

This method requires you to provide the following:

The URL of the token generation API endpoint

The URL of the refresh token generation API endpoint (optional)

The username of the user, for whom the token is being generated

The base64 encoded password of the user, for whom the token is being generated

A list of scopes for the user (optional)

The Client ID, if it is required by your OAuth server (optional)

The Client Secret, if it is required by your OAuth server (optional)

Below is the syntax to enable Password Grant.

iam:

authenticators:

- name: <Friendly name for this authenticator. E.g. oauth_2>

type: oauth2 Use OAuth protocol

grant_type: password

token_url: <Enter the URL for the token generation endpoint. E.g.

https://my-oauth/oauth/access_token>

method: POST

client_id: <Enter your client ID. E.g. 23lkjlekfjlskd90> Optional field

client_secret: <Enter your client secret. E.g. UYT9239FRE> Optional field

This section specifies how to extract the access token from the HTTP response

session_credential_extractors:

- name: access-token

type: bearer_token

location: body

path: \$\$access_token JSON path expression to the location of the access token in the response. Do not modify unless diff

This section specifies actual user information the test plan will use

users:

- name: user_1

default: true This user's credentials will be used for all authn-authz

username: <Enter the username for whom you want to extract the access token>

password_base64: <base64 password> Passwords need to be base64 encoded

scopes: Optional field with a list of scopes

E.g. - api.read

E.g. - api.write

authenticator: <friendly name of the authenticator specified above. E.g. oauth_2>

If using roles, please follow the cookie extractor with roles example above, and modify appropriately.

Client Credentials Grant

The Client Credentials Grant

(<https://www.oauth.com/oauth2-servers/access-tokens/client-credentials/>) is used for service-to-service API authentication. Use this method when testing internal APIs that do not require end-user authentication, but require service-to-service authentication.

This method requires you to provide the following:

The URL of the token generation API endpoint

The URL of the refresh token generation API endpoint (optional)

The Client ID, if it is required by your OAuth server

The Client Secret, if it is required by your OAuth server

A list of scopes for the user (optional)

Below is the syntax to enable Client Credentials Grant.

iam:

authenticators:

- name: <Friendly name for this authenticator. E.g. oauth_2>

type: oauth2 Use OAuth protocol

token_url: <Enter the URL for the token generation endpoint. E.g. https://my-oauth/oauth/access_token>

grant_type: client_credential

method: POST

client_id: <Enter your client ID. E.g. 23lkjlekfjlskd90>

client_secret: <Enter your client secret. E.g. UYT9239FRE>

This section specifies how to extract the access token from the HTTP response

session_credential_extractors:

- name: access-token

type: bearer_token

location: body

path: \$\$.access_token JSON path expression to the location of the access token in the response. Do not modify unless diff

This section specifies actual user information the test plan will use
users:

- name: user_1

default: true This user's credentials will be used for all authn-authz

scopes: Optional field with a list of scopes

E.g. - api.read

E.g. - api.write

authenticator: <friendly name of the authenticator specified above. E.g. oauth_2>

If using roles, please follow the cookie extractor with roles example above, and modify appropriately.

Providing Authorization Information for

Vertical Authorization Abuse Test

Cases

Testing for Vertical Authorization Abuse

(https://en.wikipedia.org/wiki/Privilege_escalationVertical), requires credentials for additional users, and their associated role information.

User role information for API endpoints is provided in the API catalog via the metadata.yml file (/guides/security-testing/concepts/api-catalog/metadata.yml.md). The metadata file specifies the various roles used by the API, and specific roles that apply to specific API endpoints.

While the metadata file is used to specify role information, the environment.yml file requires the provisioning of one or more users per role (as specified in the metadata file), and their respective authentication credentials.

For example if the metadata file has specified two roles (ROLE_USER, and ROLE_ADMIN), the autogenerated environment.yml file will have the below structure (assuming the default Bearer AuthN mechanism is being used).

Environment file that contains users, roles and their Authentication mechanisms that will be used by the API endpoints.

iam:

users:

List all the users, their roles, username and password, if required, in this section.

The default flag should be true if this user should be used as the default user for that role. If there are no roles, only one user should have default: True.

- name: user_1

default: true Default user for `ROLE_USER`

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

roles:

- ROLE_USER

- name: user_2

bearer_tokens:

- name: bearerAuth

value: <Enter the bearer token>

roles:

- ROLE_ADMIN

If using an authentication mechanism other than Bearer AuthN, please modify the auto generated YAML appropriately.

How do I run a Data Driven Test Plan?

Below are the high level steps for running a previously generated Data Driven Test Plan.

1. Check Test Plan State

If your Test Plan is in the Config Complete state, goto step 3 below.

If your Test Plan is in the Config in Progress state, goto step 2 below.

If your Test Plan is not in either of the above states, please contact Levo Support (support@levo.ai).

2. Configure Test Fixtures

Data Driven Test Plans may require some parameter data to be configured for the API endpoints, prior to execution.

Please follow detailed steps outlined here

(</guides/security-testing/test-your-app/test-app-security/data-driven/configure-plan-fixtures.md>),

to configure parameters via Test

Fixtures.

Your test plan's Runnable status, and number of test cases runnable will auto update as you configure required parameters.

The Test Plan will be Runnable if at least one Test Case is runnable.

You can always continue to step 3, even if you have not completed configuring parameters for all Test Suites, and Test Cases. This is OK as long as the Test Plan is in the Runnable state.

Test Cases that are not runnable, will be skipped during execution of the Test Plan.

3. Download & Configure environment.yml

Your test plan may have an auto generated environment.yml associated with it.

If it was auto generated follow the steps outlined here (</guides/security-testing/test-your-app/test-app-security/data-driven/configure-env.yml.md>), to download and configure it appropriately.

2. Execute Test Plan via CLI

Follow instructions here

(</guides/security-testing/test-your-app/test-app-security/data-driven/execute-test-plan.md>) to execute the Test Plan via the CLI (Test Runner).

3. View Test Results

In the Levo SaaS console side panel, click on Test Runs and navigate to your most recent test run results.

Running Test Plans via Levo CLI

How do I run a Zero Configuration Test Plan? ([run-zero-conf-test-plan.md](#))

How do I run a Data Driven Test Plan? ([run-data-driven-test-plan.md](#))

How do I run a Zero Configuration Test Plan?

Below are the high level steps for running a previously generated Zero Configuration Test Plan.

1. Download & Configure environment.yml

Your test plan may have an auto generated environment.yml associated with it.

If it was auto generated follow the steps outlined here

(</guides/security-testing/test-your-app/test-app-security/zero-conf/configure-env.yml.md>), to download and configure it appropriately.

2. Execute Test Plan via CLI

Follow instructions here

(</guides/security-testing/test-your-app/test-app-security/zero-conf/execute-test-plan.md>) to execute the Test Plan via the CLI (Test Runner).

3. View Test Results

In the Levo SaaS console side panel, click on Test Runs and navigate to your most recent test run results.

Common Tasks

This section describes tasks that are common when using Levo.

Providing Authentication / Authorization for Tests

(/guides/security-testing/commontasks/authn-Authz)

Providing RBAC Information for APIs (/guides/security-testing/common-tasks/api-rbac)

Running Test Plans

(/guides/security-testing/common-tasks/running-test-plans/runningtest-plans.md)

keywords: [API Catalog, OpenAPI

Specifications]

API Catalog

Levo autogenerates security and resilience tests for APIs based on schema definitions for API endpoints.

These schemas can either be auto discovered (by observing runtime traffic) or imported into Levo.

The API Catalog is a store and directory for the aforementioned schemas.

The catalog is comprised of Applications, and Services. An Application is a logical grouping of a set of API endpoints.

An Application can contain one or more Services, where a Service is another logical grouping of a set of API endpoints.

This type of organization is very common in Microservices Architecture (MSA)

(<https://aws.amazon.com/microservices/>).

keywords: [API Authorization, Roles,

Scopes]

Metadata.yml file

What is it?

Often API endpoints enforce granular authorization controls on users/clients using role based access control (RBAC) mechanisms. Effective security testing involves evaluating the proper configuration and functioning of these RBAC controls.

Currently there is no industry standard way to express RBAC information in OpenAPI specifications.

The metadata.yml allows associating RBAC information with API endpoints present in Levo's API Catalog.

Associating RBAC information with API endpoints in the API catalog is completely optional.

However, if testing authorization controls is desired, then providing RBAC information via the metadata.yml file is mandatory.

Are there other uses for this file?

RESTful APIs operate on resources, and provide CRUD operations on those resources.

Effective security also requires evaluating the proper functioning of state transitions that happen across these CRUD operations.

The metadata.yml also allows grouping API endpoints for specific resources, so that Levo can auto generate tests that evaluate the consistency of state changing CRUD operation

sequences.

How do I provide RBAC information for my API endpoints?

You can get detailed information here (</guides/security-testing/common-tasks/api-rbac>)

Configuring Test Fixtures

Test Case Level Fixtures

Above is an example of a test case for the endpoint GET
`/identity/api/v2/vehicle/{vehicleId}/location`.

This endpoint requires a valid value for path parameter `{vehicleId}` for successful invocation.

This requirement is shown in the Parameters table in the test case. Not Bound indicates that a valid value is required to execute this test case.

Levo autogenerates fixtures for these required API parameters. These autogenerated fixtures are located in the `parameters.py` code block of the test case (see below).

Here is a zoom in view of the autogenerated (commented-out) fixture for `vehicleId`:

```
@levo.fixture(name="vehicleId", location="path")  
def vehicleId():
```

```
    return "Enter Your Value Here."
```

Steps to Configure Fixture

1. Block select the fixture code in the editor, and use `CMD + /` to uncomment the fixture code.

```
@levo.fixture(name="vehicleId", location="path")  
def vehicleId():  
    return "Enter Your Value Here."
```

2. Now enter a value for the `vehicleId` that is valid for your live API endpoint that you want to test.

```
@levo.fixture(name="vehicleId", location="path")  
def vehicleId():  
    return "649acfac-10ea-43b3-907f-752e86eff2b6"
```

In the above example `649acfac-10ea-43b3-907f-752e86eff2b6` is the value that will be used for `vehicleId`, when executing this test case against a live API target.

3. Remember to save the changes you made.

You are done!

Test Fixtures for API Parameters

What are test fixtures?

Software test fixtures (https://en.wikipedia.org/wiki/Test_fixtureSoftware) initialize test functions.

They provide a fixed baseline so that tests execute reliably and produce consistent, repeatable, results.

JUnit Fixtures (<https://github.com/junit-team/junit4/wiki/Test-fixtures>), and PyTest Fixtures (<https://docs.pytest.org/en/6.2.x/fixture.html>) are pretty commonly used by modern development teams.

Initialization may setup services, state, or provide seed values to (mandatory & optional) API input parameters, so that the API invocation succeeds.

For example, effective testing of the API endpoint GET /ride_receipts/{receipt_id}, might require a known and valid value for the {receipt_id} parameter, which is present in the receipts database.

Without this known seed value, the test that exercises this API endpoint might not get a proper response from the API endpoint. Fixtures allow providing this seed value for the {receipt_id} parameter.

Levo's Test Fixtures

Levo provides fixtures for configuring seed values for various API endpoint parameters. These fixtures are available at the Test Case level, Test Suite level, and Test Plan level as shown below.

Fixtures at the Test Suite level will automatically apply to all Test Cases within that Test Suite. Fixtures at the Test Plan level will apply to all Test Suites.

Test Fixture Format

Above is an example of an API endpoint that has {vehicleId} as a path parameter. This endpoint checks if the vehicle specified by the {vehicleId} is present in its database.

If the specified vehicle is present, it returns 200 OK and the vehicle details. If not present it returns 404 Not Found and an appropriate error message.

In order to effectively test this endpoint for various security vulnerabilities, we need to know the ID of at least one legitimate vehicle present in the API server's database. Using random values for {vehicleId} will not exercise all code paths within the endpoint's implementation, largely resulting in 404 responses.

Below is an example of a test fixture that allows specifying a legitimate {vehicleId} value to be used when testing the endpoint.

```
@levo.fixture(name="vehicleId", location="path")
def vehicleId():
    return "649acfac-10ea-43b3-907f-752e86eff2b6"
```

While the example above returns a hardcoded value, the fixture could be coded to perform a database lookup, or make an API call to get the appropriate value.

Auto Generation of Fixtures

Levo auto generates test fixtures for various mandatory API parameters, and configures them with example values provided in the OpenAPI schema. Skeleton fixtures are generated

in cases where example values are not available.

You can always modify and customize the auto generated fixtures to suit specific needs.

Creating a Test Plan

Click on New Test Plan to start creating a test plan.

Choose from Zero-configuration or Data-Driven testing.

Select the application you want to create the Test Plan for from the list of your applications.

Fill in the necessary details and choose the endpoints you want to run your tests against.

Click on Next and select the categories of test you want to run from and choose from a wide variety of Tests like BOLA, SQLI, CORS, Fuzzing, etc.

Click on Generate Test Plan to finish creating the test plan.

Note: In case of Data-Driven Testing, configure the environment.yml file (you can read more about it in the next section).

Environment.yml file

Most API endpoints require some form of user/client authentication. In addition, API endpoints may also enforce granular authorization controls on users/clients using role based access control (RBAC) mechanisms.

Effective security testing requires providing valid users and their respective authentication credentials to Levo's autogenerated Test Plans.

This information can be provided in a secure, and structured manner via an environment.yml file.

How do I use environment.yml file?

The environment.yml file is autogenerated per Test Plan, and needs to be completed with appropriate user/authentication information, prior to the execution of the Test Plan.

The completed file is provided as an argument to the CLI. The CLI uses the credentials to access the target APIs and evaluate their security posture.

Are my secrets sent to Levo SaaS?

The environment.yml file contains secrets and is never sent to, or stored in Levo SaaS. This file is solely consumed by the CLI, and Levo SaaS does not have access to your secrets.

Please treat this file securely, and take all precautions necessary for handling secrets.

Tell me more

You can find more information on providing authentication/authorization information for tests here (</guides/security-testing/common-tasks/authn-authz>)

keywords: [API Security Testing, API Security Test Plan]

Test Plans

Autogenerated Test Plans are tailor-made for each API (and its associated endpoints).

API Security Test Plan

Test Plans can be generated for APIs present in either the Application or Service (API Catalog) groupings.

A Test Plan is structured as show below.

A Test plan is comprised of Test Suites. A Test Suite is focussed on a single API endpoint, and comprises of a set of Test Cases.

A Test Case has a singular objective, and tests the specific API endpoint for a specific vulnerability. For example, test the API endpoint for a SSRF (/vulnerabilities/v1/OWASPWEB-10/A10-SSRF) vulnerability, or an authorization bypass vulnerability.

Test Plan Types

At a high level test plans come in two flavors:

1. Zero Configuration (zero conf)
2. Data Driven

Zero Configuration vs. Data Driven

Time to Value

Test Category Coverage

Efficacy of Test Results

Horizontal Authorization Abuse

Coverage (BOLA / IDOR)

Vertical Authorization Abuse Coverage
(Privilege Escalation)

Zero

Data Driven

Configuration

Instant

Requires One Time Configuration

Reduced

Comprehensive

Moderate

Superior

No

Yes

Yes

Yes

All Methods.

PUT, POST, PATCH, & DELETE require additional configuration to

support stateful operations on resources.

Coverage for API Methods (GET, POST, etc.)

All Methods

Primary Use Case

Rapid

Deep / Comprehensive Assessments

Assessments

Zero Configuration

Zero Configuration test plans are instantly runnable, and only require specifying user authentication information.

These test plans use type information specified in the API schema to autogenerate data for various parameters required by the API.

For example is a string field in a POST body is of format type email, the test plan will autogenerate syntactically valid email addresses.

However there is no guarantee that the generated values are truly valid for the API endpoint being tested (the values may not exist in the backend database, etc.). Since parameter data is autogenerated, these test plans provide less security test coverage, and lower efficacy than Data Driven test plans.

For example, APIs often return 404 Not Found in lieu of 401 Unauthorized or 403 Forbidden during authentication/authorization failures. This is prevent hackers from enumerating resources.

Zero Configuration test plans cannot truly distinguish between an authentication/authorization failure and the absence of the requested resource in the API's backend database.

This results in lower efficacy and reduced test coverage (results might have false negatives).

Data Driven

As the name suggests, parameter data for APIs is end user supplied via fixtures (fixtures/test-fixtures.md).

By configuring parameter data, end users have complete control of what data is send when making API calls.

This dramatically increases efficacy of test results and test coverage, as the test plans can clearly distinguish between, authentication/authorization failures and the absence of the requested resource.

Concepts

Levo is purpose built for modern development teams, and developers.

Below are a few key concepts that should be pretty intuitive for developers and AppSec teams building API based applications.

API Catalog (api-catalog/api-catalog.md)
Test Plans (test-plans/test-plans.md)
Test Runner (test-runner.md)
Test Runs & Reports (test-run-reports.md)

CLI - Test Runner

The CLI is the component that executes the autogenerated Test Plans.
The CLI is packaged as a Docker container, and can be run on a developer's laptop or integrated into CI/CD environments.
Levo provides pre-packaged runner/actions integrations for several popular CI/CD products.

Test Run Reports

The execution results of the Test Plans are reported to Levo SaaS by the CLI.
The CLI also provides a summary of execution results in the console. This summary is also available in the CI/CD logs (when integrated with CI/CD).

Evaluate Levo using the sample application

crAPI

You can read more about crAPI here (<https://github.com/levoai/demo-apps/blob/main/crAPI/README.md>). Below are high level steps to launch and security test crAPI's APIs.

crAPI - Part I (crapi-part-1.md)

Part I covers:
Installing crAPI.
Importing crAPI's OpenAPI specifications.
Generating a tailored security test plan for crAPI.

crAPI - Part II (crapi-part-1.md)

Part II covers:
Running the test plan.
Viewing the test results.

CrAPI Sample App - Part 1

1. Install crAPI

1. Install the crAPI demo application by following instructions here (<https://github.com/levoai/demo-apps/blob/main/crAPI/docs/quick-start.md>).

2. Download

(<https://raw.githubusercontent.com/levoai/demo-apps/main/crAPI/api-specs/demo%20scenarios/onboarding-scenarios.json>) and save crAPI's OpenAPI specification

(OAS).

3. Verify crAPI is running by logging in, using one of the user credentials provided here (<https://github.com/levoai/demo-apps/blob/main/crAPI/docs/user-asset-info.md>).

2. Import crAPI APIs into Levo SaaS

1. Login into the Levo SaaS portal.

2. Click on API Catalog in the side panel and proceed to import crAPIs OAS (saved in step above).

3. In the import dialog name this API catalog as My crAPI, and complete the import.

3. Upload a metadata.yml file to enable authorization bypass (RBAC) tests

crAPI's APIs have role based access controls (RBAC). If want to validate the proper configuration and functioning of the said controls, you will need to construct a metadata.yml file and upload it to the catalog created in the previous step.

You can read more about authorization bypass tests and the metadata.yml file here (</guides/security-testing/concepts/api-catalog/metadata.yml.md>).

For your convenience, the appropriate metadata.yml for crAPI is shown below. Please upload this to catalog via the Metadata tab in the catalog UI.

metadata.yml file for crAPI that describes RBAC for API endpoints

roles:

This section captures the set of roles that are available to be associated with the API endpoints

- role: ROLE_USER

description: This role provides specific entitlements for regular users

- role: ROLE_MECHANIC

description: This role provides specific entitlements for mechanics

- role: ROLE_ADMIN

description: This role provides specific entitlements for administrators

api:

This section defines actual associations between API endpoints and roles at the global level

This can be overridden at the individual API endpoint level

roles:

- ROLE_USER

- ROLE_MECHANIC

- ROLE_ADMIN

The default role (and an associated user) that should be used to access all endpoints

This can be overridden at the individual API endpoint level

default_role: ROLE_USER

endpoints:

This section defines API endpoint specific overrides

- endpoint: GET /identity/api/v1/admin/users/find

roles:

- ROLE_ADMIN

default_role: ROLE_ADMIN

- endpoint: GET /identity/api/v2/vehicle/{vehicleId}/location

roles:

- ROLE_ADMIN

- ROLE_USER

default_role: ROLE_USER

- endpoint: GET /workshop/api/mechanic/mechanic_report

roles:

- ROLE_ADMIN

- ROLE_USER

default_role: ROLE_USER

- endpoint: GET /workshop/api/merchant/contact_mechanic

roles:

- ROLE_ADMIN

- ROLE_USER

default_role: ROLE_USER

- endpoint: GET /workshop/api/shop/orders/{order_id}

roles:

- ROLE_USER

4. Generate a security test plan for crAPI's APIs

1. Click on Test Plans in the side panel and proceed to create a test plan by clicking New Test Plan.

2. Pick Data Driven as the type of test plan to generate.

3. In the New Test Plan dialog name the plan as My crAPI Test Plan. Pick My crAPI as the API asset for this test plan.

4. Check the check box named Auto-populate API parameters for this test plan.

This uses examples in the OpenAPI specification file to auto populate test fixtures/API parameters (https://en.wikipedia.org/wiki/Test_fixtureSoftware) that are required in the generated test plan. The example values are used in making API requests during test execution.

5. Proceed to generate the test plan.

6. You will now have a test plan called My crAPI Test Plan in the Config Complete state.

7. Copy the LRN (Levo Resource Name) of My crAPI Test Plan to the clipboard.

8. Open the test plan My crAPI Test Plan, navigate to the environment.yml section, and download this file to your desktop. You can read more about the purpose of the file here (</guides/security-testing/concepts/test-plans/env-yml.md>).

CrAPI Sample App - Part 2

The test plan created was auto configured as you selected Auto-populate API parameters for this test plan in the previous step.

Just like developers run tests using JUnit, & PyTest fixtures, Levo's test plans use fixtures (https://en.wikipedia.org/wiki/Test_fixtureSoftware) to drive tests. The fixtures provide seed values for API parameters required for the proper execution of the tests. Levo used examples in the OpenAPI specification to auto populate these fixtures.

5. Install Levo CLI & Login

Levo CLI is the test runner that will execute the test plan against your running instance of crAPI. Follow the instructions here (</security-testing/test-laptop>) to install Levo CLI and authenticate it with Levo SaaS.

Skip this step if you have already completed it.

6. Execute the test plan against crAPI

Now we will use the Levo CLI to execute the test plan.

Prerequisites

Ensure you copied the LRN (Levo Resource Name) to the clipboard in the previous step.

Ensure you downloaded the environment.yml file from the test plan to your desktop.

Ensure the environment.yml file is in the same directory from which you launch Levo CLI. You may need to copy the file to the directory from where you launch the CLI.

Execute the following in the shell where you installed Levo CLI:

Use `host.docker.internal` instead of `localhost` or `127.0.0.1` if crAPI is running on your local machine.

Modify the `--target-url` value below if crAPI is running elsewhere.

```
export TEST_PLAN_LRN="<LRN value copied to your clipboard in previous steps>"
```

Execute security tests against crAPI

```
levo test --test-plan $TEST_PLAN_LRN --target-url http://host.docker.internal:8888 --env-file environment.yml
```

View the test results in the Test Runs page

1. In the Levo SaaS console side panel, click on Test Runs and navigate to your most recent test run results

2. You will notice that Levo has found failed test cases and an Broken Object Level Authorization (</vulnerabilities/v1/OWASP-API-10/A1-BOLA>) vulnerability. Navigate to the BOLA test case status, and review the summary and the logs.

Verify results using crAPI's Hackpad (Optional)

Inside crAPI, use the top level menu to navigate to Hackpad. Follow instructions in the Hackpad to verify if the IDOR finding reported by Levo is a true positive.

Congratulations! You are done.

Sample Applications For Evaluating Levo

Levo provides sample applications that help you evaluate security and resilience testing capabilities.

You can download these applications and follow instructions to test them with Levo.

Completely Ridiculous API (crAPI) ([crapi/crapi.md](#)) for security testing

MalSchema (<https://github.com/levoai/demo-apps/blob/main/MalSchema/README.rst>) for schema conformance testing

Configure environment.yml

Note: If your test plan does not have a environment.yml file associated with it you can skip this step, and proceed to the next.

Each auto generated test plan may have a environment.yml file associated with it, which provides critical authentication/authorization information for your APIs.

You can read more about this file here

(</guides/security-testing/concepts/test-plans/env-yml.md>).

You will need to configure this file with appropriate authentication/authorization information prior to executing the test plan.

1. Download the environment.yml file

In your test plan, navigate to the environment.yml section, and download this file to your desktop.

2. Configure the environment.yml file

Follow the instructions here (</guides/security-testing/common-tasks/authn-authz>) to configure appropriate user credentials/roles required to effectively test your live API endpoints.

Configure Test Plan Fixtures

1. Configure your Config in Progress test plan

Levo uses test fixtures (https://en.wikipedia.org/wiki/Test_fixtureSoftware) to provide proper values to API parameters prior to sending test traffic to the live API endpoints.

This test plan requires configuration of these test fixtures prior to execution. Proper configuration of the test plan will make it runnable.

1. Open the test plan and navigate to the test cases that require configuration (test cases are under test suites). Check the test case documentation for specific parameters that need configuration. Uncomment the auto-generated fixtures and follow this example (</guides/security-testing/concepts/test-plans/fixtures/configure-fixtures.md>), to configure values that are appropriate for your live API target.

2. Continue this process until either the test plan's state changes to Config Complete, or you have enough test cases/suites that are runnable.

3. Remember to save your changes to the test plan.

Data Driven Security Tests

The below figure describes the high level workflow for the generation and execution of Data Driven (</guides/security-testing/concepts/test-plans/test-plan-types.md>) security tests.

Just follow the steps in this guide, to generate and execute security tests against your APIs.

Execute Test Plan

1. Copy the test plan's Levo Resource Name (LRN)

From the test plans screen copy the LRN (Levo Resource Name) of your test plan to the clipboard.

2. Install Levo CLI & Login

Skip this step if you have already completed it.

Levo CLI is the test runner that will execute the test plan against your running instance of crAPI. Follow the instructions here (</security-testing/test-laptop>) to install Levo CLI and authenticate it with Levo SaaS.

3. Execute the test plan against you live API endpoints

Now we will use the Levo CLI to execute the test plan.

Prerequisites

Ensure you copied the LRN (Levo Resource Name) to the clipboard in the previous step.

Ensure you downloaded the environment.yml file (if present) from the test plan to your desktop.

Ensure the environment.yml file (if present) is in the same directory from which you launch Levo CLI. You may need to copy the file to the directory from where you launch the CLI.

Execute the following in the shell where you installed Levo CLI:

Use `host.docker.internal` instead of `localhost` or `127.0.0.1` if your API is running on your local machine.

```
levo test --test-plan <LRN value copied to clipboard > --target-url <your live API's base URL>
--env-file environment.yml
```

Note: If your test plan does not have an environment.yml file associated with it, please do not specify the `--env-file` option above.

4. View the test results in the Test Runs page

In the Levo SaaS console side panel, click on Test Runs and navigate to your most recent test run results

Congratulations! You are done.

Auto Generate Test Plan

1. Generate a security test plan for your APIs

1. Click on Test Plans in the side panel and proceed to create a test plan by clicking New Test Plan.

2. Pick Data Driven as the type of test plan to generate.

3. In the New Test Plan dialog pick a suitable name for the plan.

4. Pick the previously imported API catalog as the API asset for this test plan.

5. If your API specification has example values

(<https://swagger.io/docs/specification/adding-examples/>), and these example values will work with your live API endpoints, then

check the check box named Auto-populate API parameters for this test plan.

This uses examples in the OpenAPI specification file to auto populate test fixtures/API parameters (https://en.wikipedia.org/wiki/Test_fixtureSoftware) that are required in the generated test plan. The example values are used in making API requests during test execution.

6. Proceed to generate the test plan. The generated test plan will have coverage for several security vulnerabilities.

7. Depending on if you used Auto-populate API parameters for this test plan, in the previous step, and how comprehensive the provided examples are, your newly generated test plan with either be in the Config Complete or Config in Progress states.

8. If your test plan is in the Config Complete state, it is immediately runnable, and you can proceed to Configure environment.yml (configure-env.yml.md).

9. If your test plan is in the Config in Progress state, you will need to configure values for API parameters using test fixtures

(https://en.wikipedia.org/wiki/Test_fixtureSoftware). The next section describes the processing of configuring API parameter values via fixtures.

Import API Specifications

Levo requires OpenAPI specifications for security test generation. If you already have OpenAPI specifications, you can simply import them into the API Catalog.

Otherwise OpenAPI specifications can be generated via one of the following methods:

Auto-generate OpenAPI from live traffic via Levo's API Observability (/guides/api-observability.md) solution

If you have Postman Collections, use postman-to-openapi

(<https://github.com/levoai/postman-to-openapi>) to generate OpenAPI from your collections

If you have HAR files, you can contact support@levo.ai to have them converted to OpenAPI specifications

1. Import your APIs into Levo SaaS

If you are using an auto-generated API Catalog, pick your Application from the catalog and go to the next step.

1. Login into the Levo SaaS portal.
2. Click on API Catalog in the side panel and proceed to import you App's API specifications.
3. Select the catalog type as Application, and pick a suitable name for this catalog.
4. Complete the import, and verify if the API endpoints are visible in the catalog.

2. Upload a metadata.yml file to enable authorization bypass (RBAC) tests

If you are trying Levo for the first time, we recommend you skip this step and proceed to the next step.

If you have role based access controls (RBAC) for your APIs, and you wish to validate the proper configuration and functioning of the said controls, you will need to construct a metadata.yml file and upload it to the catalog created in the previous step.

You can read more about authorization bypass tests and the metadata.yml file here (</guides/security-testing/concepts/api-catalog/metadata.yml.md>).

Please construct an appropriate metadata.yml for your API endpoints and upload it via the Metadata tab for your API catalog in the Levo SaaS UI.

Configure environment.yml

Note: If your test plan does not have a environment.yml file associated with it you can skip this step.

Each auto generated test plan may have a environment.yml file associated with it, which provides critical authentication/authorization information for your APIs.

You can read more about this file here

(</guides/security-testing/concepts/test-plans/env.yml.md>).

You will need to configure this file with appropriate authentication/authorization information prior to executing the test plan.

1. Download the environment.yml file

Open your test plan, navigate to the environment.yml section, and download this file to your desktop.

2. Configure the environment.yml file

Follow the instructions here (</guides/security-testing/common-tasks/authn-authz>) to configure appropriate user credentials/roles required to effectively test your live API endpoints.

Execute Test Plan

1. Copy the test plan's Levo Resource Name (LRN)

From the test plans screen copy the LRN (Levo Resource Name) of your test plan to the clipboard.

2. Install Levo CLI & Login

Skip this step if you have already completed it.

Levo CLI is the test runner that will execute the test plan against your running instance of crAPI. Follow the instructions here (</security-testing/test-laptop>) to install Levo CLI and authenticate it with Levo SaaS.

3. Execute the test plan against you live API endpoints

Now we will use the Levo CLI to execute the test plan.

Prerequisites

Ensure you copied the LRN (Levo Resource Name) to the clipboard in the previous step.

Ensure you downloaded the environment.yml file (if present) from the test plan to your desktop.

Ensure the environment.yml file (if present) is in the same directory from which you launch Levo CLI. You may need to copy the file to the directory from where you launch the CLI.

Execute the following in the shell where you installed Levo CLI:

Use `host.docker.internal` instead of `localhost` or `127.0.0.1` if your API is running on your local machine.

```
levo test --test-plan <LRN value copied to clipboard> --target-url <your live API's base URL>
--env-file environment.yml
```

Note: If your test plan does not have an environment.yml file associated with it, please do not specify the --env-file option above.

4. View the test results in the Test Runs page

In the Levo SaaS console side panel, click on Test Runs and navigate to your most recent test run results

Congratulations! You are done.

Auto Generate Test Plan

1. Generate a security test plan for your APIs

1. Click on Test Plans in the side panel and proceed to create a test plan by clicking New Test Plan.

2. Pick Zero Config as the type of test plan to generate

3. In the New Test Plan dialog pick a suitable name for the plan.

4. Pick the previously imported API catalog as the API asset for this test plan.

5. Proceed to generate the test plan. The generated test plan will have coverage for several security vulnerabilities.

Zero Config Test Plans do not support horizontal authorization bypass (BOLA (</vulnerabilities/v1/OWASP-API-10/A1-BOLA>)) tests. If you skipped providing RBAC association info via the metadata.yml file, no tests for vertical authorization bypass (BFLA

(/vulnerabilities/v1/OWASP-API-10/A5-BFLA)) will be generated.

6. Your new test plan will be in the Config Complete state, and is immediately runnable. Please proceed to the next step.

Import API Specifications

Levo requires OpenAPI specifications for security test generation. If you already have OpenAPI specifications, you can simply import them into the API Catalog.

Otherwise OpenAPI specifications can be generated via one of the following methods:

Auto-generate OpenAPI from live traffic via Levo's API Observability (/guides/api-observability.md) solution

If you have Postman Collections, use postman-to-openapi

(<https://github.com/levoai/postman-to-openapi>) to generate OpenAPI from your collections

If you have HAR files, you can contact support@levo.ai to have them converted to OpenAPI specifications

1. Import your APIs into Levo SaaS

If you are using an auto-generated API Catalog, pick your Application from the catalog and go to the next step.

1. Login into the Levo SaaS portal.
2. Click on API Catalog in the side panel and proceed to import you App's API specifications.
3. Select the catalog type as Application, and pick a suitable name for this catalog.
4. Complete the import, and verify if the API endpoints are visible in the catalog.

2. Upload a metadata.yml file to enable authorization bypass (RBAC) tests

If you are trying Levo for the first time, we recommend you skip this step and proceed to the next step.

If you have role based access controls (RBAC) for your APIs, and you wish to validate the proper configuration and functioning of the said controls, you will need to construct a metadata.yml file and upload it to the catalog created in the previous step.

You can read more about authorization bypass tests and the metadata.yml file here (/guides/security-testing/concepts/api-catalog/metadata-yml.md).

Please construct an appropriate metadata.yml for your API endpoints and upload it via the Metadata tab for your API catalog in the Levo SaaS UI.

Zero Conf Security Tests

The below figure describes the high level workflow for the generation and execution of Zero Configuration (/guides/security-testing/concepts/test-plans/test-plan-types.md) security tests.

Just follow the steps in this guide, to generate and execute security tests against your APIs.

Automatically test APIs for security vulnerabilities

I want effortless and instant testing (zero-conf/zero-conf.md)

Does not require data provisioning or parameter configuration

Runs instantly

Reduced security test coverage

Reduced efficacy of test results

I want comprehensive test coverage and better efficacy (datadriven/data-driven.md)

Best security test coverage

Best efficacy of test results

Requires data/parameter configuration for API endpoints

May require associating API endpoints with RBAC info via a metadata.yml file

Using your own app to evaluate Levo's schema conformance testing

Ensure your target application is running and note down its URL.

Use host.docker.internal instead of localhost or 127.0.0.1 for targets on your local machine.

I use a live URL for my OpenAPI v3 specifications

Execute the following in the shell where you installed Levo CLI:

```
export SCHEMA_URL="<your live schema's URL>"
```

```
export TARGET_URL="<your live target's URL>"
```

Run all schema conformance tests for all my API operations

```
levo test-conformance --schema $SCHEMA_URL --target-url $TARGET_URL
```

Provide custom headers (e.g. Authorization if required)

```
levo test-conformance --schema $SCHEMA_URL --target-url $TARGET_URL -H "Authorization: Bearer <token>"
```

Now you can view the test results in the <https://levo.ai> (<https://levo.ai>) SaaS console's Test Runs page.

I use a local JSON or YAML file my OpenAPI v3 specifications

Levo CLI can accept schemas (OASv3, Swagger, etc.) as a file rather than a live URL. The Levo CLI container, mounts the user's HOME directory (on the host), as readonly directory inside the container.

Execute the following in the shell where you installed Levo CLI:

```
export SCHEMA_FILE="<your schema file's absolute path from the $HOME directory>"
```

```
export TARGET_URL="<your live target's URL>"
```

Run all schema conformance tests for all my API operations

```
levo test-conformance --schema $SCHEMA_FILE --target-url $TARGET_URL
```

Provide custom headers (e.g. Authorization if required)

```
levo test-conformance --schema $SCHEMA_FILE --target-url $TARGET_URL -H "Authorization:
```

Bearer <token>"

Now you can view the test results in the <https://levo.ai> (<https://levo.ai>) SaaS console's Test Runs page.

How do I use Levo with my own applications?

First review the Concepts (</guides/security-testing/concepts>) section. Then follow these steps:

Setup Levo CLI & sign up on [Levo.ai](https://levo.ai)

[Levo CLI for Mac OS \(/security-testing/test-laptop/test-mac-os.md\)](/security-testing/test-laptop/test-mac-os.md)

[Levo CLI for Linux \(/security-testing/test-laptop/test-linux.md\)](/security-testing/test-laptop/test-linux.md)

[Levo CLI for Windows \(/security-testing/test-laptop/test-windows.md\)](/security-testing/test-laptop/test-windows.md)

Pick your use case & follow instructions

Automatically test APIs for security vulnerabilities (<test-app-security/choices.md>)

Automatically test APIs for schema conformance (<test-app-schema-conformance.md>)

[Levo CLI in CI/CD](#)

[Embed Levo in CI/CD Quality](#)

[Gates](#)

You can embed Levo's contract & security tests in various stages of your software delivery pipeline via CI/CD Quality Gates

(<https://docs.microsoft.com/enus/azure/devops/pipelines/release/approvals/gates?view=azure-devops>).

While Levo can be embedded in any CI/CD product (via the CLI), below are first class integrations.

[GitHub Actions \(/security-testing/github-action\)](/security-testing/github-action)

[Jenkins \(/security-testing/jenkins-plugin\)](/security-testing/jenkins-plugin)

[Others](#)

Need support for a CI/CD vendor that is not listed above?

Levo's autogenerated tests can be embedded in any CI/CD product by simply wrapping the Levo CLI in a shell script that is invoked by your CI/CD vendor's job hooks.

Contact support@levo.ai for more information.

[Security Testing](#)

[Concepts \(/guides/security-testing/concepts\)](/guides/security-testing/concepts)

[Sample Application \(/guides/security-testing/test-sample-app\)](/guides/security-testing/test-sample-app)

[Testing your Own Apps \(/guides/security-testing/test-your-app/testing-your-own-apps\)](/guides/security-testing/test-your-app/testing-your-own-apps)

[Levo CLI in CI/CD \(/guides/security-testing/levocli-cicd\)](/guides/security-testing/levocli-cicd)

[Common Tasks \(/guides/security-testing/common-tasks\)](/guides/security-testing/common-tasks)

[Frictionless API Observability](#)

API Observability

Levo's frictionless & privacy-preserving API observability solution, auto discovers and auto documents all your APIs.

Here (<https://levo.ai/frictionless-api-observability/>) is a blog that describes the high level benefits.

Get Started

Key Concepts (</guides/key-concepts>)

Supported Platforms (</guides/general/supported-platforms.md>)

Quickstart (</quickstart>)

Demo Application (<demo-application.md>)

Demo Application

crAPI is an API driven sample application, that can be used to evaluate API Observability.

You can read more about crAPI here

(<https://github.com/levoai/demo-apps/blob/main/crAPI/README.md>).

1. First follow instructions in the Install Guide (</guides/install-guide>) to install the Satellite, and Sensor components successfully.
 2. Now install the crAPI demo application by following instructions here (<https://github.com/levoai/demo-apps/blob/main/crAPI/docs/quick-start.md>).
 3. Generate traffic for crAPI by browsing its web UI for at least five minutes.
 4. Go to the API Catalog in the Levo web console to see auto discovered OpenAPI specifications for crAPI. The specifications will be grouped under the Application Name you specified when installing the Sensor.
- Congratulations! You are done.

keywords: [API Security, eBPF, API Observability, Security Testing]

Guides

This section contains guides to help you get started with LevoAI's API Observability and Security Testing.

Key Concepts (</guides/key-concepts>)

API Observability (</guides/api-observability>)

Security Testing (</guides/security-testing>)

General (</guides/general>)

Install Guide (</guides/install-guide>)

Demo Application (</guides/demo-application>)

keywords: [API Security, eBPF, API Observability]

Key Concepts

API Observability

API Observability involves three components - a) Sensor, b) Satellite, and c) API Catalog.

eBPF Sensor

The sensor is a userspace process, that uses Extended Berkeley Packet Filters (eBPF) (<https://ebpf.io>) to passively capture API traffic (full HTTP payloads) from Linux workloads. The sensor works on bare metal, virtual machine, and container formats.

eBPF is used by all the modern observability & security vendors, including DATADOG (<https://www.datadoghq.com/product/network-monitoring/network-performance-monitoring/>), new relic (<https://newrelic.com/platform/kubernetes-pixie>), paloalto networks (<https://www.paloaltonetworks.com/prisma/cloud>), aqua (<https://www.aquasec.com/products/tracee/>), sysdig (<https://sysdig.com/>), Cilium (<https://cilium.io/>), etc.

Similar to network traffic mirroring

(<https://docs.aws.amazon.com/vpc/latest/mirroring/what-is-traffic-mirroring.html>) the sensor works at the Linux host level.

The sensor does not require any modifications to your application workloads. Absolutely no SDKs, no code changes, no configuration changes, no sidecars, and no runtime agents.

The sensor is not inline with the application workloads and will not impact the workload. API traffic can be aggressively sampled in high traffic environments, to limit CPU consumption by the sensor.

The sensor can be run in both production and pre-production environments. Captured API Traces (HTTP traffic) is sent to the Satellite component, for data anonymization, schema generation, and sensitive data detection/annotation.

Satellite

The Satellite runs within the customer premises or VPC, and can be run alongside application workloads, or in a separate host.

It uses sampled API traffic (API Traces) from the Sensor to:

1. Auto discover API endpoints
2. Derive (OpenAPI) schema for the discovered API endpoints
3. Detect sensitive data (PII, PSI, etc.) present in API data
4. Annotate the derived schema with sensitive data types
5. Send the API schema to Levo SaaS for API Catalog building

Your Data Stays with You!

Privacy Preserving

Privacy preserving technology ensures your API data stays with you.

Typical API observability solutions, will ingest all your API data into their SaaS, and put the burden of redacting sensitive customer data on you.

Levo's privacy preserving technology, does not ingest any of your API data into SaaS. Levo discovers and documents your APIs using only data type inferences performed in the Satellite.

API Catalog

Levo SaaS aggregates data received from the Satellite to create an API Catalog

(/guides/security-testing/concepts/api-catalog/api-catalog.md).

The API Catalog is the source of truth to answer the following questions:

What APIs do I have in my environment?

Which APIs are exposed externally?

What is the schema for my APIs?

Which APIs process sensitive data (PII, PSI, etc.)?

Which users, via which roles/scopes are accessing, which API endpoints?

Are my API schema's drifting?

The API Catalog also serves as the primary input for Levo's API security

(<https://docs.levo.ai/test-your-app/test-app-security/choices>) & contract

(<https://docs.levo.ai/test-yourapp/test-app-schema-conformance>) testing features.

Frictionless API Observability

API Observability

Levo's frictionless & privacy-preserving API observability solution, auto discovers and auto documents all your APIs.

Here (<https://levo.ai/frictionless-api-observability/>) is a blog that describes the high level benefits.

Get Started

Key Concepts (/guides/key-concepts)

Supported Platforms (/guides/general/supported-platforms.md)

Quickstart (/quickstart)

Demo Application (demo-application.md)

Demo Application

crAPI is an API driven sample application, that can be used to evaluate API Observability.

You can read more about crAPI here

(<https://github.com/levoai/demo-apps/blob/main/crAPI/README.md>).

1. First follow instructions in the Install Guide (/guides/install-guide) to install the Satellite, and Sensor components successfully.

2. Now install the crAPI demo application by following instructions here

(<https://github.com/levoai/demo-apps/blob/main/crAPI/docs/quick-start.md>).

3. Generate traffic for crAPI by browsing its web UI for at least five minutes.

4. Go to the API Catalog in the Levo web console to see auto discovered OpenAPI specifications for crAPI. The specifications will be grouped under the Application Name you specified when installing the Sensor.

Congratulations! You are done.

keywords: [API Security, eBPF, API Observability, Security Testing]

Guides

This section contains guides to help you get started with LevoAI's API Observability and Security Testing.

Key Concepts (/guides/key-concepts)

API Observability (/guides/api-observability)

Security Testing (/guides/security-testing)

General (/guides/general)

Install Guide (/guides/install-guide)

Demo Application (/guides/demo-application)

keywords: [API Security, eBPF, API Observability]

Key Concepts

API Observability

API Observability involves three components - a) Sensor, b) Satellite, and c) API Catalog.

eBPF Sensor

The sensor is a userspace process, that uses Extended Berkeley Packet Filters (eBPF) (<https://ebpf.io>) to passively capture API traffic (full HTTP payloads) from Linux workloads. The sensor works on bare metal, virtual machine, and container formats.

eBPF is used by all the modern observability & security vendors, including DATADOG (<https://www.datadoghq.com/product/network-monitoring/network-performance-monitoring/>), new relic (<https://newrelic.com/platform/kubernetes-pixie>), paloalto networks (<https://www.paloaltonetworks.com/prisma/cloud>), aqua (<https://www.aquasec.com/products/tracee/>), sysdig (<https://sysdig.com/>), Cilium (<https://cilium.io/>), etc.

Similar to network traffic mirroring

(<https://docs.aws.amazon.com/vpc/latest/mirroring/what-is-traffic-mirroring.html>) the sensor works at the Linux host level.

The sensor does not require any modifications to your application workloads. Absolutely no SDKs, no code changes, no configuration changes, no sidecars, and no runtime agents.

The sensor is not inline with the application workloads and will not impact the workload. API traffic can be aggressively sampled in high traffic environments, to limit CPU consumption by the sensor.

The sensor can be run in both production and pre-production environments. Captured API Traces (HTTP traffic) is sent to the Satellite component, for data anonymization, schema generation, and sensitive data detection/annotation.

Satellite

The Satellite runs within the customer premises or VPC, and can be run alongside application workloads, or in a separate host.

It uses sampled API traffic (API Traces) from the Sensor to:

1. Auto discover API endpoints
2. Derive (OpenAPI) schema for the discovered API endpoints
3. Detect sensitive data (PII, PSI, etc.) present in API data

4. Annotate the derived schema with sensitive data types
5. Send the API schema to Levo SaaS for API Catalog building

Your Data Stays with You!

Privacy Preserving

Privacy preserving technology ensures your API data stays with you.

Typical API observability solutions, will ingest all your API data into their SaaS, and put the burden of redacting sensitive customer data on you.

Levo's privacy preserving technology, does not ingest any of your API data into SaaS. Levo discovers and documents your APIs using only data type inferences performed in the Satellite.

API Catalog

Levo SaaS aggregates data received from the Satellite to create an API Catalog (/guides/security-testing/concepts/api-catalog/api-catalog.md).

The API Catalog is the source of truth to answer the following questions:

What APIs do I have in my environment?

Which APIs are exposed externally?

What is the schema for my APIs?

Which APIs process sensitive data (PII, PSI, etc.)?

Which users, via which roles/scopes are accessing, which API endpoints?

Are my API schema's drifting?

The API Catalog also serves as the primary input for Levo's API security (<https://docs.levo.ai/test-your-app/test-app-security/choices>) & contract (<https://docs.levo.ai/test-yourapp/test-app-schema-conformance>) testing features.

Install Satellite

1. Prerequisites

You have an account on Levo.ai (<https://app.levo.ai/login>)

Depending on the region you are installing in, you may need to use Levo.ai India-1 (<https://app.india-1.levo.ai/login>) to create account.

OS Compatibility script (/guides/general/os-compat-check.mdx) indicates the Linux host (that you want to instrument with the Sensor) is compatible.

At least 4 CPUs

At least 8 GB RAM

The Satellite URL should be reachable from the Sensor.

The Collector listens for spans from the eBPF Sensor on port 4317 using HTTP/2 (gRPC), and port 4318 using HTTP/1.1.

The Satellite listens for spans from the PCAP Sensor on port 9999 using HTTP/1.1.

2. Copy Authorization Key from Levo.ai

The Satellite uses an authorization key to access Levo.ai.

Login (<https://app.levo.ai/login>) to Levo.ai.

Click on your user profile.

Click on User Settings

Click on Keys on the left navigation panel

Click on Get Satellite Authorization Key

Copy your authorization key. This key is required in subsequent steps below.

3. Follow instructions for your platform

Install on Kubernetes (satellite-kubernetes.md)

Install on Linux host via Docker Compose (satellite-docker.mdx)

Install in AWS EC2 using Levo Satellite AMI (satellite-ami-aws-ec2.mdx)

Install in AWS EKS (satellite-aws-eks.md)

Install in AWS EKS using EC2 (satellite-aws-ecs.mdx)

Install in AWS EKS using Fargate (satellite-aws-eks-fargate.md)

Install in AWS ECS (satellite-aws-ecs.mdx)

Satellite on AWS EKS

AWS EKS supports two compute types for its nodes, EC2 and Fargate. Depending on your usecase, you can follow the installation steps given below.

Install using EC2 (satellite-aws-ecs.mdx)

Install using Fargate (satellite-aws-eks-fargate.md)

Prerequisites

eksctl (<https://eksctl.io/>) version \geq v0.152.0

Helm v3 (<https://helm.sh/docs/intro/install/>) installed and working on your local machine.

An AWS account with EKS permissions.

Install in AWS EKS using EC2

1. Setup environment variables

```
export CLUSTER_NAME='Cluster Name'
```

```
export REGION='AWS Region'
```

```
export ACCOUNT_ID='AWS Account ID'
```

2. Cluster Creation

```
read -r -d " EKS_CLUSTER <<EOF
```

```
apiVersion: eksctl.io/v1alpha5
```

```
kind: ClusterConfig
```

```
metadata:
```

```
name: ${CLUSTER_NAME}
```

```
region: ${REGION}
```

```
vpc:
```

```
subnets:
```

```
private:
```

MENTION THE SUBNETS YOU WANT TO USE FOR YOUR SATELLITE
FOR EXAMPLE:

```
us-west-2a: { id: subnet-0d09e999a579234ea }
```



```
us-west-2b: { id: subnet-0d09e999a579234eb }
nodeGroups:
- name: ng-e2e
instanceType: t2.xlarge
desiredCapacity: 1
volumeSize: 40
privateNetworking: true
EOF
echo "${EKS_CLUSTER}" > eks-cluster.yaml
eksctl create cluster -f ./configuration/eks-cluster.yaml
```

3. Connecting to the cluster

AWS EKS grants cluster admin permissions to the account from which the cluster is created. If you don't need access to the cluster for other AWS Users, you can skip this section.

Access to other AWS users in the same account can be granted via 2 ways.

Adding individual access to user accounts

Giving the permissions to a user group

Adding individuals to the cluster

This command can be run to add an individual user account to the cluster's aws-auth configmap

```
eksctl create iamidentitymapping \
--cluster ${CLUSTER_NAME} \
--region ${REGION} \
--arn <AWS ACCOUNT ARN FOR THE USER> \
--group system:masters \
--no-duplicate-arns \
--username <AWS USERNAME FOR THE USER>
```

Giving access to an IAM User Group

We create a role `developer.assume-access.role` and attach two policies to it. The first one is `EKSFullAccess` so that it has access to all the EKS resources. The second one is `developer.assume-eks-access-role.policy` that allows assuming the role.

A detailed guide on defining the roles and policies can be found here

(<https://eng.grip.security/enabling-aws-iam-group-access-to-an-eks-cluster-using-rbac>).

Once you have followed the above guide to create the roles and attach the specific policies, you can add the role to the cluster's aws-auth config map to let the developers group access the cluster

```
eksctl create iamidentitymapping \
--cluster ${CLUSTER_NAME} \
--region ${REGION} \
--arn arn:aws:iam:${ACCOUNT_ID}:role/developer.assume-access.role \
--group system:masters \
```

This needs to be run in order to grant access to the cluster.

One can Connect to the cluster by running just a single command

```
aws eks update-kubeconfig --name ${CLUSTER_NAME} --region ${REGION}> --role-arn
```

```
arn:aws:iam::${ACCOUNT_ID}:role/developer.assume-acce
```

This commands updates the kubeconfig and adds the context for the cluster and sets the current context to it. The --role argument sets the correct role and policies so that seamless access to the cluster is granted instantly.

4. Setting the cluster up

Creating an OIDC provider

Run these two commands:

```
oidc_id=$(aws eks describe-cluster --name ${CLUSTER_NAME} --region ${REGION} --query "cluster.identity.oidc.issuer" --output text | c
```

```
aws iam list-open-id-connect-providers | grep $oidc_id | cut -d "/" -f4 | cut -d "\"" -f1
```

If this returns a value, that is the OIDC ID that we need. If the statement returns nothing, run this command:

```
eksctl utils associate-iam-oidc-provider --cluster ${CLUSTER_NAME} --region ${REGION} --approve
```

This creates an OIDC Identity Provider.

Next, to create a role in AWS for the EBS CSI Driver add-on (Amazon Elastic Block Store CSI Driver (<https://docs.aws.amazon.com/eks/latest/userguide/ebs-csi.html>) manages persistent volumes in EKS) we need to run these:

```
OIDC=$(aws iam list-open-id-connect-providers | grep $oidc_id | cut -d "/" -f4 | cut -d "\"" -f1)
read -r -d " EBS_DRIVER_POLICY <<EOF
{
"Version": "2012-10-17",
"Statement": [
{
"Sid": "",
"Effect": "Allow",
"Principal": {
"Federated":
"arn:aws:iam::${ACCOUNT_ID}:oidc-provider/oidc.eks.${REGION}.amazonaws.com/id/${OIDC}
",
},
"Action": "sts:AssumeRoleWithWebIdentity",
"Condition": {
"StringEquals": {
"oidc.eks.${REGION}.amazonaws.com/id/${OIDC}:aud": "sts.amazonaws.com",
"oidc.eks.${REGION}.amazonaws.com/id/${OIDC}:sub":
"system:serviceaccount:kube-system:ebs-csi-controller-sa"
}
}
}
]
}
```

EOF

```
echo "${EBS_DRIVER_POLICY}" > aws-ebs-csi-driver-trust-policy.json
aws iam create-role \
--role-name AmazonEKS_EBS_CSI_DriverRole \
--assume-role-policy-document file://aws-ebs-csi-driver-trust-policy.json
aws iam attach-role-policy \
--policy-arn arn:aws:iam::aws:policy/service-role/AmazonEBSCSIDriverPolicy \
--role-name AmazonEKS_EBS_CSI_DriverRole
eksctl create addon --name aws-ebs-csi-driver --cluster ${CLUSTER_NAME} --region
${REGION} --service-account-role-arn arn:aws:iam::${
```

5. Install the satellite

Please follow the instructions in the Install on Kubernetes (satellite-kubernetes.md) section to install the Satellite.

Please ensure that you note down the address of the collector.

Satellite on AWS EKS using Fargate

Fargate allows us to have containers without the overhead of managing and scaling servers and clusters. AWS handles the maintenance, as well as security and health of the instances for us, which is something we would not want to spend time into.

1. Setup environment variables

```
export CLUSTER_NAME='Cluster Name'
export REGION='AWS Region'
export ACCOUNT_ID='AWS Account ID'
```

2. Cluster creation

To create a cluster using Fargate, run

```
eksctl create cluster --name ${CLUSTER_NAME} --region ${REGION} --fargate
--fargate specifies that the cluster needs to run on fargate, and initially assigns 2 fargate nodes
to us
```

It can be checked by running `kubectl get nodes`. The output would be something like this:

```
fargate-ip-192.168.1.1.<aws-region>.compute.internal
```

```
Ready
```

```
<none>
```

```
1m
```

```
v1.25
```

```
fargate-ip-192-168-1.1.<aws-region>.compute.internal
```

```
Ready
```

```
<none>
```

```
1m
```

```
v1.25
```

3. Connecting to the cluster

AWS EKS grants cluster admin permissions to the account from which the cluster is created. If

you don't need access to the cluster for other AWS Users, you can skip this section.

Access to other AWS users in the same account can be granted via 2 ways.

Adding individual access to user accounts

Giving the permissions to a user group

Adding individuals to the cluster

This command can be run to add an individual user account to the cluster's aws-auth configmap

```
eksctl create iamidentitymapping \  
--cluster ${CLUSTER_NAME} \  
--region ${REGION} \  
--arn <AWS ACCOUNT ARN FOR THE USER> \  
--group system:masters \  
--no-duplicate-arns \  
--username <AWS USERNAME FOR THE USER>
```

Giving access to an IAM User Group

We create a role `developer.assume-access.role` and attach two policies to it. The first one is `EKSFullAccess` so that it has access to all the EKS resources. The second one is `developer.assume-eks-access-role.policy` that allows assuming the role.

A detailed guide on defining the roles and policies can be found here

(<https://eng.grip.security/enabling-aws-iam-group-access-to-an-eks-cluster-using-rbac>).

Once you have followed the above guide to create the roles and attach the specific policies, you can add the role to the cluster's aws-auth config map to let the developers group access the cluster

```
eksctl create iamidentitymapping \  
--cluster ${CLUSTER_NAME} \  
--region ${REGION} \  
--arn arn:aws:iam:${ACCOUNT_ID}:role/developer.assume-access.role \  
--group system:masters \  

```

This needs to be run in order to grant access to the cluster.

One can Connect to the cluster by running just a single command

```
aws eks update-kubeconfig --name ${CLUSTER_NAME} --region ${REGION}> --role-arn  
arn:aws:iam:${ACCOUNT_ID}:role/developer.assume-acce
```

This commands updates the kubeconfig and adds the context for the cluster and sets the current context to it. The `--role` argument sets the correct role and policies so that seamless access to the cluster is granted instantly.

4. Install the satellite

Please follow the instructions in the `Install on Kubernetes (satellite-kubernetes.md)` section to install the Satellite.

Please ensure that you note down the address of the collector.

Satellite on Kubernetes

Prerequisites

Kubernetes version \geq v1.18.0

Helm v3 (<https://helm.sh/docs/intro/install/>) installed and working.

The Kubernetes cluster API endpoint should be reachable from the machine you are running Helm.

kubectl access to the cluster, with cluster-admin permissions.

At least 4 CPUs

At least 8 GB RAM

1. Setup environment variables

```
export LEVOAI_AUTH_KEY=<'Authorization Key'>
```

2. Install levoai Helm repo

```
helm repo add levoai https://charts.levo.ai && helm repo update
```

3. Create levoai namespace & install Satellite

If locating Satellite on the same cluster alongside Sensor

```
helm upgrade --install -n levoai --create-namespace \
```

```
--set global.levoai_config_override.onprem-api.refresh-token=$LEVOAI_AUTH_KEY \
```

```
levoai-satellite levoai/levoai-satellite
```

Depending on the region you are installing in, you may need to set a different Levo base URL for the satellite.

For example, if the satellite will be used with app.india-1.levo.ai, the installation command will be:

```
helm upgrade --install -n levoai --create-namespace \
```

```
--set global.levoai_config_override.onprem-api.refresh-token=$LEVOAI_AUTH_KEY \
```

```
--set global.levoai_config_override.onprem-api.url="https://api.india-1.levo.ai" \
```

```
levoai-satellite levoai/levoai-satellite
```

If locating Satellite on a dedicated cluster

You will need to expose the Satellite via either a LoadBalancer or NodePort, such that it is reachable by Sensors running in other clusters. Please modify the below command appropriately.

Please modify this command template and choose either 'LoadBalancer' or 'NodePort', prior to execution

```
helm upgrade --install -n levoai --create-namespace \
```

```
--set global.levoai_config_override.onprem-api.refresh-token=$LEVOAI_AUTH_KEY \
```

```
--set levoai-collector.service.type=<LoadBalancer | NodePort> \
```

```
--set global.levoai_config_override.onprem-api.url="https://api.india-1.levo.ai" \
```

```
levoai-satellite levoai/levoai-satellite
```

4. Verify connectivity with Levo.ai

a. Check Satellite health

The Satellite is comprised of five sub components 1) levoai-collector, 2) levoai-ion, 3) levoai-rabbitmq, 4) levoai-satellite, and 5) levoai-tagger.

Wait couple of minutes after the install, and check the health of the components by executing the following:

```
kubectl -n levoai get pods
```

If the Satellite is healthy, you should see output similar to below.

```
NAME
READY
STATUS
RESTARTS
AGE
levoai-collector-5b54df8dd6-55hq9
1/1
Running
0
5m0s
levoai-ion-669c9c4fbc-vsmmr
1/1
Running
0
5m0s
levoai-rabbitmq-0
1/1
Running
0
5m0s
levoai-satellite-8688b67c65-xppbn
1/1
Running
0
5m0s
levoai-tagger-7bbf565b47-b572w
1/1
Running
0
5m0s
```

b. Check connectivity

Execute the following to check for connectivity health:

Please specify the actual pod name for levoai-tagger below

```
kubectl -n levoai logs <levoai-tagger pod name> | grep "Ready to process; waiting for messages."
```

If connectivity is healthy, you will see output similar to below.

```
{"level": "info", "time": "2022-06-07 08:07:22,439", "line": "rabbitmq_client.py:155", "version": "fc628b50354bf94e544eef46751d44945a
```

Please contact support@levo.ai if you notice health/connectivity related errors.

5. Note down Host:Port information

If locating Satellite on the same cluster alongside Sensor

The Collector can now be reached by the Sensors running in the same cluster at `levoai-collector.levoai:4317`. Please note this, as it will be required to configure the Sensor.

If locating Satellite on a dedicated cluster

Run the below command and note the external address/port of the the Collector service. This will be required to configure the Sensor.

```
kubectl get service levoai-collector -n levoai
```

Please proceed to install Traffic Capture Sensors (`/install-traffic-capture-sensors`).

Satellite Lifecycle Management

Upgrade Satellite

Setup environment variables

```
export LEVOAI_AUTH_KEY=<'Authorization Key' from the original installation>
```

Update helm repo and upgrade installation

```
helm repo update
```

```
helm upgrade -n levoai \
```

```
--set global.levoai_config_override.onprem-api.refresh-token=$LEVOAI_AUTH_KEY \
```

```
levoai-satellite levoai/levoai-satellite
```

Uninstall Satellite

```
helm uninstall levoai-satellite -n levoai
```

After running the above command, wait until all Satellite pods have been terminated, and then run the following command to delete the rabbitmq PersistentVolumeClaim. Deleting the PVC also deletes the corresponding PersistentVolume.

```
kubectl delete pvc data-levoai-rabbitmq-0 -n levoai
```

In case the `kubectl delete pvc` command gets stuck, run the following command before deleting the PVC again:

```
kubectl patch pvc data-levoai-rabbitmq-0 -p '{"metadata":{"finalizers":null}}' -n levoai
```

Change the Authorization Key used to communicate with Levo.ai

Uninstall the Satellite.

Reinstall the Satellite with the new Authorization Key.

Change the minimum number of URLs that the satellite needs to observe to detect an API endpoint.

To detect an API endpoint, Satellite waits for at least '10' URLs to match that endpoint URL pattern. This number may cause delays in detecting API endpoints when there is not enough load.

If you want to change this number to suit your environment:

Export an environment variable LEVOAI_MIN_URLS_PER_PATTERN, and
Restart the Satellite with 'min_urls_required_per_pattern' helm config override option
For example, to set this to 3:

```
Setup environment variables
export LEVOAI_AUTH_KEY=<'Authorization Key' from the original installation>
export LEVOAI_MIN_URLS_PER_PATTERN=3
Update helm repo and upgrade installation
helm repo update
helm upgrade -n levoai \
--set global.levoai_config_override.onprem-api.refresh-token=$LEVOAI_AUTH_KEY \
--set
global.levoai_config_override.min_urls_required_per_pattern=$LEVOAI_MIN_URLS_PER_PAT
TERN \
levoai-satellite levoai/levoai-satellite
List Satellite's pods
kubectl -n levoai get pods | grep -E
'^levoai-collector|^levoai-rabbitmq|^levoai-satellite|^levoai-tagger'
Tail logs of a specific pod
kubectl -n levoai logs -f <pod name>
Troubleshooting
```

Tagger Errors

The Tagger component sends API endpoint metadata information to Levo.ai. API Observability will not function if the Tagger is in an errored state.

Please see sample output below from kubectl get pods, that shows the Tagger in an errored state.

```
NAME
READY
STATUS
RESTARTS
AGE
levoai-collector-848fb4fff9-gv8g9
1/1
Running
0
64s
levoai-rabbitmq-0
0/1
Running
0
64s
levoai-satellite-54956ccb89-5s4h2
1/1
```


Running

0

64s

levoai-tagger-799db4d9cc-89jm8

0/1

Error

1 (14s ago)

64s

Below are common error scenarios:

Authentication Errors

The Tagger component authenticates with Levo.ai using the Authorization Key. If Tagger is unable to authenticate, it will error out.

Check for authentication errors in the Tagger logs:

```
kubectl -n levoai logs <levoai-tagger-pod-id> | grep "Exception: Failed to refresh access token"
```

If there are exception messages, you have an incorrect or stale Authorization Key. Please contact support@levo.ai for further assistance.

Connectivity Errors

Check for connectivity errors in the Tagger logs:

```
kubectl -n levoai logs <levoai-tagger-pod-id> | grep "ConnectionRefusedError: [Errno 111] Connection refused"
```

If there are exception messages, Tagger is unable to connect to dependent services. It generally establishes connection after 3/4 retries. Please contact support@levo.ai for further assistance.

API Traffic Capture Filters

The Sensor allows capturing API (HTTP) traffic based on filter (include/exclude) criteria. These filters are specified in a configuration file. Please refer to Sensor Configuration (sensor-configuration.mdx) for high level structure of the file.

The sensor's fundamental unit at which filtering criteria are applied is a Linux Process.

The sensor can filter traffic based on Linux Process names, Linux Process IDs, IP address tuples associated with a Linux Process's TCP/UDP traffic, Kubernetes Pod metadata associated with the Linux Process (when running on Kubernetes), and HTTP URL/header information in traffic being processed by the Linux Process.

Below diagram shows the outcomes and precedence, when the various filtering criteria mentioned above are combined together.

API Filter Precedence

Below are details on the supported filtering criteria.

Default Excluded Ports

Configure IP Filters

IP Filter Examples

Exclude All Traffic

Exclude Specific Ports/Port Ranges
Include Specific Ports/Port Ranges
Exclude Specific IP Addresses
Include Specific IP Addresses
Exclude IP Subnets
Include IP Subnets
Capture Traffic for Specific Processes
Configure Kubernetes Pod Filters
K8s Pod Filter Examples
Trace A Single Deployment In A Specific Namespace
Ignore All Traffic Belonging To A Specific Namespace
Trace Multiple Deployments In A Specific Namespace
Trace A Specific Deployment In A Namespace, And Trace All Other Namespaces
Trace All Deployments and Statefulsets From Any Namespace
Ignore Traffic From K8s System Level Services
Configure URL Filters
URL Filter Examples
Ignore All .js API Endpoints
Ignore API Base Path /static/
Ignore API Endpoints With Query Parameter timeout
Only Trace API Endpoints Containing /users/
Only Trace GET/POST API Endpoints
Only Trace payments.com:8888 APIs
Only Trace payments.com:8888/credit/ APIs Doing GET
Trace APIs on All Subdomains of api.acme.com
Default Excluded Ports
The Sensor excludes capturing traffic from the below ports (TCP & UDP) by default.
If your API Traffic (HTTP) uses one of these ports, please see section below on how the port can be included for capture.

Standard Protocol Port

DNS

53

etcd

2379-2380

Kafka

9092-9093

mongodb

27017-27019, 28017

SQL Server

135, 4022, 1433-1434

MySQL

3306, 33060-33062

Postgress
5432-5433
RabbitMQ
5671-5672, 15672-15675, 25672, 35672-35682, 61613-61614
Redis
6379
ZooKeeper
2181, 3888, 3888
Configure IP Filters

The below sections describe common filtering scenarios with examples. In all cases the examples show the relevant snippet of the configuration file. Adapt these examples to the Helm Values config file (`../static/artifacts/sensor/config-values.yml`), if running on Kubernetes.

IP Filter Examples

Exclude All Traffic

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: drop Default policy is to drop all traffic

Exclude Specific Ports/Port Ranges

Exclude Specific Host Ports/Port Ranges

Host Port is the server listening port, where client/peer connections are accepted.

----- IP Filters:
IP/Port/Network address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: accept Default policy is to capture all traffic

entries: Specific 'entries' can override the default policy

Host Ports

Host Port is the server listening port, where client connections are accepted

- policy: drop

host-ports: 53 DNS

- policy: drop

host-ports: 2379-2380 etcd

Exclude Specific Peer Ports/Port Ranges

Peer Port is a client port used in communication with the server listening port.

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: accept Default policy is to capture all traffic

entries: Specific 'entries' can override the default policy

Peer Ports

Peer Port is the client port used in communication with the server listening port

- policy: drop

peer-ports: 9000

- policy: drop

peer-ports: 25000-29000

Include Specific Ports/Port Ranges

Include Specific Host Ports/Port Ranges

Host Port is the server listening port, where client/peer connections are accepted.

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: drop Drop all traffic except ones below

entries: Specific 'entries' can override the default policy

Host Ports

Host Port is the server listening port, where client connections are accepted

- policy: accept

host-ports: 9000

- policy: accept

host-ports: 23000-28000

Include Specific Peer Ports/Port Ranges

Peer Port is a client port used in communication with the server listening port.

----- IP Filters:
IP/Port/Network address based granular filtering of API traffic.

----- IP Filters enable granular

capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: drop Drop all traffic except ones below

entries: Specific 'entries' can override the default policy

Peer Ports

Peer Port is the client port used in communication with the server listening port

- policy: accept

peer-ports: 9000

- policy: accept

peer-ports: 25000-29000

Exclude Specific IP Addresses

Exclude Specific HOST IP Addresses

Host implies the binding IP addresses of the Server servicing the API endpoints.

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: accept Accept all traffic except ones below

entries: Specific 'entries' can override the default policy

- host-network: 10.98.76.53 Drop all traffic to/from 10.98.76.53

policy: drop

- host-network: 10.99.76.53 Drop all traffic to/from 10.99.76.53

policy: drop

Exclude Specific Peer IP Addresses

Peer implies the IP addresses of clients connecting to the Server/Host servicing the API endpoints.

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: accept Accept all traffic except ones below

entries: Specific 'entries' can override the default policy

- peer-network: 10.98.76.53 Drop all traffic to/from 10.98.76.53

policy: drop

- peer-network: 10.99.76.53 Drop all traffic to/from 10.99.76.53

policy: drop

Include Specific IP Addresses

Include Specific HOST IP Addresses

Host implies the binding IP addresses of the Server servicing the API endpoints.

----- IP Filters:
IP/Port/Network address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: drop Drop all traffic except ones below

entries: Specific 'entries' can override the default policy

- host-network: 10.98.76.53 Accept all traffic to/from 10.98.76.53

policy: accept

- host-network: 10.99.76.53 Accept all traffic to/from 10.99.76.53

policy: accept

Exclude Specific Peer IP Addresses

Peer implies the IP addresses of clients connecting to the Server/Host servicing the API endpoints.

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: drop Drop all traffic except ones below

entries: Specific 'entries' can override the default policy

- peer-network: 10.98.76.53 Drop all traffic to/from 10.98.76.53

policy: accept

- peer-network: 10.99.76.53 Drop all traffic to/from 10.99.76.53

policy: accept

Exclude IP Subnets

Entire classes of subnets can be excluded using either the CIDR or prefix notations. The examples below are for host-network subnets. The same technique is applicable for peer-network subnets.

Exclude IP Subnets - CIDR Notation

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: accept Accept all traffic except ones below

entries: Specific 'entries' can override the default policy

- host-network: 1.2.3.4/255.255.255.252 Drop all from/to this host CIDR block

policy: drop

Exclude IP Subnets - Prefix Notation

----- IP Filters:
IP/Port/Network address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: accept Accept all traffic except ones below

entries: Specific 'entries' can override the default policy

- host-network: 1.2.3.4/30 Drop all from/to this host subnet block

policy: drop

Include IP Subnets

Entire classes of subnets can be included using either the CIDR or prefix notations. The examples below are for host-network subnets. The same technique is applicable for peer-network subnets.

Include IP Subnets - CIDR Notation

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: drop Drop all traffic except ones below

entries: Specific 'entries' can override the default policy

- host-network: 1.2.3.4/255.255.255.252 Accept all to/from this host CIDR block

policy: accept

Include IP Subnets - Prefix Notation

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: drop Drop all traffic except ones below

entries: Specific 'entries' can override the default policy

- host-network: 1.2.3.4/30 Accept all to/from this host subnet block

policy: accept

Capture Traffic for Specific Processes

Traffic can be captured ONLY for specific processes, by specifying either their command name or PID.

Capture Traffic by Command Name

----- Process Filters: process command names/IDs to monitor & capture API traffic.

----- Uncomment and modify appropriately to limit capture to specific process names or IDs.

Both monitored-commands and monitored-pids support list of names & IDs respectively.

NOTE: monitored-commands and monitored-pids settings are mutually exclusive

Capture traffic from/to nginx & python3

monitored-commands:

- nginx

- python3

----- IP Filters: IP/Port/Network address based granular filtering of API traffic.

----- IP Filters enable granular capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: accept Default policy is to capture all traffic

Capture Traffic by PID

----- Process Filters: process command names/IDs to monitor & capture API traffic.

----- Uncomment and modify appropriately to limit capture to specific process names or IDs.

Both monitored-commands and monitored-pids support list of names & IDs respectively.

NOTE: monitored-commands and monitored-pids settings are mutually exclusive

Capture traffic from/to PIDs 123 & 45.

monitored-pids:

- 123

- 45

----- IP Filters: IP/Port/Network
address based granular filtering of API traffic.

----- IP Filters enable granular
capture of API traffic based on various criteria.

Default values ignore traffic from standard ports that normally do not carry HTTP traffic.

Refer to documentation on how these can be customized to suit your environment.

ip-filter-list:

default-policy: accept Default policy is to capture all traffic

Configure Kubernetes Pod Filters

The sensor allows filtering of API traffic based on the Kubernetes Pod's metadata. Examples are including/excluding API traffic from a Pod belonging to specific namespaces, deployments, statefulsets, etc.

K8s Pod Filter Configuration Section

K8s Pod filters are configured in the k8s-pod-filter-list section of the Helm Values config file (`../static/artifacts/sensor/config-values.yml`) as shown below.

sensor:

config:

----- Kubernetes Pod Filters:
Kubernetes pod properties based granular filtering of API traffic.

----- Pod Filters enable
granular capture of API traffic based on Kubernetes Pod attributes.

Rules should ideally be in decreasing order of specificity.

The first rule to match a pod's properties will be used.

k8s-pod-filter-list:

default-policy: <trace|ignore>

rules:

- policy: <trace|ignore>

namespace: <name or regex pattern>

Optional owner reference of the Pod

owner-reference:

kind: <Node|Deployment>

name: <name or regex pattern>

----- K8s Pod Filter Examples

Below are common filtering scenarios with examples. In all cases the examples show the relevant snippet of the configuration file. Adapt these examples to the Helm Values config file (`../static/artifacts/sensor/config-values.yml`).

Trace A Single Deployment In A Specific Namespace

Below example will only trace all API traffic belonging to entity-service Deployment in the levoai

Namespace, and ignore all other API traffic in the K8s cluster.

```
k8s-pod-filter-list:  
default-policy: ignore  
rules:  
- policy: trace  
namespace: levoai  
owner-reference:  
kind: Deployment  
name: entity-service
```

Ignore All Traffic Belonging To A Specific Namespace

Below example will trace all API traffic in a K8s cluster, except traffic belonging to Pods in the levoai namespace.

```
k8s-pod-filter-list:  
default-policy: trace  
rules:  
- policy: ignore  
namespace: levoai
```

Trace Multiple Deployments In A Specific Namespace

Below example will only trace all API traffic belonging to entity-service, and onboarding-service Deployments in the levoai Namespace, and ignore all other API traffic in the K8s cluster.

```
k8s-pod-filter-list:  
default-policy: ignore  
rules:  
- policy: trace  
namespace: levoai  
owner-reference:  
kind: Deployment  
name: (entity|onboarding)-service
```

Trace A Specific Deployment In A Namespace, And Trace All Other Namespaces

The below example does the following:

Traces all API traffic in all namespaces, except traffic within the crapi namespace

Trace traffic belonging to crapi-identity Deployment in the crapi namespace

```
k8s-pod-filter-list:  
default-policy: trace  
rules:  
- policy: trace  
namespace: crapi  
owner-reference:  
kind: Deployment
```

name: crapi-identity

- policy: ignore

namespace: crapi

Trace All Deployments and Statefulsets From Any Namespace

k8s-pod-filter-list:

default-policy: ignore

rules:

- policy: trace

owner-reference:

kind: Deployment

namespace: .*

- policy: trace

owner-reference:

kind: StatefulSet

name: .*

Ignore Traffic From K8s System Level Services

Below example ignores all API traffic from kube, istio and levoai namespaces. It also ignores pods created by the K8s Nodes. All other traffic is traced.

k8s-pod-filter-list:

default-policy: trace

rules:

- policy: ignore

namespace: kube-.*

- policy: ignore

namespace: istio.*

- policy: ignore

namespace: levoai

- policy: ignore

namespace: .*

owner-reference:

kind: Node

name: .*

Configure URL Filters

The sensor allows filtering of API traffic based on the API endpoint Method (operation), the API Host (Host Header), and the API endpoint's URI. The endpoint's URI can be a (Perl format) (<https://perldoc.perl.org/perlre>) regex pattern.

Please ensure that any regex metacharacters (<https://perldoc.perl.org/perlreMetacharacters>) present in the URI string expression are properly escaped (for example ?, etc.). Use an online regex evaluator (<https://regex101.com/>) to test your regex pattern if necessary.

URL Filter Configuration Section

URL filters are configured in the url-filter section of the config file as shown below.

----- URL Filters: API
parameter based granular filtering of API traffic.

-----url-filter:
'default-url-action' specifies the default behavior which can be overridden by 'rules' below.
This is a mandatory attribute that needs to be specified in order to use URL filters.
default-url-action: <trace|ignore>

YAML array of one or more rules. Order of rules matters during evaluation
rules:

'action' is mandatory. At least one of 'methods', or 'request-uri', or 'host'
MUST be specified for each rule entry
- action: <trace|ignore>

YAML array list of one or more (API operations) methods: GET, POST, PUT, PATCH, DELETE
Example: [GET], or [GET, POST, PUT, DELETE]
methods: <[GET, POST, PUT, PATCH, DELETE]>

URI of the API endpoint. Can be a (Perl format) regex pattern. Example: /foo/bar, or /bar/*
request-uri: <URI>

Hostname of the API endpoint and optionally the port. Example: levo.ai:8888, or levo.ai
host: <hostname[:port]>

The default-url-action specifies the default behavior for filtering APIs, and is a mandatory attribute. Either all API endpoints are traced, or all are ignored. This default behavior can be overridden to granularly trace or ignore specific endpoints that match rules specified in the rules sub section.

The rules sub section defines the override behavior using a YAML array list. Each entry of the array list is comprised of the following parameters:

action: mandatory parameter that accepts either trace or ignore as values

At least one, and optionally all of the below additional parameters:

methods: YAML array list of one or more (API operations) methods as values: GET, POST, PUT, PATCH, DELETE

request-uri: URI of the API endpoint as the value. Can be a (Perl format) regex pattern.

Example: /foo/bar, or /bar/*

host: Hostname of the API endpoint and optionally the port as values. Can be a (Perl format) regex pattern. Example: levo.ai:8888, or levo.ai, or a regex such as *.*.levo.ai to handle all subdomains of levo.ai

Rule entries are evaluated in the order in which they were specified. Further evaluation stops, as soon as a single rule entry matches.

URL Filter Examples

Below are common filtering scenarios with examples. In all cases the examples show the relevant snippet of the configuration file. Adapt these examples to the Helm Values config file (`../../../../static/artifacts/sensor/config-values.yml`), if running on Kubernetes.

Ignore All .js API Endpoints

The below filter will ignore all API endpoints with URIs ending with .js.

```
----- URL Filters: API
parameter based granular filtering of API traffic.
-----url-filter:
default-url-action: trace
rules:
- action: ignore
request-uri: .*\.js  Regex pattern to ignore all API endpoints which end with '.js'
```

Ignore API Base Path /static/

The below filter will ignore all API endpoints that have the base path /static/.

```
----- URL Filters: API
parameter based granular filtering of API traffic.
-----url-filter:
default-url-action: trace
rules:
- action: ignore
request-uri: /static/.*
```

Ignore API Endpoints With Query Parameter timeout

The below filter will ignore all endpoints that have the query parameter timeout. For example /users/list?timeout=60.

```
----- URL Filters: API
parameter based granular filtering of API traffic.
-----url-filter:
default-url-action: trace
rules:
- action: ignore
request-uri: .*\\?.*timeout=.*  Notice '?' is escaped with '\\?'
```

Only Trace API Endpoints Containing /users/

The below filter will ONLY trace all API endpoints that have /users/ in the path.

```
----- URL Filters: API
```

parameter based granular filtering of API traffic.

-----url-filter:
default-url-action: ignore Ignore all API endpoints by default
rules:
- action: trace
request-uri: .*/users/*

Only Trace GET/POST API Endpoints

The below filter will ONLY trace all API endpoints that perform GET or POST.

----- URL Filters: API
parameter based granular filtering of API traffic.

-----url-filter:
default-url-action: ignore Ignore all API endpoints by default
rules:
- action: trace
methods: [GET, POST]

Only Trace payments.com:8888 APIs

The below filter will ONLY trace API endpoints belonging to API host payments.com:8888.

----- URL Filters: API
parameter based granular filtering of API traffic.

-----url-filter:
default-url-action: ignore Ignore all API endpoints by default
rules:
- action: trace
host: payments.com:8888

Only Trace payments.com:8888/credit/ APIs Doing GET

The below filter will ONLY trace API endpoints belonging to API host payments.com:8888, having /credit/ as the base path, and performing GET.

----- URL Filters: API
parameter based granular filtering of API traffic.

-----url-filter:
default-url-action: ignore Ignore all API endpoints by default
rules:
- action: trace
host: payments.com:8888
methods: [GET]
request-uri: /credit/*

Trace APIs on All Subdomains of api.acme.com

The below filter will ONLY trace API endpoints belonging to all subdomains of API host acme.com. For example API endpoints belonging to payments.api.acme.com, and orders.api.acme.com, will be traced, but catalog.acme.com will be ignored.

The Host header is expressed as a Perl format regular expression.

----- URL Filters: API
parameter based granular filtering of API traffic.

-----url-filter:

default-url-action: ignore Ignore all API endpoints by default

rules:

- action: trace

host: .*\.api\.acme\.com '.' has been escaped as we are using a regex

Kubernetes Configuration

Add Tolerations and Node Selectors

Tolerations and Node Selectors for the Sensor pods may be specified via helm values. For example:

sensor:

tolerations:

- key: node-role.kubernetes.io/control-plane

operator: Exists

effect: NoSchedule

- key: "devops"

operator: "Equal"

value: "dedicated"

effect: "NoSchedule"

nodeSelector:

kubernetes.io/hostname: "mavros"

Sensor via APT Package

Install on Debian based Linux via apt

1. Install curl and gnupg

```
sudo apt install gnupg
```

```
sudo apt install curl
```

2. Configure Linux host to access Levo apt repo

```
curl -fsSL https://us-apt.pkg.dev/doc/repo-signing-key.gpg | sudo gpg --dearmor -o
```

```
/usr/share/keyrings/us-apt-repo-signing-key.gpg
```

```
echo \
```

```
"deb [arch=$(dpkg --print-architecture)
```

```
signed-by=/usr/share/keyrings/us-apt-repo-signing-key.gpg] \
```

```
https://us-apt.pkg.dev/projects/levoai apt-levo main" \
```

```
| sudo tee -a /etc/apt/sources.list.d/artifact-registry.list > /dev/null
```

sudo apt update

3. Download/install Sensor artifacts

sudo apt install levo-ebpf-sensor=0.42.1

4. Start the Sensor

Please take a look at the Running the Sensor as a Systemd Service

(/install-traffic-capture-sensors/ebpf-sensor/sensor-systemd-service) section for further instructions.

Sensor Lifecycle Management

Configure Satellite Address (host:port information)

The Satellite address is configured in /etc/levo/sensor/config.yaml. The default host:port for Satellite is localhost:4317.

Edit /etc/levo/sensor/config.yaml, and set satellite-url (under Satellite Settings) to the desired host:port value.

...

----- Satellite Settings:
----- host:port for the collector

service receiving the sensor's API traces.

mention the scheme http/https if you decide not to use gRPC for sensor satellite communication

satellite-url: <set to desired host:port value>

-----...

Configure sensor environment

The eBPF sensor environment is configured in /etc/default/levo-ebpf-sensor. The default env value is staging

Edit /etc/default/levo-ebpf-sensor, and set LEVO_ENV to the desired env value (eg. prod, qa)

Environment Variables for levo-ebpf-sensor.service

MALLOC_CONF="background_thread:true,narenas:1,tcache:false,dirtty_decay_ms:0,muzzy_decay_ms:0,abort_conf:true"

LEVO_ENV="staging"

A Sensor restart is required for this to take effect.

Start Sensor

Note: The default config file is located at: '/etc/levo/sensor/config.yaml'

sudo systemctl start levo-ebpf-sensor

Get Sensor Status

sudo systemctl status levo-ebpf-sensor

Stop Sensor

sudo systemctl stop levo-ebpf-sensor

Check Sensor Logs

journalctl -u levo-ebpf-sensor.service -b -f --since "15min ago"

If journalctl isn't providing logs, you can alternatively:

```
sudo cat syslog | grep 'levo-ebpf-sensor'
```

Show Sensor Config

```
cat /etc/levo/sensor/config.yaml
```

Uninstall Sensor

```
sudo apt remove --purge levo-ebpf-sensor
```

```
sudo apt clean
```

Manage Sensor Configuration

Please refer to Sensor Configuration

(/install-traffic-capture-sensors/common-tasks/sensor-configuration.mdx), and Applying Configuration Changes

(/install-traffic-capturesensors/common-tasks/sensor-configuration.mdxrunning-on-linux-host).

Sensor via Docker

Install on Linux host via Docker

Prerequisites

Docker Engine version 18.03.0 and above

Admin (or sudo) privileges on the Docker host

1. Install Sensor

If you are installing the Satellite and Sensor on the same Linux host, please do NOT use localhost as the satellite-address below. Use host.docker.internal, or the Linux host's IP address or domain name instead. This is required as the Sensor runs inside a Docker container, and localhost resolves to the Sensor container's IP address, instead of the Linux host.

Replace '<satellite-address>' with the values you noted down from the Satellite install

Specify below the 'Application Name' chosen earlier. Do not quote the 'Application Name'

```
sudo docker run --restart unless-stopped \  
-v /sys/kernel/debug:/sys/kernel/debug -v /proc:/host/proc \  
--add-host host.docker.internal:host-gateway \  
--privileged --detach levoai/ebpf_sensor:0.40.0 \  
--host-proc-path /host/proc/ \  
--satellite-url <satellite-address> \  
--env <'application-environment'> \  
--default-service-name <'Application Name' chosen earlier>
```

NOTE:

The default address for the collector in Docker-based Sensor installations is <https://collector.levo.ai>. This address assumes that Levo is hosting the Satellite for you, and you must also specify an organization ID when starting the sensor (with the --organization-id flag). If you wish, you may also host the Satellite yourself and specify the address

of the collector in the self-hosted Satellite to direct the Sensor's traffic to it.

2. Verify connectivity with Satellite

Execute the following command to check for connectivity health:

Please specify the actual container name for levoai-sensor below

```
docker logs <levoai-sensor container name> | grep "Initial connection with Collector"
```

If connectivity is healthy, you should see output similar to below.

```
2022/06/13 21:15:40 729071
```

```
INFO [ebpf_sensor.cpp->main:120]
```

```
Initial connection with Collector was successful.
```

Please proceed to the next step, if there are no errors.

Sensor Lifecycle Management

Uninstall Sensor

Get the container id of the Sensor

```
docker ps | grep "levoai/ebpf_sensor"
```

Remove the Sensor

```
docker rm -f <container id from docker ps step above>
```

Get Sensor Logs

Get the container id of the Sensor

```
docker ps | grep "levoai/ebpf_sensor"
```

```
sudo docker logs <container id from docker ps step above>
```

Upgrade Sensor

Uninstall Sensor

Pull new Sensor image

```
docker pull levoai/ebpf_sensor:latest
```

Reinstall Sensor

Manage Sensor Configuration

Please refer to Sensor Configuration

(/install-traffic-capture-sensors/common-tasks/sensor-configuration.mdx), and Applying Configuration Changes

(/install-traffic-capturesensors/common-tasks/sensor-configuration.mdxrunning-via-docker).

Sensor on Kubernetes

Install on Kubernetes

Prerequisites

Kubernetes version \geq v1.18.0

Helm v3 (<https://helm.sh/docs/intro/install/>) installed and working.

The Kubernetes cluster API endpoint should be reachable from the machine you are running Helm.

kubectl access to the cluster, with cluster-admin permissions.

1. Install levoai Helm repo

```
helm repo add levoai https://charts.levo.ai && helm repo update
```

2. Create levoai namespace & install Sensor

Replace 'hostname|IP' & 'port' with the values you noted down from the Satellite install

If Sensor is installed on same cluster as Satellite, use 'levoai-haproxy'

If they are installed on different clusters, the haproxy service should be exposed so that it is reachable by the sensor. Use the exposed address as the value for satellite-url.

Specify below the 'Application Name' chosen earlier and Organization ID (copy from levo platform).

```
helm upgrade levoai-sensor levoai/levoai-ebpf-sensor \
--install \
--namespace levoai \
--create-namespace \
--set sensor.config.default-service-name=<'Application Name' chosen earlier> \
--set sensor.config.satellite-url=<hostname|IP:port> \
--set sensor.config.organization-id=<your-org-id> \
--set sensor.levoEnv=<'Application environment'>
```

3. Verify connectivity with Satellite

i. Check Sensor health

Check the health of the Sensor by executing the following:

```
kubectl -n levoai get pods | grep levoai-sensor
```

If the Sensor is healthy, you should see output similar to below.

```
levoai-sensor-747fb4aaa9-gv8g9
1/1
Running
0
1m8s
```

ii. Check connectivity

Execute the following command to check for connectivity health:

Please specify the actual pod name for levoai-sensor below

```
kubectl -n levoai logs <levoai-sensor pod name> | grep "Initial connection with Collector"
```

If connectivity is healthy, you should see output similar to below.

```
2022/06/13 21:15:40 729071
```

```
INFO [ebpf_sensor.cpp->main:120]
```

```
Initial connection with Collector was successful.
```

Please contact support@levo.ai if you notice health/connectivity related errors.

NOTE:

The default address for the satellite url in helm installations is levoai-haproxy. This address assumes that the Satellite is installed in the same cluster (and namespace) as the Sensor. If they are installed on different clusters, the haproxy service should be exposed so that it is reachable by the sensor. Use the exposed address as the value for satellite-url. If you wish to, you may also request Levo to host the Satellite for you. In this case, you will need to set the satellite-url to https://collector.levo.ai and specify an organization ID (organization-id) via helm values.

helm upgrade --set sensor.levoEnv=<your-application-environment> --set sensor.config.satellite-url=https://collector.levo.ai --set se
Please proceed to the next step, if there are no errors.

Sensor as a Systemd Service
Running the Sensor as a Systemd Service

1. Configure Satellite Address

The Satellite (collector) address is configured in /etc/levo/sensor/config.yaml.

NOTE:

The default address for the collector in Systemd installations is https://collector.levo.ai. This address assumes that Levo is hosting the Satellite for you, and you must also specify an organization ID (organization-id) via the config file. If you wish, you may also host the Satellite yourself and specify the address of the collector in the self-hosted Satellite to direct the Sensor's traffic to it.

Edit /etc/levo/sensor/config.yaml, and set satellite-url (under Satellite Settings) to the address noted from the Satellite install.

...

----- Satellite Settings:

Levo Organization ID. This must be specified when the collector is hosted by Levo.

organization-id: ""

host:port for the collector service receiving the Sensor's API traces.

satellite-url: <Use the default (https://collector.levo.ai) or set to a custom address>

...

Note: If you change the Satellite address later, you have to restart the Sensor, since it's not a hot property.

2. Configure Application Name

The Application Name is configured in /etc/levo/sensor/config.yaml.

Edit /etc/levo/sensor/config.yaml, and set default-service-name to the Application Name chosen

earlier.

----- Default Application

Name:

Auto discovered API endpoints and their OpenAPI specifications are show in the API Catalog grouped under this application name. The application name helps segregate and group API endpoints from different environments.

default-service-name: <'Application Name' chosen earlier>

Configure sensor environment

The eBPF sensor environment is configured in /etc/default/levo-ebpf-sensor. The default env value is staging

Edit /etc/default/levo-ebpf-sensor, and set LEVO_ENV to the desired env value (eg. prod, qa)

Environment Variables for levo-ebpf-sensor.service

```
MALLOC_CONF="background_thread:true,narenas:1,tcache:false,dirty_decay_ms:0,muzzy_decay_ms:0,abort_conf:true"
```

```
LEVO_ENV="staging"
```

Note: If you change the Application Name later, you have to restart the Sensor, since it's not a hot property.

3. Start the Sensor

```
sudo systemctl start levo-ebpf-sensor
```

4. Verify connectivity with Satellite

```
sudo journalctl -u levo-ebpf-sensor.service -b -f
```

If 'journalctl' isn't tailing logs, use syslog:

```
sudo cat /var/log/syslog | grep 'levo-ebpf-sensor'
```

Connection Success

If connectivity is healthy, you should see output similar to below.

```
2022/06/13 21:15:40 729071
```

```
INFO [ebpf_sensor.cpp->main:120]
```

```
Initial connection with Collector was successful.
```

Connection Failures

If the Sensor is unable to connect with the Satellite, you will notice log entries similar to the one below. Please contact support@levo.ai for assistance.

Initial connection with Collector failed. However, the sensor will keep attempting to send future traces.

```
[OTLP TRACE GRPC Exporter] Export() failed: failed to connect to all addresses
```

Please proceed to the next step, if there are no errors.

5. Sensor's resource limits

By default, sensor is restricted to use up to 50% of CPU and 2GB memory.

If you ever need to change these limits, you need to modify CPUQuota and MemoryMax in the below systemd config file under [Service] section:

1. Open the config file `/usr/lib/systemd/system/levo-ebpf-sensor.service` and modify CPUQuota and MemoryMax

```
sudo vi /usr/lib/systemd/system/levo-ebpf-sensor.service
```

For example,

If you want to limit sensor's CPU usage to 0.75 of a core, then set CPUQuota=75%. You can set CPUQuota=200% to go upto two full cores of CPU.

If you want to limit sensor's memory usage to 1GB, then set MemoryMax=1G

2. Reload the config

```
systemctl daemon-reload
```

3. Restart the sensor

```
sudo systemctl restart levo-ebpf-sensor
```

Sensor via YUM Package

Install on RPM based Linux Distributions via yum

1. Configure the package manager

Configure yum to access Levo's RPM packages using the following command:

```
sudo tee -a /etc/yum.repos.d/levo.repo << EOF
```

```
[levo]
```

```
name=Levo.ai
```

```
baseurl=https://us-yum.pkg.dev/projects/levoai/yum-levo
```

```
enabled=1
```

```
repo_gpgcheck=0
```

```
gpgcheck=0
```

```
EOF
```

2. Install the eBPF Sensor

Install the eBPF Sensor from Levo's RPM repository.

1. Update the list of available packages:

```
sudo yum makecache
```

1. Install the package in your repository.

```
sudo yum install levo-ebpf-sensor-0.42.1
```

Enter y when prompted.

3. Start the Sensor

Please take a look at the Running the Sensor as a Systemd Service

(/install-traffic-capture-sensors/ebpf-sensor/sensor-systemd-service) section for further

instructions.

Install PCAP Sensor

Prerequisites

Refer `pcap-filter-guide` (<https://www.tcpdump.org/manpages/pcap-filter.7.html>) to apply filters.

NOTE: You need to have the satellite installed to configure the sensor to point to it. If you haven't done it already, head over to `Install Satellite (/install-satellite)` Make sure the satellite is able to listen on port 9999 Edit Inbound Rules to accept port 9999 in case the satellite is running on an AWS instance.

Follow instructions for your platform

Install on Fargate (`/install-traffic-capture-sensors/pcap-sensor/sensor-fargate`)

Install via Docker (`/install-traffic-capture-sensors/pcap-sensor/sensor-docker`)

Install on Kubernetes (`/install-traffic-capture-sensors/pcap-sensor/sensor-kubernetes`)

Sensor via Docker

Install via Docker

Prerequisites

Docker Engine version 18.03.0 and above

Admin (or sudo) privileges on the Docker host

```
sudo docker run --net=host --rm -it levoai/pcap-sensor:0.1.1 \
```

```
./bin/init apidump \
```

```
--satellite-url "your satellite url (http(s)://hostname|IP:port)" \
```

```
--levo-env "your application environment (staging, production etc.)" \
```

```
--levoai-org-id "your levo org id"
```

Specify additional flags in the command

```
--trace-export-interval "trace export interval in seconds (default 10)"
```

```
--rate-limit "number of traces per minute"
```

```
--filter "pcap filter string, eg. port 8080 and (not port 8081)"
```

```
--host-allow "host allow regex"
```

```
--path-allow "path allow regex"
```

```
--host-exclusions "host exclude regex"
```

```
--path-exclusions "path exclude regex"
```

Sensor on Fargate

Prerequisites

AWS profile access key and secret access key saved at path `~/.aws/credentials` file

The profile should have all the required permissions as listed here

Install Sensor on Fargate

The pcap Sensor can be installed as a sidecar on an existing AWS task by adding to its task definition via the AWS Console.

The steps to add the sensor to your task are as follows

Go to Task Definitions

Select the required task definition

Click on Create revision with JSON

Add the given JSON object under ContainerDefinitions

Replace the values for satellite-url, levo-env and levoai-org-id in entrypoint.

Replace the values for Environment and LogConfiguration as per your requirement.

Set the cpu limit as number of CPU Units (Note: 1 core = 1024 CPU Units)

Set the memory limit in Mib (Note: memory should not exceed the Task memory limit)

```
{
"name": "levo-pcap-sensor",
"image": "levoai/pcap-sensor:0.1.1",
"cpu": 512,
"memory": 512,
"portMappings": [],
"essential": false,
"entryPoint": [
"/bin/init",
"apidump",
"--satellite-url",
"< INSERT SATELLITE URL (http(s)://hostname|IP:port) >",
"--levo-env",
"<INSERT APPLICATION ENVIRONMENT (staging, production etc.)>",
"--levoai-org-id",
"< INSERT LEVO ORG ID >",
"--rate-limit",
"<INSERT NUMBER OF TRACES PER MINUTE>"
],
"environment": [
{
"name": "LEVO_AWS_REGION",
"value": "< INSERT AWS REGION (us-west-2) >"
}
],
"mountPoints": [],
"volumesFrom": [],
"logConfiguration": {
"logDriver": "awslogs",
"options": {
"awslogs-group": "< INSERT LOGS IDENTIFIER (/ecs/your-application-pcap) >",
"awslogs-create-group": "true",
"awslogs-region": "< INSERT AWS REGION (us-west-2) >",
"awslogs-stream-prefix": "ecs-pcap"
}
}
}
```



```
}  
}
```

Specify additional flags in the entrypoint

--trace-export-interval

default 10s

--rate-limit

number of traces per minute

--filter

eg. port 8080 and (not port 8081)

--host-allow

regex for allowed hosts

--path-allow

regex for allowed paths

--host-exclusions

regex for excluded hosts

--path-exclusions

regex for excluded paths

AWS Permissions needed

Add the AmazonECS_FullAccess policy to get access to all the necessary permissions.

Action

Resource

ec2:DescribeRegions

*

ecs:ListClusters

*

, or restricted to account like

ecs:DescribeClusters

arn:aws:ecs:::cluster/*

ecs:ListTaskDefinitionFamilies *

ecs:DescribeTaskDefinition

*

ecs:RegisterTaskDefinition

*

ecs:ListServices

*

*, or restricted to your account, or restricted

ecs:DescribeServices

to the cluster you selected

*, or restricted to your account, or restricted

ecs:UpdateService

to the cluster you selected

Purpose

Find the list of AWS regions you have enabled. (If not present, defaults to a precompiled list.)
Find the available ECS clusters.
Look up the names of the available ECS clusters.
Find the available task definitions.
Read the existing task definition in order to copy it.
Write a new version of the task definition.
Find the available services.
Identify which services are using the task definition you selected.
Update and restart the service using the new task definition.

Action

Resource

Purpose

*, or restricted to your account, or restricted

Mark the service as having been updated by Levoai.
to the cluster you selected

ecs:TagResource

Sensor on Kubernetes

Install on Kubernetes as daemonset

Prerequisites

Kubernetes version \geq v1.18.0

Helm v3 (<https://helm.sh/docs/intro/install/>) installed and working.

The Kubernetes cluster API endpoint should be reachable from the machine you are running Helm.

kubectl access to the cluster, with cluster-admin permissions.

1. Install levoai helm repo

```
helm repo add levoai https://charts.levo.ai && helm repo update
```

2. Create levoai namespace and install pcap-sensor

Replace 'hostname|IP' & 'port' with the values you noted down from the Satellite install

If Sensor is installed on same cluster as Satellite, use 'http://levoai-satellite:9999'

Specify below the 'Application Name' chosen earlier.

```
helm upgrade levoai-pcap-sensor levoai/levoai-pcap-sensor \
--install \
--namespace levoai \
--create-namespace \
--set sensor.config.levoaiOrgId="your Levo Org ID" \
--set sensor.config.satelliteUrl="http(s)://hostname|IP:port"
```

```
--set sensor.config.levoEnv="your application environment (staging, production etc.)"
```

Set additional configs

```
sensor.config.traceExportInterval="trace export interval in seconds (default 10)"
```

```
sensor.config.rateLimit="rate limit number in traces/min (default 1000)"
```

```
sensor.config.filter="pcap filter string, eg. port 8080 and (not port 8081)"
```

```
sensor.config.hostAllow="host allow regex"
```

```
sensor.config.pathAllow="path allow regex"
```

```
sensor.config.hostExclusions="host exclusion regex"
```

```
sensor.config.pathExclusions="path exclusion regex"
```

AWS API Gateway

Logs-based Instrumentation

Tailing Logs with CloudWatch

You may use CloudWatch Logs to instrument your AWS API Gateway endpoints.

The following script has been provided as an example to help you configure logging for your API Gateway endpoints.

Levo's Log Parser (/install-log-parsing-sensors) can be configured to parse the logs and send them to Levo.

```
!/usr/bin/env bash
```

```
log_group_name=levo/api-gateway-logs
```

```
aws logs create-log-group --log-group-name $log_group_name
```

```
aws logs put-retention-policy --log-group-name $log_group_name --retention-in-days 7
```

```
log_group_arn=$(aws logs describe-log-groups --log-group-name-prefix $log_group_name  
--query 'logGroups[0].arn' --output text)
```

```
aws apigatewayv2 update-stage --api-id 'your-apigateway-api-id' --stage-name '$default'  
--access-log-settings "DestinationArn=$log_gr
```

```
aws logs tail --follow $log_group_name
```

Streaming Logs with CloudWatch and Amazon Data Firehose

You may also use Amazon Data Firehose to stream live access logs to Levo's satellite.

1. Configure a CloudWatch log group for APIs in API Gateway (using the above example script)
2. Create a Firehose stream to send incoming events to a publicly accessible satellite endpoint
3. Connect the CloudWatch log group to the Firehose stream

Please contact support@levo.ai if you are interested in this setup.

CloudFront Lambda@Edge Instrumentation

You may configure AWS CloudFront with your API Gateway endpoints as the origin, and use Lambda@Edge functions to intercept and capture traffic.

Please visit the following links for more information

Setting up API Gateway with a CloudFront distribution

(<https://repost.aws/knowledge-center/api-gateway-cloudfront-distribution>)

Setting up Levo's CloudFront Lambda@Edge Functions

(/install-traffic-capture-sensors/aws-cloudfront)

AWS CloudFront

Lambda@Edge functions to ingest traffic from AWS CloudFront distributions.

Installation

Pre-requisites

Install the AWS CLI (version 2) by following the AWS docs

(<https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>).

You have sufficient permissions on AWS to create and deploy Lambda@Edge functions.

The Satellite has been successfully set up and is reachable (via HTTPS) from the worker.

Creating the Lambda Functions using the AWS CLI

Obtain your organization's ID from <https://app.levo.ai/settings/organizations>

(<https://app.levo.ai/settings/organizations>) or by clicking on your profile picture in Levo's dashboard, and navigating to User Settings -> Organizations.

Run the install.sh script in the repository.

```
git clone https://github.com/levoai/aws-cloudfront-lambda
```

```
cd aws-cloudfront-lambda
```

```
LEVO_ORG_ID=<value> ./install.sh
```

Associating the Lambdas with a CloudFront Distribution

1. Go to the AWS CloudFront Console

(<https://us-east-1.console.aws.amazon.com/cloudfront/v4/home/distributions>) and select your distribution.

2. Click on the "Behaviors" tab, then click on the "Create Behaviour" button.

3. Configure the behaviour and ensure that the following properties are set:

Path pattern: Use * to send all JSON payloads to Levo, or use a more specific API pattern

Origin and origin groups: The origin for which the traffic should be sent

Allowed HTTP methods: GET, HEAD, OPTIONS, PUT, POST, PATCH, DELETE

Cache policy: Set this to any policy as per your requirements

Function Associations

Origin request:

Function type: Lambda@Edge

Function ARN: Paste the "Request Handler ARN" value printed by the install.sh script

Include body: Yes

Origin response:

Function type: Lambda@Edge

Function ARN: Paste the "Response Handler ARN" value printed by the install.sh script

4. Click on the "Create behaviour" button to save the configuration.

That's all! Within a few minutes, you should start seeing API catalogs in your Levo dashboard.

AWS Traffic Mirroring

i. Prerequisites

Satellite has been successfully installed with traffic mirroring listener.

You have noted down the Satellite's Elastic Network Interface (target ENI) id.

You have noted down the Source Elastic Network Interface (source ENI) id, usually the Load Balancer ENI.

The Satellite is reachable from the source where you are mirroring traffic from.

Setup Levo CLI with AWS credentials (/security-testing/test-laptop)

ii. Creating mirroring session using Levo CLI

In order to create the traffic mirroring in aws you have to run:

```
levo mirror create
```

The CLI will ask for some inputs. First it will ask for the Elastic Network Interface resource id of the source instance from which you want to mirror the traffic.

```
? What is the source Network Interface resource id? [your-source-eni-for-traffic-mirroring]
```

```
Getting source mirroring details...
```

Then CLI will ask for the Elastic Network Interface resource id of the target satellite instance you want to mirror the traffic to.

```
? What is the target Network Interface resource id? [eni-for-satellite-running-traffic-listener]
```

```
Getting source mirroring details...
```

```
Initializing traffic mirroring... creating traffic mirroring filter if necessary.
```

```
Looking for an existing traffic mirror target...
```

```
Looking for eni-***** in us-west-2
```

```
Then it will ask you to name the traffic mirroring session so you can identify it.
```

```
? How do you want to name the mirroring session? [your-mirroring-session-name]
```

```
Creating traffic mirroring session...
```

Done. Now traffic should be mirrored from your source network interface into the Levo satellite.

ii. Listing mirroring session using Levo CLI

```
foo@bar:~$ levo mirror list
```

```
my-mirroring-session-1
```

```
my-mirroring-session-2
```

```
my-mirroring-session-3
```

iii. Delete a mirroring session using Levo CLI

```
foo@bar:~$ levo delete my-mirroring-session-1
```

```
Sesion successfully deleted!
```

Azure API Management

Policy-based Instrumentation

Pre-requisites

You have sufficient permissions on Azure to configure API Management policies.

The Satellite has been successfully set up and is reachable (via HTTPS) from the resource group.

Installation

To instrument your Azure API Management endpoints, the following steps are required:

1. Configuring named values
2. Adding the instrumentation policy

Configuring Named Values

Follow the steps in the official Azure docs to add named values to your API Management instance

(<https://learn.microsoft.com/en-us/azure/api-management/api-management-howtoproperties?tabs=azure-portaladd-a-plain-or-secret-value-to-api-management>).

The following named values must be configured:

Name

Description

Your organization's ID.

Obtain your organization's ID from <https://app.levo.ai/settings/organizations>

LevoOrgId

(<https://app.levo.ai/settings/organizations>) or by clicking on your profile picture in Levo's dashboard, and navigating

to User Settings -> Organizations.

LevoTracesEndpoint The URL to which traces should be sent, e.g. <https://collector.levo.ai>.

LevoEnv

The environment in which the apps will show up in Levo's dashboard, e.g. production or staging.

Adding the Policy

Follow the steps in the official Azure docs to add a policy to your API Management instance (<https://learn.microsoft.com/en-us/azure/api-management/api-management-howtoproperties?tabs=azure-portaladd-a-plain-or-secret-value-to-api-management>).

Copy the contents of the policy.xml file in the [levoai/azure-apim-policy](https://github.com/levoai/azure-apim-policy)

(<https://github.com/levoai/azure-apim-policy>) repository on GitHub and paste it into the policy editor.

Ensure that the policy is added at the API Scope

(<https://learn.microsoft.com/en-us/azure/api-management/set-edit-policies?tabs=editorapi-scope>).

Logs-based Instrumentation

You may also use Azure API Management Logs to instrument your APIs. Visit the Log Parser ([/install-log-parsing-sensors](#)) page for more details.

Cloudflare Worker

Prerequisites

You are using Cloudflare for DNS, and you have proxying (<https://developers.cloudflare.com/dns/manage-dns-records/reference/proxied-dns-records/>) enabled.

You have sufficient permissions on Cloudflare to create workers and configure worker routes for your website.

The Satellite has been successfully set up and is reachable (via HTTPS) from the worker.

Deploying the Worker

Using the CLI

Follow the steps below to deploy the worker to your account.

You can obtain your organization's ID from <https://app.levo.ai/settings/organizations> (<https://app.levo.ai/settings/organizations>) or by clicking on your profile picture in Levo's dashboard, and navigating to User Settings -> Organizations.

Clone the worker repository

```
git clone https://github.com/levoai/cf-worker.git
```

cd into the repository

```
cd cf-worker
```

Install all dependencies

```
yarn
```

Authenticate with Cloudflare

```
npx wrangler login
```

Deploy the worker

```
npx wrangler deploy
```

Add your organization ID as a secret

```
echo <VALUE> | npx wrangler secret put LEVO_ORG_ID
```

That's it! The worker has been added to your Cloudflare account.

You must also add LEVO_SATELLITE_URL as an environment variable for the worker if you are hosting the Satellite yourself.

Check the repository's README (<https://github.com/levoai/cf-worker/blob/main/README.md>) for a list of all supported variables.


Configuring Websites to use the Worker

Follow the instructions in the Cloudflare Docs

(<https://developers.cloudflare.com/workers/configuration/routing/routes/set-up-a-route>).

When adding a worker route, ensure that the failure mode is set to "Fail open" to allow requests to bypass the worker in case of unexpected errors or if the daily request limit

(<https://developers.cloudflare.com/workers/platform/limits/daily-request>) runs out.



```
route" style={{ display: 'block', margin: 'auto', paddingTop: '24px'}} />
```

Install Traffic Capture Sensors

Depending on your environment, you may choose to install a different Levo sensor to suit your needs.

eBPF Sensor (Recommended)

You should install the eBPF sensor (</guides/key-conceptsebpf-sensor>) if:

You have access to the node / VM / machine where your application workloads are running

In addition to your publicly exposed services, you want to instrument internal applications which do not have public API endpoints

Click here for the installation instructions (</install-traffic-capture-sensors/ebpf-sensor>).

PCAP Sensor

You should install the pcap sensor if:

Your application workloads are deployed on a Serverless compute architecture (like AWS Fargate)

Click here for the installation instructions (</install-traffic-capture-sensors/pcap-sensor>).

AWS Traffic Mirroring

Use this if you want to use traffic mirroring to instrument your application workloads.

Click here for the installation instructions (</install-traffic-capture-sensors/aws-traffic-mirroring>).

Cloudflare Worker

You may install Levo's Cloudflare Worker if:

You are using Cloudflare for DNS, and you have proxying

(<https://developers.cloudflare.com/dns/manage-dns-records/reference/proxied-dns-records/>) enabled.

Click here for the installation instructions (</install-traffic-capture-sensors/cloudflare-worker>).

AWS CloudFront Lambda@Edge

You may install Levo's CloudFront Lambda@Edge functions if:

You are using CloudFront as a CDN for your API endpoints.

Note that CloudFront does not provide access to the API endpoint response bodies.

Click here for the installation instructions (</install-traffic-capture-sensors/aws-cloudfront>).

AWS API Gateway

You may instrument your AWS API Gateway endpoints with CloudWatch Logs.

However, CloudWatch only provides endpoints access logs and API endpoint request and response bodies will not be available.

Click here for the installation instructions (</install-traffic-capture-sensors/aws-api-gateway>).

Azure API Management Policy

You should install Levo's Azure API Management policy if:
Your API endpoints are managed by Azure API Management.
Click here for the installation instructions
(/install-traffic-capture-sensors/azure-api-management).

Quickstart

If you want a quick glimpse of Levo's API Observability without a full installation, check out the Quickstart page (/quickstart).

Sensor on MacOS

This guide provides comprehensive instructions for installing the Levo Satellite, Sensor and Log Parser components together as a single container on a MacOS host.

Follow instructions for your specific platform/method below:

Install on Linux host via Docker

Install via Docker

Prerequisites

Docker Engine version 18.03.0 and above

1. Install Levo-all (Sensor, Satellite and Log Parser)

This section provides information on the optional environment variables that can be set to customize the properties of the sensor-satellite configuration.

The Sensor-Satellite setup can be run with the following docker command -

```
docker run -e LEVOAI_AUTH_KEY=<your-auth-key> \
```

```
-e LEVOAI_ORG_ID=<your-org-id> \
```

```
--net=host \
```

```
-v ./logs:/mnt/levo/logs
```

```
levoai/levo-all
```

Required Environment Variables

LEVOAI_AUTH_KEY

Description: The Satellite CLI authorization key from app.levo.ai

Default: ""

LEVOAI_ORG_ID

Description: Organization ID for your specific organization in your app.

Default: ""

Optional Environment Variables

The following environment variables can be configured to modify the behavior of the Sensor-Satellite setup:

LEVO_FILTER

Description: Set a filter for specific data.

Default: ""

LEVO_TRACE_EXPORT_INTERVAL

Description: Interval for exporting traces.

Default: 0.0

LEVO_RATE_LIMIT_NUMBER

Description: Set the rate limit number.

Default: 0.0

LEVO_HOST_ALLOW_RE

Description: Regular expression for allowed hosts.

Default: ""

LEVO_PATH_ALLOW_RE

Description: Regular expression for allowed paths.

Default: ""

LEVO_HOST_EXCLUSIONS_RE

Description: Regular expression for excluded hosts.

Default: ""

LEVO_PATH_EXCLUSIONS_RE

Description: Regular expression for excluded paths.

Default: ""

LEVO_ORG_ID

Description: Set the organization ID.

Default: ""

LEVO_APP_ENVIRONMENT

Description: Set the application environment.

Default: "staging"

LEVOAI_SATELLITE_PORT

Description: Set the port for the LevoAI satellite.

Default: 9999

LEVOAI_MODE

Description: Set the mode of the LevoAI system (e.g., "single-node").

Default: "single-node"

LEVOAI_DEBUG_ENABLED

Description: Enable or disable debug mode.

Default: false

LEVOAI_DEBUG_PORT

Description: Set the port for debugging.

Default: 12345

LEVOAI_DEBUG_SERVER_HOST

Description: Set the host for the debug server.

Default: "host.docker.internal"

LEVOAI_LOG_LEVEL

Description: Set the log level (e.g., "INFO").

Default: "INFO"

ENABLE_LOG_PARSER

Description: Enable or disable the log parser.

Default: true

LEVOAI_BASE_URL

Description: Set the base URL for the Levo.ai Platform API.

Default: "https://api.levo.ai"

APP_NAME

Description: Set the application name.

Default: "app-logs-DATE-TIME"

ENV_NAME

Description: Set the environment name.

Default: "staging"

Note

Setting these environment variables is optional and can be set according to your specific requirements before deploying the Sensor-Satellite setup.

Log Parser

List of supported log parsers

Nginx

Apache

Azure API Gateway

Note

Make sure logs directories are structured as per the supported log parsers.

The logs directory should be mounted to the /mnt/levo/logs directory in the container.

Nginx logs should be mounted to /mnt/levo/logs/nginx.

Apache logs should be mounted to /mnt/levo/logs/apache.

Azure API Gateway logs should be mounted to /mnt/levo/logs/azure.

Common Tasks

Generating CLI Authorization Keys

Accessing Organization IDs

Generating CLI Authorization Keys

The Levo CLI is packaged within CI/CD plugins that are embedded in quality gates, that run security/resilience tests.

The CLI uses an authorization key to access Levo.ai. Follow instructions below to generate a key.

Login (<https://app.levo.ai/login>) to Levo.ai

Click on your user profile

Click on User Settings

Click on Keys on the left navigation panel

Click on Get CLI Authorization Key

Now copy & save your authorization key, to be used in the CI/CD plugin of your choice

Accessing Organization IDs

Levo allows signed-in users to belong to more than one organization. Each organization has a unique ID. Below are instructions on fetching the ID for a specific organization.

Fetch ORG ID

Login (<https://app.levo.ai/login>) to Levo.ai
Click on your user profile
Click on User Settings
Click on Organizations on the left navigation panel
Now copy & save the ID for the Organization of your preference
This ID will be used within 3rd party integrations like CI/CD plugins, etc.

keywords: [CI/CD Integrations, API
Integrations, Continuous Security Testing]
Integrations

Levo allows you to embed API security/resilience testing into development workflows by integrating with with your preferred third-party development tools.

Below are details on various supported integrations.

JIRA (</integrations/jira>)
Splunk (</integrations/splunk>)
Slack (</integrations/slack>)
Okta (</integrations/okta>)
Webhooks (</integrations/webhooks>)

Jira

Atlassian JIRA Integration

This integration allows JIRA tickets to be created/viewed directly from Test Run failures reported by Levo. Below are links to common tasks.

[Add JIRA Integration](#)

[Creating JIRA Tickets From Vulnerability Page](#)

[Assign / Unassign JIRA Tickets From Vulnerability Page](#)

[Add JIRA Integration](#)

1. Prerequisites

Ensure you have a JIRA account, and note down the URL for the JIRA service.

Create an API integration token in your Atlassian account as shown below.

Copy the API Token

Identify the JIRA Project that will be the recipient for the tickets created from Levo, and note down the project's Key name.

2. Enable JIRA Integration

In the Levo SaaS console, navigate to the Integrations screen as shown below and click on Jira tile.

Configure the JIRA integration following the steps below. Specify the Project Key rather than the Project Name in the screens below.

Save the settings to enable the integration

Congratulations! You have successfully enabled the JIRA integration. Below are steps to a) create JIRA tickets from failed test runs, and b) view linked JIRA tickets from failed test cases.

Creating JIRA Tickets From Vulnerability Page

Follow the below steps to create a JIRA ticket for a specific Vulnerability reported from a Test Run.

Navigate to the Vulnerabilities page.

Navigate to the specific Vulnerability, and click on the Create Ticket icon.

Complete the dialog appropriately to create a JIRA ticket

Optionally verify if the ticket was successfully created in JIRA. Click on Jira Ticket link to open the ticket in a new browser tab.

Assign / Unassign JIRA Tickets From Vulnerability Page

Vulnerabilities that are linked with a JIRA ticket, will have a User Icon as shown below. Clicking on the icon will open the dialog, select user to assign/unassign user to a ticket.

Okta

Configure Okta Integration using SAML

Follow these steps to configure the Okta Integration for your Levo organization. This will allow you to use Okta to login to the platform.

1. Navigate to Levo console IAM settings section and click on "Configure Okta" button.
2. Copy ACS Url and Entity ID to be used during the Okta app creation.
3. Navigate to the application creation screen in your Okta admin console.
4. Add the Descope app from the Okta Integration Marketplace.
5. When you first add the integration, set the name as Levo and click Done:
6. Under Sign On > Advanced Sign-on Settings use the values we got in step 2:
7. Once you've added the app, expand the attributes field in the SAML section and add the following mapping:
 8. Under Assignments in the same section, add the relevant User and Group assignments to your new application.
9. Go to Sign on methods > SAML 2.0 > Metadata details, to locate and copy your Okta Metadata URL.
10. And as a last step in the Okta app creation edit the logo of the application and replace it with the Levo logo (../assets/Integrations/Okta/levo-logo.png) so you can identify it in your Okta dashboard.
11. Go back to Levo console and fill in the details to connect your Okta app.

Connection Name: AN identifier for the connection.

Domain: Email domain of your organization users.

Okta Metadata Url: The Metadata URL you copied from Okta.

After that, you should be able to use this custom app to login to Levo using Okta.

Slack

Add Slack Integration

1. Prerequisites

Sign in to your Slack and navigate to the channel where you want to receive notifications.

Click on the channel name, then select "Integrations" > "Add an app".

Search for "Incoming WebHooks" and install it.

Select the channel for posting messages and click "Add Incoming WebHooks integration".

Copy the Webhook URL provided.

2. Enable Slack Integration

In the Levo SaaS console, navigate to the Integrations screen as shown below and click on Slack tile.

Configure the Slack integration following the steps below.

Select the event types for Changelog Notifications and Vulnerability Notifications that you wish to receive.

Changelog Event Types: New Application, New Endpoint, New Sensitive Type.

Vulnerability Notification Types: Vulnerability Created, Vulnerability Reopened, Vulnerability Closed.

Paste the Webhook URL copied from Slack.

After configuring your preferences, save to activate the Slack integration.

Splunk

Add Splunk Integration

1. Prerequisites

Sign in to your Splunk where you like to receive notifications.

Copy the Webhook URL and HEC Token.

2. Enable Splunk Integration

In the Levo SaaS console, navigate to the Integrations screen as shown below and click on Splunk tile.

Configure the Splunk integration following the steps below.

Select the event types for Changelog Notifications and Vulnerability Notifications that you wish to receive.

Changelog Event Types: New Application, New Endpoint, New Sensitive Type.

Vulnerability Notification Types: Vulnerability Created, Vulnerability Reopened, Vulnerability Closed.

Paste the Webhook URL and HEC Token copied from Splunk.

After configuring your preferences, save to activate the Splunk integration.

Custom Webhooks

Add Custom Webhook Integration

1. Prerequisites

Make sure you have the necessary webhook related information where you like to receive notifications.

Copy the Webhook URL and API Key.

2. Enable Splunk Integration

In the Levo SaaS console, navigate to the Integrations screen as shown below and click on Webhooks tile.

Configure the Webhooks integration following the steps below.

Select the event types for Changelog Notifications and Vulnerability Notifications that you wish to receive.

Changelog Event Types: New Application, New Endpoint, New Sensitive Type.

Vulnerability Notification Types: Vulnerability Created, Vulnerability Reopened, Vulnerability Closed.

Paste the Webhook URL and API Key copied earlier.

After configuring your preferences, save to activate the Webhook integration.

Quickstart

Evaluate Levo's API Observability in Action with your favourite tools.

Quickstart on Mac ([quickstart-mac.md](#))

Quickstart on Windows ([quickstart-kubernetes.md](#))

Quickstart with Minikube ([quickstart-minikube.md](#))

Quickstart with Burp ([quickstart-burp-plugin.md](#))

Quickstart with OWASP ZAP ([quickstart-zap-addon.md](#))

Quickstart with MITM proxy in Docker ([quickstart-mitm.md](#))

If you are looking for comprehensive install instructions (for all supported platforms), please refer to the Install Guide ([../guides/install-guide/install-guide.md](#)).

keywords: [API Security, ZAP, OWASP,
Linux, macOS, Windows, API Observability]

Quickstart with Burp

The Levo.ai add-on for Burp Plugin allows building OpenAPI specs with the traffic sent or proxied via Burp Suite.

You can install the Levo.ai Burp Integration Plugin from the BApp Store:
<https://portswigger.net/bappstore/e1772ac14930453b98d5bff8c4f8b0cd>
(<https://portswigger.net/bappstore/e1772ac14930453b98d5bff8c4f8b0cd>).

1. Set Levo satellite's URL

2. Set org id

You can copy Organization Id from "User Settings" page from Levo Dashboard top right section.

3. Enable sending traffic to Levo

keywords: [API Security, ZAP, OWASP,
Linux, macOS, Windows, API Observability, Kubernetes]

Quickstart on Kubernetes

This quickstart guide will help you install the LevoAI eBPF Sensor on a Kubernetes cluster.

Prerequisites

Kubernetes version \geq v1.18.0

Helm v3 (<https://helm.sh/docs/intro/install/>) installed and working.

The Kubernetes cluster API endpoint should be reachable from the machine you are running Helm.

kubectl access to the cluster, with cluster-admin permissions.

At least 4 CPUs

At least 8 GB RAM

Copy Authorization Key from Levo.ai

The Satellite uses an authorization key to access Levo.ai.

Login (<https://app.levo.ai/login>) to Levo.ai.

Click on your user profile.

Click on User Settings

Click on Keys on the left navigation panel

Click on Get Satellite Authorization Key

Copy your authorization key. This key is required in subsequent steps below.

Add Helm Charts Repository

```
helm repo add levoai https://charts.levo.ai && helm repo update
```

Add LevoAI Auth Key

```
export LEVOAI_AUTH_KEY=<'Authorization Key' copied earlier>
```

Install Satellite

```
helm upgrade --install -n levoai --create-namespace \
```

```
--set global.levoai_config_override.onprem-api.refresh-token=$LEVOAI_AUTH_KEY \
```

```
levoai-satellite levoai/levoai-satellite
```

Check satellite connectivity

Execute the following to check for connectivity health:

Please specify the actual pod name for levoai-tagger below

```
kubectl -n levoai logs <levoai-tagger pod name> | grep "Ready to process; waiting for messages."
```


If connectivity is healthy, you will see output similar to below.

```
{"level": "info", "time": "2022-06-07 08:07:22,439", "line": "rabbitmq_client.py:155", "version": "fc628b50354bf94e544eef46751d44945a"}
```

Install eBPF Sensor

Replace 'hostname|IP' & 'port' with the values you noted down from the Satellite install

If Sensor is installed on same cluster as Satellite, use 'levoai-collector.levoai:4317'

Specify below the 'Application Name' chosen earlier.

```
helm upgrade levoai-sensor levoai/levoai-ebpf-sensor \
--install \
--namespace levoai \
--create-namespace \
--set sensor.config.default-service-name=<'Application Name' chosen earlier> \
--set sensor.config.satellite-url=<hostname|IP:port>
--set sensor.config.env=<'Application environment'>
```

Check sensor health

Please specify the actual pod name for levoai-sensor below

```
kubectl -n levoai logs <levoai-sensor pod name> | grep "Initial connection with Collector"
```

If connectivity is healthy, you should see output similar to below.

```
2022/06/13 21:15:40 729071
```

```
INFO [ebpf_sensor.cpp->main:120]
```

```
Initial connection with Collector was successful.
```

Please contact support@levo.ai if you notice health/connectivity related errors.

keywords: [API Security, ZAP, OWASP,
Linux, macOS, Windows, API Observability]

Quickstart on Mac / Laptop

Prerequisites

Docker Engine version 18.03.0 and above

Copy Authorization Key from Levo.ai

The Levo-all uses an authorization key to access Levo.ai.

Login (<https://app.levo.ai/login>) to Levo.ai.

Click on your user profile.

Click on User Settings

Click on Keys on the left navigation panel

Click on Get Satellite Authorization Key

Copy your authorization key. This key is required in subsequent steps below.

Install Levo-all (Sensor, Satellite and Log Parser)

The Sensor-Satellite setup can be run with the following docker command -

```
docker run -e LEVOAI_AUTH_KEY=<your-auth-key> \  
-e LEVOAI_ORG_ID=<your-org-id> \  
--net=host \  
-v ./logs:/mnt/levo/logs  
levoai/levo-all
```

Required Environment Variables

LEVOAI_AUTH_KEY

Description: The Satellite CLI authorization key from app.levo.ai

Default: ""

LEVOAI_ORG_ID

Description: Organization ID for your specific organization in your app.

Default: ""

Note

For more information on the environment variables, refer to the Environment Variables (/install-traffic-capture-sensors/sensor-on-macosrequired-environment-variables) section.

For more information on log parser, refer to the Log Parser (/install-traffic-capture-sensors/sensor-on-macoslog-parser) section.

Please contact support@levo.ai if you notice health/connectivity related errors.

keywords: [API Security, ZAP, OWASP,
Linux, macOS, Windows, API Observability]

Quickstart with Minikube

Run the following command to find out which driver your minikube installation is using:

```
minikube profile list
```

```
|-----|-----|-----|-----|-----|-----|-----|-----|  
| Profile  
| VM Driver | Runtime |  
IP  
| Port | Version | Status  
| Nodes | Active |  
|-----|-----|-----|-----|-----|-----|-----|  
| minikube | docker  
| docker  
| 192.168.49.2 | 8443 | v1.27.3 | Running |  
1 | *  
|  
|-----|-----|-----|-----|-----|-----|-----|
```

The second column in the output, VM Driver, should list the minikube driver of your existing minikube profile.

If you want to test the eBPF Sensor in minikube, we recommend using minikube with a VM driver (e.g. kvm2 or virtualbox). Or if you are already using a VM, you may run minikube on bare metal (with the "none" driver).

You can find the full list of minikube drivers in the minikube docs (<https://minikube.sigs.k8s.io/docs/drivers/>).

Based on your minikube driver, follow the instructions below:

Bare-metal or VM-based driver

Follow the standard Kubernetes installation instructions

(/install-traffic-capture-sensors/ebpf-sensor/sensor-kubernetes).

Docker driver

Since the eBPF Sensor needs access to the /proc folder on the host, there are additional steps to ensure that the directory is mounted correctly inside the Sensor container for running it in minikube with the Docker driver.

First, run:

```
minikube mount /proc:/ggproc
```

Then, in a new terminal window, run:

```
helm repo add levoai https://charts.levo.ai && helm repo update
```

```
helm pull levoai/levoai-ebpf-sensor --untar
```

```
cd levoai-ebpf-sensor/
```

```
sed -i "s/path: /proc/path: /ggproc/" templates/deployment.yaml
```

```
helm upgrade --install levoai-sensor . -n levoai
```

```
keywords: [API Security, eBPF, macOS,
```

```
Windows, API Observability]
```

```
import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';
```

```
Quickstart with MITM proxy
```

```
Quickstart instructions for evaluating API Observability on Laptops/Desktops running Mac OSX or Windows.
```

```
Levo Sensor Package for
```

```
OSX/Windows
```

```
Since Mac OSX and Windows do not support eBPF (https://ebpf.io), Levo provides a Sensor package (Docker based install), to enable quick evaluation on these platforms. This
```

```
Sensor package gets visibility into your API traffic, by reverse proxying
```

```
(https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/) traffic between your API
```

```
Client and
```

```
API Server.
```

```
Your estimated completion time is 10 minutes.
```

1. Prerequisites

```
Docker Engine version `18.03.0` and above
```

```
Admin (or `sudo`) privileges on the Docker host
```

```
Forever Free Account on Levo.ai (https://levo.ai/levo-signup/)
```

```
Command line terminal with Bash or Bash compatible shell
```

```
Docker Engine version `18.03.0` and above
```

```
Admin privileges on the Docker host
```

```
Forever Free Account on Levo.ai (https://levo.ai/levo-signup/)
```

Docker containers MUST be allowed to connect to the internet. Please check Firewall settings
PowerShell terminal

2. Setup Test API Service

API Observability auto discovers APIs and generates OpenAPI specifications for all API endpoints, by observing API traffic between your API Client and API Server.

If you do not have a test API Service/Application, you can use the sample application (/guides/demo-application) provided by Levo.

a. Note down the base URL for your test API Server/Service.

For example, if you are running the sample application (crAPI) on your laptop, the base URL would be `http://localhost:8888`. If your local test API Server uses HTTPS the base URL for example, would be `https://localhost/`.

Since the Sensor package runs in a container, addresses like `localhost`, `127.0.0.1`, etc., that refer to the Docker host, must be translated to ones, that can be resolved correctly to point to the Docker host inside the container. Please specify `host.docker.internal` instead of `localhost` or `127.0.0.1` in the base URL.

In essence, if your base URL is `http://localhost:<port>` or `http://127.0.0.1:<port>`, you will need to specify `http://host.docker.internal:<port>` instead below.

b. Export your API Server/Service URL in your terminal.

```
export SERVICE_ADDRESS=<http://host:port/base-path>  
$env:SERVICE_ADDRESS="<http://host:port/base-path>"
```

3. Copy Authorization Key from Levo.ai

The Sensor package uses an authorization key to access Levo.ai. Follow instructions below to copy & export the key.

Login (<https://app.levo.ai/login>) to Levo.ai.

Click on your user profile.

Click on User Settings

Click on Keys on the left navigation panel

Click on Get Satellite Authorization Key

Now copy your authorization key.

Export the copied Authorization Key in your terminal.

```
export LEVOAI_AUTH_KEY=<'Authorization Key' copied above>  
$env:LEVOAI_AUTH_KEY="<'Authorization Key' copied above>"
```

4. Pick an Application Name

Auto discovered API endpoints and their OpenAPI specifications are show in the API Catalog (/guides/security-testing/concepts/api-catalog/api-catalog.md), grouped under an

application name. The application name helps segregate and group API endpoints from different API servers, similar to how file folders work in an operating system.

a. Pick a descriptive name which will be used in the subsequent step below. For example: my-test-api-server.

b. Export the Application Name in your terminal.

```
export LEVOAI_SERVICE_NAME=<'Application Name' chosen above>
$env:LEVOAI_SERVICE_NAME="<'Application Name' chosen above>"
```

5. Download - Docker Compose file

Execute the following in your terminal:

```
import BrowserOnly from '@docusaurus/BrowserOnly';
export function CurlScript(props) { var curlCmd = "curl"; if (props	curlCmd) return (
<BrowserOnly fallback={
Loading...
}> {} => (
-s -o proxy-docker-compose.yml {window.location.protocol + '/' + window.location.host +
'/artifacts/satellite/proxy-docker-compose
}); }
export function DownloadLink() { return ( <BrowserOnly fallback={
Loading...
}> {} => ( <a href={window.location.protocol + '/' + window.location.host +
'/artifacts/satellite/proxy-docker-compose.yml'}> here )) ); }
```

If prefer to download the Docker Compose file via your browser, you can download it .

6. Install Sensor Package via Docker Compose

Execute the following in your terminal (where you previously downloaded the Docker Compose file):

```
docker compose -f proxy-docker-compose.yml pull && docker compose -f
proxy-docker-compose.yml up -d
docker compose -f .\proxy-docker-compose.yml pull
docker compose -f .\proxy-docker-compose.yml up -d
```

7. Verify Connectivity with Levo.ai

The Sensor package contains both the (proxy based) Sensor and Satellite. Follow steps below to check the Satellite health and connectivity to Levo.ai.

a. Check Satellite Health

The Satellite is comprised of four sub components 1) levoai-collector, 2) levoai-rabbitmq, 3)levoai-satellite, and 4) levoai-tagger.

Wait couple of minutes after the install, and check the health of the components by executing the following:

```
docker ps -f name=levoai
```

If the Satellite is healthy, you should see output similar to below.

```
CONTAINER ID
IMAGE
COMMAND
CREATED
STATUS
5a54d8efe672
levoai/proxy:latest
"docker-entrypoint.s..."
50 seconds ago
Up 37 seconds
PORTS
8767c62db6cb
levoai/satellite:latest
"python -OO /opt/lev..."
50 seconds ago
Up 37 seconds
dcb187e00ff2
levoai/satellite:latest
"gunicorn --capture-..."
50 seconds ago
Up 37 seconds
0.0.0.0:9999->9999/tcp
169ceecf0263
rabbitmq:3.10.5-management
"docker-entrypoint.s..."
50 seconds ago
Up 49 seconds (healthy)
4369/tcp, 5671/tcp, 0
0.0.0.0:8081->8081/tcp
```

b. Check Connectivity

Execute the following to check for connectivity health:

```
docker logs levoai-tagger 2>&1 | grep "Ready to process; waiting for messages."
```

```
docker logs levoai-tagger 2>&1 | sls "Ready to process; waiting for messages."
```

If connectivity is healthy, you will see output similar to below:

```
{"level": "info", "time": "2022-06-07 08:07:22,439",
"line": "rabbitmq_client.py:155", "version": "fc628b50354bf94e544eef46751d44945a2c55bc",
"module": "/opt/levoai/e7s/src/python/levoai_e7s/satellite/rabbitmq_client.py",
"message": "Ready to process; waiting for messages."}
```

Please contact support@levo.ai if you notice health/connectivity related errors.

8. Generate Application Traffic

The Sensor picks up API traffic that is HTTP\1.x based. There has to be some consistent load on your API endpoints for them to be auto discovered and documented.

a. Point Your API Client to the Sensor

The Sensor acts as a reverse proxy

(<https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/>) for your API Server. You will need to point your API Client to the Sensor.

The Sensor will proxy the traffic to your test API Server/Service.

The Sensor listens on `http://127.0.0.1:9080` (`http://127.0.0.1:9080`). Please point your API Client (Web Browser, Postman (<https://www.postman.com/>), curl (<https://curl.se/>), etc.) to this address (instead of the API Server's address).

If your API Server uses HTTP/s (TLS), the Sensor will use HTTP/s when proxying traffic to it. However your API Client will need to use HTTP when talking to the Sensor.

If you are using `/etc/hosts` (or equivalent in Windows) to resolve the IP address of your API Server, please edit the appropriate `/etc/hosts` entry to point to `127.0.0.1` (IP address of the Sensor).

b. Generate Traffic

Please ensure you exercise your API endpoints several times using using your API Client. Use a load generator to generate consistent traffic, if necessary.

c. Verify API Traffic Capture

Check the logs of Satellite's Tagger sub-component.

```
docker logs levoai-tagger 2>&1 | grep "Consuming the span"
```

```
docker logs levoai-tagger 2>&1 | sls "Consuming the span"
```

If API Traffic is correctly being processed, you will see a lot of log entries containing the term Consuming the span.

9. View Auto-discovered OpenAPI Specifications

The API Catalog (`/guides/security-testing/concepts/api-catalog/api-catalog.md`) in `Levo.ai` should be auto populated in a matter of minutes (after your API endpoints are being exercised consistently).

The API Catalog will contain your auto discovered API endpoints and their OpenAPI schemas, all grouped under the Application Name you chose earlier.

Congratulations! You have successfully auto discovered and auto documented API endpoints in your application.

Common Tasks

Shutdown Sensor

Execute the following in the directory where you downloaded the Docker Compose file:

`docker compose -f proxy-docker-compose.yml down`

Change Sensor Listen Port

The Sensor by default listens on TCP port 9080 (interface address 127.0.0.1). If this conflicts with a port being used by another application, you can change it by following the instructions below.

Shutdown (`quickstart-mitm.mdshutdown-sensor`) the Sensor (if running)

Export your desired port in your terminal

```
export LEVOAI_PROXY_PORT=<Your desired port number>
```

```
$env:LEVOAI_PROXY_PORT="<Your desired port number>"
```

- [Start](./quickstart-mitm.md#install-sensor-package-via-docker-compose) the Sensor

keywords: [API Security, ZAP, OWASP,

Linux, macOS, Windows, API Observability]

Quickstart with OWASP ZAP

The Levo.ai add-on for ZAP allows building OpenAPI specs with the traffic sent or proxied via ZAP.

This guide assumes that you have signed up for a Levo account (<https://app.levo.ai/signup>) and have installed a recent version of ZAP (<https://www.zaproxy.org/download/>) (> 2.12.0).

Here are the steps you need to follow to start building OpenAPI specs with Levo and ZAP:

1. The OpenAPI spec is built by sending anonymized API traces to Levo. You may run the Satellite (a set of services which receives and processes the traces) locally using `docker` or `minikube`, or on AWS with an AMI provided by Levo.

Click here for instructions on installing the satellite (`/install-satellite`).

Please ensure that ZAP is able to reach the satellite at the configured listening port (the default is 9999).

2. Launch ZAP and install the Levo.ai add-on from the ZAP Marketplace (<https://www.zaproxy.org/addons/>). You may need to restart ZAP after the add-on is installed.

Screenshot of the Levo.ai button

in ZAP's main toolbar

3. If the add-on is successfully installed, you should see a new button in the main toolbar.

Clicking on it will toggle sending traffic to Levo's satellite.

4. Navigate to Tools → Options → Levo.ai in ZAP and enter the URL pointing to the satellite (e.g. `http://localhost:9999`).

Screenshot of the Levo.ai

Options Panel in ZAP

5. Ensure that the Levo button is enabled in the toolbar, and you are good to go! Start browsing your website using ZAP and you should start seeing auto-discovered applications in your Levo dashboard in a few minutes.

Levo CLI Command Reference

`help`

Show help message for the CLI,

levo --help

version

Show the current version of the CLI.

levo --version

login

levo login [options] <arguments>

Authenticate the CLI with Levo's SaaS portal.

Options:

-v, --verbosity [NOTSET|DEBUG|INFO|WARNING|ERROR|CRITICAL]

OPTIONAL Accept all of the Python's log

level values: CRITICAL, ERROR, WARNING,

INFO, DEBUG, and NOTSET (all case

insensitive).

-k, --key TEXT

Specify an authorization key to login with.

Go to <https://app.dev.levo.ai/settings/keys>

to get your authorization key.

-o, --organization TEXT

OPTIONAL Specify the id of the organization

you want to work with.

-h, --help

Show this message and exit.

The login command might ask you for a CLI Authorization Key, that is used to authenticate the CLI with Levo.ai. This key can be retrieved from User Profile-->User Settings->Keys (<https://app.dev.levo.ai/settings/keys>).

You will need an account on Levo.ai (<https://levo.ai/levo-signup/>) to retrieve the key.

The login command stores authentication tokens in the `$HOME/.config/configstore/levo.json` file (on the Docker host). This file is only accessible by the user who owns the `$HOME` directory. Treat this file as do with any secrets.

logout

levo logout [options]

Removes the local login config file.

Options:

-v, --verbosity [NOTSET|DEBUG|INFO|WARNING|ERROR|CRITICAL]

OPTIONAL Accept all of the Python's log

level values: CRITICAL, ERROR, WARNING,

INFO, DEBUG, and NOTSET (all case

insensitive).

-h, --help

Show this message and exit.

The logout command removes the `$HOME/.config/configstore/levo.json` file (on the Docker Host). This file contains authentication tokens, and other local state persisted by the CLI.

test-conformance

levo test-conformance [options] <arguments>

Perform schema conformance tests against API endpoints specified in the target-url.

Options:

--schema TEXT

--schema must specify a valid URL or file path (accessible from the CLI container) that points to an Open API / Swagger specification.

--target-url TEXT

[required]

--target-url must specify a valid URL pointing to a live host that implements the endpoints specified by --schema.

--disable-reporting-to-saas

[required]

Do not send test reports to Levo's SaaS portal.

-H, --header TEXT

Custom header that will be used in all requests to the target server. Example: -H "Authorization: Bearer 123" .

--show-errors-tracebacks

Show full tracebacks for internal errors.

--ignore-ssl-verify TEXT

Controls whether the test run verifies the server's SSL certificate.

-v, --verbosity [NOTSET|DEBUG|INFO|WARNING|ERROR|CRITICAL]

OPTIONAL Accept all of the Python's log level values: CRITICAL, ERROR, WARNING, INFO, DEBUG, and NOTSET (all case insensitive).

--export-junit-xml FILENAME

Export test results as JUnit XML

-h, --help

Show this message and exit.

Levo CLI runs as a Docker container and by default mounts the current working directory on the host file system as read/write. If specifying a schema file as an argument, please provide a path that is accessible by the CLI container.

Do not use 127.0.0.1 or localhost as arguments of the --target-url, as these will not resolve correctly within the CLI container. Please use host.docker.internal instead.

Usage Examples

```
levo test-conformance --target-url http://host.docker.internal:9000/ --schema ./malschema.json
```

```
levo test-conformance --target-url http://host.docker.internal:9000/ --schema
```

```
http://host.docker.internal:9000/api/openapi.json
```

```
test
```

```
levo test [options] <arguments>
```

Execute a test plan against the specified target-url.

Options:

--target-url TEXT

--target-url must specify a valid URL pointing to a live host that implements the endpoints that are present in the test plan.

[required]

--disable-reporting-to-saas

Do not send test reports to Levo's SaaS portal.

--test-plan TEXT

--test-plan must specify a valid Levo Resource Name (LRN) or a path to a Levo Test Plan folder (accessible from the CLI container).

-H, --header TEXT

[required]

Custom header that will be used in all requests to the target server. Example: -H "Authorization: Bearer 123" .

--show-errors-tracebacks
Show full tracebacks for internal errors.

--env-file TEXT
Path to YAML file with environment definitions (AuthN/AuthZ info, etc.). This file must be accessible from the CLI container.

--ignore-ssl-verify TEXT
Controls whether the test run verifies the server's SSL certificate.

-v, --verbosity [NOTSET|DEBUG|INFO|WARNING|ERROR|CRITICAL]

-d, --suite-execution-delay INTEGER
Adds a delay between test suite execution

--request-timeout INTEGER
Timeout for the http request made to the API

--export-junit-xml FILENAME
Export test results as JUnit XML

-h, --help
Show this message and exit.

Levo CLI runs as a Docker container and by default mounts the current working directory on the host file system as read/write. If specifying a Test Plan folder as an argument, please provide a path that is accessible by the CLI container.

Do not use 127.0.0.1 or localhost as arguments of the --target-url, as these will not resolve correctly within the CLI container. Please use host.docker.internal instead.

Authentication credentials and user role information might be required by some Test Plans for proper execution. This is to be provided using the --env-file option.

Please refer to details on Authentication/Authorization (</guides/security-testing/concepts/test-plans/env-yml.md>).

Usage Examples

Using a local test plan folder levo test --target-url host.docker.internal:8888 --test-plan ./my-test-plan-folder --env-file

./environment.yml

using a LRN (Levo Resource Name) for a test plan located in Levo SaaS levo test --target-url host.docker.internal:8888 --test-plan

demo:app/Demo_crAPI:tp/Demo_crAPI --env-file ./environment.yml

Here demo:app/Demo_crAPI:tp/Demo_crAPI is the LRN for a test plan located in Levo SaaS.

test-plan

Test Plan management sub commands.

run

This is an alias of the levo test command.

levo test-plan run [options] <arguments>

Run a test plan against the specified target-url.

Options:

--target-url TEXT

--target-url must specify a valid URL pointing to a live host that implements the endpoints that are present in the test plan.

[required]

--disable-reporting-to-saas

Do not send test reports to Levo's SaaS portal.

--test-plan TEXT

--test-plan must specify a valid Levo Resource Name (LRN) or a path to a Levo Test Plan folder (accessible from the CLI container).

-H, --header TEXT

[required]

Custom header that will be used in all requests to the target server. Example: -H "Authorization: Bearer 123" .

--show-errors-tracebacks

Show full tracebacks for internal errors.

--env-file TEXT

Path to YAML file with environment definitions (AuthN/AuthZ info, etc.). This file must be accessible from the CLI

container.

--ignore-ssl-verify TEXT

Controls whether the test run verifies the server's SSL certificate.

-v, --verbosity [NOTSET|DEBUG|INFO|WARNING|ERROR|CRITICAL]

-d, --suite-execution-delay INTEGER

Adds a delay between test suite execution

--request-timeout INTEGER

Timeout for the http request made to the API

--export-junit-xml FILENAME

Export test results as JUnit XML

-h, --help

Show this message and exit.

This command is a functional equivalent of the levo test command. Please see constraints and examples outlined for that command.

export-env

The environment file is used to specify authentication credentials, and optional role(s) information (for authorization tests). Please refer to Authentication/Authorization (</guides/security-testing/concepts/test-plans/env-yml.md>).

levo test-plan export-env [OPTIONS] <arguments>

Export the environment file of a test plan from Levo SaaS to the local file system.

Options:

--lrn TEXT

The LRN of the test plan, whose environment file you want to export.

--local-dir TEXT

[required]

Path to a local directory where the environment file is to be exported. The local directory must be accessible from the CLI container. If not specified, the test plan is exported to the current working directory.

-v, --verbosity [NOTSET|DEBUG|INFO|WARNING|ERROR|CRITICAL]

Accept all of the Python's log level values:

CRITICAL, ERROR, WARNING, INFO, DEBUG, and

NOTSET (all case insensitive).

-h, --help

Show this message and exit.

Levo CLI runs as a Docker container and by default mounts the current working directory on the host file system as read/write.

Usage Examples

```
levo test-plan export-env --lrn "acme-gizmo-org:ws/buchi:app/Demo_crAPI:tp/Demo_crAPI"
--local-dir ./
```

Additional Notes

Usage with a proxy

Option 1: Copy proxy CA certificate

The CLI container does not have access to the host's CA certificates. If you are using a proxy with a self-signed certificate, you can copy the CA certificate to `$HOME/.config/configstore/ca-cert.pem` on the host. This directory is mounted as a volume in the CLI container in the alias. The CLI will read this file if it exists and load it into the container's CA certificate store.

Option 2: Use the `--ignore-ssl-verify` option

You can use the `--ignore-ssl-verify` flag with the `levo` command to disable SSL verification for all requests made by the CLI, for example:

```
levo --ignore-ssl-verify test --target-url https://crapi.levo.ai --app-lrn your-app
```

The usage of this option is discouraged unless absolutely necessary.

Adding the `--ignore-ssl-verify` flag after the `test` subcommand, e.g. `levo test --ignore-ssl-verify`, will cause SSL verification to be ignored only for the requests sent to the target server.

Upgrading Levo CLI

Levo CLI is shipped as a Docker image. There are versioned Levo CLI images, and also tagged images with tags `latest` & `stable`. While you can pick the specific version of the image you want, it is recommended that you use the stable image.

Follow instructions below for your platform.

Note: if you update the `levo` alias, please remember to persist it in the shell's profile.

Mac OS

To get the latest stable image type the following in a terminal:

```
docker pull levoai/levo:stable
```

To select a specific version of the image and update the alias (where `x.x.x` is the version):

```
docker pull levoai/levo:<x.x.x>
```

```
alias levo='docker run --rm -v $HOME/.config/configstore:/home/levo/.config/configstore:rw -v
```

```
$HOME/.aws:/home/levo/.aws -v $PWD:/home
```

Linux

To get the latest stable image type the following in a terminal:

```
docker pull levoai/levo:stable
```

To select a specific version of the image and update the alias (where x.x.x is the version):

```
docker pull levoai/levo:<x.x.x>
```

```
alias levo='docker run --rm --add-host=host.docker.internal:`ip route|awk "/docker0/ { print $9 }"` -v $HOME/.config/configstor
```

Windows

To get the latest stable image type the following in a terminal:

```
docker pull levoai/levo:stable
```

To select a specific version of the image and update the alias (where x.x.x is the version):

```
docker pull levoai/levo:<x.x.x>
```

```
Function Launch_Levo {docker run --rm -v
```

```
`${HOME}/.config/configstore:/home/levo/.config/configstore:rw -v ${pwd}:/home/levo/work:rw -
```

Levo CLI (aka Test Runner)

The CLI is the component that executes the autogenerated Test Plans. The CLI can be run on a developer's laptop or integrated into CI/CD environments.

The CLI is packaged as a Docker container, and is available for Mac OS, Windows, and Linux.

Install Levo CLI for Mac OS ([/security-testing/test-laptop/test-mac-os](#))

Install Levo CLI for Linux ([/security-testing/test-laptop/test-linux](#))

Install Levo CLI for Windows ([/security-testing/test-laptop/test-windows](#))

Levo CLI Command Reference ([/security-testing/test-laptop/levo-cli-command-reference](#))

Test on Linux

Prerequisites

Use of Levo CLI requires Docker (min version: 18.03.0)

Linux version that supports Docker

Ensure that you are able to launch and use Docker containers, and network connectivity works

Bash or Bash compatible shell

ip command installed (if missing see notes below)

Instructions to Setup Levo CLI

Open a terminal (bash) window and type the following commands to setup an alias:

```
mkdir -p $HOME/.config/configstore
```

```
alias levo='docker run --rm \
```

```
--add-host=host.docker.internal:`ip route|awk "/docker0/ { print $9 }"` \
```

```
--mount type=bind,source=$HOME/.config/configstore,target=/home/levo/.config/configstore \
```

```
-v $HOME/.aws:/home/levo/.aws \
```

```
-v $PWD:/home/levo/work:rw \
```

```
-e LOCAL_USER_ID=$(id -u) \
```

```
-e LOCAL_GROUP_ID=$(id -g) \
```

```
-e TERM=xterm-256color \
```



```
-ti levoai/levo:stable'
```

Depending on the region your apps are in, you may need to set a different Levo base URL for the satellite.

For example, if the CLI will be used with app.india-1.levo.ai, use the following alias:

```
alias levo='docker run --rm \  
--add-host=host.docker.internal:`ip route|awk "/docker0/ { print $9 }"` \  
--mount type=bind,source=$HOME/.config/configstore,target=/home/levo/.config/configstore \  
-v $HOME/.aws:/home/levo/.aws \  
-v $PWD:/home/levo/work:rw \  
-e LOCAL_USER_ID=$(id -u) \  
-e LOCAL_GROUP_ID=$(id -g) \  
-e LEVO_BASE_URL=https://api.india-1.levo.ai \  
-e TERM=xterm-256color \  
-ti levoai/levo:stable'
```

Now signup and create an account on Levo.ai (<https://Levo.ai>) via the CLI:

```
levo login
```

Notes

Use of sudo with Docker may be required for your installation. Please refer to: Run docker as non-root user

(<https://docs.docker.com/engine/install/linux-postinstall/managedocker-as-a-non-root-user>)

The CLI container mounts your current working directory as R/W. This directory is used to read schema files, and export test plans etc.

Please note that the alias is only available in the current terminal session. If you want to persist this across sessions, you need to persist this in the shell's profile (.bashrc, etc.). Please refer to the shell documentation.

ip command can be installed as shown below:

- Debian: sudo apt install iproute2

- Fedora/CentOS: sudo yum -y install iproute

- Arch: sudo pacman -S iproute2

Upgrade Instructions ([levo-cli-upgrade-instructions.mdlinux](#))

Test on Mac OS

Prerequisites

Use of Levo CLI requires Docker (min version: 18.03.0)

OSX version that supports Docker

Ensure that you are able to launch and use Docker containers, and network connectivity works

Instructions to Setup Levo CLI

Open a terminal (zsh) window and type the following commands to setup an alias:

```
alias levo='docker run --rm \  
-v $HOME/.config/configstore:/home/levo/.config/configstore:rw \  
-v $HOME/.aws:/home/levo/.aws \  

```

```
-v $PWD:/home/levo/work:rw \  
-e TERM=xterm-256color \  
-ti levoai/levo:stable'
```

Depending on the region your apps are in, you may need to set a different Levo base URL for the satellite.

For example, if the CLI will be used with app.india-1.levo.ai, use the following alias:

```
alias levo='docker run --rm \  
-v $HOME/.config/configstore:/home/levo/.config/configstore:rw \  
-v $HOME/.aws:/home/levo/.aws \  
-v $PWD:/home/levo/work:rw \  
-e TERM=xterm-256color \  
-e LEVO_BASE_URL=https://api.india-1.levo.ai \  
-ti levoai/levo:stable'
```

Now signup and create an account on Levo.ai (<https://Levo.ai>) via the CLI:

```
levo login
```

Notes

The CLI container mounts your current working directory as R/W. This directory is used to read schema files, and export test plans etc.

Please note that the alias is only available in the current terminal session. If you want to persist this across sessions, you need to persist this in the shell's profile (.bashrc, .zshrc, etc.). Please refer to the shell documentation.

Upgrade Instructions (levo-cli-upgrade-instructions.mdlinux)

Test on Windows

Prerequisites

Windows 10 OS that supports Docker

Use of Levo CLI requires Docker for Windows (min version: 3.0.0)

Ensure that you are able to launch and use Docker containers, and network connectivity works

Instructions to Setup Levo CLI

Open a powershell window and type the following commands to setup an alias:

```
Function Launch_Levo {docker run --rm -v  
${HOME}/.config/configstore:/home/levo/.config/configstore:rw -v ${pwd}:/home/levo/work:rw -  
Set-Alias -Name levo -Value Launch_Levo
```

Depending on the region your apps are in, you may need to set a different Levo base URL for the satellite.

For example, if the CLI will be used with app.india-1.levo.ai, use the following alias:

```
Function Launch_Levo {docker run --rm -v  
${HOME}/.config/configstore:/home/levo/.config/configstore:rw -v ${pwd}:/home/levo/work:rw -  
Set-Alias -Name levo -Value Launch_Levo
```

Now signup and create an account on Levo.ai (<http://Levo.ai>) via the CLI:

```
levo login
```

Notes

The CLI container mounts your current working directory as R/W. This directory is used to read schema files, and export test plans etc.

Please note that the alias is only available in the current powershell session. If you want to persist this across sessions, you need to persist this in the powershell profile. Please refer to the powershell documentation.

Upgrade Instructions (levo-cli-upgrade-instructions.mdlinux)

Running Tests from Catalog

Levo provides you with the ability to run a variety of tests on your API endpoints using the Run Tests feature.

Navigate to the Applications tab and choose an Application you want to run tests against.

Click on the Run Tests button on the bottom right side of the screen. You can choose to Run On Cloud: The tests will be run on Levo's platform, i.e., the requests to the target server are made by Levo. This means that the application must be exposed via a publicly reachable domain or IP address.

Run OnPrem: The tests will run on your premise. Head to the testrunners (testrunner.md) page to know how to install Testrunners.

Select Runnable Endpoints on the next screen.

Note: You can configure non-runnable endpoints by manually entering sample values for mandatory parameters.

Click on Next and select the categories of test you want to run from and choose from a wide variety of Tests like BOLA, SQLI, CORS, Fuzzing, etc.

Enter a Target URL to run tests against, e.g. <http://crapi.levo.ai> and click on Run Tests to start the tests' execution.

GitHub Action

Levo's security/contract tests can be embedded in quality gates via GitHub Actions (<https://docs.github.com/en/actions>).

Below are examples of embedding Levo's autogenerated tests in GitHub CI/CD via pre-built actions. You have two choices.

Execute Test Plans

Execute Standalone Contract Tests

Execute Test Plans Via Actions

Prerequisites

Forever Free Account on Levo.ai

A runnable Levo Test Plan (</guides/security-testing/concepts/test-plans>)

Action Configuration

The pre-built action for executing Test Plans requires the following configuration settings:

authorization-key : Specify your CLI authorization key here. Refer to [Generating CLI Authorization Keys \(/integrations/common-tasks.mdgenerating-cli-authorizationkeys\)](/integrations/common-tasks.mdgenerating-cli-authorizationkeys) for instructions on fetching your key

organization-id : Specify your Organization ID here. Refer to [Accessing Organization IDs \(/integrations/common-tasks.mdaccessing-organization-ids\)](/integrations/common-tasks.mdaccessing-organization-ids) for instructions on fetching your ID

target : The base URL of the Application/API under test

plan : Specify the LRN of your Levo Test Plan (from the Levo console) here. The LRN uniquely identifies the Test Plan to execute

LRN

base64_env : This is an OPTIONAL setting.

If you are using an environment file

(</guides/security-testing/test-your-app/test-app-security/data-driven/configure-env.yml>) to define authentication details, you add the

contents of the file here in BASE64 encoded format.

report : This is an OPTIONAL setting.

This setting controls the reporting of test results to the Levo Cloud. If you do not want to send test results to the Levo Cloud, set this to false. The default value is true.

cli_extra_args : This is an OPTIONAL setting.

Use this setting to pass extra CLI arguments like headers or the verbosity level. Please use `\"` to escape quotes.

E.g. `cli_extra_args: "-H \"Authorization: Bearer 1234\" --verbosity INFO"`

Here is a sample Test Plan Action with its configuration:

```
- uses: levoai/actions/test-plan@v1-beta
```

with:

Authorization key required to execute the Levo CLI. Please refer to

<https://app.levo.ai/settings/keys> to get your authorization

`authorization-key: <'Specify your CLI authorization key here'>`

The ID of your organization in Levo dashboard. Please refer to

<https://app.levo.ai/settings/organization> to get your organization

`organization-id: <'Specify your organization ID here'>`

The base URL of the Application/API under test.

`target: <'Specify the target base URL here'>`

Test plan LRN. You can get this value from the test plan section in the Levo SaaS console.

`plan: <'Specify your Test Plan's LRN here'>`

[OPTIONAL] Base64 encoded environment file content.

`base64_env: <'The contents of your environment file as a BASE64 encoded string here'>`

[OPTIONAL] If you do not want to report the result of this execution to the Levo cloud, set this value to false. Default: true

report: <true|false>

[OPTIONAL] Use this option to pass extra CLI arguments like headers or verbosity.

Please use `\\` to escape quotes.

E.g. `cli_extra_args: "-H \\\"Authorization: Bearer 1234\\\" --verbosity INFO"`

`cli_extra_args: <"Specify any extra arguments here">`

Job Outputs

This pre-built Action produces the below Outputs

(<https://docs.github.com/en/actions/using-jobs/defining-outputs-for-jobs>), which can be referenced by downstream Actions/Jobs.

outputs:

success:

description: ' of successful test cases'

failed:

description: ' of failed test cases'

skipped:

description: ' of skipped test cases'

Execute Standalone Schema Conformance Tests (aka Contract Tests) Via Actions

Prerequisites

Forever Free Account on Levo.ai

Action Configuration

The pre-built action for executing standalone Schema Conformance Tests requires the following configuration settings:

`authorization-key` : Specify your CLI authorization key here. Refer to [Generating CLI Authorization Keys \(/integrations/common-tasks.mdgenerating-cli-authorizationkeys\)](#) for instructions on fetching your key

`organization-id` : Specify your Organization ID here. Refer to [Accessing Organization IDs \(/integrations/common-tasks.mdaccessing-organization-ids\)](#) for instructions on fetching your ID

`schema` : The URL or file path of the (under test) API's OpenAPI schema (YAML or JSON format)

`target` : The base URL of the Application/API under test

`report` : This is an OPTIONAL setting.

This setting controls the reporting of test results to the Levo Cloud. If you do not want to send test results to the Levo Cloud, set this to false. The default value is true.

`cli_extra_args` : This is an OPTIONAL setting.

Use this setting to pass extra CLI arguments like headers or the verbosity level. Please use `\\` to escape quotes.

E.g. `cli_extra_args: "-H \\\"Authorization: Bearer 1234\\\" --verbosity INFO"`

Here is a sample Schema Conformance Test Action with its configuration:

- uses: levoai/actions/schema-conformance@v1-beta

with:

Authorization key required to execute the Levo CLI. Please refer to <https://app.levo.ai/settings/keys> to get your authorization

authorization-key: <'Specify your CLI authorization key here'>

The ID of your organization in the Levo dashboard. Please refer to <https://app.levo.ai/settings/organization> to get your organi

organization-id: <'Specify your organization ID here'>

The URL or file path of the API's OpenAPI schema.

schema: '<URL of schema|File path of schema>'

The base URL of the Application/API under test.

target: '<Specify the target base URL here>'

[OPTIONAL] If you do not want to report the result of this execution to the Levo cloud, set this value to false. Default: true

report: <true|false>

[OPTIONAL] Use this option to pass extra CLI arguments like headers or verbosity.

Please use `\\` to escape quotes.

E.g. cli_extra_args: "-H \\\"Authorization: Bearer 1234\\\" --verbosity INFO"

cli_extra_args: <"Specify any extra arguments here">

Job Outputs

This pre-built Action produces the below Outputs

(<https://docs.github.com/en/actions/using-jobs/defining-outputs-for-jobs>), which can be referenced by downstream Actions/Jobs.

outputs:

success:

description: ' of successful test cases'

failed:

description: ' of failed test cases'

Jenkins Plugin

Levo's security/contract tests can be embedded in Jenkins quality gates via Levo's Jenkins plugin (<https://plugins.jenkins.io/levo/>).

Prerequisites

Forever Free Account on Levo.ai

A runnable Levo Test Plan (</guides/security-testing/concepts/test-plans/test-plans.md>)

A Levo CLI Authorization Key. Refer to instructions here

(</integrations/common-tasks.md#generating-cli-authorization-keys>)

Installation

Below are the installation options:

Using the GUI (<https://www.jenkins.io/doc/book/managing/plugins/from-the-web-ui>): From your Jenkins dashboard navigate to Manage Jenkins > Manage Plugins and select the Available tab. Locate the plugin by searching for levo, and install it

Using the CLI tool (<https://github.com/jenkinsci/plugin-installation-manager-tool>):

```
jenkins-plugin-cli --plugins levo:33.vc34b_8f81dc9a
```

Using direct upload (<https://www.jenkins.io/doc/book/managing/plugins/advanced-installation>).

Download one of the releases (<https://plugins.jenkins.io/levo/releases>) and upload it to your Jenkins instance

Running Levo Test Plans Via Freestyle Projects (Jobs)

Follow the steps below to create a build job, that executes a Levo Test Plan against your build target.

1. Create a Freestyle project and name it appropriately
2. Optionally configure the General, Build Triggers, and Build Environment sections based on your preferences

3. Add Levo Test Plan build step to Build Steps

Jenkins Build Step

4. Configure the build step as shown below:

Levo Build Step Config

- i. Test Plan

Copy the LRN of your Levo Test Plan (from the Levo console), and paste it in the Test Plan section. The LRN uniquely identifies the Test Plan to execute.

LRN

- ii. Target

Specify the API target that needs to be tested here. This is usually the base URL of your API.

- iii. Extra CLI Arguments (optional)

Please refer to the CLI Command Reference

(</security-testing/test-laptop/levo-cli-command-reference.md>). Specify any optional arguments based on your preferences here.

- iv. Generate JUnit Reports

If you would to generate build results (Test Plan execution results) in standard JUnit format (<https://www.ibm.com/docs/en/developer-for-zos/14.1.0?topic=formats-junit-xmlformat>), check the box titled Generate JUnit Reports.

- v. Levo Credentials

You will need to specify the Levo CLI Authorization Key here. The Jenkins Credentials Provider Plugin (<https://plugins.jenkins.io/credentials/>) is utilized to securely store the API key.

Jenkins Credentials

Click on the "Add" button next to the credentials dropdown

Select your domain

Select "Levo CLI Credentials" for Kind

Select your Scope based on your preferences

Enter your Organization ID in the Organization Id text box. Refer to [Accessing Organization IDs \(/integrations/common-tasks.md#accessing-organization-ids\)](/integrations/common-tasks.md#accessing-organization-ids) for instructions on fetching your ID

Enter your CLI authorization key in the CLI Authorization Key textbox. Refer to [Generating CLI Authorization Keys \(/integrations/common-tasks.md#generating-cli-authorization-keys\)](/integrations/common-tasks.md#generating-cli-authorization-keys) for

instructions on fetching your key

Click Add to save the credentials

Finally select the credential you just added

vi. Environment Secret Text

If you are using an environment file

(/guides/security-testing/test-your-app/test-app-security/data-driven/configure-env.yml) to define authentication details, you add those

details as a secret file here.

Environment File for Jenkins

Click on the "Add" button next to the Environment Secret Text dropdown

Select your domain

Select "Secret file" for Kind

Select your Scope based on your preferences

Import your environment.yml file using the file chooser dialog

Click Add to save the environment.yml as a secret file

Now select the secret file you just added

5. Optionally configure Post-build Actions

6. Save/Apply your Freestyle Project configuration. You are done!

Test Runner

Levo provides you with the ability to run security tests on your Application endpoints.

With Levo you can run security tests

on Cloud

on Premises

To run security tests we need three things

A target service URL which should be reachable

Valid configuration for authenticated endpoints

Valid API endpoint parameters

If the target service is publicly reachable, tests can be run directly from Levo Cloud

But if you want to test some internal services which are not publicly reachable, you can use the testrunner.

The testrunner can be installed in your premise.

Once you start security tests from the UI, the testrunner will pull those tests and execute them in your premise. This way if you want to test internal APIs, you can install our testrunner service.

Installation

You can install the testrunner

via helm in Kubernetes environment

via docker

Prerequisites

You will need an authorization token and the organization-id for which you want to run security

tests

Either helm or docker installed based on the installation process.

Install testrunner via helm on Kubernetes

To get the authorization token and organization-id, follow instructions here

Add Levo Helm repo

```
helm repo add levoai https://levoai.github.io/helm-charts/
```

Run following command to install testrunner helm chart

```
helm install \
```

```
--set key="auth-key" \
```

```
--set orgId="organization id" \
```

```
--set levoBaseUrl="https://api.levo.ai" \
```

```
testrunner levoai/testrunner
```

Depending on the region your apps are in, you may need to set a different Levo base URL.

For example, if the testrunner will be used with app.india-1.levo.ai, use the following alias:

```
helm install \
```

```
--set key="auth-key" \
```

```
--set orgId="organization id" \
```

```
--set levoBaseUrl="https://api.india-1.levo.ai" \
```

```
testrunner levoai/testrunner
```

Install testrunner via Docker

To get the authorization token and organization-id, follow instructions here

If you have docker running on your machine, you can start the testrunner with a simple docker run command

```
mkdir -p $HOME/.config/configstore
```

```
alias levo='docker run --rm \
```

```
--add-host=host.docker.internal:`ip route|awk "/docker0/ { print $9 }"` \
```

```
--mount type=bind,source=$HOME/.config/configstore,target=/home/levo/.config/configstore \
```

```
-v $HOME/.aws:/home/levo/.aws \
```

```
-v $PWD:/home/levo/work:rw \
```

```
-e LOCAL_USER_ID=$(id -u) \
```

```
-e LOCAL_GROUP_ID=$(id -g) \
```

```
-e LEVO_BASE_URL=https://api.levo.ai \
```

```
-e TERM=xterm-256color \
```

```
-ti levoai/levo:stable'
```

```
levo start --key "auth-key" --organization "orgId"
```

Depending on the region your apps are in, you may need to set a different Levo base URL.

For example, if the testrunner will be used with app.india-1.levo.ai, use the following alias:

```
alias levo='docker run --rm \
```

```
--add-host=host.docker.internal:`ip route|awk "/docker0/ { print $9 }"` \
```

```
--mount type=bind,source=$HOME/.config/configstore,target=/home/levo/.config/configstore \
```

```
-v $HOME/.aws:/home/levo/.aws \
```

```
-v $PWD:/home/levo/work:rw \
```

```
-e LOCAL_USER_ID=$(id -u) \  
-e LOCAL_GROUP_ID=$(id -g) \  
-e LEVO_BASE_URL=https://api.india-1.levo.ai \  
-e TERM=xterm-256color \  
-ti levoai/levo:stable'
```

Get Authorization token and Organization ID

Login (<https://app.levo.ai/login>) to Levo.ai

To get the authorization token

Click on User Settings

Click on Keys on the left navigation panel

Click on Get CLI Authorization Key

To get the Organization ID

Click on User Settings

Click on Organizations on the left navigation panel

Click on Copy for whichever organization you want to run tests for

If you are an India Customer

India Login (<https://app.india-1.levo.ai/login>) for Levo.ai

Install Code Analysis Tools

Prerequisites

Docker is installed on your machine.

Ensure that you are able to launch and use Docker containers, and network connectivity works.

api.levo.ai is reachable from the host machine

Instructions to setup code-scanning

Install Levo CLI (</security-testing/test-laptop>), which contains the commands to scan code

Once the CLI is installed, you can

Scan your code and identify REST API endpoints.

Scan the project directory to look for existing OpenAPI/Swagger specs.

Scan Code and identify REST API Endpoints

Login to Levo CLI

```
levo login
```

Enter the CLI authorization key and select the organization.

Once logged in, go to the project directory where you want to run code-scanning

```
cd <your_project_directory>
```

Inside the project directory, run the below CLI command

```
levo scan code \  

```

```
--dir <relative path to directory you wish to scan> \  

```

```
--app-name <name of the app you wish to see on Dashboard> \  

```

```
--env-name <the environment to which your app should belong> \  

```

```
--language <programming language used in repository, default is java>
```

In the `--dir` option, you can specify the relative subdirectory path (`./path/to/sub-directory`) if you want to scan only a part of the project, or simply DOT (`.`) for the

current directory.

Use the --help option to know the list of available options

If there are REST endpoints in the code, they will be imported to the Levo Dashboard, under the given app-name.

Scan project directory to fetch and import OpenAPI/Swagger specs

Login to Levo CLI

levo login

Enter the CLI authorization key and select the organization.

Once logged in, go to the project directory where you want to scan for openAPI specs.

```
cd <your_project_directory>
```

Inside the project directory, run the below CLI command

```
levo scan schema \
```

```
--dir <relative path to directory you want to scan>
```

```
\
```

```
--env-name <the environment to which your app should belong>
```

In the --dir option, you can specify the relative subdirectory path (./path/to/sub-directory) if you want to scan only a part of the project, or simply DOT (.) for the current directory.

Use the --help option to know the list of available options

If there are OpenAPI/Swagger specs in the project directory, they will be imported to Levo Dashboard.

The App Name will be the same as the title of the OpenAPI/Swagger spec.

Github Action

Prerequisites

An account on Levo.ai

An application code repository on GitHub (Currently Java and Python is supported)

Action Configuration

The pre-built action for executing Scan Code requires the following configuration settings:

authorization-key : Specify your CLI authorization key here. Refer to [Generating CLI Authorization Keys \(/integrations/common-tasks.mdgenerating-cli-authorizationkeys\)](#) for instructions on fetching your key

organization-id : Specify your Organization ID here. Refer to [Accessing Organization IDs \(/integrations/common-tasks.mdaccessing-organization-ids\)](#) for instructions on fetching your ID

saas-url : The URL of the Levo SaaS instance. Default value is <https://api.levo.ai>. For India, use <https://api.india-1.levo.ai>.

app-name : The name of the application you want to see on the Levo Dashboard

env-name : This is an OPTIONAL setting. The environment to which your app should belong. Default value is staging.

Here is a sample Scan Code Action with its configuration:

- name: Levo Scan Repo
uses: levoai/actions/scan@v2.3.0
with:

Authorization key required to execute the Levo CLI. Please refer to <https://app.levo.ai/settings/keys> to get your authorization

authorization-key: <'Specify your CLI authorization key here'>

The ID of your organization in Levo dashboard. Please refer to <https://app.levo.ai/settings/organization> to get your organization

organization-id: <'Specify your organization ID here'>

[OPTIONAL] The environment to which your app should belong. Default: staging.
saas-url: "https://api.dev.levo.ai"

The name of the application you want to see on the Levo Dashboard.

app-name: <'Application Name here'>

[OPTIONAL] The environment to which your app should belong. Default: staging.

env-name: <'Environment Name here'>

Job Outputs

This pre-built Action produces the below Outputs

(<https://docs.github.com/en/actions/using-jobs/defining-outputs-for-jobs>), which can be referenced by downstream Actions/Jobs.

outputs:

scan-success: <'true/false'>

Install Log Parsing Sensors

Access Logs Based Instrumentation

i. Pre-requisites

Satellite has been successfully installed.

You have noted down the Satellite's hostname:port or ip-address:port information.

The Satellite is reachable (via HTTP/s) from the location where you are going to install the log-parser.

ii. Pick an Application Name

Auto discovered API endpoints and their OpenAPI specifications are shown in the API Catalog (</guides/security-testing/concepts/api-catalog>), grouped under an application name.

The application name helps segregate and group API endpoints from different environments, similar to how file folders work in an operating system.

Pick a descriptive name which will be used in the subsequent step below. For example:

my-test-app.

iii. Follow instructions for your platform

Follow instructions for your specific platform/method below:

Install on Linux host via Docker

Install on Linux host via Docker

Prerequisites

Docker Engine version 18.03.0 and above

1. Install Log Parser

If you are installing the Satellite and Log Parser on the same Linux host, please do NOT use localhost as the hostname below. Use the Linux host's IP address, or domain name instead. This is required as the Log Parser runs inside a Docker container, and localhost resolves to the Log Parser container's IP address, instead of the Linux host.

Replace '<SATELLITE_URL>' with the values you noted down from the Satellite install

Specify below the 'APP_NAME'. Do not quote the 'APP_NAME'.

Environment Name is optional. If not specified, it defaults to 'staging'

```
docker run --rm -d --name=log-parser \  
-v ./logs:/mnt/levo/logs \  
-e LEVO_SATELLITE_URL=<LEVO_SATELLITE_URL> \  
-e LEVOAI_ORG_ID=<LEVOAI_ORG_ID> \  
-e APP_NAME=<APP_NAME> \  
-e ENV_NAME=<ENV_NAME> \  
levoai/log-parser
```

NOTE:

The default address for the satellite in Docker-based Log Parser installations is <https://satellite.levo.ai>.

In case of levo hosted satellite, it is necessary that you must also specify an organization ID (LEVOAI_ORG_ID).

If you wish, you may also host the Satellite yourself and specify the address of the satellite to direct the Log Parser's data to it.

2. Verify connectivity with Satellite

Execute the following command to check for connectivity health:

Please specify the actual container name for log-parser below
docker logs log-parser | grep "starting fluentd"

If connectivity is healthy, you should see output similar to below.

```
2024-02-22 01:27:06 +0000 [info]: starting fluentd-1.16.3 pid=7 ruby="3.2.2"  
2024-02-22 01:27:06 +0000 [info]: 0 starting fluentd worker pid=16 ppid=7 worker=0  
2024-02-22 01:27:06.831947051 +0000 fluent.info:  
{\"pid\":16,\"ppid\":7,\"worker\":0,\"message\":\"starting fluentd worker pid=16 ppid=7 worke  
Please proceed to the next step, if there are no errors.
```

description: Continuous API Security Assurance. Observe,
detect, protect, early! slug: /

Welcome to Levo!

Levo comprises of two components, a Sensor which runs alongside your application workloads, and a Satellite. The Sensor captures live traffic from your environment and sends it to the Satellite for processing.

Levo can host the Satellite for you (reach out to support@levo.ai (mailto:support@levo.ai)), or you can host it yourself.

Signup with your enterprise email (<https://app.levo.ai/signup>)

OS Platform Compatibility Check (</guides/general/os-compat-check>)

Install Satellite (</install-satellite>)

Install Traffic Capture Sensors (</install-traffic-capture-sensors>)