Symmetric Searchable Encryption component

Deployment manual version 0.4.0

Component name: SSE – Symmetric Searchable Encryption

Component deployment name: sse

Changelog

Table 1 Document Change History

Version	Date	Pages	Author	Modification
0.1	5/3/2020	9	Hai-Van Dang	Initial release: deployment with docker in cloud or locally
0.2	30/3/2020	9	James Bowden	Deployment with docker compose
0.3	11/08/2020	12	Hai-Van Dang	Minio deployment Deployment with MySQL Deployment with database-as-a-service
0.4	15/12/2020	12	Hai-Van Dang	Minio deployment with SSL support Providing technical notes on re-compiling SJCL javascript, and modifying sjcl python package

1. Table of Contents

1. Terminology

Terminology/ Abbreviation	Explanation
End-user	User who uploads/ searches data
SSE server	Server which stores encrypted data
Trusted Authority (TA)	Server which stores metadata necessary for upload/ search encrypted data
SSE client	An application which utilizes SSE javascript APIs, i.e. sse.js, to upload/ search over encrypted data
sse.js	The provided javascript APIs
SSE database	Database of SSE server
TA database	Database of TA server

2. Introduction

This manual instructs how to deploy Symmetric Searchbale Encryption components, i.e. SSE server, Trusted Authority, and an example SSE client, i.e. a web application, which uses SSE services, in docker containers. They are assumingly deployed in three different machines, where port 80 is reserved for each application (i.e. SSE server, TA, SSE client), and port 5432 is reserved for each own database server (i.e. SSE database server, TA database server). If deployed in the same machine, further port configurations need to be done in order to make sure each application runs in different port, and there is no port conflict of the two databases, i.e. SSE database and TA database.

For a local deployment, please refer to the section 4 for deploying three applications in development environment, and section 5 for deploying them with docker compose.

3. Deployment in separate machines

3.1. SSE Server

SSE Server stores encrypted data, and support to search over encrypted data.

1. Configure database name, database user and password in deploy.env file

```
DJANGO_LOGLEVEL=[log level of SSE server application]
```

```
TA_SERVER=[URL of Restful API URL provided by TA to perform search
function]

DB_NAME=[Name of database]

DB_USER=[Database username]

DB_PASSWORD=[Database user password]

DB_HOST=[Name of postgres container, i.e. sse-postgres]

DB_PORT=[Port of postgres database, i.e. 5432]

ALLOWED_HOSTS=[List of IP addresses, which are allowed to access services
provided by SSE server]
```

where

- Name of postgres container is the container name which will be defined in step (3), i.e. sse-postgres
- DJANGO_LOGLEVEL: This log level describes the severity of the messages that the logger will handle. Python defines the following log levels:
    - DEBUG: Low level system information for debugging purposes
    - INFO: General system information
    - WARNING: Information describing a minor problem that has occurred.
    - ERROR: Information describing a major problem that has occurred.
    - CRITICAL: Information describing a critical problem that has occurred.

Example:

```
DJANGO_LOGLEVEL=DEBUG
TA_SERVER=http://[IP address of TA server]/api/v1/search/
DB_NAME=db1
DB_USER=user1
DB_PASSWORD=pwd1
DB_HOST=sse-postgres
DB_PORT=5432
ALLOWED_HOSTS='*'
```

2. Define a docker network, where the two docker containers, i.e. database and SSE application, reside in.

```
docker network create sse-network
```

3. Initialize Postgres SQL database as a docker container

```
sudo docker run -d -p 5432:5432 --name sse-postgres --net sse-network -e
PGDATA:/var/lib/postgresql/data/ -v [path to folder which persistenly
stores data]:/var/lib/postgresql/data/ -e POSTGRES_USER=[SSE database
username] -e POSTGRES_PASSWORD=[SSE database password] -e POSTGRES_DB=[SSE
database name] postgres
```

where:

- 5432 is the listening port of Postgres SQL server
- sse-network is the name of docker network which is created in step (2)
- sse-postgres is a chosen name of the container
- [path to folder which persistently stores data] is a chosen folder to store data. This helps to preserve data even if docker container is shut down.
- [SSE database username] is a chosen username
- [SSE database password] is a chosen password
- [SSE database name] is a chosen database name

Example:

```
sudo docker run -d -p 5432:5432 --name sse-postgres --net sse-network -e
PGDATA:/var/lib/postgresql/data/ -v
/home/ubuntu/data/data/ssedata:/var/lib/postgresql/data/ -e
POSTGRES_USER=user1 -e POSTGRES_PASSWORD=pwd1 -e POSTGRES_DB=db1 postgres
```

4. **Build and deploy SSE Server as a docker container**

```
sudo docker build -t sse [folder contains SSE server source code]

sudo docker run -d -it -p 80:8080 --name sse-server --net sse-network --
env-file [folder contains SSE server source code]/deploy.env sse
```

where:

- 80: listening port on the host machine
- 8080: listening port on the SSE container, which is configured in "entrypont.sh" of the source code
- sse-network is the name of docker network which is created in step (2)
- "deploy.env" file contains configuration details of the server

All the logs will be written to standard output, which can be seen through the command

```
docker logs sse-server
```

where

- sse-server is the name of the container defined above in step (3).

In order to check if you deploy SSE server successfully, please access to http://[IP address of SSE server]/api/v1/, which should return the list of supported APIs like below

```
{
    "ciphertext": {
        "list_endpoint": "/api/v1/ciphertext/",
        "schema": "/api/v1/ciphertext/schema/"
    },
    "map": {
        "list_endpoint": "/api/v1/map/",
        "schema": "/api/v1/map/schema/"
    },
    "search": {
        "list_endpoint": "/api/v1/search/",
        "schema": "/api/v1/search/schema/"
    }
}
```

3.2. Trusted Authority (TA)

Trusted Authority stores metadata, which are needed for uploading/ searching data.

1. Configure database name, database user and password in deploy.env file

```
DJANGO_LOGLEVEL=[log level of SSE server application]
TA_SERVER=[URL of Restful API URL provided by TA to perform search
function]
DB_NAME=[Name of TA database]
DB_USER=[TA database username]
DB_PASSWORD=[TA database user password]
DB_HOST=[Name of postgres container, i.e. ta-postgres]
DB_PORT=[Port of postgres database, i.e. 5432]
ALLOWED_HOSTS=[List of IP addresses, which are allowed to access services
provided by TA server]
```

where

- Name of postgres container is the container name which will be defined in step (3), i.e. ta-postgres
- DJANGO_LOGLEVEL: This log level describes the severity of the messages that the logger will handle. Python defines the following log levels:
    - DEBUG: Low level system information for debugging purposes
    - INFO: General system information
    - WARNING: Information describing a minor problem that has occurred.
    - ERROR: Information describing a major problem that has occurred.
    - CRITICAL: Information describing a critical problem that has occurred.

Example:

```
DJANGO_LOGLEVEL=DEBUG
DB_NAME=tadb1
DB_USER=tauser1
DB_PASSWORD=tapwd1
DB_HOST=ta-postgres
DB_PORT=5432
```

```
ALLOWED_HOSTS='*'
```

2. Define a docker network, where the two docker containers, i.e. database and TA application, reside in.

```
docker network create ta-network
```

3. Initialize Postgres SQL database as a docker container

```
sudo docker run -d -p 5432:5432 --name ta-postgres --net ta-network -e
PGDATA:/var/lib/postgresql/data/ -v [path to folder which persistenly
stores data]:/var/lib/postgresql/data/ -e POSTGRES_USER=[TA database
username] -e POSTGRES_PASSWORD=[TA database password] -e POSTGRES_DB=[TA
database name] postgres
```

where:

- 5432 is the listening port of Postgres SQL server
- sse-network is the name of docker network which is created in step (2)
- sse-postgres is a chosen name of the container
- [path to folder which persistenly stores data] is a chosen folder to store data. This helps to preserve data even if docker container is shut down.
- [TA database username] is the chosen username in deloy.env file
- [TA database user password] is the chosen password in deploy.env file
- [TA database name] is a chosen database name in deploy.env file

Example:

```
sudo docker run -d -p 5432:5432 --name ta-postgres --net ta-network -e
PGDATA:/var/lib/postgresql/data/ -v
/home/ubuntu/data/ta:/var/lib/postgresql/data/ -e POSTGRES_USER=ta1 -e
POSTGRES_PASSWORD=tapwd1 -e POSTGRES_DB=tadb1 postgres
```

4. Build and deploy TA server as a docker container

```
sudo docker build -t ta [folder contains TA source code]

sudo docker run -d -it -p 80:8000 --name ta-server --net ta-network --env-
file [folder contains TA source code]/deploy.env ta
```

where:

- ta-network is the name of docker network which is created in step (1)
- 80: listening port on the host machine
- 8000: listening port on the TA container, which is configured in “entrypont.sh” of the source code
- “deploy.env” file contains configuration details of the server

All the logs will be written to standard output, which can be seen through the command

```
docker logs ta-server
```

where

- ta-server is the name of the container defined above in step (3).

In order to test if TA server is successfully deployed, you can try to access http://[IP address of TA]/api/v1/, which return the list of supported APIs like below

```
{
    "fileno": {
        "list_endpoint": "/api/v1/fileno/",
        "schema": "/api/v1/fileno/schema/"
    },
    "longrequest": {
        "list_endpoint": "/api/v1/longrequest/",
        "schema": "/api/v1/longrequest/schema/"
    },
    "search": {
        "list_endpoint": "/api/v1/search/",
        "schema": "/api/v1/search/schema/"
    },
    "searchno": {
        "list_endpoint": "/api/v1/searchno/",
        "schema": "/api/v1/searchno/schema/"
    }
}
```

3.3. SSE Client (example application)

This web application is an example application, which shows how to utilize the provided JS library, i.e. sse.js, to upload/ searching data.

1. Configure URL of TA and SSE, salt value and iv value (which are needed for encryption) in deploy.env file.

2. Build and deploy SSE Client application as a docker container

```
sudo docker build -t sse-client [folder contains web application source
code]

sudo docker run -d -it -p 80:80 --name sseclient --env-file [folder
contains web application source code]/deploy.env sse-client

# Fill contanst values (url_ta_ip,etc.) in sse.js with values from
environment variables, which are defined in deploy.env

sudo docker exec -it sseclient bash -c 'sed -i -e "s|ta_url|$ta_url|"
sse/static/js/sse.js'

sudo docker exec -it sseclient bash -c 'sed -i -e "s|sse_url|$sse_url|"
sse/static/js/sse.js'
```

```
sudo docker exec -it sseclient bash -c 'sed -i -e
"s|salt_value|$salt_value|" sse/static/js/sse.js'

sudo docker exec -it sseclient bash -c 'sed -i -e "s|iv_value|$iv_value|"
sse/static/js/sse.js'
```

where:

- 80: listening port on the host machine, and on the SSE client container (which is configured in "entrypont.sh" of the source code)
- "deploy.env" contains configurations for URL of TA server (ta_url), URL of SSE srever (sse_url), salt value and iv value. The salt and iv values are used for encrypting/ decrypting, which are shared between users and TA.

4. Local deployment in development mode

The following paragraphs instruct how to run the three applications, i.e. SSE server, TA, SSE client, in development mode. In this scenario, the applications will not be run in docker containers. Furthermore, SQLite is used as database instead of Postgres SQL.

4.1. SSE Server

1. Re-configure database to use SQLite instead of Postgres SQL in "SQLServer/settings.py" in the source code

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3', # for local test
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'), # for local test
        #'ENGINE': 'django.db.backends.postgresql_psycopg2',
        #'NAME': os.environ['DB_NAME'],# database name
        #'USER': os.environ['DB_USER'], # user name
        #'PASSWORD': os.environ['DB_PASSWORD'], # user password
        #'HOST': os.environ['DB_HOST'], # postgres server
        #'PORT': os.environ['DB_PORT'],  # postgres port
    }
}
```

2. Initialize database in SQLite
This needs to run once when database needs to be initialized (e.g. for the 1st run of SSE server application, or after deleting the previous database, i.e. db.sqlite3 file).

```
python3 manage.py migrate
```

3. Create environment variables, i.e. DJANGO_LOGLEVEL, TA_SERVER url, ALLOWED_HOSTS, which SSE server needs to access

```
source local.sh
```

where local.sh is provided in the source code

4. Run SSE server in development mode

```
python3 manage.py runserver 0.0.0.0:8080
```

4.2. TA Server

1. Re-configure database to use SQLite instead of Postgres SQL in “TA/settings.py” in the source code

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3', # for local test
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'), # for local test
        #'ENGINE': 'django.db.backends.postgresql_psycopg2',
        #'NAME': os.environ['DB_NAME'],# database name
        #'USER': os.environ['DB_USER'], # user name
        #'PASSWORD': os.environ['DB_PASSWORD'], # user password
        #'HOST': os.environ['DB_HOST'], # postgres server
        #'PORT': os.environ['DB_PORT'],  # postgres port
    }
}
```

2. Initialize database in SQLite
This needs to run once when database needs to be initialized (e.g. for the 1st run of TA server application, or after deleting the previous database, i.e. db.sqlite3 file)

```
python3 manage.py migrate
```

3. Run SSE server in development mode

```
python3 manage.py runserver 0.0.0.0:8000
```

4.3. SSE client

Run SSE client in development mode

```
# Fill contanst values (url_ta_ip,etc.) in sse.js with values from
environment variables, which are defined in deploy.env

sed -i -e "s|ta_url|$ta_url|" sse/static/js/sse.js

sed -i -e "s|sse_url|$sse_url|" sse/static/js/sse.js

sed -i -e "s|salt_value|$salt_value|" sse/static/js/sse.js
sed -i -e "s|iv_value|$iv_value|" sse/static/js/sse.js
sudo python3 manage.py runserver 0.0.0.0:80
```

5. Local deployment with docker-compose

It is possible to deploy all the SSE components on a single host using docker-compose. The following section describes how to do that.

1. Get the docker-compose definition from github

```
git clone https://github.com/somnonetz/asclepios-sse-docker-compose
```

2. Configure the local environment

```
cd asclepios-sse-docker-compose
cp .env.example .env
```

Edit the .env file to modify the default config

3. Build the docker images

```
docker-compose build
```

4. Run the SSE service

```
docker-compose up
```

Once it has successfully booted, each components is accessible on the host as follows:

- SSE Client - http://localhost:80
- SSE Server - http://localhost:8080
- TA - http://localhost:8000

6. Minio deployment

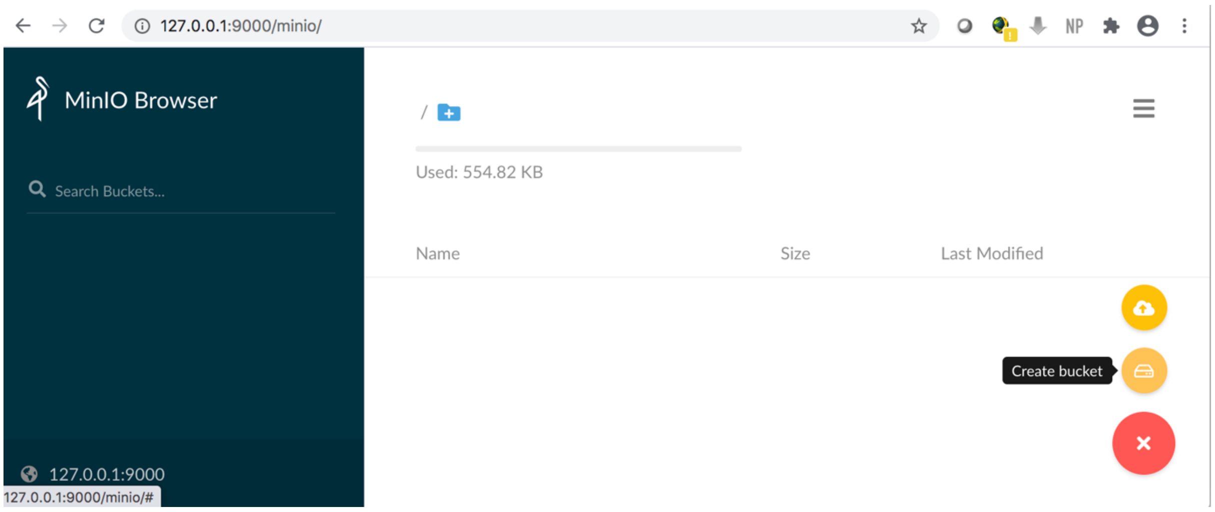
A. As docker container without SSL

1. Deploy as a docker container:

```
docker run -p 9000:9000 --name minio \
        -e 'MINIO_ACCESS_KEY=choose_access_key_here' \
        -e 'MINIO_SECRET_KEY=choose_secret_key_here' \
        -v folder_in_host_machine:/data \
        minio/minio server /data
```

If deployed successfully, you can access Minio server at http://127.0.0.1:9000 with the chosen access key and secret key.

2. Create a bucket:



B. As a gateway to Amazon S3 without SSL

Assuming that you've created an Amazon S3 bucket. You can run Minio as a gateway to the created Amazon S3 with the following command.

```
docker run -p 9000:9000 --name minio-s3 \
        -e 'MINIO_ACCESS_KEY=access_key_from_aws' \
        -e 'MINIO_SECRET_KEY=access_secret_from_aws' \
        minio/minio gateway s3
```

C. As docker container with SSL using self-signed certificate

1. Generate self-signed certificate using OpenSSL

Example:

```
openssl genrsa -out private.key 2048

openssl req -new -x509 -days 3650 -key private.key -out public.crt -subj
"/C=US/ST=state/L=location/O=organization/CN=<domain.com>"
```

2. Create a folder to store data in the host machine

```
mkdir data
```

3. Create a folder to store private.key and public.crt in host machine

```
mkdir config
mkdir config/certs
```

4. Copy private.key and public.crt to config/certs

Run minio as a docker container with SSL enabled

```
docker run -p 9000:9000 --name minio \
        -e 'MINIO_ACCESS_KEY=choose_access_key_here ' \
        -e 'MINIO_SECRET_KEY=choose_secret_key_here ' \
        -v data:/data \
        -v config:/root/.minio \
        minio/minio server /data
```

If deployed successfully, you can access Minio server at https://127.0.0.1:9000 with the chosen access key and secret key.

7. Deployment SSEServer/ TA with MySQL/ Postgres

Database engine can be changed easily with the existing template files.

In order to change to MySQL, copy the template files then re-build the docker image for TA and/ or SSEServer.

```
cp template_Dockerfile_mysql Dockerfile
cp template_requirements_mysql.txt requirements.txt
cp template_entrypoint_mysql.sh entrypoint.sh
cp template_settings_mysql.py TA/settings.py
```

In order to change to Postgres, copy the template files then re-build the docker image for TA and/ or SSEServer.

```
cp template_Dockerfile_postgres Dockerfile
cp template_requirements_postgres.txt requirements.txt
cp template_entrypoint_postgres.sh entrypoint.sh
cp template_settings_postgres.py TA/settings.py
```

8. Deployment SSEServer/ TA with external database-as-a-service

In order to use database-as-a-service, for instance Amazon Relational Database Service, at first you need to create a database at the CSP, for i.e. AWS. Then you change the environment variables DB_HOST and DB_PORT correspondingly.

Example:
DB_HOST=….rds.amazonaws.com
DB_PORT=3306

9. Technical notes

a. Re-compile SLCL javascript

In the SSE client implementation, we rely on SJCL library for cryptographic functions (encryption, decryption, hashing, etc.). The SJCL source code can be found at https://github.com/bitwiseshiftleft/sjcl. However, we have compiled only necessary sub-libraries in https://github.com/bitwiseshiftleft/sjcl/tree/master/core for our need instead of using the ready-to-use compiled js at https://cdnjs.com/libraries/sjcl. The reason we compiled ourselves is the ready-to-use compiled js does not contain functions for progressive encryption and decryption.

In order to compile sjcl js, we download sjcl from https://github.com/bitwiseshiftleft/sjcl, and add a few sub-libraries (marked in red color) into config.mk as below, then invoke *make* command line to build it:

```
SOURCES= core/sjcl.js core/aes.js core/bitArray.js core/codecString.js core/codecHex.js core/codecBase32.js core/codecBase64.js core/sha256.js core/ccm.js core/ocb2.js core/gcm.js core/hmac.js core/pbkdf2.js core/random.js core/convenience.js core/exports.js core/ocb2progressive.js core/codecBytes.js core/sha1.js
COMPRESS= core_closure.js
```

The compiled sjcl.js is copied into sse/static/js folder.

b. Modify SLCL python package

In the TA implementation, we rely on sjcl-python package (https://github.com/berlincode/sjcl) for cryptographic functions (encryption, decryption, hashing). However, the *encrypt* function in sjcl-python does not allow to provide SALT and IV as parameters (https://github.com/berlincode/sjcl/blob/master/sjcl/sjcl.py#L156), which is incompatible with the encryption function in SJCL javascript. In order to make the encryption at SSE client and TA compatible, we have modified sjcl-python package so that the *encrypt* function accepts SALT and IV values as parameters. Therefore, instead of using the ready-to-use python package sjcl (https://pypi.org/project/sjcl/), we use the modified version which locates at *sjcl-0.2.1/sjcl.*