Eclipse KUKSA audit

Technical Report

Reference 23-11-1425-REP
Version 1.2
Date 2024/01/15

Quarkslab

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

Legal Notice

This report reflects the work and results obtained within the duration of the audit on the specified scope (see Section 4.2) and as agreed between the OSTIF, Eclipse Foundation, Eclipse KUKSA committers, and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure the code is bug or vulnerability free.

1. Project Information

Document history			
Version	**Date**	**Details**	**Authors**
1.2	2024/01/15	Minor fixes	Pauline Sauder
1.1	2024/01/09	Minor fixes	Laurent Laubin
1.0	2023/12/14	Initial Version	Damien Aumaitre Victor Houal Laurent Laubin Madigan Lebreton

Quarkslab		
Contact	**Role**	**Contact Address**
Frédéric Raynal	CEO	fraynal@quarkslab.com
Ramtine Tofighi Shirazi	Project Manager	mrtofighishirazi@quarkslab.com
Pauline Sauder	Project Manager	psauder@quarkslab.com
Damien Aumaitre	R&D Engineer	daumaitre@quarkslab.com
Victor Houal	R&D Engineer Apprentice	vhoual@quarkslab.com
Laurent Laubin	R&D Engineer	llaubin@quarkslab.com
Madigan Lebreton	R&D Engineer	mlebreton@quarkslab.com

OSTIF		
Contact	**Role**	**Contact Address**
Amir Montazery	Managing Director	amir@ostif.org
Derek Zimmer	Executive Director	derek@ostif.org

Eclipse Foundation		
Contact	**Role**	**Contact Address**
Mikaël Barbero	Head of Security	mikael.barbero@eclipse-foundation.org
Marta Rybczynska	Security Team	marta.rybczynska@eclipse-foundation.org

Eclipse KUKSA committers		
Contact	**Role**	**Contact Address**
Erik Jaegervall	Senior Software Architect	erik.jaegervall@se.bosch.com
Sven Erik Jeroschewski	Software Developer	svenerik.jeroschewski@bosch.com

2. Executive summary

Quarkslab audited the Eclipse KUKSA project. The goal of the audit was to assist Eclipse KUKSA committers to increase the security posture of the project. The project codebase was assessed on a specific scope defined with the Eclipse Foundation and OSTIF. This assessment was achieved during an allocated amount of time in order to find issues and vulnerabilities, using automated tools, fuzzing and manual review.

2.1 Disclaimer

This report reflects the work and results obtained within the duration of the audit on the specified scope and as agreed between the OSTIF, Eclipse Foundation, Eclipse KUKSA committers, and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure the code is bug or vulnerability free.

2.2 Findings summary

ID	Description	Perimeter
HIGH-1	A feeder can crash the databroker.	Databroker Register-Datapoints endpoint
HIGH-2	Any user can crash the databroker	Databroker subscriptions
MED-1	Recent values can be overwritten with old values.	Databroker entries
LOW-1	Databroker-specific entries can be modified remotely.	Databroker version
LOW-2	Client can subscribe to unavailable scope and waits for data that will never be sent.	Databroker subscriptions
LOW-3	An expired token can leak information about entries update timestamp.	Databroker subscriptions
LOW-4	Entries metadata can be read by every client and feeder.	Databroker entries metadata
LOW-5	Malicious JWT access token can crash a thread of the databroker	Databroker JWT handling
LOW-6	A malicious Protobuf error message can trigger an unhandled error	Python SDK - Protobuf messages parsing
LOW-7	Datapoint.from_message() does not check if no value is provided.	Python SDK - Protobuf messages parsing
LOW-8	DataEntry.From_message() and ValueRestriction.	Python SDK - Protobuf messages parsing

LOW-9	ValueRestriction without type field	Python SDK - Protobuf messages parsing
LOW-10	No check on timestamp uint value	Python SDK - Protobuf messages parsing
INFO-1	A vulnerability is known for the `atty` crate on Windows.	Databroker dependencies
INFO-2	Multiple databroker dependencies are out of date.	Databroker dependencies
INFO-3	Subscription channels remain open after token expiration	Databroker subscriptions
INFO-4	UpdateDatapoints request format is not unified with other endpoints.	Databroker UpdateDatapoints endpoint
INFO-5	`debug_assert` in `executor.rs`	Databroker subscriptions
INFO-6	Slow input in `glob::to_regex()`	Databroker

Severity: critical, high, medium, low, info

2.3 Recommendations and action plan

ID	Recommendation	Perimeter
HIGH-1	Set an upper-bound limit on the number of entries that the databroker can handle.	Databroker RegisterDatapoints endpoint
HIGH-2	We think that having a full-fledged SQL parser as a core dependency is not required given this use case. Maybe it should be considered to move to a custom DSL tailored for the task. It will reduce the attack surface.	Databroker subscriptions
MED-1	A timestamp check must be done before updating an entry value.	Databroker entries
LOW-1	Deny the write access on the three databroker-specific entries.	Databroker version
LOW-2	Return the error to the subscriber and delete the `TODO` comment in broker.rs#L835-L839.	Databroker subscriptions
LOW-3	In brokers.rs#L805-L845, consider adding an expiration check before notifying the client. If the token is expired, consider closing the connection.	Databroker subscriptions
LOW-4	Restrict the metadata access according to the JWT permissions scope.	Databroker entries metadata

LOW-5	Consider using a Rust-safe addition.	Databroker JWT handling
LOW-6	Check the presence of the attribute before using it.	Python SDK - Protobuf messages parsing
LOW-7	Check the field value is not `None` before using it	Python SDK - Protobuf messages parsing
LOW-8	Check the content type when dealing with `Oneof` Field	Python SDK - Protobuf messages parsing
LOW-9	Check the field value is not `None` before using it	Python SDK - Protobuf messages parsing
LOW-10	Catch exception on timestamp parsing	Python SDK - Protobuf messages parsing
INFO-1	To our knowledge, the databroker is not meant to be run on Windows, moreover this dependency is only used for testing purposes.	Databroker dependencies
INFO-2	Consider upgrading to the last up-to-date version for each dependency.	Databroker dependencies
INFO-3	Close the communication channel when the token is expired.	Databroker subscriptions
INFO-4	Consider unifying the gRPC endpoint request format to retrieve databroker entries.	Databroker UpdateDatapoints endpoint
INFO-5	Remove the `debug_assert` and the `todo` statements.	Databroker subscriptions
INFO-6	As a rule of thumb, regular expressions should not be controlled from the outside. We recommend to move to a custom parser to handle the parsing of the scope.	Databroker

Severity: critical, high, medium, low, info

3. Reading Guide

This reading guide describes the different sections present in this report and gives some insights about the information contained in each of them and how to interpret it.

3.1 Executive summary

The executive summary presents the results of the assessment in a non-technical way, summarizing all the findings and explaining the associated risks. For each vulnerability, a severity level is provided as well as a description and one or more mitigations, as shown below.

ID	Description	Category
CRIT 1	Description of vulnerability #1	Injection
HIGH 4	Description of vulnerability #4	Remote code execution
MED 3	Description of vulnerability #3	Denial of Service
LOW 2	Description of vulnerability #2	Information leak

Severity: critical, high, medium, low, info

Each vulnerability is identified throughout this document by a unique identifier `VULN-<ID>`, where *ID* is a number. Every vulnerability identifier present in the vulnerabilities summary table is a clickable link that leads to the corresponding technical analysis that details how it was found (and exploited if it was the case). Severity levels are explained in section 3.4.

The executive summary also provides an action plan with a focus on the identified *quick wins*, some specific mitigations that would drastically improve the security of the assessed system.

3.2 Introduction

The introduction recalls the context in which the assignment has been performed. It details the objectives set by the customer, the target of evaluation and the expected deliverables.

It also recalls the agreed scope of work including the different assets that must be assessed, the type of tests the auditors are allowed to perform as well as the type of tests or actions that are forbidden regarding the context of the assessment.

Last, the final planning of the assignment is detailed in this section recalling when the assessment started and ended as well as the different key steps and meetings dates.

3.3 Methodology

The introduction is followed by this section detailing the methodology followed by the evaluators and the different steps of the assessment. This section also details the choices made by the auditors during the execution of the assessment and the reasons why they made them.

3.4 Metrics definition

This report uses specific metrics to rate the severity, impact and exploitability of each identified vulnerability.

3.4.1 Impact

The impact is assessed regarding the information an attacker can access by exploiting a vulnerability but also the operational impact such an attack can have. The following table summarizes the different levels of impact we are using in this report and their meanings in terms of information access and availability.

Critical	Allows a total compromise of the assessed system, allowing an attacker to read or modify the data stored in the system as well as altering its behavior.
High	Allows an attacker to impact significantly one or more components, giving access to sensitive data or offering the attacker a possibility to pivot and attack other connected assets.
Medium	Allows an attacker to access some information, or to alter the behavior of the assessed system with restricted permissions.
Low	Allows an attacker to access non-sensitive information, or to alter the behavior of the assessed system and impact a limited number of users.

3.4.2 Exploitability

The exploitability of a vulnerability is evaluated by taking the following criterias in consideration:

- **Access conditions:** the vulnerability may require the attacker to have physical access to the targeted asset or to be present in the same network for instance, or can be directly exploited from the Internet.
- **Required skills:** an attacker may need specific skills to exploit the vulnerability.
- **Known available exploit**: when a vulnerability has been published and an exploit is available, the probability a non-skilled attacker would find it and use it is pretty high.

The following table summarizes the different level of exploitability of a vulnerability:

Critical	The vulnerability is easy to exploit even from an unskilled attacker and has no specific access conditions.
High	The vulnerability is easy to exploit but requires some specific conditions to be met (specific skills or access).
Medium	The vulnerability is not trivial to discover and exploit, requires very specific knowledge or specific access (internal network, physical access to an asset).
Low	The vulnerability is very difficult to discover and exploit, requires highly specific knowledge or authorized access.

3.4.3 Severity

The severity of a vulnerability is defined by its impact and its exploitability, following the following table:

		Impact			
		●●●●	●●●○	●●○○	●○○○
Exploitability	●●●●	Critical	Critical	High	Medium
	●●●○	Critical	High	High	Medium
	●●○○	High	High	Medium	Low
	●○○○	Medium	Medium	Low	Low

Contents

4. Introduction

4.1 Overview of Eclipse KUKSA

The official KUKSA website [1] defines the project as the following:

> *The open Eclipse KUKSA project aims to provide shared building blocks for the Software Defined Vehicles that can be shared across the industry. That millions of lines of code should go into generating customer value and not reinvented wheels. KUKSA tries to provide you with a solid set of wheels that can act a solid foundation for a variety of competing products and services. In that sense KUKSA components encourage cooperation on the plumbing, enabling competition and faster innovation cycles on the customer-value creating procelain.*

The KUKSA Vehicle Abstraction Layer named KUKSA.val is the main solution of the project. It provides software components for working with in-vehicle signals. Those in-vehicle signals are represented using the Vehicle Signal Specification [2] (VSS). KUKSA.val provides reading and writing capabilities on vehicle data through a set of applications:

- KUKSA.val databroker;
- KUKSA.val Server;
- KUKSA.val Python client SDK;
- KUKSA.val Feeders and Providers.

4.2 Scope of the audit

The scope of the audit was focused on the KUKSA.val databroker and the Python client SDK. Those two components are both distributed in the *eclipse/kuksa.val* GitHub repository. The third party dependencies are out of the scope of the audit, but their proper usage is in scope. More details on the audit scope will be given in the threat model.

The Table 1. shows the version on which the audit was conducted.

Project	kuksa.val
Repository	https://github.com/eclipse/kuksa.val
Commit hash	6690ca970a2f7dd8245bd00f6b10788eaad755b5
Commit date	2023/10/27
Tag	0.4.1

Table 1. Audit scope details

5. Methodology

5.1 Threat model

The threat model is the initial step of the audit. It provides an overview of the project's work. More importantly, this step identifies the project's purposes and critical functionalities.

Then, high-level attack scenarios can be extrapolated from these critical functionalities. The resulting scenarios will guide the next steps of the audit.

Identifying the audited code base critical features and assets permits the creation of realistic scenarios. A world-like approach is important to identify the most relevant attack vectors and vulnerabilities.

5.2 Static analysis

5.2.1 Automated static analysis

This part of the audit aims to run several automated security tools on the audited code base.

Most of these tools are open-source and could be integrated in a continuous integration workflow.

5.2.2 Manual review

The manual review is a complex process. It can be seen as multiple iterations of the following workflow:

- understanding of the inner workings of part of the code base
- imagining an attack scenario based on the code base understanding and the threat model
- testing the scenario by static analysis or dynamic tests
- validating or denying the attack scenario

This process aims to identify technical and logical vulnerabilities.

5.3 Dynamic analysis

Dynamic analysis is mainly done through fuzzing. Further details on fuzzing are given in Section 8.1.1.

This process aims to identify technical vulnerabilities. It complements the manual review by automating vulnerability tests.

6. Threat Model

6.1 Overview

KUKSA.val allows clients to access data from their vehicles. A databroker is embedded and centralizes the data. Then, clients can interface with the databroker to get and set, if possible, those data. Figure 1. shows a general workflow about how clients access data from the KUKSA databroker.

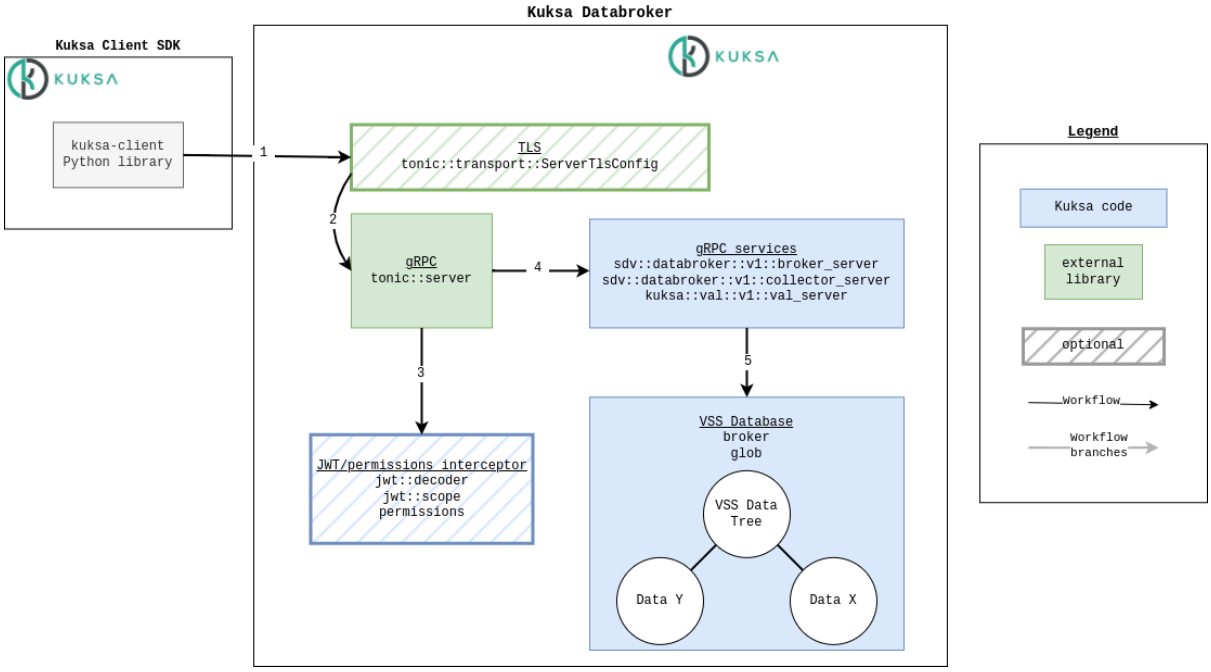

Figure 1. General workflow: client access data

Based on the previous workflow, we identified some critical parts of KUKSA databroker and KUKSA Client SDK code. Once critical code was identified, we produced a threat model that will guide our audit and provide priorities to assess during the allocated time frame. Figure 2. shows an overview of the defined threat model.

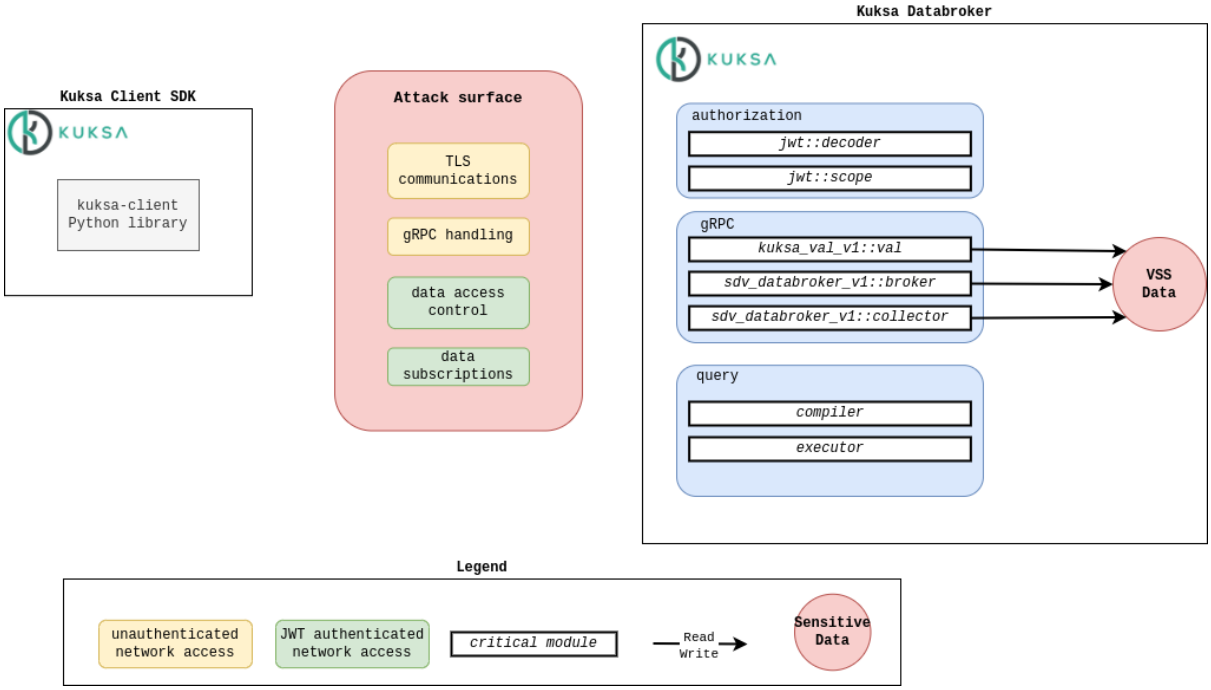

Figure 2. Threat model: attack surface overview

Note: "data access control" is set as "JWT authenticated network access" because the JWT signature is verified before parsing its values to get the permissions. The signature verification acts as an authentication.

This threat model is divided into 4 main scenarios that will guide the audit. These scenarios assess critical parts of the code identified in the attack surface.

6.2 Scenario 1

The first scenario aims to assess the TLS implementation. The client SDK and the databroker exchange sensitive pieces of information such as the JWT that allows the client to access data.

For example, if the client SDK allows TLS communications with untrusted CA, then an attacker could be able to use a Man-In-The-Middle attack to steal the client's JWT.

Ensuring that TLS is correctly implemented both on the client and server side is mandatory.

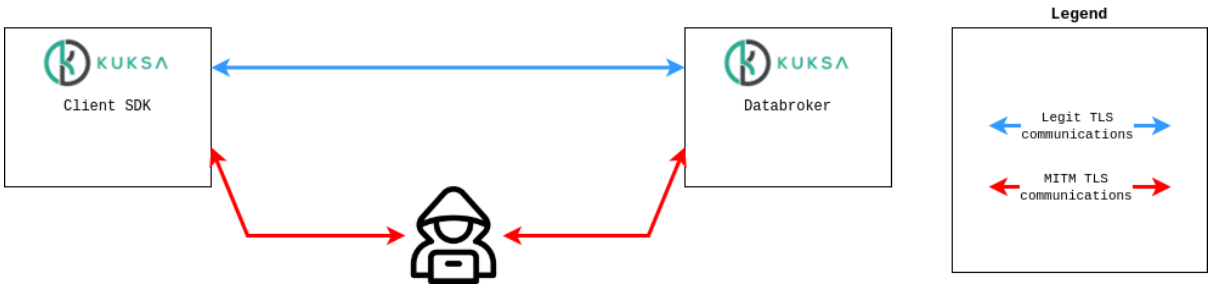

Figure 3. Threat model: Scenario 1

6.3 Scenario 2

The second scenario aims to assess the JWT implementation and the permissions handling in KUKSA.val databroker.

For example, a client allowed to access a single data point of the VSS tree with read-only permissions should not be able to read/write other data. Writing the datapoint he has read access to should not be possible either.

Regular expressions are used to parse the *scope* field of the JWT. If not set correctly, this regex could lead to unexpected behaviors.

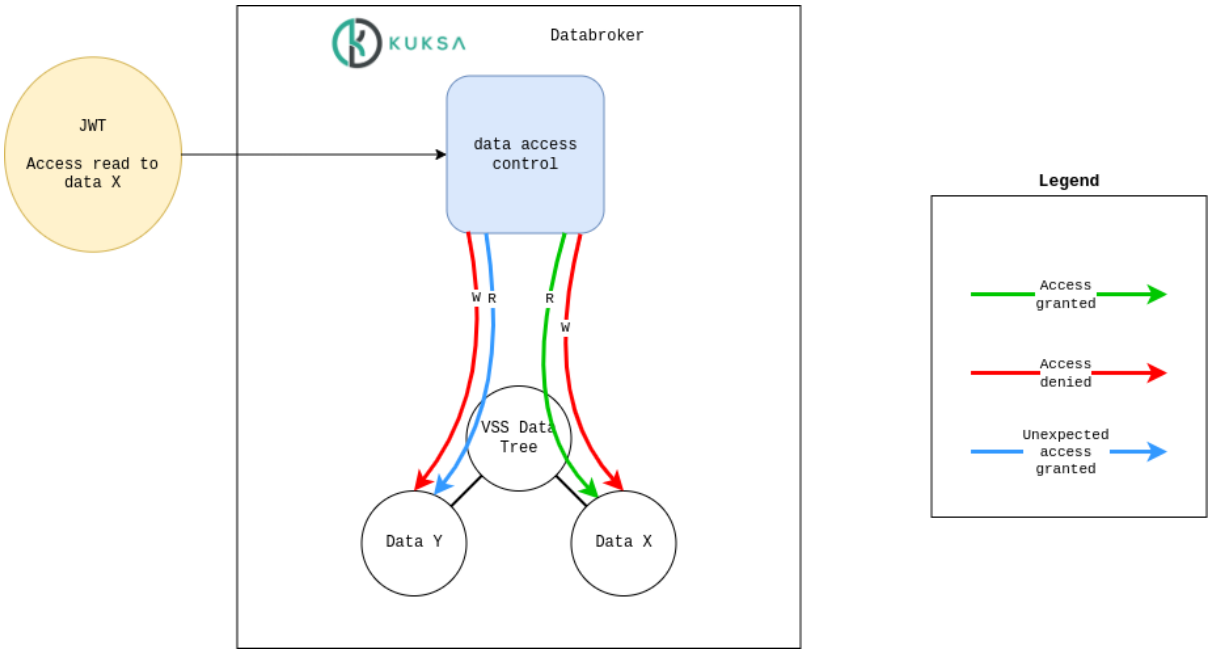

Figure 4. Threat model: Scenario 2

6.4 Scenario 3

The third scenario aims to find security issues in the KUKSA databroker.

Those security issues could be exploited remotely in case of a malicious client or a MITM attack.

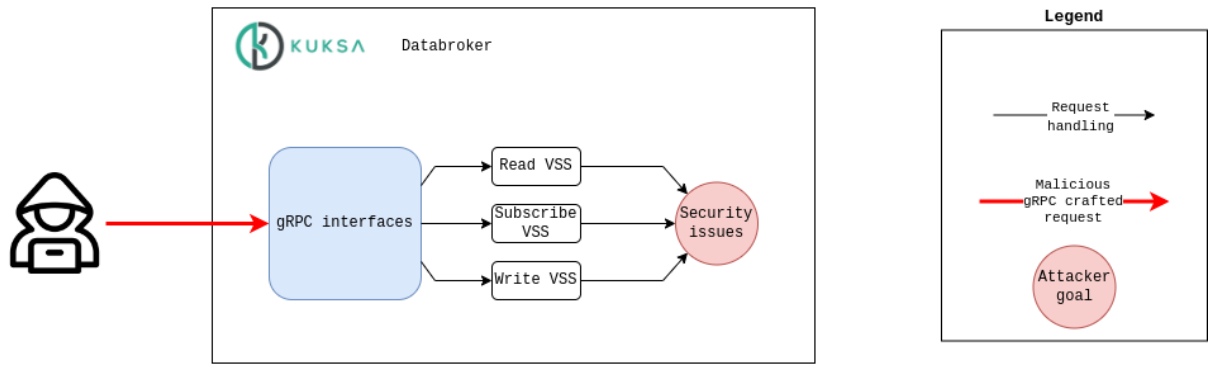

Figure 5. Threat model: Scenario 3

6.5 Scenario 4

The fourth scenario aims to find security issues on the KUKSA client SDK written in Python.

Those security issues could be exploited remotely in the case of a malicious databroker or a MITM attack.

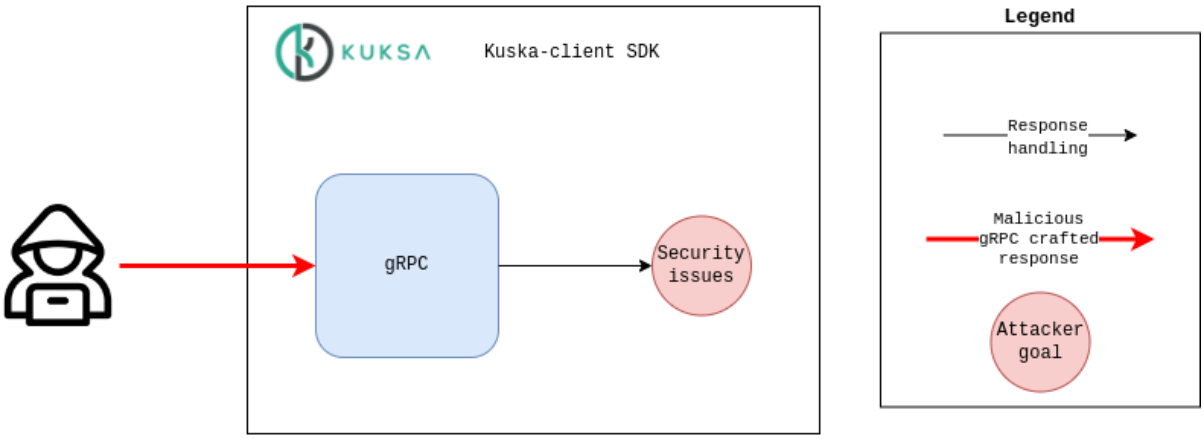

Figure 6. Threat model: Scenario 4

6.6 Hypothesis

KUKSA.val provides in-vehicle software components. The configuration and use of these components should follow the official documentation. Moreover, users of KUKSA.val must base a threat and risk analysis to decide what is required and feasible for their specific use case.

There exist several attack vectors. But some of them are purely theoretical. They would not be exploitable or exploited in real-world usage of the KUKSA.val components.

Hypotheses have been made and applied to the four defined scenarios. These hypotheses aims to fit the scenarios with the real-world usage of KUKSA.val components.

6.6.1 KUKSA.val misconfigurations

Incorrect configurations of the KUKSA.val components can lead to unsecured architecture.

For example, the use of the `--insecure` flag on the databroker allows communications without TLS. An attacker could easily use a Man-In-The-Middle attack to read and modify the exchanged data.

Hypothesis: The KUKSA.val components are configured following the documented best practices.

Note: The databroker is configured to use TLS and JWT.

6.6.2 JWT cryptographic keys

Cryptographic keys are needed to sign the emitted JSON Web Tokens. These keys should be managed securely. A leaked key would make the entire permission mechanism of the databroker useless.

As the management of these keys is not under KUKSA.val's control, they are considered managed in a secure way and not compromised.

Hypothesis: Cryptographic keys used to generate the access tokens are not compromised.

6.6.3 TLS cryptographic keys

Cryptographic keys are needed to establish secure TLS communications. These keys should be managed securely. A leaked key would make the databroker TLS communications unsecured.

As the management of these keys are not under KUKSA.val's control, they are considered managed securely and not compromised.

Hypothesis: Cryptographic keys used to secure communications with TLS are not compromised.

6.6.4 VSS Parsing

At startup, a VSS structure is provided as a JSON file to the databroker. Then, this VSS structure is parsed to create the databroker database.

Exploiting a vulnerability in this initial VSS parsing could be possible. It would require access on the databroker machine to modify the JSON file passed as an argument. In the current architecture of KUKSA.val, this attack vector seems highly unlikely to be used as it requires a high level of access to the target and wouldn't give further access to an attacker.

Hypothesis: VSS parsing vulnerabilities are not exploitable.

6.6.5 Supply chain attacks

Compromising Kuksa codebase by compromising one of its dependencies or by directly adding malicious code (in a contribution for example) is considered out-of-scope of the audit.

Hypothesis: The future compromise of a dependency is not considered in the scope of this audit.

7. Static analysis

7.1 Automated static analysis

Several linters and static checkers were selected and tested on KUKSA.val. These tools were used to find issues but also to see if they could be integrated into the project workflows. The following open-source tools have been selected:

- `Clippy` [3]: Clippy provides a collection of lints used to catch common mistakes and improve Rust code.
- `cargo-audit` [4]: cargo-audit audits the project dependencies. It looks for crates with vulnerabilities reported to the RustSec Advisory Database [5].
- `cargo-outdated` [6]: cargo-outdated displays dependencies that are out of date.

7.1.1 Clippy

Clippy has several options which are not executed by default. The default clippy command `cargo clippy` did not return any result.

The following clippy command was used to allow several options:

```
$ cargo clippy --no-deps -A clippy::all -W clippy::integer_arithmetic  -W
↪  clippy::string_slice -W clippy::expect_used  -W clippy::fallible_impl_from -W
↪  clippy::get_unwrap -W clippy::index_refutable_slice  -W
↪  clippy::indexing_slicing -W clippy::match_on_vec_items -W
↪  clippy::match_wild_err_arm  -W clippy::missing_panics_doc -W clippy::panic -W
↪  clippy::panic_in_result_fn  -W clippy::unreachable -W clippy::unwrap_in_result
↪  -W clippy::unwrap_used
```

Seven types of warning are triggered by Clippy in the databroker.

- `arithmetic operation that can potentially result in unexpected side-effects`: jwt/decoder.rs#L134
- `used `unwrap()` on a `Result` value`: jwt/scope.rs#L34, lib.rs#L34
- `used `expect()` on a `Result` value`: glob.rs#L75, glob.rs#L109, lib.rs#L38, main.rs#L38, main.rs#L39, main.rs#L41
- `used `expect()` on an `Option` value`: main.rs#L317, main.rs#L318
- `indexing into a string may panic if the index is within a UTF-8 character`: grpc/server.rs#L52
- `slicing may panic`: grpc/server.rs#L52
- `indexing may panic`: grpc/sdv_databroker_v1/broker.rs#L154

These warnings will be further investigated during the audit.

Clippy is already integrated in KUKSA.val databroker build workflow.

The options of Clippy presented previously could be added to this workflow to improve the overall security of the project.

7.1.2 cargo-audit

In the audited scope, `cargo-audit` reported only one finding:

```
Crate:     atty
Version:   0.2.14
Warning:   unsound
Title:     Potential unaligned read
Date:      2021-07-04
ID:        RUSTSEC-2021-0145
URL:       https://rustsec.org/advisories/RUSTSEC-2021-0145
Dependency tree:
atty 0.2.14
   cucumber 0.19.1
       databroker 0.4.1
```

INFO	INFO-1 A vulnerability is known for the `atty` crate on Windows.
Exploitability	**Impact**
Perimeter	Databroker dependencies
Prerequisites	
Description	
A vulnerability in the `atty` dependency used by the databroker is known. The RUSTSEC-2021-0145 [7] indicates that the issue only affects Windows operating system.	
Recommendations	
To our knowledge, the databroker is not meant to be run on Windows, moreover this dependency is only used for testing purposes.	

Cargo-audit is not integrated in KUKSA.val workflows. It can be integrated by using the audit-check action [8].

7.1.3 cargo-outdated

Cargo-outdated found multiple dependencies that are not using the last major version.

```
$ cargo outdated --depth=1
databroker
================
```

```
Name                Project  Compat  Latest   Kind         Platform
----                -------  ------  ------   ----         --------
axum                0.6.20   ---     0.7.2    Normal       ---
cucumber            0.19.1   ---     0.20.2   Development  ---
jsonwebtoken        8.3.0    ---     9.2.0    Normal       ---
serde               1.0.190  ---     1.0.193  Normal       ---
serde_json          1.0.107  ---     1.0.108  Normal       ---
sqlparser           0.16.0   ---     0.40.0   Normal       ---
tracing-subscriber  0.3.17   ---     0.3.18   Normal       ---
uuid                1.5.0    ---     1.6.1    Normal       ---
vergen              8.2.5    ---     8.2.6    Build        ---
```

INFO	INFO-2 Multiple databroker dependencies are out of date.
Exploitability	**Impact**
Perimeter	Databroker dependencies
Prerequisites	
Description	
The databroker does not use the most up-to-date version of several dependencies.	
Recommendations	
Consider upgrading to the last up-to-date version for each dependency.	

7.2 Manual review

7.2.1 TLS implementation

According to the KUKSA.val documentation, the databroker and the Python client SDK should support both insecure and secure via TLS connections. As mentioned in the threat model, TLS cryptographic keys are out of the audit scope and are considered secure.

In KUKSA.val current architecture, mutual authentication is not supported. Clients authenticate the server using its TLS certificate, but the server does not use TLS to authenticate clients.

TLS Server

rustls crate

The workspace `Cargo.lock` indicates that the version `0.9.2` of the `tonic` crate is used as the databroker gRPC implementation. This crate has a `tls` feature, which is enabled in the databroker `Cargo.toml`. The feature provides TLS capabilities to the gRPC implementation based on the `rustls` [9] crate.

The `rustls` crate is defined as:

> *Rustls is a TLS library that aims to provide a good level of cryptographic security, requires no configuration to achieve that security, and provides no unsafe features*

or obsolete cryptography.

The reputation of `rustls` matches its description. An audit [10] of `rustls` was conducted in 2020 and can be retrieved from the official GitHub repository. No outdated or unsecured features are embedded in the library. Moreover, no security issue has been discovered in the `0.21.8` version used by the databroker.

The `rustls` crate provides mutual authentication capabilities. These capabilities are not exploited by the databroker in its current state.

TLS configuration

When starting the databroker, the TLS server certificate and the associated keypair should be passed as command-line arguments. The following command-line is a valid example:

```
$ ./target/release/databroker --tls-cert ./tls-srv.crt --tls-private-key
↪  ./tls-srv.key
```

TLS client

The Python kuksa-client uses the `grpc` library. This library provides TLS functionalities through the use of the `BoringSSL` library.

Testing the TLS client showed that the server certificate must be issued by the trusted certificate authorities. If not, connections to the TLS server are not possible. As such, an attacker cannot read or write the intercepted TLS traffic without compromising the client or the server.

7.2.2 JWT and permission handling

The databroker handles permissions through the use of access tokens. Json Web Tokens (JWT) are used as OAuth 2.0 Bearer Tokens to encode the relevant parts of the access tokens. JWT are emitted and signed by an external authorization infrastructure which is out-of-scope of this audit.

The KUKSA.val JWT access tokens are structured with a header and a data structure. This format is defined in the RFC 9068 [11]. An access token encodes several pieces of information such as the issuer of the token, its expiration date, and its scope. Permissions are encoded into the scope field, following the scope format defined in the OAuth 2.0 RFC 6749 [12]. The following lines show a valid data structure for a KUKSA.val access token:

```
{
    "sub": "local dev",
    "iss": "createToken.py",
    "aud": [
        "kuksa.val"
    ],
    "iat": 1516239022,
    "exp": 1767225599,
```

```
    "scope": "actuate provide read create"
}
```

In the current implementation of KUKSA.val access tokens, only the `RS256` algorithm is supported. `RSASSA-PKCS1-v1_5 SHA-256` signatures are supported to authenticate access tokens. As the library `jsonwebtoken` used by the databroker supports other algorithms, this limit is specific to the databroker. Hardcoding the supported algorithm to `RS256` makes the known JWT authentication bypass via algorithm confusion attack unexploitable.

JWT in gRPC requests

When requesting the databroker, clients should attach their access token to the HTTP/gRPC request by adding an authorization header. This header should respect the following format, with `TOKEN` being the access token encoded in Base64 format:

```
Authorization: Bearer TOKEN
```

The `authorization` first character should correspond to `Bearer`, otherwise, an error is returned to the client with the message `Invalid auth token`. If those first character are set, then the JWT content is decoded with the `jsonwebtoken` crate. This decoding will verify that the JWT is correctly signed. If it is not, an error is returned to the client with the message `Invalid auth token: DecodeError("InvalidToken")`.

The databroker correctly handles the `authorization` field and the JWT decoding. Errors are returned to the gRPC client if the decoding fails.

Permissions

Permissions on the databroker data are encoded in the `scope` field of the access tokens. In the current state of KUKSA.val databroker, four permissions are available:

- **Read**: This permission allows reading values from specific entries of the databroker.
- **Actuate**: This permission allows reading and writing the `actuator_target` value of specific entries.
- **Provide**: This permission allows reading and writing the `datapoint` value of specific entries.
- **Create**: This permission allows reading and creating specific entries.

VSS paths are attached to these permissions. A path corresponds to a part of the VSS tree. These paths are decoded from the `scope` field of the JWT to be translated into regular expressions in the databroker. Resulting regexps are used to match the requested datapoints. Access is granted if the requested datapoint matches the permission regexps, otherwise, a permission error is returned.

The following lines show a valid data structure for a JWT access token with read-only access on 2 VSS paths:

```
{
    "sub": "local dev",
    "iss": "createToken.py",
    "aud": [
        "kuksa.val"
    ],
    "iat": 1516239022,
    "exp": 1767225599,
    "scope": "read:Vehicle.Speed read:Vehicle.ADAS.*"
}
```


In the databroker, the following structure is derived from the previous JWT access token:

```
Permissions {
  expires_at: Some(SystemTime { tv_sec: 1767225599, tv_nsec: 0 }),
  read: Regexps(RegexSet(["^Vehicle\\.Speed(?:\\..+)?$",
  ↪  "^Vehicle\\.ADAS\\.[^.\\s\\:]+$"])),
  actuate: Nothing,
  provide: Nothing,
  create: Nothing
}
```

No special characters can be passed through the scope field to modify the behavior of the resulting regular expressions. The JWT `scope` field elements are checked to ensure that they are only composed of the `*` character or a set of letters and numbers. The decoding of the `scope` elements are well handled by the databroker.

When trying to access a path of the VSS tree on which the JWT has no permission, the databroker returns this type of error:


```
{
    "error": null,
    "errors": [
        {
        "error": {
            "code": 403,
            "message": "Access was denied for Vehicle.ADAS.ABS.IsEnabled",
            "reason": "forbidden"
        },
        "path": "Vehicle.ADAS.ABS.IsEnabled"
        }
    ]
}
```


The databroker handles correctly the permissions attached to the JWT access token. Clients trying to access unauthorized scope receive a gRPC response indicating that the access was denied.

7.2.3 Databroker gRPC handling

The KUKSA.val databroker supports 3 gRPC services:

- **kuksa.val.v1:VAL**: This interface provides 4 endpoints.
- **sdv.databroker.v1:Broker**: This interface provides 4 endpoints, mainly used by clients.
- **sdv.databroker.v1:Collector**: This interface provides 3 endpoints, mainly used by feeders and providers.

Some of the endpoints of the 3 gRPC services can provide the same functionality. For example, reading the current value from a VSS path can be done with the `VAL:Get` and the `Broker:GetDatapoints` endpoints. The structure of the databroker source code shows that the same logic will be used in both endpoints. This means that a vulnerability found on one of the endpoints could impact other endpoints.

Tools

In this chapter, the gRPC endpoints of the databroker are investigated. To test the databroker, we used a gRPC client that allows crafting custom inputs based on protobuf files. The tool used is an open-source project named Evans [13] and is available on GitHub.

In the following sections, we will provide examples of requests with Evans and the resulting responses from the databroker. The commands executed will be given so that it is possible to replay the requests.

The initial setup of Evans is the following:

```
# Start Evans from a shell
$ ./evans --path ./kuksa_databroker/databroker-proto/proto/ --proto
↪  kuksa/val/v1/val.proto --host localhost --port 55555  repl

# In Evans, set the authorization header with a JWT token
$ header authorization="Bearer eyJhb...Crmb71Y"
```

kuksa.val.v1:VAL service

The `kuksa.val.v1:VAL` service provides 4 endpoints that will be investigated in the following paragraphs.

Get

This gRPC endpoint provides the capability to read VSS tree entries. An entry is a structure that contains the current value, the target value (for actuators only), and the metadata.

The following shows the Evans request to get the entry values.

```
$ kuksa.val.v1.VAL@localhost:55555> call --add-repeated-manually --emit-defaults
↪  --dig-manually --enrich Get
  yes
  dig down
```

```
<repeated> entries::path (TYPE_STRING) => Vehicle.Speed
  VIEW_ALL
  no
  no
```

The following shows the databroker response content with the requested entries.


```
{
    "entries": [
    {
        "actuatorTarget": null,
        "metadata": {
        "dataType": "DATA_TYPE_FLOAT",
        "description": "Vehicle speed.",
        "entryType": "ENTRY_TYPE_SENSOR",
        "valueRestriction": null
        },
        "path": "Vehicle.Speed",
        "value": {
        "float": 12,
        "timestamp": "2023-12-06T13:46:23.806837137Z"
        }
    }
    ],
    "error": null,
    "errors": []
}
```


Set

This gRPC endpoint provides the capability to write the values of existing VSS tree entries.

The protobuf format offers the capability to modify the values of the current value, the target value, and the metadata. Forging a request to modify the metadata of a VSS entry can be done with Evans. But modifying metadata is not supported by the databroker. The databroker will return an `OK` code without error, but the metadata of the entry will not be modified.

The current value of an entry can be modified through this endpoint. The following example shows the modification request of the `Vehicle.Speed` current value.

```
$ call --add-repeated-manually --emit-defaults --dig-manually --enrich Set
  yes
  dig down
  dig down
<repeated> updates::entry::path (TYPE_STRING) => Vehicle.Speed
  dig down
  dig down
<repeated> updates::entry::value::timestamp::seconds (TYPE_INT64) => 1701871499
<repeated> updates::entry::value::timestamp::nanos (TYPE_INT32) => 0
  float
<repeated> updates::entry::value::float (TYPE_FLOAT) => 123.456
```

```
  skip
  skip
  yes
  FIELD_VALUE
  no
  no
```

The databroker returns a JSON structure indicating that no errors were met.

```
{
  "error": null,
  "errors": []
}
```

The current value can then be read from the Python kuksa-client.

```
$ Test Client> getValue Vehicle.Speed
{
    "path": "Vehicle.Speed",
    "value": {
        "value": 123.45600128173828,
        "timestamp": "2023-12-06T14:04:49+00:00"
    }
}
```

The timestamp attached to the value corresponds to the one set in the `Set` request. It appears that the databroker does not check this field before storing the current value. This behavior can be disruptive as old values could overwrite recent values, especially in overloaded networks.

The following example shows how to overwrite our previous value from 2023 with a value from 1970.

```
$ call --add-repeated-manually --emit-defaults --dig-manually --enrich Set
  yes
  dig down
  dig down
<repeated> updates::entry::path (TYPE_STRING) => Vehicle.Speed
  dig down
  dig down
<repeated> updates::entry::value::timestamp::seconds (TYPE_INT64) => 123456
<repeated> updates::entry::value::timestamp::nanos (TYPE_INT32) => 0
  float
<repeated> updates::entry::value::float (TYPE_FLOAT) => 234.567
  skip
  skip
  yes
  FIELD_VALUE
  no
  no
```

```
{
    "error": null,
    "errors": []
}
```

Reading the `Vehicle.Speed` values from the python client return the following. Notice that the date is from 1970.

```
$ Test Client> getValue Vehicle.Speed
{
    "path": "Vehicle.Speed",
    "value": {
        "value": 234.56700134277344,
        "timestamp": "1970-01-02T10:17:36+00:00"
    }
}
```

This vulnerability concerns several endpoints:

- `Set` endpoint from the **kuksa.val.v1:VAL** service
- `SetDatapoints` endpoint from the **sdv.databroker.v1:Broker** service
- `UpdateDatapoints` endpoint from the **sdv.databroker.v1:Collector** service
- `StreamDatapoints` endpoint from the **sdv.databroker.v1:Collector** service

MEDIUM	MED-1 Recent values can be overwritten with old values.	
Exploitability	●●○○	**Impact** ●●○○
Perimeter	Databroker entries	
Prerequisites	`Provide` or `Actuate` permissions	

Description

The protobuf structure provides a timestamp when an entry value is updated. As the provided timestamp is not checked against the timestamp of the previous entry value, the databroker can overwrite a recent value with an old one. Network latency can heavily impact the value updated in the databroker.

Recommendations

A timestamp check must be done before updating an entry value.

Subscribe

This gRPC endpoint allows clients to subscribe to a VSS entry path. When a field of this entry is modified, the new values will be automatically sent to the subscribers. This endpoint offers only subscribing capabilities without conditions.

The JWT expiration is correctly handled for this endpoint. A user can subscribe before the expiration of its token. He will receive the updates of the subscribed entry until his token expires. Once his token has expired, no more data will be sent to the user. However, the communication channel remains active even if the token is expired.

INFO	INFO-3 Subscription channels remain open after token expiration
Exploitability	**Impact**
Perimeter	Databroker subscriptions
Prerequisites	`Read` permission
Description	
When a client subscribes to an entry, he receives a gRPC response when this entry is updated. If he subscribes before his token expiration, he will not receive updates after expiration but the communication channel will remain open.	
Recommendations	
Close the communication channel when the token is expired.	

GetServerInfo

This gRPC endpoint is used to identify the version of the databroker.

At the start of the databroker, 3 entries are registered. These entries are `Kuksa.Databroker.GitVersion`, `Kuksa.Databroker.CargoVersion` and `Kuksa.Databroker.CommitSha`. They allow a client to identify the version of the databroker.

A user allowed to write on the `Kuksa` scope can modify the values of these three entries, by using one of the writing gRPC endpoints such as `Set`. As they are specific entries set by the databroker, these 3 entries should be read-only. The following example shows the modification of this value from the Python client.

```
$ Test Client> getValue Kuksa.Databroker.CommitSha
{
    "path": "Kuksa.Databroker.CommitSha",
    "value": {
        "value": "6690ca970a2f7dd8245bd00f6b10788eaad755b5",
        "timestamp": "2023-12-05T15:10:46.005114+00:00"
    }
}

$ Test Client> setValue Kuksa.Databroker.CommitSha 000000000000
OK

$ Test Client> getValue Kuksa.Databroker.CommitSha
{
    "path": "Kuksa.Databroker.CommitSha",
```

```
    "value": {
        "value": "000000000000",
        "timestamp": "2023-12-06T15:24:36.243281+00:00"
    }
}
```

LOW	LOW-1 Databroker-specific entries can be modified remotely.
Exploitability	●○○○ **Impact** ●●○○
Perimeter	Databroker version
Prerequisites	`Provide` permission on `Kuksa`
Description	
A user with write permissions on `Kuksa` VSS path can modify the values of the databroker version. The version by the databroker will then be incorrect and could lead to incompatibility issues with other clients.	
Recommendations	
Deny the write access on the three databroker-specific entries.	

sdv.databroker.v1:Broker service

The `sdv.databroker.v1:Broker` service provides 4 endpoints that will be investigated in the following paragraphs. This service is mostly used by clients.

GetDatapoints

This gRPC endpoint only provides reading capabilities on the current value of an entry. The metadata and target value of an entry cannot be read with this endpoint.

The source code used by this endpoint is pretty similar to the one used by the `Get` of the `VAL` service. No issues related to this endpoint were found.

SetDatapoints

This gRPC endpoint only provides writing capabilities on the target value of an entry. The metadata and current value of an entry can't be written with this endpoint.

This endpoint fits with the `Actuate` permission. Clients will use this duo to request a modification of the vehicle state through the databroker and the corresponding provider.

The vulnerability MED-1 affects this gRPC endpoint.

Permission segregation between actuating and providing actions are correctly handled on the databroker. An attempt to modify the current value of an entry through this gRPC endpoint will return the following error to the client.

```
{
    "errors": {
    "Vehicle.Speed": "ACCESS_DENIED"
    }
}
```

Subscribe

This gRPC provides subscription capabilities to clients. SQL syntax can be used by the client to subscribe to an entry with custom conditions.

A user can try to use SQL requests to bypass the permission mechanism. The subscription mechanism seems to trigger an internal error with this type of request. For example, the following SQL request tries to access `Vehicle.Speed` on which the user has no read access.

```
SELECT Vehicle.ADAS.ABS.IsEngaged WHERE Vehicle.Speed < 12
```

The databroker triggers an internal error when `Vehicle.Speed` is accessed and the user has no access right on it. But, the returned error is then ignored and an `Ok()` statement is returned. This led to opening a communication channel with the client with no databroker subscriptions. The vulnerability scope is located at broker.rs#L835-L839.

LOW	LOW-2 Client can subscribe to unavailable scope and waits for data that will never be sent.
Exploitability	●○○○ **Impact** ●○○○
Perimeter	Databroker subscriptions
Prerequisites	`Read` permission
Description	
Subscriptions with SQL comparison conditions that try to access out-of-permissions scope data are failing, but no error is sent to the client and the communication channel is still open. This leads clients to await data from the open channel, but the relevant subscriptions are not registered server-side. A `TODO` comment in the source indicates that the error must be sent to the subscriber.	
Recommendations	
Return the error to the subscriber and delete the `TODO` comment in broker.rs#L835-L839.	

The INFO-3 issue also affects this gRPC endpoint. It appears that on this endpoint, the expiration of a token has a different impact. The communication channel remains open, but the SQL queries are executed every time an update happens. If a client subscribed to an SQL query before the expiration of his JWT, the query will still be executed after expiration and a response will be sent to the client. As the token is expired, the values will not be readable. However, the client will be able to know the time at which an entry has been modified.

The issue is due to a lack of timestamp checks before notifying clients. This check could be added to the `notify` function at broker.rs#L805-L845.

The following logs show the result of an expired subscription. The entry is updated every second, and the client with an expired token still receives the subscription callbacks.

```
{
    "fields": {
    "Vehicle.Speed": {
        "floatValue": 8,
        "timestamp": "2023-11-20T09:23:18.460571145Z"
    }
    }
}

{
    "fields": {
    "Vehicle.Speed": {
        "floatValue": 9,
        "timestamp": "2023-11-20T09:23:19.475708680Z"
    }
    }
}
// HERE EXPIRATION OF THE JWT TOKEN
{
    "fields": {
    "Vehicle.Speed": {
        "failureValue": "NOT_AVAILABLE",
        "timestamp": "2023-11-20T09:23:20.490804601Z"
    }
    }
}

{
    "fields": {
    "Vehicle.Speed": {
        "failureValue": "NOT_AVAILABLE",
        "timestamp": "2023-11-20T09:23:21.505369154Z"
    }
    }
}
```

LOW	LOW-3 An expired token can leak information about entries update timestamp.
Exploitability	●●○○ **Impact** ●○○○
Perimeter	Databroker subscriptions
Prerequisites	`Read` permission

Description

By subscribing to a datapoint before the token expiration, a user can be informed when this datapoint is written once its token expires. This is possible because token expiration is not checked before calling back the subscriber.

Recommendations

In brokers.rs#L805-L845, consider adding an expiration check before notifying the client. If the token is expired, consider closing the connection.

GetMetadata

This gRPC endpoint returns the metadata attached to an entry. The client must provide the VSS path of the entry.

The metadata of every entry is accessible by clients. Reading metadata is always allowed, even if the `scope` field of the client JWT excludes the entry path.

LOW	LOW-4 Entries metadata can be read by every client and feeder.
Exploitability	●○○○ **Impact** ●○○○
Perimeter	Databroker entries metadata
Prerequisites	Valid JWT

Description

Entries metadata access is not restricted. A user with restricted access rights can read the metadata of all the entries stored in the databroker.
An attacker could use this leak of information to gain further knowledge of the target, for example by discovering entries that are specific to a vehicle model.

Recommendations

Restrict the metadata access according to the JWT permissions scope.

sdv.databroker.v1:Collector service

The `sdv.databroker.v1:Collector` service provides 3 endpoints that will be investigated in the following paragraphs. This service is mainly used by the providers and the feeders embedded in the vehicle.

UpdateDatapoints

This gRPC endpoint allows to modify the current value of an entry.

The vulnerability MED-1 affects this gRPC endpoint.

The request format of this endpoint is different from the others. To select an entry, the VSS path is commonly used. But this gRPC message format requires the `id` number at which the entry is registered in the databroker. To retrieve this number from the VSS path, the client should first send a request to the `sdv.databroker.v1:Broker:GetMetadata` endpoint.

INFO	INFO-4 UpdateDatapoints request format is not unified with other endpoints.
Exploitability	**Impact**
Perimeter	Databroker UpdateDatapoints endpoint
Prerequisites	
Description	
`UpdateDatapoints` endpoint use the `id` to identify entries. This is not unified with other gRPC endpoints which use the VSS path. `UpdateDatapoints` requires a call to `GetMetadata` or `RegisterDatapoints` to retrieve the `id` field which is unstable.	
Recommendations	
Consider unifying the gRPC endpoint request format to retrieve databroker entries.	

StreamDatapoints

This gRPC endpoint is used by providers and feeders to open a communication channel with the databroker. The channel allows to continuously feed new values into the databroker.

The JWT expiration is handled such that providing a new value after expiration returns the `ACCESS_DENIED` error. However, the communication channel remains open even if the token has expired. This can be tested with an access token that has an upcoming expiration date.

The vulnerability MED-1 affects this gRPC endpoint.

RegisterDatapoints

This gRPC endpoint allows providers and feeders to register new entries in the VSS tree. When a new entry is registered, its `id` used by the databroker is returned.

If a provider tries to register a new entry to an existing path, the `id` of the entry located at the path is returned. The new entry data are discarded. The `id` field is a 32-bit integer managed with the `atomic` crate.

On the databroker side, entries are stored in a memory `HashMap<i32, Entry>`. Every entry is stored in memory and the `RegisterDatapoints` gRPC endpoint allows creating new entries remotely. From an attacker view, `RegisterDatapoints` provides a direct impact on the memory size of the databroker process. We developed a custom Rust client which goal was to

create as many entries as possible. In our test with a 32GB memory machine, 29 million entries were created before the databroker gets killed by the operating system.

Our custom Rust client is available as part of the delivered archive.

HIGH	HIGH-1 A feeder can crash the databroker.	
Exploitability	●●○○	**Impact** ●●●○
Perimeter	Databroker RegisterDatapoints endpoint	
Prerequisites	`Create` permission	
Description		
The number of entries in the databroker is not upper bounded. Entries are stored in a hashmap kept in the databroker memory. A malicious client can register new datapoints to grow the databroker memory consumption. Modern operating systems have protection and will kill the databroker process when its memory consumption becomes too high.		
Recommendations		
Set an upper-bound limit on the number of entries that the databroker can handle.		

7.2.4 The KUKSA Python SDK

As an introduction, the **KUKSA Python SDK** is part of the **KUKSA.val** repository, which also contains the databroker. Mixing the server and client components of a same project in a single repository, especially when they are not developed in the same language and therefore do not contain common code, is not a good software engineering practice. We noted during our audit and the release of version 0.4.2 that the client had been moved to a dedicated repository, which is definitely a good point.

The SDK contains the Python code required to communicate with either KUKSA Server or KUKSA Databroker, handling gRPC and WebSocket protocols, TLS, authorization, etc. The various services exposed by the broker, like getting or setting values for example, are available through a Python class (`KuksaClientThread`) easy to use, handling the backend communication in a dedicated thread. The entire package is a pure Python implementation, and depends on well-known libraries like **grpcio** and **websockets**, which are regularly audited. Available on **PyPI**, it can be easily used in a personal project, via a simple `pip install kuksa-client` or by cloning the official repository.

The code is easy to read, and uses good practices, for example **typing** [14], or **docstring** with well-explained information when necessary. Some linters are executed on the code, and are under discussion to be added to the CI (see this discussion [1]). Various unit tests are also present.

The SDK also contains a command-line interface. This CLI enables us to interact with the broker, to query, update, or subscribe to values. This is a useful tool, which uses the **cmd2** [15]

[1] https://github.com/eclipse/kuksa.val/pull/597#discussion_r1269023945

python package as the CLI backend, natively providing commands to run script or enter a host shell :

```
Test Client> help
...
Uncategorized
=============
alias  help     macro  quit          run_script  shell
edit   history  py     run_pyscript  set         shortcuts
```

All these commands are not required in a classic usage of the kuksa-library, and it adds some content the end user would not expect. In the foreground, the `kuksa-client` command in the path allowing you to bounce on a shell. It also adds various Python dependencies, and potential security issues in those dependencies.

We would suggest to split the Python client in two packages, one for the kuksa-library, and one for the kuksa-client.

8. Dynamic analysis

8.1 Fuzzing the databroker

8.1.1 Why fuzzing?

Both fuzzing and property testing are ways of automatically testing code. Where unit tests typically test some expected set of behavior, automated tests can be more rigorous and exhaustive - making them far more likely to weed out bugs - especially those you didn't even conceive of. Fuzzing and property testing have a lot in common with each other. Where they tend to differ is how tests are driven.

With fuzzing you typically use some external agent to test your program. Fuzzers usually can instrument the code under test, and make use of tools such as sanitizers to check whether invariants are violated. Fuzzers will also keep track of which input leads to which branches being hit in code, and tailor their input to cover as many branches as possible. This process can take time and is why it often pays off to run fuzzers for extended periods or even continuously.

Property testing typically works the other way around: property testing is typically done by including a library in your test code which allows you to drive the tests yourself. Rather than looking for crashes or sanitizer failures, the emphasis is more on checking your implementation using a strategy.

While fuzzing tends to be more thorough, property testing tends to be faster to execute [16].

8.1.2 Structured inputs

It's possible to combine fuzzing with structured inputs using arbitrary crate. The way a fuzzer typically works is that it generates random data which is passed to a program over some channel. But with arbitrary this randomness can be more clearly channeled: it can take the arbitrary stream of data, and use it to create structured types.

8.1.3 Automated testing strategies

These are some common testing strategies:

- **roundtrip testing**: generate a message, pass it to the encoder, then pass the encoder's output to the decoder. The decoder's output should be the same as the original message.
- **differential testing**: test the program against a "known good" implementation of a similar program (also known as an "oracle").
- **invariant testing**: test that a certain property always holds. This can be a universal property like: "my program didn't crash". But invariants can be specific too.

8.1.4 Methodology used for Kuksa

For the audit we chose to only do fuzzing, using proptest [17] or other verification methods like kani [18] requires a deep knowledge of the code we didn't have during the audit.

Since we didn't have a clear model to use as an oracle, the strategy we used was to maximize our coverage. Given the safe nature of Rust and the absence of *unsafe code* in the code base, the failures we expected were mostly panic leading to DOS (Denial Of Service).

We decided to use `cargo fuzz` [19] which is a cargo subcommand for fuzzing with `libFuzzer` [20]. `libFuzzer` is an in-process, coverage-guided, evolutionary fuzzing engine.

`libFuzzer` is linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzzing entrypoint (a.k.a. target function); the fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data to maximize the code coverage.

Since some functions are typed, we decided to go with a structured approach. The raw bytes generated by the fuzzer are interpreted using the `arbitrary` crate [21] to generate an instance of the input.

The `arbitrary` crate provides an `Arbitrary` trait to produce well-typed, structured values, from raw bytes buffers.

A fuzzing harness is developed to bridge the gap between how the fuzzer expects input to occur and how input happens in the application.

Writing a harness with `cargo fuzz` is quite easy: you need to provide a closure to the `fuzz_target` macro. This macro can take parameters that implement the `Arbitrary` trait.

Since `libfuzzer` is in maintenance mode, our harnesses have a feature that can be used to use the shim provided by `libafl` [22].

8.1.5 Harnesses

We developed 12 harnesses targeting various parts of the application. We were interested in all the functions that can take parameters provided by a potential attacker.

During the cartography of the application, we determined 3 objects that can be of interest:

- the JWT token
- the VSS that can be provided
- the gRPC endpoints

Attacking the JWT token requires the ability to sign an arbitrary token. Since the deployment of infrastructure required to sign the token is not provided by Kuksa, we wanted to check if the handling of token was robust enough.

To do that we targeted 3 functions:

- `glob::to_regex`
- `authorization::jwt::scope::parse_whitespace_separated`

- `authorization::jwt::decoder::Permissions::try_from`

For the VSS, it is only provided during the startup of the databroker. To maximize the coverage we developed 2 harnesses both of them targeting the `vss::parse_vss_from_str` functions.

Regarding the gRPC endpoints, 2 GRPC servers are available: `BrokerServer` and `CollectorServer`.

`BrokerServer` has 4 endpoints:

- `get_datapoints`
- `set_datapoints`
- `subscribe`
- `get_metadata`

`CollectorServer` has 3 endpoints:

- `update_datapoints`
- `stream_datapoints`
- `register_datapoints`

We used the arbitrary crate to call directly the server implementation with a gRPC request.

We didn't find a way to fuzz the `stream_datapoints` endpoint without spawning a full gRPC server which will make the fuzzing highly inefficient. So this harness was not used during the fuzzing campaign.

We also decided to target the `query::compiler::compile` function since this function parses arbitrary SQL strings.

To have better coverage we also designed a harness to target specifically the databroker API.

We'll now explain how these harnesses work.

`glob::to_regex`

This harness is really simple since the target function takes a string as input.

```
    fuzz_target!(|data: &str| {
     _ = glob::to_regex(data);
});
```

`authorization::jwt::scope::parse_whitespace_separated`

This harness is really simple since the target function takes a string as input.

```
fuzz_target!(|data: &str| {
    let _ = parse_whitespace_separated(data);
});
```

`authorization::jwt::decoder::Permissions::try_from`

This harness is a bit more complex because we leverage the `Arbitrary` trait to generate valid claims. We wrap the `jwt::Claims` as a new type to keep the implementation of `Arbitrary` inside the harness. It's not mandatory and it'll be best to have that in the databroker crate with a configuration selector.

```
#[derive(Debug)]
struct Claims(jwt::Claims);

impl<'a> arbitrary::Arbitrary<'a> for Claims {
    fn arbitrary(u: &mut arbitrary::Unstructured<'a>) -> arbitrary::Result<Self> {
        let claims = jwt::Claims {
            sub: String::arbitrary(u)?,
            iss: String::arbitrary(u)?,
            aud: Vec::arbitrary(u)?,
            iat: u64::arbitrary(u)?,
            exp: u64::arbitrary(u)?,
            scope: String::arbitrary(u)?,
        };
        Ok(Self(claims))
    }
}

fuzz_target!(|data: Claims| {
    _ = Permissions::try_from(data.0);
});
```

`vss::parse_vss_from_str`

We have 2 harnesses for this function. The first one is quite standard.

```
fuzz_target!(|data: &str| {
    _ = parse_vss_from_str(data);
});
```

The second leverages `Arbitrary` trait to craft a json-like string.

```
fuzz_target!(|data: JsonString| {
    _ = parse_vss_from_str(data.0.as_str());
});
```

`query::compiler::compile`

This harness is also quite standard. If the fuzzer manages to craft a valid query, we try to execute it.

```
fuzz_target!(|data: &str| {
    let test_compilation_input = TestCompilationInput {};
    if let Ok(compiled_query) = compile(data, &test_compilation_input) {
        let execution_input1 = TestExecutionInput {
        };
        _ = compiled_query.execute(&execution_input1);
    }
});
```

grpc_get_datapoints

All gRPC endpoints are asynchronous, so we need to start a runtime to call the future. It is not very efficient and it's something that clearly can be improved. The gRPC request is also wrapped in a new type to craft arbitrary requests. We also need to inject into gRPC extensions the corresponding permissions.

```
#[derive(Debug)]
struct Request(GetDatapointsRequest);

impl<'a> arbitrary::Arbitrary<'a> for Request {
    fn arbitrary(u: &mut arbitrary::Unstructured<'a>) -> arbitrary::Result<Self> {
        let request = GetDatapointsRequest {
            datapoints: Vec::arbitrary(u)?,
        };
        Ok(Self(request))
    }
}

fuzz_target!(|data: Request| {
    let _runtime = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            let broker = DataBroker::default();
            let mut request = tonic::Request::new(data.0);
            request
                .extensions_mut()
                .insert(permissions::ALLOW_ALL.clone());
            let _ = broker.get_datapoints(request).await;
        });
});
```

grpc_set_datapoints

This harness is quite similar to the previous ones.

```
#[derive(Debug)]
struct Request(SetDatapointsRequest);
```

```
impl<'a> arbitrary::Arbitrary<'a> for Request {
    fn arbitrary(u: &mut arbitrary::Unstructured<'a>) -> arbitrary::Result<Self> {
        let datapoints: HashMap<String, Point> = HashMap::arbitrary(u)?;
        let datapoints: HashMap<String, Datapoint> =
            HashMap::from_iter(datapoints.into_iter().map(|(i, p)| (i, p.0)));

        let request = SetDatapointsRequest { datapoints };
        Ok(Self(request))
    }
}

fuzz_target!(|data: Request| {
    let _runtime = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            let broker = DataBroker::default();
            let mut request = tonic::Request::new(data.0);
            request
                .extensions_mut()
                .insert(permissions::ALLOW_ALL.clone());
            let _ = broker.set_datapoints(request).await;
        });
});
```

`grpc_subscribe`

This harness is quite similar to the previous ones.

```
#[derive(Debug)]
struct Request(SubscribeRequest);

impl<'a> arbitrary::Arbitrary<'a> for Request {
    fn arbitrary(u: &mut arbitrary::Unstructured<'a>) -> arbitrary::Result<Self> {
        let request = SubscribeRequest {
            query: String::arbitrary(u)?,
        };
        Ok(Self(request))
    }
}

fuzz_target!(|data: Request| {
    let _runtime = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            let broker = DataBroker::default();
            let mut request = tonic::Request::new(data.0);
            request
                .extensions_mut()
```

```
                .insert(permissions::ALLOW_ALL.clone());
            let _ = Broker::subscribe(&broker, request).await;
        });
});
```

`grpc_get_metadata`

This harness is quite similar to the previous ones.

```
fuzz_target!(|data: Request| {
    let _runtime = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            let broker = DataBroker::default();
            let mut request = tonic::Request::new(data.0);
            request
                .extensions_mut()
                .insert(permissions::ALLOW_ALL.clone());
            let _ = broker.get_metadata(request).await;
        });
});
```

`grpc_update_datapoints`

This harness is quite similar to the previous ones.

```
#[derive(Debug)]
struct Request(UpdateDatapointsRequest);

impl<'a> arbitrary::Arbitrary<'a> for Request {
    fn arbitrary(u: &mut arbitrary::Unstructured<'a>) -> arbitrary::Result<Self> {
        let datapoints: HashMap<i32, Point> = HashMap::arbitrary(u)?;
        let datapoints: HashMap<i32, Datapoint> =
            HashMap::from_iter(datapoints.into_iter().map(|(i, p)| (i, p.0)));

        let request = UpdateDatapointsRequest { datapoints };
        Ok(Self(request))
    }
}

fuzz_target!(|data: Request| {
    let _runtime = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            let broker = DataBroker::default();
            let mut request = tonic::Request::new(data.0);
            request
```

```
                .extensions_mut()
                .insert(permissions::ALLOW_ALL.clone());
            let _ = broker.update_datapoints(request).await;
        });
});
```

`grpc_register_datapoints`

This harness is quite similar to the previous ones.

```
#[derive(Debug)]
struct Request(RegisterDatapointsRequest);

impl<'a> arbitrary::Arbitrary<'a> for Request {
    fn arbitrary(u: &mut arbitrary::Unstructured<'a>) -> arbitrary::Result<Self> {
        let list: Vec<Point> = Vec::arbitrary(u)?;
        let list: Vec<RegistrationMetadata> =
        ↪  Vec::from_iter(list.into_iter().map(|p| p.0));

        let request = RegisterDatapointsRequest { list };

        Ok(Self(request))
    }
}

fuzz_target!(|data: Request| {
    let _runtime = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            let broker = DataBroker::default();
            let mut request = tonic::Request::new(data.0);
            request
                .extensions_mut()
                .insert(permissions::ALLOW_ALL.clone());
            let _ = broker.register_datapoints(request).await;
        });
});
```

Databroker API

This harness is more complex. We use the fuzzer to generate a sequence of functions to call inside the `databroker`.

Our goal is to target the following functions:

- `add_entry`
- `get_entry_by_id`
- `get_entry_by_path`

- `get_datapoint`
- `update_entries`
- `subscribe`
- `subscribe_query`
- `get_id_by_path`
- `get_datapoint_by_path`
- `get_metadata`
- `get_metadata_by_path`

To do so, we leverage once again the `Arbitrary` trait to generate a `Context` used as input for the fuzzer.

This `Context` will have a list of entries we want to initially add to the database and a list of operations we want to apply to the database.

```
#[derive(arbitrary::Arbitrary, Debug)]
struct Context {
    initial: Vec<AddEntry>,
    ops: Vec<Ops>,
}

#[derive(arbitrary::Arbitrary, Debug)]
enum Ops {
    AddEntry(AddEntry),
    GetEntryById(GetEntryById),
    GetEntryByPath(GetEntryByPath),
    GetDatapoint(GetDatapoint),
    UpdateEntries(UpdateEntries),
    Subscribe(Subscribe),
    SubscribeQuery(SubscribeQuery),
    GetIdByPath(GetIdByPath),
    GetDatapointByPath(GetDatapointByPath),
    GetMetadata(GetMetadata),
    GetMetadataByPath(GetMetadataByPath),
}
```

The structure of the fuzzer is then quite standard. We still need to set up a runtime to run our function.

```
fuzz_target!(|data: Context| {
    let _runtime = tokio::runtime::Builder::new_current_thread()
        .enable_all()
        .build()
        .unwrap()
```

```
        .block_on(fuzz(data));
});

async fn fuzz(data: Context) {
    let broker = DataBroker::default();
    let broker = broker.authorized_access(&permissions::ALLOW_ALL);

    let Context { initial, ops } = data;
```

We add the initial values to the database.

```
    let mut ids = Vec::new();
    for i in initial {
        ...
        if let Ok(id) = broker
            .add_entry(
                name,
                data_type,
                change_type,
                entry_type,
                description,
                allowed,
            )
            .await
        {
            ids.push(id);
        }
    }

    if ids.is_empty() {
        return;
    }
```

And then we call the functions with the arguments generated by the fuzzer.

```
    for op in ops {
        match op {
            Ops::AddEntry(i) => {
                ...
                if let Ok(id) = broker
                    .add_entry(
                        name,
                        data_type,
                        change_type,
                        entry_type,
                        description,
                        allowed,
                    )
                    .await
                {
```

```
                ids.push(id);
            }
        }
        Ops::GetEntryById(i) => {
            _ = broker.get_entry_by_id(i.id.id).await;
        }
        Ops::GetEntryByPath(i) => {
            _ = broker.get_entry_by_path(&i.path).await;
        }
        Ops::GetDatapoint(i) => {
            _ = broker.get_datapoint(i.id.id).await;
        }
        Ops::UpdateEntries(i) => {
            let updates: HashMap<i32, EntryUpdate> = i
                .entries
                .into_iter()
                .map(|(k, v)| (k.id, v.into()))
                .collect();
            _ = broker.update_entries(updates).await;
        }
        Ops::Subscribe(i) => {
            let valid_entries: HashMap<i32,
            ↪  HashSet<databroker::broker::Field>> = i
                .entries
                .into_iter()
                .map(|(k, v)| {
                    (
                        k.id,
                        v.into_iter()
                            .map(|v| <Field as
                            ↪  Into<databroker::broker::Field>>::into(v))
                            .collect(),
                    )
                })
                .collect();
            _ = broker.subscribe(valid_entries).await;
        }
        Ops::SubscribeQuery(i) => {
            _ = broker.subscribe_query(&i.query).await;
        }
        Ops::GetIdByPath(i) => {
            _ = broker.get_id_by_path(&i.path).await;
        }
        Ops::GetDatapointByPath(i) => {
            _ = broker.get_datapoint_by_path(&i.path).await;
        }
        Ops::GetMetadata(i) => {
            _ = broker.get_metadata(i.id.id).await;
        }
        Ops::GetMetadataByPath(i) => {
            _ = broker.get_metadata_by_path(&i.path).await;
        }
```

```
        }
    }
}
```

This way we are able to have a very good coverage of the databroker API.

8.1.6 Methodology

We developed each harness by following these steps:

- Run harness for some time
- Stop fuzzing
- Mimimize corpus
- Check coverage to see if enough code is covered
- Improve fuzzer
- Rinse and repeat

8.1.7 Fuzzing campaign

Once we were quite confident with our harnesses, we ran them on a dedicated server for a 24h period. We set up the fuzzing campaign to stop the harnesses if no coverage were discovered after 1h and if more than 250 crashes were found.

The following table summarizes the results. The corpus for each harness is also minimized.

harness	corpus	crashes	coverage	features
`databroker`	7413	272	11281	39538
`glob_to_regex`	6547	10	11885	65288
`grpc_get_datapoints`	149	0	1252	3333
`grpc_get_metadata`	57	0	1041	1408
`grpc_register_datapoints`	205	0	4862	7082
`grpc_set_datapoints`	873	0	2014	6668
`grpc_subscribe`	5675	283	5903	20124
`grpc_update_datapoints`	767	0	2146	6985
`parse_vss_from_json`	2909	0	3860	22213
`parse_vss_from_str`	3924	0	3615	14210
`permissions_try_from`	1173	0	6878	27812
`query_compiler_compile`	3083	369	4574	17987
`scope_parse_whitespace_separated`	229	0	4093	5691

The following harnesses were stopped because no new coverage was found after 1h:

- `scope_parse_whitespace_separated`
- `permissions_try_from`
- `parse_vss_from_str`
- `grpc_get_datapoints`

- `grpc_set_datapoints`
- `grpc_get_metadata`
- `grpc_update_datapoints`
- `grpc_register_datapoints`

The following harnesses were still finding coverage after 24h of fuzzing:

- `glob_to_regex`

The following harnesses produced crashes:

- `query_compiler_compile`
- `grpc_subscribe`
- `databroker`

The following harness produced slow inputs:

- `glob_to_regex`

8.1.8 Coverage

To obtain the global coverage from all the fuzzers we need to merge each coverage into a global one. Since the coverage is gathered by LLVM, it is quite easy to merge it by using `cargo profdata` and `cargo cov`.

```
$ cargo +nightly profdata -- merge -sparse \
    coverage/scope_parse_whitespace_separated/coverage.profdata \
    coverage/permissions_try_from/coverage.profdata \
    coverage/query_compiler_compile/coverage.profdata \
    coverage/databroker/coverage.profdata \
    coverage/glob_to_regex/coverage.profdata \
    coverage/parse_vss_from_str/coverage.profdata \
    coverage/parse_vss_from_json/coverage.profdata \
    coverage/grpc_get_datapoints/coverage.profdata \
    coverage/grpc_set_datapoints/coverage.profdata \
    coverage/grpc_subscribe/coverage.profdata \
    coverage/grpc_get_metadata/coverage.profdata \
    coverage/grpc_update_datapoints/coverage.profdata \
    coverage/grpc_register_datapoints/coverage.profdata \
    -o merged.profdata
$ cargo +nightly cov -- show \
    -object={{binaries_coverage_path}}/scope_parse_whitespace_separated \
    -object={{binaries_coverage_path}}/permissions_try_from \
    -object={{binaries_coverage_path}}/query_compiler_compile \
    -object={{binaries_coverage_path}}/databroker \
    -object={{binaries_coverage_path}}/glob_to_regex \
    -object={{binaries_coverage_path}}/parse_vss_from_str \
    -object={{binaries_coverage_path}}/parse_vss_from_json \
    -object={{binaries_coverage_path}}/grpc_get_datapoints \
```

```
        -object={{binaries_coverage_path}}/grpc_set_datapoints \
        -object={{binaries_coverage_path}}/grpc_subscribe \
        -object={{binaries_coverage_path}}/grpc_get_metadata \
        -object={{binaries_coverage_path}}/grpc_register_datapoints \
        -object={{binaries_coverage_path}}/grpc_update_datapoints \
        -instr-profile merged.profdata \
        -ignore-filename-regex="rustc/.*" \
        -ignore-filename-regex=".cargo/registry/.*" \
        -ignore-filename-regex="databroker-proto/.*" \
        -ignore-filename-regex="databroker/fuzz/.*" \
        -output-dir merged -format html
```

We obtain the following coverage. It is quite good since the fuzzers manage to trigger a good part of the application. The majority of what is missing is either implementation not covered by the fuzzers (like the `Debug` implementation) or code we couldn't trigger because crashes happened before (like the SQL executor).

Coverage Report

Created: 2023-12-04 10:58

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
.cargo/git/checkouts/arbitrary-json-492b08e17393dd46/b7a3461/src/lib.rs	53.33% (8/15)	68.52% (37/54)	66.67% (46/69)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/authorization/jwt/decoder.rs	20.00% (2/10)	46.67% (28/60)	40.00% (26/65)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/authorization/jwt/scope.rs	33.33% (2/6)	90.57% (48/53)	77.42% (24/31)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/authorization/mod.rs	0.00% (0/5)	0.00% (0/8)	0.00% (0/9)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/broker.rs	53.60% (67/125)	74.95% (754/1006)	74.10% (512/691)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/glob.rs	66.67% (4/6)	92.45% (49/53)	93.33% (28/30)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/grpc/kuksa_val_v1/conversions.rs	0.00% (0/9)	0.00% (0/264)	0.00% (0/112)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/grpc/kuksa_val_v1/val.rs	0.00% (0/31)	0.00% (0/549)	0.00% (0/299)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/grpc/sdv_databroker_v1/broker.rs	40.00% (10/25)	58.95% (112/190)	61.06% (69/113)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/grpc/sdv_databroker_v1/collector.rs	26.09% (6/23)	55.23% (95/172)	50.65% (39/77)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/grpc/sdv_databroker_v1/conversions.rs	54.55% (6/11)	32.44% (73/225)	42.55% (60/141)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/grpc/server.rs	0.00% (0/13)	0.00% (0/131)	0.00% (0/53)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/lib.rs	0.00% (0/3)	0.00% (0/17)	0.00% (0/6)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/permissions.rs	76.92% (20/26)	64.33% (110/171)	60.00% (69/115)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/query/compiler.rs	45.45% (5/11)	32.70% (104/318)	41.03% (64/156)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/query/executor.rs	18.18% (2/11)	12.78% (23/180)	17.54% (20/114)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/query/expr.rs	14.29% (1/7)	21.05% (4/19)	15.00% (3/20)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/types.rs	37.50% (6/16)	1.83% (6/327)	4.64% (11/237)	- (0/0)
kuksa/kuksa.val/kuksa_databroker/databroker/src/vss.rs	90.00% (63/70)	92.06% (290/315)	92.21% (296/321)	- (0/0)
Totals	**47.75% (202/423)**	**42.14% (1733/4112)**	**47.65% (1267/2659)**	**- (0/0)**

Generated by llvm-cov -- llvm version 17.0.4-rust-1.76.0-nightly

Figure 1. Fuzzing global coverage

8.1.9 Crashes triage

Since the fuzzing campaign produced a lot of crashes (even if we reduced their amount), we used `casr` (CASR: Crash Analysis and Severity Report) [23] to reduce them. This tool will try to reproduce the crashes and triage them in buckets sharing the same root cause.

In our case, several harnesses produced crashes or failing inputs.

- `permissions_try_from`
- `query_compiler_compile`

- `grpc_subscribe`
- `databroker`
- `glob_to_regex`

For the `databroker` harness, 245 crashes were sorted in 8 clusters. All clusters are various stack exhaustion in the `sqlparser` crate.

For the `grpc_subscribe` harness, 244 crashes were sorted in 27 clusters. All clusters are also various stack exhaustion in the `sqlparser` crate.

Details are available in the appendix A.

8.1.10 Findings

Overflow when adding duration to instant in `Permissions::try_from`

The first crash found by the fuzzer lies in the implementation of `Permissions::try_from` function.

```
thread '<unnamed>' panicked at library/std/src/time.rs:601:31:
overflow when adding duration to instant

Failing input:

        artifacts/permissions_try_from/crash-d6ad6ccef535c90f

Output of `std::fmt::Debug`:

        Claims(
            Claims {
                sub: "Z",
                iss: "",
                aud: [
                    "",
                ],
                iat: 13301911657259272913,
                exp: 14344505347206954180,
                scope: "",
            },
        )
```

The root cause is hinted at in the Rust documentation: *mathematical operations like add may panic if the underlying structure cannot represent the new point in time.*

```rust
impl TryFrom<Claims> for Permissions {
    type Error = Error;

    fn try_from(claims: Claims) -> Result<Self, Self::Error> {
```

```
        let scopes =
        ↪  scope::parse_whitespace_separated(&claims.scope).map_err(|err| match
        ↪  err {
            scope::Error::ParseError => Error::ClaimsError,
        })?;

        ...

        permissions = permissions
            .expires_at(std::time::UNIX_EPOCH +
            ↪  std::time::Duration::from_secs(claims.exp));

        permissions.build().map_err(|err| match err {
            PermissionsBuildError::BuildError => Error::ClaimsError,
        })
    }
}
```

Our recommendation is to use the `checked_add` function to do the addition.

LOW	LOW-5 Malicious JWT access token can crash a thread of the databroker	
Exploitability	●○○○	**Impact** ●○○○
Perimeter	Databroker JWT handling	
Prerequisites	JWT crafting capability	
Description		
Timestamp addition doesn't support numbers that are too big and will panic on the databroker side at jwt/scope.rs#L134.		
Recommendations		
Consider using a Rust-safe addition.		

Stack exhaustion in `query::compiler::compile`

```
$ cargo fuzz run query_compiler_compile
↪  artifacts/query_compiler_compile/crash-7b38f34b0d58a1526fa09cc20ea3169df633e33c
↪
...
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 547689466
INFO: Loaded 1 modules   (1408673 inline 8-bit counters): 1408673 [0x555b64034630,
↪  0x555b6418c4d1),
INFO: Loaded 1 PC tables (1408673 PCs): 1408673 [0x555b6418c4d8,0x555b6570aee8),
target/x86_64-unknown-linux-gnu/release/query_compiler_compile: Running 1 inputs 1
↪  time(s) each.
Running:
↪  artifacts/query_compiler_compile/crash-7b38f34b0d58a1526fa09cc20ea3169df633e33c
```

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==655278==ERROR: AddressSanitizer: stack-overflow on address 0x7ffec09cef20 (pc
↪  0x555b5feb28b8 bp 0x7ffec09d0af0 sp 0x7ffec09cef20 T0)
    #0 0x555b5feb28b8
    ↪  (kuksa.val/kuksa_databroker/databroker/fuzz/target/x86_64-unknown-linux-gnu/release/query_
    ↪  (BuildId: 15994ef7afeb661301822cc00f3dae293c3aa4df)
    #1 0x555b5feb8a1e
    ↪  (kuksa.val/kuksa_databroker/databroker/fuzz/target/x86_64-unknown-linux-gnu/release/query_
    ↪  (BuildId: 15994ef7afeb661301822cc00f3dae293c3aa4df)
    ...
 SUMMARY: AddressSanitizer: stack-overflow
  ↪  (kuksa.val/kuksa_databroker/databroker/fuzz/target/x86_64-unknown-linux-gnu/release/query_com
  ↪  (BuildId: 15994ef7afeb661301822cc00f3dae293c3aa4df)
==655278==ABORTING
```

The root cause is the parsing of the SQL expression, the `sqlparser` crate crashes because it exceeds the number of stacks (256 by default).

```
match sqlparser::parser::Parser::parse_sql(&dialect, sql)
```

This issue is well-known and patched in the `sqlparser` repository [24]. We decided to update the `sqlparser` crate to the latest version available (0.40.0 at the time of the writing). This specific crash was fixed but the fuzzer continues to find issues even on the latest version (See the appendix for more details). All issues are stack exhaustions in various parts of the crate.

Here is an example of another input producing the stack exhaustion.

```
└─ cargo fuzz fmt grpc_subscribe triage/grpc_subscribe/cl1/crash-040ece4bda39820ab6988d6d10
37dbc7e1f2709f

Output of `std::fmt::Debug`:

Request(
    SubscribeRequest {
        query: "ASSERT((-\"\"(((((SF++++++++++++++++++++++++++++++++++++++++xxx(xxx(+++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++(++++++++++++++++++++++++++++++++++xxx(
xxxxxxxxxxxxxZ(((((++++++++++++++++++++++++++++++++++++++++++++++++++++++++[++++++++++++
++++++++++++++++++++++xxx(xxx(+++NA++++++++++++++++++++++++++++++++++++++++++++++++xxx(x
xx(++++++++++++++++++++++++++++++++++++++++++++++++++++++++(++++++++++++++++++++++++++++
+++++++xxx(xxxxxxxxxxxxxZ(((((++++++++++++++++++++++++++++++++++++++++++++++++++++++++++[
++++++++++++++++++++++++++++++++++xxx(xxx(++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++xxx(xx|\nW\n||+++++++++++++++T((-\"\"(((((
SF++++++++++++++++++++++++++++++++++++++++xxx(xxx(++++++++++++++++++++++++++++++++++++++
+++++++++++++++(++++++++++++++++++++++++++++++++++++xxx(xxxxxxxxxxxxxZ(((((++++++++++++",
    },
)
```

Figure 1. Failing SQL query

Some users also reported crashes due to stack exhaustion [25].

HIGH	HIGH-2 Any user can crash the databroker		
Exploitability	●●●○	**Impact**	●●●○
Perimeter	Databroker subscriptions		
Prerequisites	`Read` permissions		

Description

A crafted SQL request sent through the `Subscribe` gRPC endpoint can lead to a `stack exhaustion` error resulting in a crash in the databroker.
This vulnerability is still present (at the date of the writing) in the `sqlparser` library.

Recommendations

We think that having a full-fledged SQL parser as a core dependency is not required given this use case. Maybe it should be considered to move to a custom DSL tailored for the task. It will reduce the attack surface.

Unresolved literal should result in compilation error in `query::executor::Expr`

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1032748402
INFO: Loaded 1 modules   (1403216 inline 8-bit counters): 1403216 [0x55a4d2596b90,
↪  0x55a4d26ed4e0),
INFO: Loaded 1 PC tables (1403216 PCs): 1403216 [0x55a4d26ed4e0,0x55a4d3c569e0),
target/x86_64-unknown-linux-gnu/release/grpc_subscribe: Running 1 inputs 1 time(s)
↪  each.
Running: artifacts/grpc_subscribe/crash-6f4f1b03ce518e6baa48b2292fe6cc5fde226774
thread '<unnamed>' panicked at
↪  kuksa.val/kuksa_databroker/databroker/src/query/executor.rs:106:17:
Unresolved literal should result in compilation error
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
==126395== ERROR: libFuzzer: deadly signal
    #0 0x55a4cd5d0de1
    ↪  (kuksa.val/kuksa_databroker/databroker/fuzz/target/x86_64-unknown-linux-gnu/release/grpc_s
    ↪  (BuildId: 350cd0b3d9fd0ead431c16ece2834bdc1619bf43)
    #1 0x55a4cd6ad0dd
    ↪  (kuksa.val/kuksa_databroker/databroker/fuzz/target/x86_64-unknown-linux-gnu/release/grpc_s
    ↪  (BuildId: 350cd0b3d9fd0ead431c16ece2834bdc1619bf43)
    #2 0x55a4cd6c49d9
    ↪  (kuksa.val/kuksa_databroker/databroker/fuzz/target/x86_64-unknown-linux-gnu/release/grpc_s
    ↪  (BuildId: 350cd0b3d9fd0ead431c16ece2834bdc1619bf43)
...
NOTE: libFuzzer has rudimentary signal handlers.
      Combine libFuzzer with AddressSanitizer or similar for better crash reports.
SUMMARY: libFuzzer: deadly signal


Error: Fuzz target exited with exit status: 77

Request(
```

```
    SubscribeRequest {
        query: "\nSELECT\n\nOWITUES\n",
    },
)
```

The root cause is in the `query::executor::Expr::execute` function. A `debug_assert` is still present. Also, there is a `todo!()` which will cause a panic when called.

```
impl Expr {
    pub fn execute(&self, input: &impl ExecutionInput) -> Result<DataValue,
    ↪  ExecutionError> {
        match &self {
            Expr::Datapoint { name, data_type: _ } => Ok(input.lookup(name)),
            Expr::Alias { expr, .. } => expr.execute(input),
            ...
            Expr::Cast {
                expr: _,
                data_type: _,
            } => todo!(),
            Expr::UnresolvedLiteral { raw: _ } => {
                debug_assert!(
                    false,
                    "Unresolved literal should result in compilation error"
                );
                Err(ExecutionError::TypeError(
                    "Unresolved literal found while executing query".to_string(),
                ))
            }
        }
    }
}
```

INFO	INFO-5 `debug_assert` in `executor.rs`
Exploitability	**Impact**
Perimeter	Databroker subscriptions
Prerequisites	
Description	
A `debug_assert` statement is executed at line 106 of `executor.rs` .	
Recommendations	
Remove the `debug_assert` and the `todo` statements.	

Several slow inputs in `glob::to_regex`

`glob_to_regex` didn't produce any crashes but rather very slow inputs. For example, some input takes 50s to execute.

```
$ cargo fuzz run glob_to_regex
↪  artifacts/glob_to_regex/slow-unit-2641e328ba32e8c122cc584497f9a8d1565ab211
   Finished release [optimized + debuginfo] target(s) in 0.19s
   Finished release [optimized + debuginfo] target(s) in 0.20s
    Running `target/x86_64-unknown-linux-gnu/release/glob_to_regex
     ↪  -artifact_prefix=kuksa.val/kuksa_databroker/databroker/fuzz/artifacts/glob_to_regex/
     ↪  artifacts/glob_to_regex/slow-unit-2641e328ba32e8c122cc584497f9a8d1565ab211`
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2306060899
INFO: Loaded 1 modules   (1394821 inline 8-bit counters): 1394821 [0x560e72492730,
↪  0x560e725e6fb5),
INFO: Loaded 1 PC tables (1394821 PCs): 1394821 [0x560e725e6fb8,0x560e73b2f808),
target/x86_64-unknown-linux-gnu/release/glob_to_regex: Running 1 inputs 1 time(s)
↪  each.
Running:
↪  artifacts/glob_to_regex/slow-unit-2641e328ba32e8c122cc584497f9a8d1565ab211
Executed
↪  artifacts/glob_to_regex/slow-unit-2641e328ba32e8c122cc584497f9a8d1565ab211 in
↪  49256 ms
***
*** NOTE: fuzzing was not performed, you have only
***       executed the target code on a fixed set of inputs.
***
```

Here is the input file.

Figure 1. Slow input

We didn't manage to trigger these code paths but it is known that the ability to craft user-controlled regular expressions can lead to complex state machines which will result in a long evaluation time.

INFO	INFO-6 Slow input in `glob::to_regex()`
Exploitability	**Impact**
Perimeter	Databroker
Prerequisites	

Description

A crafted input sent to `glob::to_regex()` seems to slow down the function execution. We were not able to exploit it remotely.

Recommendations

As a rule of thumb, regular expressions should not be controlled from the outside. We recommend to move to a custom parser to handle the parsing of the scope.

8.1.11 Integration to OSS Fuzz

Since all the fuzzers are based on `cargo fuzz`, integration in OSS Fuzz is straightforward. The required steps are described in the OSS-Fuzz documentation [26].

8.2 Fuzzing the KUKSA Python SDK

The fourth scenario of the thread model suggested an attacker able to hijack the gRPC communication sent to the client, either by setting up a man in the middle attack, or by taking control of the databroker.

The Python client is using the following services exposed by the Databroker:

```
service VAL {
  rpc Get(GetRequest) returns (GetResponse);
  rpc Set(SetRequest) returns (SetResponse);
  rpc Subscribe(SubscribeRequest) returns (stream SubscribeResponse);
  rpc GetServerInfo(GetServerInfoRequest) returns (GetServerInfoResponse);
}
```

Each of these services will result in a Protobuf response which will be parsed by a dedicated function. For example, the `GetServerInfo` will result in a call to the `ServerInfo.from_message` defined by:

```
@dataclasses.dataclass
class ServerInfo:
  name: str
  version: str

  @classmethod
  def from_message(cls, message: val_pb2.GetServerInfoResponse):
    return cls(name=message.name, version=message.version)
```

We can easily write some Python code to see how the parser behaves:

```
In [1]: from kuksa_client.grpc import *
   ...: from kuksa.val.v1 import val_pb2
   ...:
   ...: resp = val_pb2.GetServerInfoResponse(name="fakesrv",version=None)
   ...: ServerInfo.from_message(resp)
Out[1]: ServerInfo(name='fakesrv', version='')
```

An interesting point to note in the previous snippet is that a field not marked as optional in the protobuf message definition, `version` in this example, can still be missing from the Protobuf message. One notes that in the context of the `ServerInfo` parser, there is no side effect.

8.2.1 Error response returned by the databroker

The `Get` and `Set` RPC will both start by checking if the broker replies with an error code. Indeed, the client could ask to read or modify an unknown VSS entry, or require special authorizations unsatisfied by the current client context.

Error messages are defined by the following proto definition:

```
message Error {
  uint32 code    = 1;
  string reason  = 2;
  string message = 3;
}
```

In the meantime, the Python function checking if a response contains an error from the databroker, whether responding to an RPC Get or Set calls, does not verify if the `code` field is present in the error message:

```
def _raise_if_invalid(self, response):
  if response.HasField('error'):
      error = json_format.MessageToDict(
          response.error, preserving_proto_field_name=True)
  else:
      error = {}
  ...
  if (error and error['code'] is not http.HTTPStatus.OK) # <-- error['code'] ?
```

So by crafting a response message containing an error without `code` field:

```
def Get(self, request, context):
  resp = val_pb2.GetResponse.FromString(b'\x1a\r\x12\x06reason\x1a\x03msg')
  return resp
```

We can raise a Python exception not handled by the framework:

```
$ kuksa-client
Connecting to VSS server at 127.0.0.1 port 55555 using KUKSA GRPC protocol.
...
gRPC channel connected.
Test Client> getValue Vehicle.Speed
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/lib/python3.11/threading.py", line 1038, in _bootstrap_inner
    self.run()
  File "/.../kuksa_client/__init__.py", line 103, in run
    self.loop.run_until_complete(self.backend.mainLoop())
  ...
  File "/.../kuksa_client/grpc/__init__.py", line 578, in _process_get_response
    self._raise_if_invalid(response)
  File "/.../kuksa_client/grpc/__init__.py", line 638, in _raise_if_invalid
    if (error and error['code'] is not http.HTTPStatus.OK) or any
                  ~~~~~^^^^^^^^
KeyError: 'code'
{"error": "Timeout"}
```

Please note that once a non-expected error, like the previous one, is thrown, any other query

from the client will result in a timeout error message, even if the databroker is restored to its normal state.

LOW	LOW-6 A malicious Protobuf error message can trigger an unhandled error
Exploitability	●○○○ **Impact** ●○○○
Perimeter	Python SDK - Protobuf messages parsing
Prerequisites	A protobuf message embedding an error without code field
Description	
Sending this message will result in an error and block further queries to the broker.	
Recommendations	
Check the presence of the attribute before using it.	

8.2.2 The DataEntry type

If the `Set` RPC does nothing more than checking there is no error code in the response, obviously, the `Get` RPC expects some content. We can ask the databroker for a list of values, so the response should contain a list of answers. Indeed, the `GetResponse` message is defined by:

```
message GetResponse {
  repeated DataEntry entries     = 1;
  repeated DataEntryError errors = 2;
  Error error                    = 3;
}
```

In the meantime, the `Subscribe` RPC will return a `SubscribeResponse` message, which basically is just a list of `EntryUpdate`:

```
message EntryUpdate {
    DataEntry entry       = 1;
    repeated Field fields = 2;
}
...
message SubscribeResponse {
  repeated EntryUpdate updates = 1;
}
```

So in the end, whether we call `Set` or `Subscribe`, this will result in a `DataEntry` type parsing, which basically describes a VSS entry. This protobuf type is quite complex, containing other complex subtypes like `Datapoint` and `Metadata`, the latter also having quite complex definitions.

8.2.3 The EntryUpdate.from_message() function

The `EntryUpdate` type is the "highest" object exposing a `from_message` function which takes as input a raw protobuf message. Indeed, the `GetResponse` message is parsed by `BaseVSSClient._process_get_response`, just by checking if there is an error in the response, and then parsing the `DataEntry` entries. Concerning an `EntryUpdate`, the parsing is done by the following code :

```
  def from_message(cls, message: val_pb2.EntryUpdate):
return cls(
    entry=DataEntry.from_message(message.entry),
    fields=[Field(field) for field in message.fields],
)
```

In the end, this makes the `DataEntry.from_message function` an interesting candidate for fuzzing, in order to obtain the best code coverage, and to avoid all the requirements of creating an instance of `BaseVSSClient`.

We decided to use **Atheris**[27] as the fuzzing engine. First, because this is a coverage-guided Python based off of **libFuzzer** [20], but also because it is the recommended solution to integrate a Python project on OSS-Fuzz [28].

Using Atheris is as easy as installing the **pip** package and running a small snippet of Python code like :

```
import sys
import atheris

with atheris.instrument_imports():
  from kuksa.val.v1 import val_pb2,types_pb2
  from kuksa_client.grpc import *

def TestOneInput(data):
  de = types_pb2.DataEntry.FromString(data)
  DataEntry.from_message(de)

atheris.Setup(sys.argv, TestOneInput)
atheris.Fuzz()
```

However, this code will crash on the first test case in the `types_pb2.DataEntry.FromString` function. Indeed, the protobuf generated code expects a **structured content**. Sending a buffer of null bytes is enough to crash the protobuf DataEntry initialization :

```
  === Uncaught Python exception: ===
  DecodeError: Error parsing message
  Traceback (most recent call last):
    File "/fuzzme.py", line 13, in TestOneInput
      de = types_pb2.DataEntry.FromString(data)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
  DecodeError: Error parsing message

  ==24554== ERROR: libFuzzer: fuzz target exited
  SUMMARY: libFuzzer: fuzz target exited
  MS: 0 ; base unit: 0000000000000000000000000000000000000000
```

Here comes another good point for Atheris, being based off of libFuzzer, it supports **libprotobuf-mutator** [29]. The overall idea is to use the protobuf definition to mutate the fields with type-valid value.

Using **atheris_libprotobuf_mutator**, we just have to slightly modify the previous code to provide to the fuzzer engine the proto definition we want to mutate :

```
import sys
import atheris
import atheris_libprotobuf_mutator

from kuksa.val.v1 import val_pb2,types_pb2
from kuksa_client.grpc import *

@atheris.instrument_func
def TestOneProtoInput(msg):
  # msg will be a valid types_pb2.DataEntry
  # we just try to parse it ...
  d = DataEntry.from_message(msg)

if __name__ == '__main__':
  atheris_libprotobuf_mutator.Setup(sys.argv,
    TestOneProtoInput, proto=types_pb2.DataEntry)
  atheris.Fuzz()
```

Few seconds after being started, the fuzzer identified **4 distinct test cases** triggering uncaught Python Exception, which are presented in the following parts. We then let the fuzzer run for half a day, reaching half a billion testcases, without discovering new issues :

```
root@1b5679b00512:/src/kuksa.val/kuksa-client# python3 fuzzme.py
...
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 495542489
...
#32768  pulse  cov: 54 ft: 73 corp: 14/999b lim: 4096 exec/s: 16384 rss: 54Mb
...
#536870912      pulse  cov: 54 ft: 73 corp: 14/999b lim: 4096 exec/s: 11698 rss:
↪  55Mb
```

The following sections describe those four issues.

Datapoint without value

```
=== Uncaught Python exception: ===
TypeError: getattr(): attribute name must be string
Traceback (most recent call last):
  File
  ↪  "/usr/local/lib/python3.8/site-packages/atheris_libprotobuf_mutator/helpers.py",
  ↪  line 69, in TestOneProtoInputImpl
    test_one_proto_input(msg)
  File "fuzz-from_message.py", line 12, in TestOneProtoInput
    d = DataEntry.from_message(msg)
  File "/usr/local/lib/python3.8/site-packages/kuksa_client/grpc/__init__.py",
  ↪  line 449, in from_message
    entry_kwargs['value'] = Datapoint.from_message(message.value)
  File "/usr/local/lib/python3.8/site-packages/kuksa_client/grpc/__init__.py",
  ↪  line 312, in from_message
    value=getattr(message, message.WhichOneof('value')),
TypeError: getattr(): attribute name must be string

==84== ERROR: libFuzzer: fuzz target exited
SUMMARY: libFuzzer: fuzz target exited
MS: 1 CustomCrossOver-; base unit: adc83b19e793491b1c6ea0fd8b46cd9f32e592fc
...
value {\012}\012
```

LOW	LOW-7 Datapoint.from_message() does not check if no value is provided.
Exploitability	●○○○ **Impact** ●○○○
Perimeter	Python SDK - Protobuf messages parsing
Prerequisites	Forging a Datapoint
Description	
`getattr(message, message.WhichOneof('value'))` will throw an Error if no value is provided in `types_pb2.Datapoint`.	
Recommendations	
Check the field value is not `None` before using it	

Crafted ValueRestrictions field

`ValueRestriction` contains only `oneof` specific value restriction (`Int`, `UInt`, `String`). If for example the type is `ValueRestrictionString`, it will not contain a min or max field. Checking the protobuf object field will result in an error, HasField throws an error if you check for an attribute not defined in the proto definition.

```
=== Uncaught Python exception: ===
ValueError: Protocol message ValueRestrictionString has no "min" field.
Traceback (most recent call last):
```

```
  File
  ↪  "/usr/local/lib/python3.8/site-packages/atheris_libprotobuf_mutator/helpers.py",
  ↪  line 69, in TestOneProtoInputImpl
    test_one_proto_input(msg)
  File "fuzz-from_message.py", line 12, in TestOneProtoInput
    d = DataEntry.from_message(msg)
  File "/usr/local/lib/python3.8/site-packages/kuksa_client/grpc/__init__.py",
  ↪  line 460, in from_message
    entry_kwargs['metadata'] = Metadata.from_message(message.metadata)
  File "/usr/local/lib/python3.8/site-packages/kuksa_client/grpc/__init__.py",
  ↪  line 171, in from_message
    if value_restriction.HasField(field):
ValueError: Protocol message ValueRestrictionString has no "min" field.

==698== ERROR: libFuzzer: fuzz target exited
SUMMARY: libFuzzer: fuzz target exited
MS: 2 InsertByte-Custom-; base unit: adc83b19e793491b1c6ea0fd8b46cd9f32e592fc
...
metadata {\012  value_restriction {\012    string {\012      allowed_values:
↪  \",\"\012    }\012  }\012}\012
```

LOW	LOW-8 DataEntry.From_message() and ValueRestriction.
Exploitability	●○○○ **Impact** ●○○○
Perimeter	Python SDK - Protobuf messages parsing
Prerequisites	Forging invalid ValueRestriction in DataEntry
Description	
ValueRestrictionString and ValueRestrictionInt/Uint/Float shares some parsing code leading to issues with min and max fields	
Recommendations	
Check the content type when dealing with `Oneof` Field	

ValueRestriction without type field

```
=== Uncaught Python exception: ===
TypeError: getattr(): attribute name must be string
Traceback (most recent call last):
  File
  ↪  "/usr/local/lib/python3.8/site-packages/atheris_libprotobuf_mutator/helpers.py",
  ↪  line 69, in TestOneProtoInputImpl
    test_one_proto_input(msg)
  File "fuzz-from_message.py", line 12, in TestOneProtoInput
    d = DataEntry.from_message(msg)
  File "/usr/local/lib/python3.8/site-packages/kuksa_client/grpc/__init__.py",
  ↪  line 460, in from_message
    entry_kwargs['metadata'] = Metadata.from_message(message.metadata)
```

```
  File "/usr/local/lib/python3.8/site-packages/kuksa_client/grpc/__init__.py",
  ↪  line 167, in from_message
    value_restriction = getattr(
TypeError: getattr(): attribute name must be string

==746== ERROR: libFuzzer: fuzz target exited
SUMMARY: libFuzzer: fuzz target exited
MS: 6
↪  ChangeByte-Custom-CustomCrossOver-CustomCrossOver-CustomCrossOver-CustomCrossOver-;
↪  base unit: adc83b19e793491b1c6ea0fd8b46cd9f32e592fc
...
actuator_target {\012  uint32: 0\012}\012metadata {\012  deprecation: \"\"\012
↪  value_restriction {\012  }\012}\012
```

LOW	LOW-9 ValueRestriction without type field
Exploitability	●○○○ **Impact** ●○○○
Perimeter	Python SDK - Protobuf messages parsing
Prerequisites	Forging invalid ValueRestriction in DataEntry

Description

ValueRestriction parsing supposes there is a `Oneof` type field

Recommendations

Check the field value is not `None` before using it

Converting Uint64 to timestamp can result in invalid date

```
=== Uncaught Python exception: ===
OverflowError: Python int too large to convert to C int
Traceback (most recent call last):
  File
  ↪  "/usr/local/lib/python3.8/site-packages/atheris_libprotobuf_mutator/helpers.py",
  ↪  line 69, in TestOneProtoInputImpl
    test_one_proto_input(msg)
  File "/data/kuksa.val/kuksa-client/fuzzme/fuzzme-pb2.py", line 13, in
  ↪  TestOneProtoInput
    d = Datapoint.from_message(msg)
  File "/usr/local/lib/python3.8/site-packages/kuksa_client/grpc/__init__.py",
  ↪  line 315, in from_message
    timestamp=message.timestamp.ToDatetime(
  File
  ↪  "/usr/local/lib/python3.8/site-packages/google/protobuf/internal/well_known_types.py",
  ↪  line 232, in ToDatetime
    delta = datetime.timedelta(
OverflowError: Python int too large to convert to C int

==1971== ERROR: libFuzzer: fuzz target exited
SUMMARY: libFuzzer: fuzz target exited
```

```
MS: 6 CrossOver-Custom-CustomCrossOver-CustomCrossOver-ChangeBit-Custom-; base
↪  unit: adc83b19e793491b1c6ea0fd8b46cd9f32e592fc
...
timestamp {\012  seconds: -9223372036854775808\012}\012uint32_array {\012}\012
```

LOW	LOW-10 No check on timestamp uint value	
Exploitability	●○○○	**Impact** ●○○○
Perimeter	Python SDK - Protobuf messages parsing	
Prerequisites	Forging invalid ValueRestriction in DataEntry	
Description		
Timestamp field conversion of values greater than 2**38 triggers an error		
Recommendations		
Catch exception on timestamp parsing		

Depending on the value, we can trigger different kind of error with an invalid timestamp value. If the value is between 2^{38} and 2^{48}, this triggers the following error :

```
File ~/.../site-packages/google/protobuf/internal/well_known_types.py:232, in
↪  Timestamp.ToDatetime(self, tzinfo)
215 """Converts Timestamp to a datetime.
...
--> 232 delta = datetime.timedelta(
    233     seconds=self.seconds,
    234     microseconds=_RoundTowardZero(self.nanos, _NANOS_PER_MICROSECOND),
    235 )
OverflowError: days=1628906115; must have magnitude <= 999999999}
```

8.2.4 Coverage

The following table summarizes the coverage.

Name	Stmts	Miss	Cover
`/src/kuksa.val/kuksa_certificates/__init__.py`	3	0	100%
`fuzz-from_message.py`	13	0	100%
`kuksa/__init__.py`	0	0	100%
`kuksa/val/__init__.py`	0	0	100%
`kuksa/val/v1/__init__.py`	0	0	100%
`kuksa/val/v1/types_pb2.py`	64	52	19%
`kuksa/val/v1/val_pb2.py`	38	26	32%
`kuksa/val/v1/val_pb2_grpc.py`	43	23	47%
`kuksa_client/__init__.py`	46	21	54%
`kuksa_client/_metadata.py`	25	5	80%
`kuksa_client/cli_backend/__init__.py`	25	19	24%
`kuksa_client/grpc/__init__.py`	547	276	50%
TOTAL	804	422	48%

If these results may suggest that only half of the code was covered, one must keep in mind that we focused on a malicious broker sending fake responses to the gRPC queries. This means that all the code preparing queries and sending data to the broker was not covered, in particular all the `to_message()` functions from the various dataclasses. By looking at the coverage report provided with the deliverables, we can note that the entire code dealing with protobuf messages parsing has been covered.

To be complete, we considered adding conversion functions to protobuf messages to the harness, which could be easily done by calling `to_message()` on the result of the parsing of the samples sent by the fuzzer :

```
@atheris.instrument_func
def TestOneProtoInput(msg):
  d = DataEntry.from_message(msg)
  res = d.to_message()
```

Trying this harness triggered a lot of errors, the code was not intended to handle VSS Python DataEntry not well defined. And from a security point of view, this makes no sense. Indeed, an attacker who could generate a fake Python DataClass would already control the Python client.

9. Conclusion

To conclude, Quarkslab found several vulnerabilities in the KUKSA.val codebase, thanks to automated tools and manual investigations. Some of these issues can be exploited in real-world use cases to impact the databroker availability.

Moreover, Quarkslab provided leads and strategies on how to implement static and dynamic security analysis of the KUKSA.val databroker. Once implemented, these strategies will enhance the overall security level of the KUKSA project.

9. Bibliography

[1] *Kuksa - About*. URL: https://eclipse-kuksa.github.io/kuksa-website/about/ (visited on 12/07/2023).

[2] *Vehicle Signal Specification*. URL: https://covesa.github.io/vehicle_signal_specification/ (visited on 12/07/2023).

[3] *GitHub - rust-lang/rust-clippy: Clippy*. URL: https://github.com/rust-lang/rust-clippy (visited on 12/07/2023).

[4] *RustSec: cargo audit*. URL: https://crates.io/crates/cargo-audit (visited on 12/07/2023).

[5] *GitHub - rustsec/advisory-db: repository of security advisories filed against Rust crates*. URL: https://github.com/RustSec/advisory-db/ (visited on 12/07/2023).

[6] *RustSec: cargo outdated*. URL: https://crates.io/crates/cargo-outdated (visited on 12/07/2023).

[7] *RUSTSEC-2021-0145 - atty: Potential unaligned read*. URL: https://rustsec.org/advisories/RUSTSEC-2021-0145 (visited on 12/07/2023).

[8] *GitHub - rustsec/audit-check: Rust audit-check Action*. URL: https://github.com/rustsec/audit-check (visited on 12/07/2023).

[9] *GitHub - rustls/rustls: Rustls is a modern TLS library written in Rust*. URL: https://github.com/rustls/rustls (visited on 12/07/2023).

[10] *GitHub - rustls/rustls: Security review*. URL: https://github.com/rustls/rustls/blob/main/audit/TLS-01-report.pdf (visited on 12/07/2023).

[11] *RFC 9068 - JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens*. URL: https://datatracker.ietf.org/doc/html/rfc9068 (visited on 12/07/2023).

[12] *RFC 6749 - The OAuth 2.0 Authorization Framework*. URL: https://datatracker.ietf.org/doc/html/rfc6749 (visited on 12/07/2023).

[13] *GitHub - ktr0731/evans: more expressive universal gRPC client*. URL: https://github.com/ktr0731/evans (visited on 12/07/2023).

[14] *typing — Support for type hints*. URL: https://docs.python.org/3/library/typing.html (visited on 12/07/2023).

[15] *cmd2 - immersive interactive command line applications*. URL: https://github.com/python-cmd2/cmd2 (visited on 12/07/2023).

[16] *bridging fuzzing and property testing*. URL: https://blog.yoshuawuyts.com/bridging-fuzzing-and-property-testing/ (visited on 12/05/2023).

[17] *GitHub - proptest-rs/proptest: Hypothesis-like property testing for Rust*. URL: https://github.com/proptest-rs/proptest (visited on 12/06/2023).

[18] *GitHub - model-checking/kani: Kani Rust Verifier.* URL: https://github.com/model-checking/kani (visited on 12/05/2023).

[19] *GitHub - rust-fuzz/cargo-fuzz: Command line helpers for fuzzing.* URL: https://github.com/rust-fuzz/cargo-fuzz (visited on 12/05/2023).

[20] *libFuzzer - a library for coverage-guided fuzz testing. — LLVM 18.0.0git documentation.* URL: https://llvm.org/docs/LibFuzzer.html (visited on 12/05/2023).

[21] *GitHub - rust-fuzz/arbitrary: Generating structured data from arbitrary, unstructured input.* URL: https://github.com/rust-fuzz/arbitrary/ (visited on 12/05/2023).

[22] *N/A.* URL: https://github.com/AFLplusplus/LibAFL/tree/main/libafl_libfuzzer (visited on 12/05/2023).

[23] *GitHub - ispras/casr: Collect crash (or UndefinedBehaviorSanitizer error) reports, triage, and estimate severity.* URL: https://github.com/ispras/casr (visited on 12/11/2023).

[24] *Prevent stack overflows by limiting recursion by 46bit - Pull Request 501 - sqlparser-rs/sqlparser-rs - GitHub.* URL: https://github.com/sqlparser-rs/sqlparser-rs/pull/501 (visited on 12/05/2023).

[25] *Stack overflow in `Statement::to_string` for deeply nested expresions - Issue 984 - sqlparser-rs/sqlparser-rs - GitHub.* URL: https://github.com/sqlparser-rs/sqlparser-rs/issues/984 (visited on 12/05/2023).

[26] *OSS-Fuzz - Integrating a Rust project.* URL: https://google.github.io/oss-fuzz/getting-started/new-project-guide/rust-lang/ (visited on 12/07/2023).

[27] *Atheris: A Coverage-Guided, Native Python Fuzzer.* URL: https://github.com/google/atheris (visited on 12/07/2023).

[28] *Integrating a Python project in OSS-Fuzz.* URL: https://google.github.io/oss-fuzz/getting-started/new-project-guide/python-lang/ (visited on 12/07/2023).

[29] *libprotobuf-mutator, a library to randomly mutate protobuffers.* URL: https://github.com/google/libprotobuf-mutator (visited on 12/07/2023).

[30] *GitHub - rust-embedded/cargo-binutils: Cargo subcommands to invoke the LLVM tools shipped with the Rust toolchain.* URL: https://github.com/rust-embedded/cargo-binutils (visited on 12/05/2023).

A. Appendix

A.1 Databroker

A.2 Manual review

Rust client to register an infinite number of datapoints

We created a custom client that tries to register 1 billion entries to the databroker. To reproduce this client, you must:

- initialize a cargo project
- import `proto/`
- copy the following code

Import the following lines in `Cargo.toml`:

```
[package]
name = "client"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
↪  https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
prost = "0.12.3"
prost-types = "0.12.3"
tokio = { version = "1.34.0", features = ["full"] }
tonic = "0.10.2"

[build-dependencies]
tonic-build = "0.10.2"
protobuf-src = "1.1.0"
```

Import the following lines in `lib.rs`:

```
#![allow(unknown_lints)]
#![allow(clippy::derive_partial_eq_without_eq)]
pub mod sdv {
    pub mod databroker {
        pub mod v1 {
            tonic::include_proto!("sdv.databroker.v1");
        }
    }
}
```

Import the following lines in `main.rs`:

```
use client::sdv::databroker::v1::collector_client::CollectorClient;
use client::sdv::databroker::v1::RegisterDatapointsRequest;
use client::sdv::databroker::v1::RegistrationMetadata;
use client::sdv::databroker::v1::DataType;
use client::sdv::databroker::v1::ChangeType;
use tonic;
use tokio;


#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // connection to server
    let channel = tonic::transport::Channel::from_static("http://127.0.0.1:55555")
    .connect()
    .await?;
    // creating gRPC client from channel
    let mut client = CollectorClient::new(channel);
    let mut i: u64 = 0;
    let limit: u64 = 1000000000; // 1_000_000_000 entries

    while i < limit {
        let mut registrations: Vec<RegistrationMetadata> = Vec::new();
        for _ in 0..10000 {
            i += 1;
            registrations.push(
                RegistrationMetadata {
                    name: String::from("Vehicle.test")+&(i.to_string()),
                    data_type: DataType::Bool.into(),
                    description: String::from("Description here"),
                    change_type: ChangeType::Static.into(),
                }
            );

        }
        let request = tonic::Request::new(
            RegisterDatapointsRequest {
            list: registrations,
            },
        );
        // sending request and waiting for response
    let response = client.register_datapoints(request).await?.into_inner();
    println!("RESPONSE={:?}", response);

    }

    Ok(())
}
```

A.2.1 Fuzzing

Prerequisites

```
$ apt install llvm-dev # to have symbols in stack traces
$ rustup component add llvm-tools-preview rust-src
$ cargo install cargo-fuzz
$ cargo install cargo-binutils
# not mandatory unless a fuzzing campaign is planned
$ cargo install just
$ cargo install zellij
$ cargo install pueue
$ cargo install casr
```

Setup

Apply patches to 0.4.1 tree

```
$ git clone https://github.com/eclipse/kuksa.val.git
$ cd kuksa.val
$ git checkout v0.4.1
$ git apply [patch_dir]/*
$ cd kuksa_databroker/databroker/fuzz
```

Usage

Run `cargo fuzz run` to start a fuzzing session for a given harness

```
$ cargo +nightly fuzz run [harness]
...
```

`cargo fuzz` provides a command to display the coverage of the fuzzer. First you need to gather the coverage for all the files in corpus:

```
$ cargo +nightly fuzz coverage [harness]
```

Then you can view covered lines with `cargo cov` (installed with `cargo-binutils` [30]).

```
$ cargo +nightly cov -- show
↪  target/x86_64-unknown-linux-gnu/coverage/x86_64-unknown-linux-gnu/release/{{harness}}
↪  -instr-profile fuzz/coverage/{{harness}}/coverage.profdata
↪  -ignore-filename-regex="rustc/.*" -ignore-filename-regex=".cargo/registry/.*"
↪  -output-dir {{outdir}} -format html
```

Since we are using the `arbitrary` crate, reproduction can be a bit tricky. `cargo fuzz` stores in the corpus (and in the crashes directory) the raw inputs generated by the fuzzer.

We can't use these inputs directly, we first need to convert them to be able to use them. `cargo fuzz` provides a `fmt` command to output a debug representation of an input.

A `Justfile` is provided with the common commands used during a fuzzing campaign.

A.2.2 Triage

```
$ just triage databroker
rm -rf triage/databroker
casr-libfuzzer -i artifacts/databroker -o triage/databroker --
↪  target/x86_64-unknown-linux-gnu/release/databroker
10:16:23 [INFO] Analyzing 272 files...
10:16:23 [INFO] Generating CASR reports...
10:16:23 [INFO] Using 6 threads
...
10:16:55 [INFO] Deduplicating CASR reports...
10:16:59 [INFO] Number of reports before deduplication: 245. Number of reports
↪  after deduplication: 38
10:16:59 [INFO] Clustering CASR reports...
10:16:59 [INFO] Number of clusters: 8
==> <cl1>
Crash: fuzz/triage/databroker/cl1/crash-018829fee875c2ec0a02ba6f6eca588100caa521
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 9
Cluster summary -> StackOverflow: 9
==> <cl2>
Crash: fuzz/triage/databroker/cl2/crash-03e2661bdc07d243e7f9d6e28532456aafe49c60
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 2
Cluster summary -> StackOverflow: 2
==> <cl3>
Crash: fuzz/triage/databroker/cl3/crash-15334043a92e935b72ecdde2c8794ef4e1a2aff9
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenize
  Similar crashes: 2
Crash: fuzz/triage/databroker/cl3/crash-1c65d4f685b9bc32e263d29e838512e172c148a5
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenize
  Similar crashes: 2
Cluster summary -> StackOverflow: 4
==> <cl4>
Crash: fuzz/triage/databroker/cl4/crash-1ee085bba64a256c19a081bedaf36763b5984bfe
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 2
Cluster summary -> StackOverflow: 2
==> <cl5>
Crash: fuzz/triage/databroker/cl5/crash-01c347cfb4c4238f752c827f2be4ce4906b83644
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/mod.rs
  Similar crashes: 11
Cluster summary -> StackOverflow: 11
==> <cl6>
Crash: fuzz/triage/databroker/cl6/crash-37e8705ccf15a7093d85a04ede4fa6ad5c99f1cd
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenizer.rs:296:5
  Similar crashes: 2
Cluster summary -> StackOverflow: 2
==> <cl7>
Crash: fuzz/triage/databroker/cl7/crash-00c89c1a738ecf6ab812069e0dc616b7bd377270
  casrep: NOT_EXPLOITABLE: StackOverflow:
  ↪  sqlparser-0.40.0/src/parser/mod.rs:2595:28
```

```
  Similar crashes: 5
Crash: fuzz/triage/databroker/cl7/crash-e494da5a277fe5fb13e83b604c3128c81613d143
  casrep: NOT_EXPLOITABLE: StackOverflow:
  ↪  sqlparser-0.40.0/src/parser/mod.rs:2630:13
  Similar crashes: 1
Crash: fuzz/triage/databroker/cl7/crash-fa120f8cad156e5f42f6c82acaeabd507a17dc22
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenizer.rs:366:9
  Similar crashes: 1
Cluster summary -> StackOverflow: 7
==> <cl8>
Crash: fuzz/triage/databroker/cl8/crash-ce41173d24dc36dcb6c5599000f4ee7b52db5bf8
  casrep: NOT_EXPLOITABLE: StackOverflow:
  ↪  sqlparser-0.40.0/src/parser/mod.rs:2572:32
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
SUMMARY -> StackOverflow: 38
```

```
$ cargo fuzz fmt databroker
↪  triage/databroker/cl1/crash-018829fee875c2ec0a02ba6f6eca588100caa521

Output of `std::fmt::Debug`:

Context {
    initial: [
        AddEntry {
            name: "g",
            data_type: Int8Array,
            change_type: Static,
            entry_type: Actuator,
            description: "\u{8}",
            allowed: None,
        },
    ],
    ops: [
        SubscribeQuery(
            SubscribeQuery {
                query: "MERGE\tDUT(f++++++++++++++ X+++++++DEC++++++++
                ++++++++++-+DEC++++++++++++++++++++++++++++++++++++++
                ++++DEC++++++++++++++++++-+DEC(f++++++++++++++++++++
                +++++++DEC++++++++++++++++++-+DEC(f++++++++++++++ X++M
                ERGE++++++++++-+DEC(fC++++++++++++++++++-+DEC++++++++
                ++++++++++++++++++++++++++++++++DEC++++++++++++++++
                ++-+DEC(f++++++++++++++ X++++++++++++++++-+DEC(f+++++
                +++T(f++++++++++++++ X+++++++DEC++++++++++DEC+++++++++
                +++++++++-+DEC(f++++++++++++++ X++++++++++++++++-+DEC
                (f++++++++++++++++++++++++++++++++++++++++++DEC+++++++
                ++++++++++-+DEC(f++++++++++++++ X++++++++++++++++-+D
                EC(f++++++++T(f++++++++++++++ X+++++++DEC++++++++++DEC
                ++++++++++++++++++-+DEC(f++++++++++++++ X++++++++++++
                ++++-+DEC(f++++++++++++++++++++++++++++++++++++++++++
```

```
                +++++++++DE++++++++++++++++++++T(f++++++++++++++ X+++++
                ++DEC+++++++++DEC++++++++++++++++++-+DEC(f++++++++++
                +++ X+++++++++++++++++-+DEC(f++++++DO+++++++++++++++
                ++++++++++++++++++++++++++++DE+++++++++++++++ X+++++++
                ++++++++++-+DEC(fC++++++++++++++++++-+DEC++++++++++++
                ++++++++++++++++++++++++++++++DEC++++++++++++++++++-+
                DEC(f++++++++++++ X+++++++++++++++++-+DEC(f++++++++T
                (f++++++++++++ X+++++++DEC++++++++++DEC++++++++++++
                ++++++++++++++++++++++DEC++++++++++++++++++-+DEC(f+++
                +++++++++++++++++++++++++DEC++++++++++++++++++-+DEC(f
                ++++++++++++ X++MERGE++++++++++-+DEC(fC++++++++++++
                ++++++++++++++++++++++++++++++DE++++++++++++++++++T(
                f++++++++++++ X+++++++DEC++++++++++DEC++++++++++++++
                ++++-+DEC(f++++++++++++ X+++++++++++++++++-+DEC(f+++
                +++DO++++++++++++++++++++++++++++++++++++++++++++++DE+++
                ++++++++++++ X+++++++++++++++++-+DEC(fC++++++++++++++
                ++++-+DEC++++++++++++++++++++++++++++++++++++++++++++DE
                C++++++++++++++++++-+DEC(f++++++++++++ X++++++++++++
                +++++-+DEC(f++++++++T(f++++++++++++++ X+++++++DEC+++++
                +++++DEC++++++++++++++++++++++++++++++++++++DEC+++++++
                ++++++++++++-+DEC(f++++++++++++++++++++++++++++DEC++++
                +++++++++++++++-+DEC(f++++++++++++ X++MERGE++++++++++
                -+DEC(fC++++++++++++++++++-+DEC++++++++++++++++++++++
                ++++++++++++++++++++DEC++++++++++++++++++-+DEC(f+++++
                ++++++++ X+++++++++++++++++-+DEC(f++++++++T(f++++++++
                +++++ X+++++++DEC++++++++++DEC++++++++++++++++++-+DEC
                (f++++++++++++ X+++++++++++++++++-+DEC(f++++++++++++
                ++++++++++++++++++++++++++++++++++++++++DE++++++++++++
                +++++++T(f++++++++++++++ X+++++++DEC++++++++++DEC+++++
                ++++++++++++++-+DEC(f++++++++++++++ X+++++++++++++++-
                +DEC(f++++++DO++++++++++++++++++++++++++++++++++++++++
                ++++DE+++++++++++++++ X++++++++++++",

            },
        ),
        GetMetadataByPath(
            GetMetadataByPath {
                path: "\u{12}",
            },
        ),
        GetEntryByPath(
            GetEntryByPath {
                path: "",
            },
        ),
    ],
}
```

```
just triage grpc_subscribe
rm -rf triage/grpc_subscribe
```

```
casr-libfuzzer -i artifacts/grpc_subscribe -o triage/grpc_subscribe --
↪  target/x86_64-unknown-linux-gnu/release/grpc_subscribe
10:20:55 [INFO] Analyzing 275 files...
10:20:55 [INFO] Generating CASR reports...
10:20:55 [INFO] Using 6 threads
...
10:21:26 [INFO] Number of reports before deduplication: 244. Number of reports
↪  after deduplication: 120
10:21:26 [INFO] Clustering CASR reports...
10:21:27 [INFO] Number of clusters: 27
==> <cl1>
Crash: fuzz/triage/grpc_subscribe/cl1/crash-040ece4bda39820ab6988d6d1037dbc7e1f2
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 10
Cluster summary -> StackOverflow: 10
==> <cl2>
Crash: fuzz/triage/grpc_subscribe/cl2/crash-7cbf9bd1523318796261214376a923659927
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 4
Cluster summary -> StackOverflow: 4
==> <cl3>
Crash: fuzz/triage/grpc_subscribe/cl3/crash-2233c5d0c17ecfaa24fe260fdc8fc595d1a0
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 4
Cluster summary -> StackOverflow: 4
==> <cl4>
Crash: fuzz/triage/grpc_subscribe/cl4/crash-159426f639b500f3d60c595b7161ec320d57
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenize
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl5>
Crash: fuzz/triage/grpc_subscribe/cl5/crash-2fbe10031130fde34ed573bbff84699931ab
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 5
Cluster summary -> StackOverflow: 5
==> <cl6>
Crash: fuzz/triage/grpc_subscribe/cl6/crash-a9445486e411c75fd6f1d1869c956c73e3b6
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 2
Cluster summary -> StackOverflow: 2
==> <cl7>
Crash: fuzz/triage/grpc_subscribe/cl7/crash-26ab345be362be20e803fa20bb5012727839
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 2
Cluster summary -> StackOverflow: 2
==> <cl8>
Crash: fuzz/triage/grpc_subscribe/cl8/crash-aacf74e86eaa15d4da8f60df39a1874eac32
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl9>
Crash: fuzz/triage/grpc_subscribe/cl9/crash-463c248c850e18981b5c240f8bb08f43cd6b
```

```
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 5
Cluster summary -> StackOverflow: 5
==> <cl10>
Crash: fuzz/triage/grpc_subscribe/cl10/crash-cd6985eaba3005cd7482aa26eba187ec73d
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl11>
Crash: fuzz/triage/grpc_subscribe/cl11/crash-28ea048b0b1baa6a01628e7dc673bbafee8
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 6
Cluster summary -> StackOverflow: 6
==> <cl12>
Crash: fuzz/triage/grpc_subscribe/cl12/crash-888768198b19ee069140084ef63e66f69a6
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl13>
Crash: fuzz/triage/grpc_subscribe/cl13/crash-6e6d745ad501dd39a4360e8800920f88ab3
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenize
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl14>
Crash: fuzz/triage/grpc_subscribe/cl14/crash-07a2702744dbe8edd3948f8fe901bf41487
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 5
Cluster summary -> StackOverflow: 5
==> <cl15>
Crash: fuzz/triage/grpc_subscribe/cl15/crash-00cfd0b866400b5f485ed3e6eb6eee40a50
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 16
Cluster summary -> StackOverflow: 16
==> <cl16>
Crash: fuzz/triage/grpc_subscribe/cl16/crash-a251d03010390cea3b5d21c1920ea2194b1
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 1
Crash: fuzz/triage/grpc_subscribe/cl16/crash-64e4c97790726f673b6fc41d7e38561da3d
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 1
Cluster summary -> StackOverflow: 2
==> <cl17>
Crash: fuzz/triage/grpc_subscribe/cl17/crash-85f10c6c00eb56b41146950c4a5621a53bf
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 5
Cluster summary -> StackOverflow: 5
==> <cl18>
Crash: fuzz/triage/grpc_subscribe/cl18/crash-0b085987354a4f6eb61ac413499ed10354a
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 4
Cluster summary -> StackOverflow: 4
==> <cl19>
```

```
Crash: fuzz/triage/grpc_subscribe/cl19/crash-5d6138ca14779f326241b9c97ad7a52a3bd
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 2
Crash: fuzz/triage/grpc_subscribe/cl19/crash-575b40b70ac694fe6f9662f1db7e709c048
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 1
Crash: fuzz/triage/grpc_subscribe/cl19/crash-01aaec6fac2baa2ecbc22e6ded751933fcf
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 5
Cluster summary -> StackOverflow: 8
==> <cl20>
Crash: fuzz/triage/grpc_subscribe/cl20/crash-786c03bc309a1f033902e4326351c314a0f
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenize
  Similar crashes: 3
Cluster summary -> StackOverflow: 3
==> <cl21>
Crash: fuzz/triage/grpc_subscribe/cl21/crash-8622ccdcbe7ad6f17ecde2aa4a9f789e343
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenize
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl22>
Crash: fuzz/triage/grpc_subscribe/cl22/crash-62bf553303ceaa8486f78e7acca2b336bdc
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/dialect/
  Similar crashes: 1
Crash: fuzz/triage/grpc_subscribe/cl22/crash-00673ef0318299c0f75952d4366ec2e512c
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 17
Cluster summary -> StackOverflow: 18
==> <cl23>
Crash: fuzz/triage/grpc_subscribe/cl23/crash-9375e6ff5922fa98714ac5ffe1149a470ce
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl24>
Crash: fuzz/triage/grpc_subscribe/cl24/crash-c2e8be7d4554df575d430d826b17b1aaef0
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/parser/m
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl25>
Crash:
↪  fuzz/triage/grpc_subscribe/cl25/crash-65820996ea0e8ef5d4593b6557feb5f7fd4a5aa0
  casrep: NOT_EXPLOITABLE: StackOverflow:
  ↪  sqlparser-0.40.0/src/parser/mod.rs:2572:32
  Similar crashes: 1
Cluster summary -> StackOverflow: 1
==> <cl26>
Crash:
↪  fuzz/triage/grpc_subscribe/cl26/crash-0349194f82cf17f2c792ee6e6aa113afc3290e72
  casrep: NOT_EXPLOITABLE: StackOverflow:
  ↪  sqlparser-0.40.0/src/parser/mod.rs:2629:9
  Similar crashes: 1
```

```
Crash:
↪  fuzz/triage/grpc_subscribe/cl26/crash-a83275d214570214cd2f9c9cd1eff911499cebf0
  casrep: NOT_EXPLOITABLE: StackOverflow: sqlparser-0.40.0/src/tokenizer.rs
  Similar crashes: 1
Crash:
↪  fuzz/triage/grpc_subscribe/cl26/crash-166e9dfa6d0d03f93cbe6185efc82494d118c6fe
  casrep: NOT_EXPLOITABLE: StackOverflow: /sqlparser-0.40.0/src/tokenizer.rs:366:9
  Similar crashes: 2
Cluster summary -> StackOverflow: 4
==> <cl27>
Crash:
↪  fuzz/triage/grpc_subscribe/cl27/crash-1d8453bf5acb86b1100ff50a7ce822f75adb0466
  casrep: NOT_EXPLOITABLE: StackOverflow:
  ↪  sqlparser-0.40.0/src/parser/mod.rs:2572:32
  Similar crashes: 8
Cluster summary -> StackOverflow: 8
SUMMARY -> StackOverflow: 120
```

```
   cargo fuzz fmt grpc_subscribe
   ↪  triage/grpc_subscribe/cl1/crash-040ece4bda39820ab6988d6d1037dbc7e1f2709f

Output of `std::fmt::Debug`:

Request(
    SubscribeRequest {
        query:
        ↪  "ASSERT((-\"\"(((((SF++++++++++++++++++++++++++++++++++++++++xxx(xxx(+++++
        ++++++++++++++++++++++++++++++++++++++++++++++++++++++
(++++++++++++++++++++++++++++++++++++xxx(xxxxxxxxxxxxxxZ(((((++++++++++++++++++++++
++++++++++++++++++++++++++++++++++[++++++++++++++++++
++++++++++++++++xxx(xxx(+++NA++++++++++++++++++++++++++++++++++++++++++++++++++xxx(x
xx(++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++(++++++++++++++++++++++++++++++++++xxx(xxxxxxxxxxxxxxZ(((((+++++++++++++++++
++++++++++++++++++++++++++++++++++++++++[++++++++++++++
++++++++++++++++++++xxx(xxx(++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++xxx(xx|\nW\n||+++++++++++++++T((-\"\"(((((
SF++++++++++++++++++++++++++++++++++++++++xxx(xxx(++++++++++++++++++++++++++++++++
++++++++++++++++++++(++++++++++++++++++++++++++++++++++++++++xxx(xxxxxxxxxxxxxxZ(((((
++++++++++++",
    },
)
```

```
    just triage query_compiler_compile
rm -rf triage/query_compiler_compile
casr-libfuzzer -i artifacts/query_compiler_compile -o
↪  triage/query_compiler_compile --
↪  target/x86_64-unknown-linux-gnu/release/query_compiler_compile
10:30:49 [INFO] Analyzing 369 files...
10:30:49 [INFO] Generating CASR reports...
10:30:49 [INFO] Using 6 threads
```

```
    10:31:16 [INFO] Deduplicating CASR reports...
10:31:18 [INFO] Number of reports before deduplication: 361. Number of reports
↪  after deduplication: 1
10:31:18 [INFO] There are less than 2 CASR reports, nothing to cluster.
==> <query_compiler_compile>
Crash:
↪  fuzz/triage/query_compiler_compile/crash-00af9ec26b9dbceaf0bc742f6857dda96220e4f2
  casrep: NOT_EXPLOITABLE: RustPanic: databroker/src/query/executor.rs:106:17
  Similar crashes: 1
Cluster summary -> RustPanic: 1
SUMMARY -> RustPanic: 1
```

```
    cargo fuzz fmt query_compiler_compile
    ↪  triage/query_compiler_compile/crash-00af9ec26b9dbceaf0bc742f6857dda96220e4f2

Output of `std::fmt::Debug`:

"SELECT\t1oooomJ"
```