YouTestMe

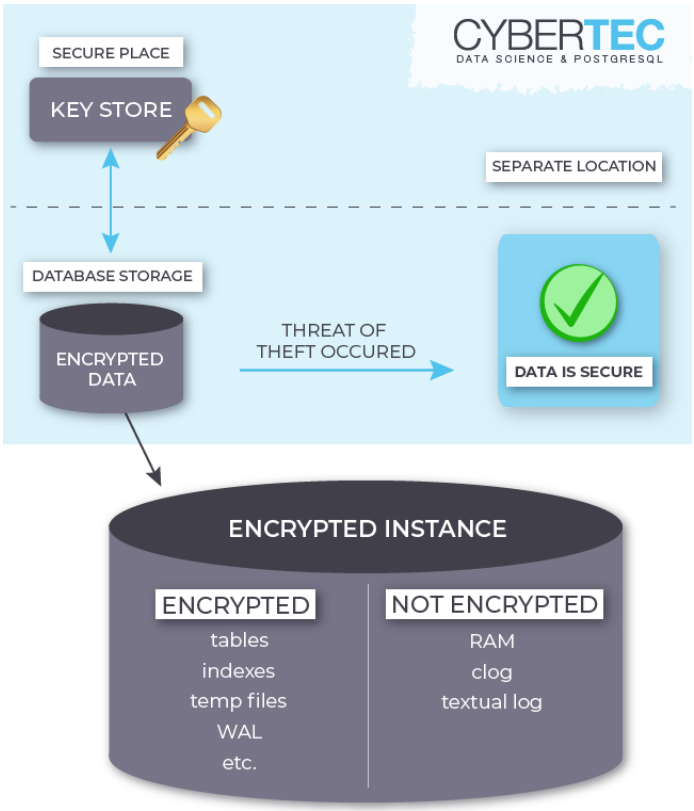
PostgreSQL Transparent Data Encryption

Table of Contents

1 Introduction

Transparent Data Encryption offers encryption at the file level. TDE solves the problem of protecting data at rest, encrypting databases on the hard drive, and consequently on backup media.

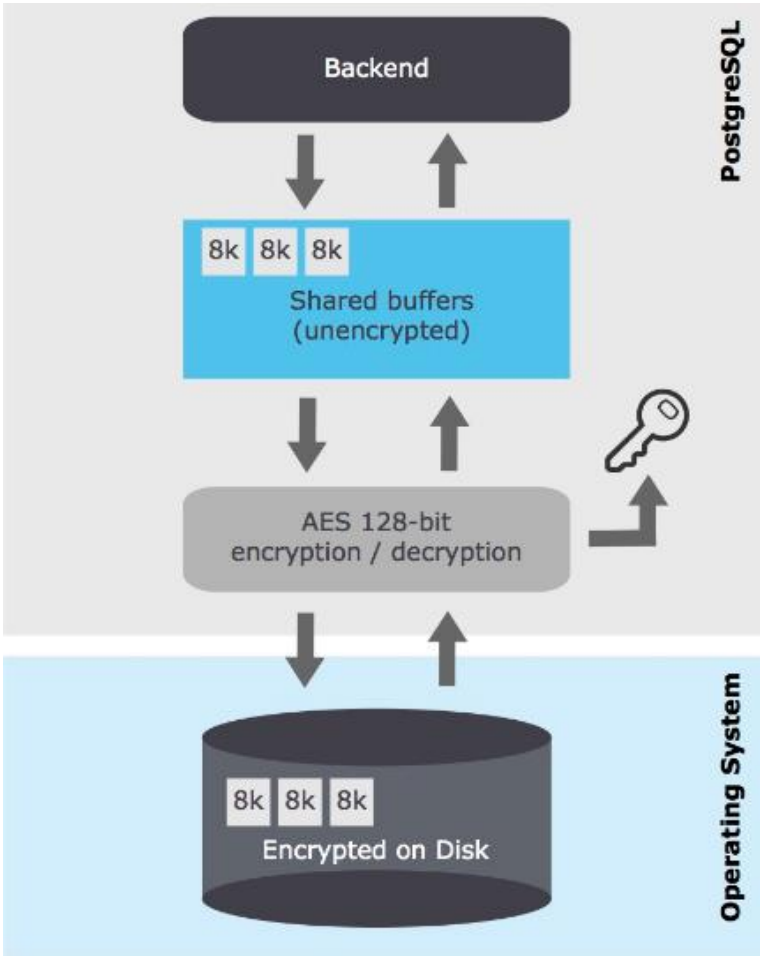
After applying a database patch for TDE encryption, PostgreSQL encrypts data (both relations and write-ahead log) when writing to disk and decrypt it when reading. The encryption is transparent, so the application sees no difference between the encrypted and unencrypted clusters.

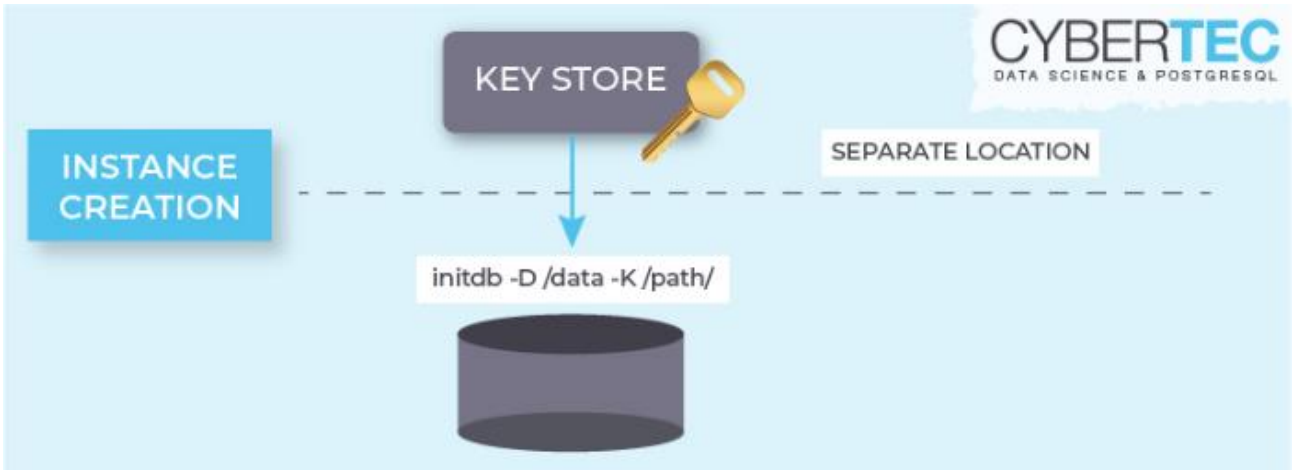


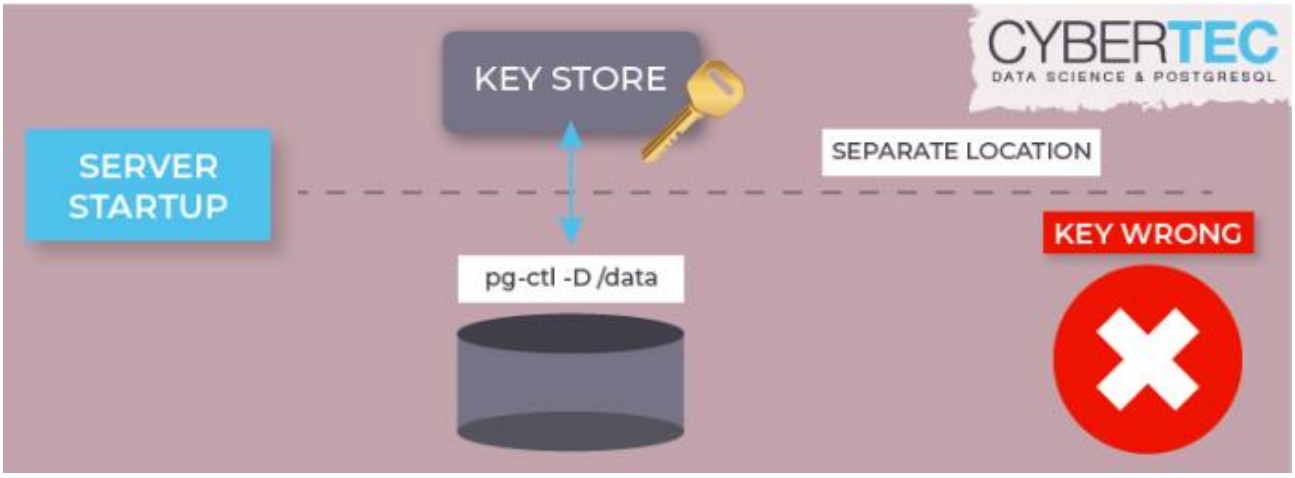
2 How does the encryption work

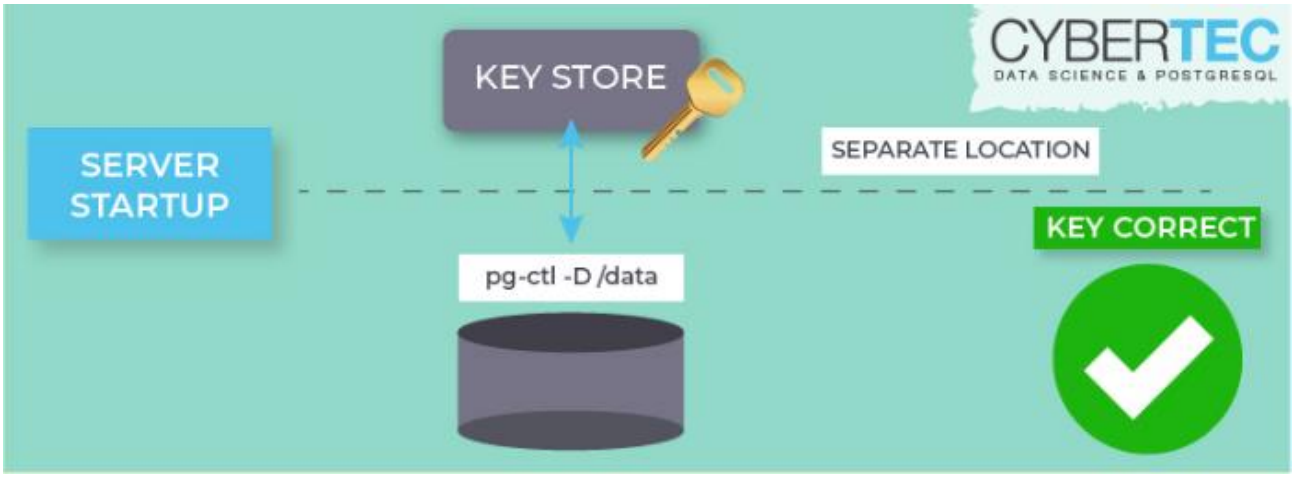
The idea behind the patch is to store all the files making up a PostgreSQL cluster securely on disk in an encrypted format (data-at-rest encryption) and then decrypt blocks as they are read from disk. This only requires that the database is initialized with encryption in mind and that the key used for initializing the database is accessible to the server during start-up.

The encryption key can be provided in two ways – through an environment variable or through a special configuration parameter specifying a custom key setup command for implementing special security requirements.




CYBERTEC
DATA SCIENCE & POSTGRESQL
KEY STORE
INSTANCE CREATION
SEPARATE LOCATION
initdb -D /data -K /path/


CYBERTEC
DATA SCIENCE & POSTGRESQL
KEY STORE
SERVER STARTUP
SEPARATE LOCATION
KEY WRONG
pg-ctl -D /data


CYBERTEC
DATA SCIENCE & POSTGRESQL
KEY STORE
SERVER STARTUP
SEPARATE LOCATION
KEY CORRECT
pg-ctl -D /data

3 Details

For encryption, a 128-bit AES algorithm in XTS mode is used, sometimes called also XTS-AES. It's a block cipher with a "tweak" for extra security and adheres to IEEE P1619 standard. The key needs to be provided to the server during every start-up, and when it doesn't match, the server will refuse to start. Encrypted will be more or less everything - heap files (tables, indexes, and sequences), xlog (as they also contain data), clog, temporary files being generated during the execution of a larger query.

Performance penalty incurred by encryption/decryption depends heavily on concrete use cases. For cases where the working set fits well into PostgreSQL shared buffers, it isn't very important, though.

4 Expectations on performance

Naturally, one agrees to compromise on performance when going for encryption, as there are no free lunches. Here things are a bit fuzzy – one can expect a very big performance hit if your workload is IO-oriented (logging, etc.). On the other hand, on typical server hardware, when the active dataset stays more or less in shared buffers, the performance penalty will be minor and not noticeable.

5 Comparative analysis

The comparative test was performed on two PostgreSQL database servers: standard PostgreSQL database (PG1) and PostgreSQL database with TDE (PG2).

1. Determine which directory number relates to which database:

```
SELECT oid as object_id, datname as database_name FROM pg_database;
```

Which returns:

```
object_id | database_name
------------+----------------
        1 | template1
    14799 | template0
    14804 | postgres
    20886 | test
```

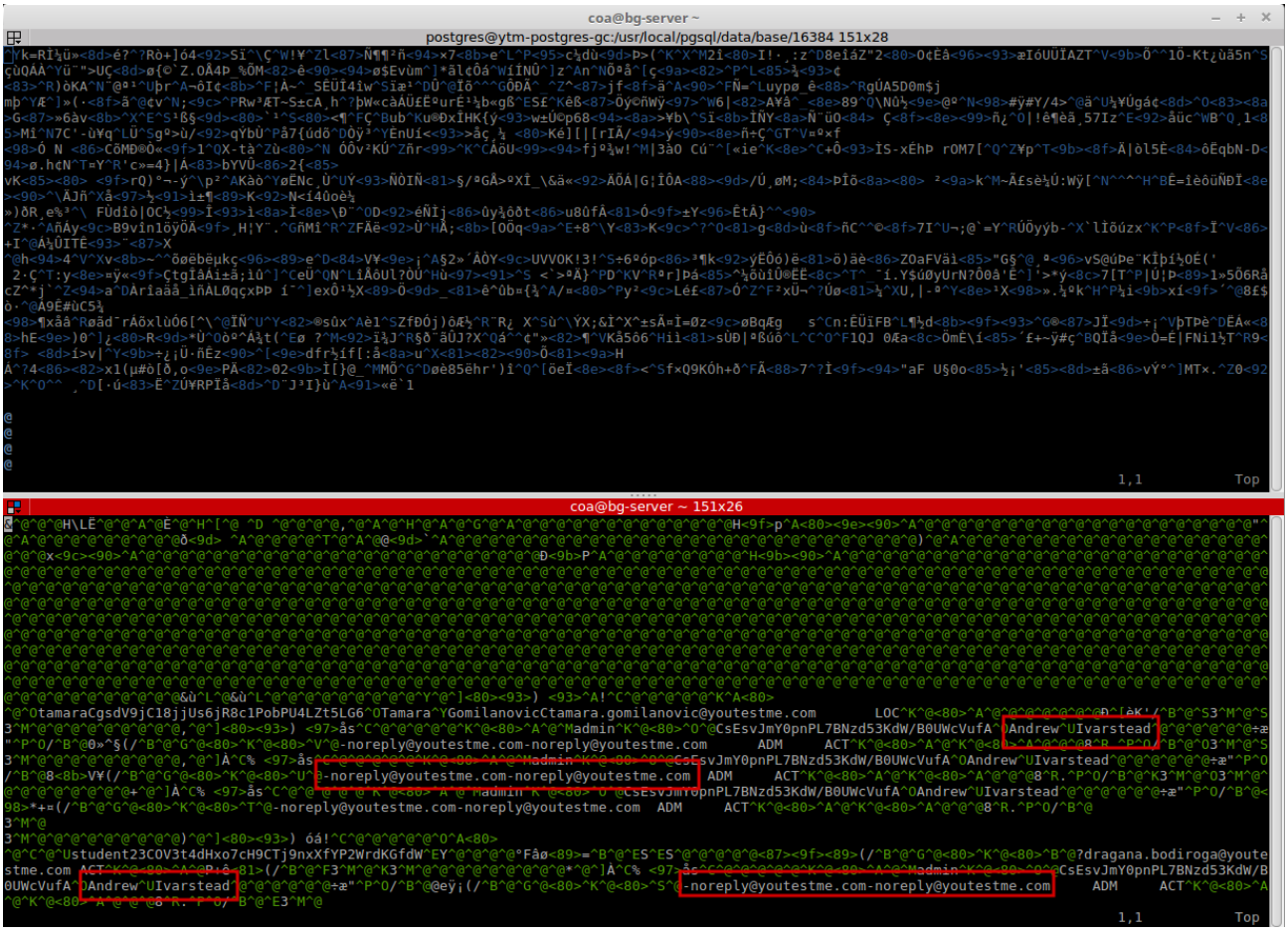
2. To find out which file contains the table data 'test_data', you can do the following:

Start up psql then:

```
\c test
SELECT pg_relation_filepath('test_data');

pg_relation_filepath
----------------------
base/20886/186770
```

3. After you located the file on both database servers, open them in some text editor and compare their content:



The example above presents different file content for the same table. In an unencrypted database can be noticed some personal information stored in plaintext, unlike the other PostgreSQL database that implements the TDE patch.

4. The alternative approach will search for a specific pattern in all PostgreSQL files (name or email of the known user in the database).

   For this purpose, you can use the following Linux command in the terminal:

```
find /usr/local/pgsql/data -type f -exec grep -ail Jessica {} \;
```