OpenTelemetry Collector Fuzzing Audit

In collaboration with the Cloud Native Computing Foundation (CNCF) and the OpenTelemetry Collector maintainers

Adam Korczynski & David Korczynski, Ada Logics

8th November 2024

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

Contents

About Ada Logics **1**

CNCF security and fuzzing audits **3**

Executive summary **4**

Strategic recommendations **5**
- Ensure continuity . . . 5
- Ensure that complex code paths are tested . . . 5
- Consider CI fuzzing . . . 5

OpenTelemetry Collector fuzzing **6**
- Architecture . . . 6

OpenTelemetry Collector Fuzzers **9**
- OpenTelemetry Collector fuzzers . . . 9
- OpenTelemetry Collector Contrib fuzzers . . . 10

CNCF security and fuzzing audits

This report details a fuzzing audit commissioned by the CNCF for the OpenTelemetry project. Fuzzing is an integral part of ensuring the security, stability, and integrity of CNCF's ecosystem. CNCF continues to use state-of-the-art techniques to secure its projects and carry out manual audits. Over the last few years, CNCF has invested in security audits, fuzzing, and software supply chain security, which have helped proactively discover and fix hundreds of issues.

Fuzzing is an effective technique for finding security and reliability issues in software. CNCF's efforts have enabled more than twenty CNCF projects to add fuzz testing through a series of dedicated fuzzing audits. More than 350 bugs have been found through fuzzing of CNCF projects. The fuzzing efforts of CNCF have focused on enabling continuous fuzzing of projects to ensure continued security analysis, which is done by way of the open-source fuzzing project OSS-Fuzz.

CNCF continues to work in this space and will further increase investment to improve security across its projects and community. Future work focuses on integrating fuzzing into more projects, enabling sustainable fuzzer maintenance, increasing maintainer involvement and enabling fuzzing to find more vulnerabilities in memory-safe languages. Maintainers who are interested in getting fuzzing integrated into their projects or have questions about fuzzing are encouraged to visit the dedicated CNCF-fuzzing repository https://github.com/cncf/cncf-fuzzing where questions and queries are welcome.

Executive summary

In this engagement, Ada Logics worked on setting up a fuzzing suite for OpenTelemetrys collector. At the time this engagement started, OpenTelemetrys Collector did not have any fuzz testing, and the goal of this fuzzing audit was to build the infrastructure that would support integration into OSS-Fuzz and then improve the fuzzing efforts continuously. The integration into OSS-Fuzz allows the OpenTelemetry Collectors fuzzers to run continuously during and after our work during this audit has concluded. This is important for OpenTelemetry Collector, since it will continue to be fuzz tested as the project changes over time.

We carried out this work in collaboration with the OpenTelemetry maintainers, which allowed us to efficiently get the fuzz tests checked into the Openelemetry Collectors source tree. This is arguably the best way to store a projects fuzzers in general, as the maintainers have the highest degree of control of the fuzzing suite.

OpenTelemetry is a comprehensive ecosystem that consists of many different components that are implemented in many different languages. In collaboration with the OpenTelemetry security team, we made it a goal to create a reference implementation of an ideal fuzzing setup for the OpenTelemetry Collector, which other subprojects in the OpenTelemetry ecosystem could adopt at a later time. As such, this report details the work we did and gives an insight into the actionable steps other OpenTelemetry subprojects can take to adopt fuzzing at an ideal state. Below in this report, we have a section called "Strategic recommendations"; This section includes practical guidelines for other subprojects to adopt fuzzing.

Strategic recommendations

This section is intended both for the Collector project and as a reference for other sub-projects in the OpenTelemetry ecosystem.

Ensure continuity

In this engagement, we integrated the OpenTelemetry Collector into OSS-Fuzz. We did that at the onset of the engagement to build the fuzzing efforts around a central component of continuity. Continuity should be a central component of any fuzzing setup, as fuzzing takes time to create confidence in the absence of bugs in the code and to find deep bugs in the target code base. As such, we recommend that the OpenTelemetry Collector project ensures continuity by maintaining the fuzzers so that they don't break with changes in the source code. At the same time, we recommend that other efforts in the OpenTelemetry ecosystem likewise start with continuity and are built around OSS-Fuzz. OpenTelemetry is now integrated into OSS-Fuzz, and other OpenTelemetry projects can build upon that integration.

Ensure that complex code paths are tested

Fuzzing performs well against code paths that carry out complex processing routines. OpenTelemetry Collector processes raw data into structured data types in multiple places, and we've applied fuzzing to these cases. For other projects in the OpenTelemetry ecosystem, we recommend starting with test coverage for complex APIs and methods.

Consider CI fuzzing

OSS-Fuzz regularly builds the OpenTelemetry Collector fuzzers against the latest `main` branch. Over time, OSS-Fuzz will accumulate a large corpus and will start each fuzz cycle by reaching far into the call tree of the fuzzers' entry points. As such, if bugs are added to the code base, the fuzzers are likely to find them before the code base evolves much further. However, it can be beneficial to test the code even before it is merged by testing it in the subprojects CI pipeline. We recommend OSS-Fuzz's CIFuzz, which uses its accumulated corpus: https://google.github.io/oss-fuzz/getting-started/continuous-integration.

OpenTelemetry Collector fuzzing

In this section, we present details on the OpenTelemetry Collector fuzzing setup. We first present a high-level view of the overall fuzzing architecture and how it supports running the fuzzers continuously. We then enumerate the fuzzers we wrote during the audit, and finally, we go into detail regarding the crashes that the fuzzers found.

OpenTelemetry Collector is a standalone agent that users typically deploy on their host next to their application. It is vendor and platform agnostic. The OpenTelemetry Collector is responsible for collecting telemetry data from the users application and supports communication via the OpenTelemetrys instrumentation, APIs or SDKs. It allows users to build pipelines for their telemetry data; In other words, users can pipe their telemetry data to services of their choice. The OpenTelemetry Collector splits this into three high-level components: 1) Receivers that receive data from data sources, 2) Processors can modify the data and pipe it to the desired exporter, and finally 3) Exporters that send the telemetry data to a destination.

OpenTelemetry Collector is a standalone agent that users typically deploy on their host next to their application. It is vendor and platform-agnostic. The OpenTelemetry Collector collects telemetry data from the user's application and supports communication via the OpenTelemetry instrumentation, APIs or SDKs. It allows users to build pipelines for their telemetry data; In other words, users can pipe their telemetry data to services of their choice. The OpenTelemetry Collector splits this into three high-level components: 1) Receivers that receive data from data sources, 2) Processors can modify the data and pipe it to the desired exporter, and finally, 3) Exporters that send the telemetry data to a destination.

The OpenTelemetry Collector has a repository at https://github.com/open-telemetry/opentelemetry-collector, which holds the core functionality needed to use the collector. In addition, the OpenTelemetry community maintains a repository where the community maintains receivers, processors and exporters for specific providers at https://github.com/open-telemetry/opentelemetry-collector-contrib. In this fuzzing audit, we added fuzzing to both repositories, and we added all fuzzers to the OpenTelemetry Collectors OSS-Fuzz integration.

Architecture

A central component in OpenTelemetry Collectors' fuzzing infrastructure is its integration into OSS-Fuzz. The OpenTelemetry Collector source code and its fuzz tests are the two core elements that OSS-Fuzz uses to fuzz OpenTelemetry Collector. The following diagram shows how OSS-Fuzz uses these two elements and what happens when an issue is found/fixed.

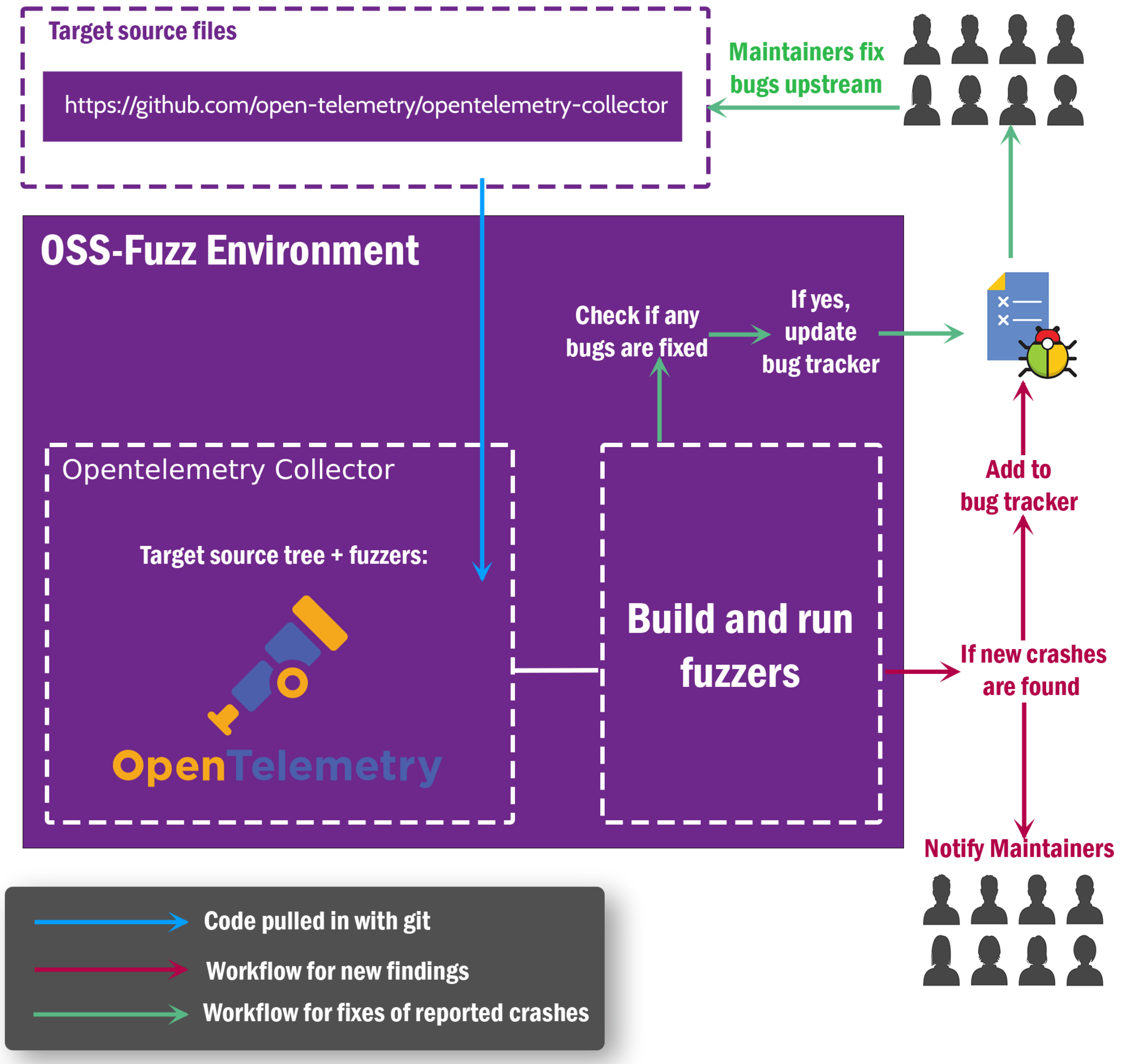
The current OSS-Fuzz set up builds the fuzzers by cloning the upstream OpenTelemetry Collector GitHub repositories to get the latest OpenTelemetry Collector source code and its fuzz tests. OSS-Fuzz then builds the fuzzers against the cloned OpenTelemetry Collector source code, which ensures that the fuzzers always run against the latest OpenTelemetry Collector commit.

This build cycle happens daily, and OSS-Fuzz will verify whether any existing bugs have been fixed. If OSS-fuzz finds that any bugs have been fixed, OSS-Fuzz marks the crashes as fixed in the Monorail bug tracker and notifies maintainers.

In each fuzzing iteration, OSS-Fuzz uses its corpus accumulated from previous fuzz runs. If OSS-Fuzz detects any crashes when running the fuzzers, OSS-Fuzz performs the following actions:

1. A detailed crash report is created.
2. An issue in the Monorail bug tracker is created.
3. An email is sent to maintainers with links to the report and relevant entries in the bug tracker.

OSS-Fuzz has a 90-day disclosure policy, meaning that a bug becomes public in the bug tracker if it has not been fixed. The detailed report is never made public. The OpenTelemetry Collector maintainers will fix issues upstream, and OSS-Fuzz will pull the latest OpenTelemetry Collector main branch the next time it performs a fuzz run and verify that a given issue has been fixed.


Target source files
https://github.com/open-telemetry/opentelemetry-collector
Maintainers fix bugs upstream
OSS-Fuzz Environment
Check if any bugs are fixed
If yes, update bug tracker
Opentelemetry Collector
Target source tree + fuzzers:
OpenTelemetry
Build and run fuzzers
Add to bug tracker
If new crashes are found
Notify Maintainers
Code pulled in with git
Workflow for new findings
Workflow for fixes of reported crashes

OpenTelemetry Collector Fuzzers

In this section, we present an overview of the OpenTelemetry Collector fuzzers and the parts of OpenTelemetry Collector they test. We wrote 22 fuzz tests for the core OpenTelemetry Collector project and 17 fuzz tests for the OpenTelemetry Collector Contrib project.

The first table contains the fuzzers for the core OpenTelemetry Collector project with the subdirectory where they are found. The second table contains the fuzzers for the OpenTelemetry Collector Contrib project.

OpenTelemetry Collector fuzzers

Repository: https://github.com/open-telemetry/opentelemetry-collector

#	Name	OpenTelemetry Collector Subdir
1	FuzzReceiverHandlers	`receiver/otlpreceiver`
2	FuzzUnmarshalMetrics	`pdata/pmetric`
3	FuzzRequestUnmarshalJSON	`pdata/pmetric/pmetricotlp`
4	FuzzResponseUnmarshalJSON	`pdata/pmetric/pmetricotlp`
5	FuzzRequestUnmarshalProto	`pdata/pmetric/pmetricotlp`
6	FuzzResponseUnmarshalProto	`pdata/pmetric/pmetricotlp`
7	FuzzRequestUnmarshalJSON	`pdata/ptrace/ptraceotlp`
8	FuzzResponseUnmarshalJSON	`pdata/ptrace/ptraceotlp`
9	FuzzRequestUnmarshalProto	`pdata/ptrace/ptraceotlp`
10	FuzzResponseUnmarshalProto	`pdata/ptrace/ptraceotlp`
11	FuzzUnmarshalJSONTraces	`pdata/ptrace`
12	FuzzUnmarshalPBTraces	`pdata/ptrace`
13	FuzzUnmarshalJsonLogs	`pdata/plog`
14	FuzzUnmarshalPBLogs	`pdata/plog`
15	FuzzRequestUnmarshalJSON	`pdata/plog/plogotlp`
16	FuzzResponseUnmarshalJSON	`pdata/plog/plogotlp`
17	FuzzRequestUnmarshalProto	`pdata/plog/plogotlp`

#	Name	OpenTelemetry Collector Subdir
18	FuzzResponseUnmarshalProto	`pdata/plog/plogotlp`
19	FuzzRequestUnmarshalJSON	`pdata/pprofile/pprofileotlp`
20	FuzzResponseUnmarshalJSON	`pdata/pprofile/pprofileotlp`
21	FuzzRequestUnmarshalProto	`pdata/pprofile/pprofileotlp`
22	FuzzResponseUnmarshalProto	`pdata/pprofile/pprofileotlp`

OpenTelemetry Collector Contrib fuzzers

Repository: https://github.com/open-telemetry/opentelemetry-collector-contrib

#	Name	OpenTelemetry Collector Contrib Subdir
1	FuzzProcessTraces	`processor/groupbyattrsprocessor`
2	FuzzProcessLogs	`processor/groupbyattrsprocessor`
3	FuzzProcessMetrics	`processor/groupbyattrsprocessor`
4	FuzzConsumeLogs	`processor/logdedupprocessor`
5	FuzzConsumeTraces	`processor/probabilisticsamplerprocessor`
6	FuzzConsumeLogs	`processor/probabilisticsamplerprocessor`
7	FuzzProcessTraces	`processor/sumologicprocessor`
8	FuzzProcessLogs	`processor/sumologicprocessor`
9	FuzzProcessMetrics	`processor/sumologicprocessor`
10	FuzzConsumeTraces	`processor/tailsamplingprocessor`
11	FuzzHandleReq	`receiver/cloudflarereceiver`
12	FuzzParseRequest	`receiver/lokireceiver/internal`
13	FuzzHandleReq	`receiver/mongodbatlasreceiver`
14	FuzzParseTraceV2Request	`receiver/sapmreceiver`
15	FuzzHandleDatapointReq	`receiver/signalfxreceiver`
16	FuzzHandleRawReq	`receiver/splunkhecreceiver`
17	FuzzHandleReq	`receiver/webhookeventreceiver`