Bypass Apache Superset restrictions to perform SQL injections

Posted Thu 10 October 2024
Author Mathieu Farrell
Category Pentest
Tags pentest, vulnerability, Apache, Python3, SQL, 2024

The following article explains how during an audit we took a look at Apache Superset and found bypasses (by reading the PostgreSQL documentation) for the security measures implemented.

What is Apache Superset?

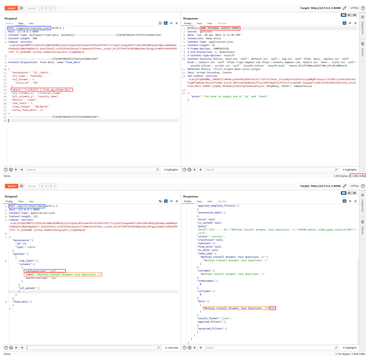
Apache Superset is an open-source platform that facilitates data exploration and visualization. Superset offers a code-free interface for rapid charting to explore data without writing complex SQL queries. It also features a web-based SQL editor for more advanced analysis. Despite the ability to perform SQL queries, not everything is granted to users, as security and data compartmentalization are taken into consideration.

Context

As part of a web audit, an application developed by one of our clients was presenting graphics and analytics token from Apache Superset (version 4.0.1, latest version available at the time of writing) within its web interface. By analyzing interactions between the application and Burp, we realized that it was possible to interact with the Apache Superset API.

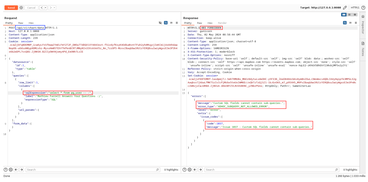
As we tried to interact with the API, we quickly identified multiple ways in which we could control some SQL queries executed by the DBMS by playing with the following API routes:

- `/superset/explore_json/` (Time Based converted to Error Based using `xpath_exists(xpath, xml [, nsarray])`, which evaluates XPath 1.0 expressions.)
- `/api/v1/chart/data`



It is important to note that executing SQL queries within Apache Superset is normal, as it is the primary purpose of the application and is not considered a vulnerability. The purpose of this blogpost is to show the weaknesses identified regarding the security measures implemented to prevent an attacker from executing arbitrary SQL requests.

As soon as a user (and therefore an attacker) tries to execute arbitrary requests, these requests are blocked by the security mechanism.



Let's look at the code

To identify the reason for this blocking, we simply had to retrieve the source code from GitHub. After a quick code review, the following snippets appear to check that the query does not contain any subqueries.

File: superset/models/helpers.py
Function: `validate_adhoc_subquery()`

```
...

def validate_adhoc_subquery(
    sql: str,
    database_id: int,
    default_schema: str,
) -> str:
    """
    Check if adhoc SQL contains sub-queries or nested sub-queries with table.

    If sub-queries are allowed, the adhoc SQL is modified to insert any applicable RLS
    predicates to it.

    :param sql: adhoc sql expression
    :raise SupersetSecurityException: if sql contains sub-queries or
    nested sub-queries with table
    """
    statements = []
    for statement in sqlparse.parse(sql):
        if has_table_query(statement):
            if not is_feature_enabled("ALLOW_ADHOC_SUBQUERY"):
                raise SupersetSecurityException(
                    SupersetError(
                        error_type=SupersetErrorType.ADHOC_SUBQUERY_NOT_ALLOWED_ERROR,
                        message=_("Custom SQL fields cannot contain sub-queries."),
                        level=ErrorLevel.ERROR,
                    )
                )
            statement = insert_rls_in_predicate(statement, database_id, default_schema)
        statements.append(statement)

    return ";\n".join(str(statement) for statement in statements)

...
```

And as it is shown, function `validate_adhoc_subquery()` calls function `has_table_query()`.

File: superset/sql_parse.py
Function: `has_table_query()`

```
...

def has_table_query(token_list: TokenList) -> bool:
    """
    Return if a statement has a query reading from a table.

        >>> has_table_query(sqlparse.parse("COUNT(*)")[0])
        False
        >>> has_table_query(sqlparse.parse("SELECT * FROM table")[0])
        True

    Note that queries reading from constant values return false:

        >>> has_table_query(sqlparse.parse("SELECT * FROM (SELECT 1)")[0])
        False

    """
    state = InsertRLSState.SCANNING
    for token in token_list.tokens:
        # Ignore comments
        if isinstance(token, sqlparse.sql.Comment):
            continue

        # Recurse into child token list
        if isinstance(token, TokenList) and has_table_query(token):
            return True

        # Found a source keyword (FROM/JOIN)
        if imt(token, m=[(Keyword, "FROM"), (Keyword, "JOIN")]):
            state = InsertRLSState.SEEN_SOURCE

        # Found identifier/keyword after FROM/JOIN
        elif state == InsertRLSState.SEEN_SOURCE and (
            isinstance(token, sqlparse.sql.Identifier) or token.ttype == Keyword
        ):
            return True

        # Found nothing, leaving source
        elif state == InsertRLSState.SEEN_SOURCE and token.ttype != Whitespace:
            state = InsertRLSState.SCANNING

    return False

...
```

The function `has_table_query()` uses the library `sqlparse` to parse the SQL query that is to be executed in order to validate or invalidate the query before execution. The feature operates as follows, the query elements are tokenized and the query is invalidated if it contains forbidden elements.

If we take the screenshot given in the previous section, we can see that it was probably the use of the `FROM` clause that triggered the security mechanism.

Read some documentation and find a bypass

After setting up the lab with a few commands referenced in the documentation, and realized that the default DBMS was PostgreSQL, we got interested in how we could bypass the security to execute arbitrary queries.

```
git clone https://github.com/apache/superset
cd superset
docker compose -f docker-compose-image-tag.yml up
```

We looked at the PostgreSQL documentation and identified the following functions:

- `query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)`, map the contents of relational tables to XML values.
- `query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)`, produce XML data mappings and the corresponding XML Schema in one document (or forest), linked together.
- `table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)`, map the contents of relational tables to XML values.
- `table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)`, produce XML data mappings and the corresponding XML Schema in one document (or forest), linked together.
- `database_to_xml(nulls boolean, tableforest boolean, targetns text)`, produce analogous mappings of entire schemas or the entire current database.

The above functions take strings as parameters, execute them as SQL queries and format the result into the expected output (XML). We can therefore call these functions to execute arbitrary requests. What happens is that during parsing (by `sqlparse`), our malicious queries are considered as strings (function parameter) and are "tokenized" as such, that's why function `has_table_query` does not detect the injection.

Vulnerability report

We had two injection points (/superset/explore_json/, /api/v1/chart/data) and a bypass technique. All we had to do now was forward the information to the people in charge of the project.

After reporting the vulnerabilities to the Apache Superset security team, Daniel Vaz Gaspar (remediation developer) informed us that, in parallel of our report, he had received another one from Mike Yushkovskiy referring to one of our two injection points (/api/v1/chart/data) and the same bypass using function `query_to_xml()`. A patch was being deployed for version 4.0.2 and a CVE number (CVE-2024-39887) had already been reserved.

However, the exploitation of the bypass via our second endpoint had not been identified, and the list of additional functions (`query_to_xml_and_xmlschema()`, `table_to_xml()`, `table_to_xml_and_xmlschema()`, `database_to_xml()`) would be implemented to improve the patch and will be deployed in next version (version 4.1.0).

Disclosure timeline

- 2024/05/20 - Vulnerability reported to Apache.
- 2024/05/21 - Apache Superset acknowledges the report and says they are already working on a fix for the `query_to_xml()` vuln and will look into the other.
- 2024/06/06 - Quarkslab Vulnerability Reports Team (QVRT) tells Apache Superset that more functions should be added to the deny list that was implemented to address the `query_to_xml()` vulnerability.
- 2024/06/07 - Apache Superset acknowledges the prior email and indicate they will address the comment.
- 2024/07/02 - Apache Superset tells QVRT they couldnt repro one of the issues pointed out in the last email.
- 2024/07/05 - QVRT says we will double check if the fix resolved the metioned issue. Asked if there is an estimated date for releasign fixes.
- 2024/07/12 - Apache Superset informes the remaining fixes have been committed.
- 2024/07/22 - QVRT asks if Apache Superset plans to publish an security advisory or new release and the estimated date for it.
- 2024/08/19 - Apache Superset tells QVRT that they plan to release both a security advisory and a new release in 2 or 3 weeks.
- 2024/10/10 - This blog post is published.
- 2024/11/01 - Apache Superset 4.1.0 released with the fix for CVE-2024-39887.

If you would like to learn more about our security audits and explore how we can help you, get in touch with us!