Transparent Data Encryption

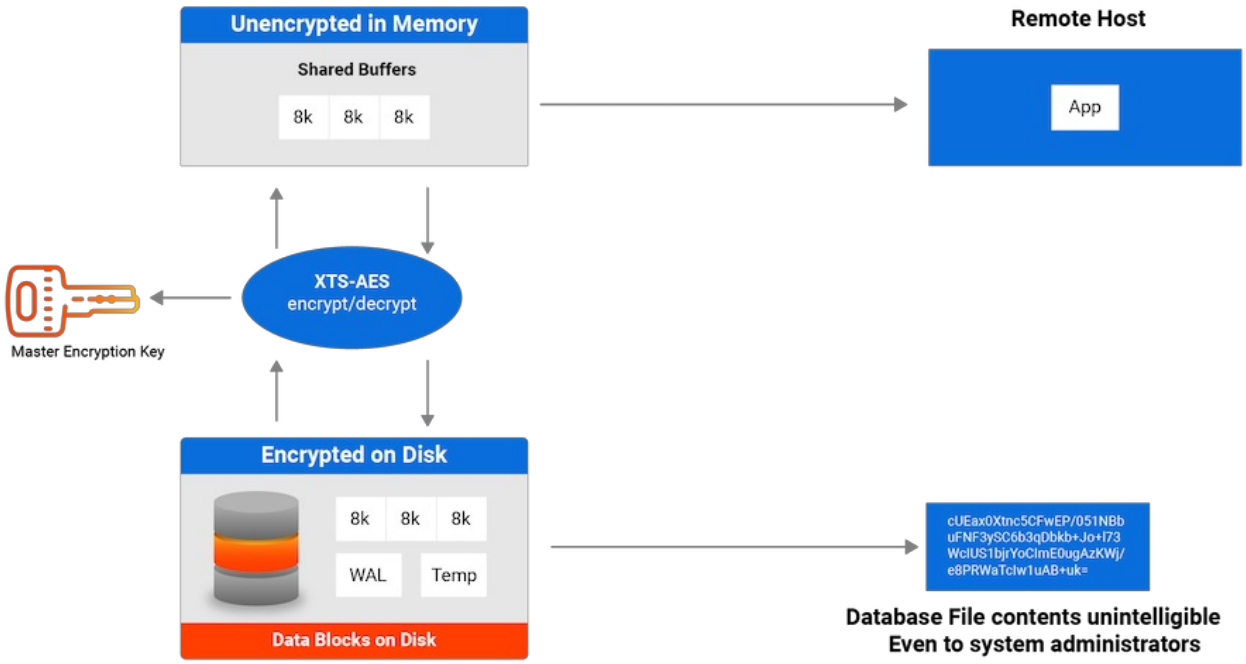
Version 15

Built at 2024-05-09T17:41:05

1 Transparent Data Encryption

Transparent data encryption (TDE) is an optional feature supported by EDB Postgres Advanced Server and EDB Postgres Extended Server from version 15.

It encrypts any user data stored in the database system. This encryption is transparent to the user. User data includes the actual data stored in tables and other objects as well as system catalog data such as the names of objects.



What's encrypted with TDE?

TDE encrypts:

- The files underlying tables, sequences, indexes, including TOAST tables and system catalogs, and including all forks. These files are known as *data files*.
- The write-ahead log (WAL).
- Various temporary files that are used during query processing and database system operation.

> **Implications**
>
> - Any WAL fetched from a server using TDE, including by streaming replication and archiving, is encrypted.
> - A physical replica is necessarily encrypted (or not encrypted) in the same way and using the same keys as its primary server.
> - If a server uses TDE, a base backup is automatically encrypted.

The following aren't encrypted or otherwise disguised by TDE:

- Metadata internal to operating the database system that doesn't contain user data, such as the transaction status (for example, pg_subtrans and pg_xact).
- The file names and file system structure in the data directory. That means that the overall size of the database system, the number of databases, the number of tables, their relative sizes, as well as file system metadata such as last access time are all visible without decryption.
- Data in foreign tables.
- The server diagnostics log.
- Configuration files.

> **Implications**
>
> Logical replication isn't affected by TDE. Publisher and subscriber can have different encryption settings. The payload of the logical replication protocol isn't encrypted. (You can use SSL.)

How does TDE affect performance?

The performance impact of TDE is low. For details, see the Transparent Data Encryption Impacts on EDB Postgres Advanced Server 15 blog.

How does TDE work?

TDE prevents unauthorized viewing of data in operating system files on the database server and on backup storage. Data becomes unintelligible for unauthorized users if it's stolen or misplaced.

Data encryption and decryption is managed by the database and doesn't require application changes or updated client drivers.

EDB Postgres Advanced Server and EDB Postgres Extended Server provide hooks to key management that's external to the database. These hooks allow for simple passphrase encrypt/decrypt or integration with enterprise key management solutions. See Securing the data encryption key for more information.

How does TDE encrypt data?

Starting with version 16, EDB TDE introduces the option to choose between AES-128 and AES-256 encryption algorithms during the initialization of the Postgres cluster. The choice between AES-128 and AES-256 hinges on balancing performance and security requirements. AES-128 is commonly advised for environments where performance efficiency and lower power consumption are pivotal, making it suitable for most applications. Conversely, AES-256 is recommended for scenarios demanding the highest level of security, often driven by regulatory mandates.

TDE uses AES-128-XTS or AES-256-XTS algorithms for encrypting data files. XTS uses a second value, known as the *tweak value*, to enhance the encryption. The XTS tweak value with TDE uses the database OID, the relfilenode, and the block number.

For write-ahead log (WAL) files, TDE uses AES-128-CTR or AES-256-CTR, incorporating the WAL's log sequence number (LSN) as the counter component.

Temporary files that are accessed by block are also encrypted using AES-128-XTS or AES-256-XTS. Other temporary files are encrypted using AES-128-CBC or AES-256-CBC.

How is data stored on disk with TDE?

In this example, the data in the `tbfoo` table is encrypted. The `pg_relation_filepath` function locates the data file corresponding to the `tbfoo` table.

```
insert into tbfoo values
('abc','123');
INSERT 0 1

select
pg_relation_filepath('tbfoo');

 pg_relation_filepath
----------------------
 base/5/16416
```

Grepping the data looking for characters doesn't return anything. Viewing the last five lines returns the encrypted data:

```
$ hexdump -C 16416 | grep abc
$

$ hexdump -C 16416 | tail -5
00001fc0  c8 0f 1d c8 9a 63 3d dc  7d 4e 68 98 b8 f2 5e 0a  |.....c=.}Nh...^.|
00001fd0  9a eb 20 1d 59 ad be 94  6e fd d5 6e ed 0a 72 8c  |.. .Y...n..n..r.|
00001fe0  7b 14 7f de 5b 63 e3 84  ba 6c e7 b0 a3 86 aa b9  |{...[c...l......|
00001ff0  fe 4f 07 50 06 b7 ef 6a  cd f9 84 96 b2 4b 25 12  |.O.P...j.....K%.|
00002000
```

2 Securing the data encryption key

The key for transparent data encryption (the data key) is normally generated by `initdb` and stored in a file `pg_encryption/key.bin` under the data directory. This file actually contains several keys that are used for different purposes at run time. However, in terms of the data key, it contains a single sequence of random bytes.

Without any further action, this file contains the key in plaintext, which isn't secure. Anyone with access to the encrypted data directory has access to the plaintext key, which defeats the purpose of encryption. Therefore, this setup is suitable only for testing purposes.

To secure the data key properly, “wrap” it by encrypting it with another key. Broadly, you can use two approaches to arrange this:

- Protect the data key with a passphrase. A wrapping key is derived from the passphrase and used to encrypt the data key.
- The wrapping key is stored elsewhere, for example, in a key management system, also known as a key store. This second key is also called the *key-wrapping key* or *master key*.

If you don't want key wrapping, for example for testing, then you must set the wrap and unwrap commands to the special value `-`. This setting specifies to use the key from the file without further processing. This approach differs from not setting a wrap or unwrap command at all, and from setting either/both to an empty string. Having no wrap or unwrap command set when transparent data encryption is used results in a fatal error when running an affected utility program.

Postgres leaves this configuration up to the user, which allows tailoring the setup to local requirements and integrating with existing key management software or similar. To configure the data key protection, you must specify a pair of external commands that take care of the wrapping (encrypting) and unwrapping (decryption).

Using a passphrase

You can protect the data key with a passphrase using the openssl command line utility. The following is an example that sets up this protection:

```
initdb -D datadir -y --key-wrap-command='openssl enc -e -aes-128-cbc -pbkdf2 -out "%p"' --key-unwrap-command='openssl enc -d -aes-128-cbc -pbkdf2 -in "%p"'
```

This example wraps the randomly generated data key (done internally by initdb) by encrypting it using the AES-128-CBC (AESKW) algorithm. The encryption uses a key derived from a passphrase using the PBKDF2 key derivation function and a randomly generated salt. The terminal prompts for the passphrase. (See the openssl-enc manual page for details about these options. Available options vary across versions.) The placeholder `%p` is replaced with the name of the file to store the wrapped key.

The unwrap command performs the opposite operation. initdb doesn't need the unwrap operation. However, it stores it in the `postgresql.conf` file of the initialized cluster, which uses it when it starts up.

The key wrap command receives the plaintext key on standard input and needs to put the wrapped key at the file system location specified by the `%p` placeholder. The key unwrap command needs to read the wrapped key from the file system location specified by the `%p` placeholder and write the unwrapped key to the standard output.

Utility programs like pg_rewind and pg_upgrade operate directly on the data directory or copies, such as backups. These programs also need to be told about the key unwrap command, depending on the circumstances. They each have command-line options for this purpose.

To simplify operations, you can also set the key wrap and unwrap commands in environment variables. These are accepted by all affected applications if you don't provide the corresponding command line options. For example:

```
PGDATAKEYWRAPCMD='openssl enc -e -aes-128-cbc -pbkdf2 -out "%p"'
PGDATAKEYUNWRAPCMD='openssl enc -d -aes-128-cbc -pbkdf2 -in "%p"'
export PGDATAKEYWRAPCMD PGDATAKEYUNWRAPCMD
```

Key unwrap commands that prompt for passwords on the terminal don't work when the server is started by pg_ctl or through service managers such as systemd. The server is detached from the terminal in those environments. If you want an interactive password prompt on server start, you need a more elaborate configuration that fetches the password using some indirect mechanism.

For example, for systemd, you can use `systemd-ask-password`:

```
PGDATAKEYWRAPCMD="bash -c 'openssl enc -e -aes-128-cbc -pbkdf2 -out %p -pass file:<(sudo systemd-ask-password --no-tty)'"
PGDATAKEYUNWRAPCMD="bash -c 'openssl enc -d -aes-128-cbc -pbkdf2 -in %p -pass file:<(sudo systemd-ask-password --no-tty)'"
```

You also need an entry like in `/etc/sudoers`:

```
postgres ALL = NOPASSWD: /usr/bin/systemd-ask-password
```

Using a key store

You can use the key store in an external key management system to manage the data encryption key. The tested and supported key stores are:

- Amazon AWS Key Management Service (KMS)
- Google Cloud - Cloud Key Management Service
- HashiCorp Vault (KMIP Secrets Engine and Transit Secrets Engine)

- Microsoft Azure Key Vault
- Thales CipherTrust Manager

AWS Key Management Service example

Create a key with AWS Key Management Service:

```
aws kms create-key
aws kms create-alias --alias-name alias/pg-tde-master-1 --target-key-id "..."
```

Use the `aws kms` command with the `alias/pg-tde-master-1` key to wrap and unwrap the data encryption key:

```
PGDATAKEYWRAPCMD='aws kms encrypt --key-id alias/pg-tde-master-1 --plaintext fileb:///dev/stdin --
output text --query CiphertextBlob | base64 -d > "%p"'
PGDATAKEYUNWRAPCMD='aws kms decrypt --key-id alias/pg-tde-master-1 --ciphertext-blob fileb://"%p" --
output text --query Plaintext | base64 -d'
```

> **Note**
>
> Shell commands with pipes, as in this example, are problematic because the exit status of the pipe is that of the last command. A failure of the first, more interesting command isn't reported properly. Postgres handles this somewhat by recognizing whether the wrap or unwrap command wrote nothing. However, it's better to make this more robust. For example, use the `pipefail` option available in some shells or the `mispipe` command available on some operating systems. Put more complicated commands into an external shell script or other program instead of defining them inline.

Alternatively, you can use the crypt utility to wrap and unwrap the data encryption key:

```
PGDATAKEYWRAPCMD='crypt encrypt aws --out %p --region us-east-1 --kms alias/pg-tde-master-1'
PGDATAKEYUNWRAPCMD='crypt decrypt aws --in %p --region us-east-1'
```

Azure Key Vault example

Create a key with Azure Key Vault:

```
az keyvault key create --vault-name pg-tde --name pg-tde-master-1
```

Use the `az keyvault` command with the `pg-tde-master-1` key to wrap and unwrap the data encryption key:

```
PGDATAKEYWRAPCMD='crypt encrypt azure --vaultURL https://pg-tde.vault.azure.net --name pg-tde-master-1
--version fa2bf368449e432085318c5bf666754c --out %p'
PGDATAKEYUNWRAPCMD='crypt decrypt azure --vaultURL https://pg-tde.vault.azure.net --name pg-tde-master-
1 --version fa2bf368449e432085318c5bf666754c --in %p'
```

This example uses crypt. You can't use the Azure CLI directly for this purpose because it lacks some functionality.

Google Cloud KMS example

Create a key with Google Cloud KMS:

```
gcloud kms keys create pg-tde-master-1 --location=global --keyring=pg-tde --purpose=encryption
```

Use the `az keyvault` command with the `pg-tde-master-1` key to wrap and unwrap the data encryption key:

```
PGDATAKEYWRAPCMD='gcloud kms encrypt --plaintext-file=- --ciphertext-file=%p --location=global --
keyring=pg-tde --key=pg-tde-master-1'
PGDATAKEYUNWRAPCMD='gcloud kms decrypt --plaintext-file=- --ciphertext-file=%p --location=global --
keyring=pg-tde --key=pg-tde-master-1'
```

Alternatively, you can use the crypt utility to wrap and unwrap the data encryption key:

```
PGDATAKEYWRAPCMD='crypt encrypt gcp --out=%p --location=global --keyring=pg-tde --key=pg-tde-master-1 -
-project your-project-123456'
PGDATAKEYUNWRAPCMD='crypt decrypt gcp --in=%p --location=global --keyring=pg-tde --key=pg-tde-master-1
--project your-project-123456'
```

HashiCorp Vault Transit Secrets Engine example

```
# enable once
vault secrets enable transit

# create a key (pick a name)
vault write -f transit/keys/pg-tde-master-1

PGDATAKEYWRAPCMD='base64 | vault write -field=ciphertext transit/encrypt/pg-tde-master-1 plaintext=- >
%p'
PGDATAKEYUNWRAPCMD='vault write -field=plaintext transit/decrypt/pg-tde-master-1 ciphertext=- < %p |
base64 -d'
```

Key rotation

To change the master key, manually run the unwrap command specifying the old key. Then feed the result into the wrap command specifying the new key. Equivalently, if the data key is protected by a passphrase, to change the passphrase, run the unwrap command using the old passphrase. Then feed the result into the wrap command using the new passphrase. You can perform these operations while the database server is running. The wrapped data key in the file is used only on startup. It isn't used while the server is running.

Building on the example in Using a passphrase, which uses openssl, to change the passphrase, you can:

```
cd $PGDATA/pg_encryption/
openssl enc -d -aes-128-cbc -pbkdf2 -in key.bin | openssl enc -e -aes-128-cbc -pbkdf2 -out key.bin.new
mv key.bin.new key.bin
```

With this method, the decryption and the encryption commands ask for the passphrase on the terminal at the same time, which is awkward and confusing. An alternative is:

```
cd $PGDATA/pg_encryption/
openssl enc -d -aes-128-cbc -pbkdf2 -in key.bin -pass pass:<replaceable>ACTUALPASSPHRASE</replaceable>
| openssl enc -e -aes-128-cbc -pbkdf2 -out key.bin.new
mv key.bin.new key.bin
```

This technique leaks the old passphrase, which is being replaced anyway. openssl supports a number of other ways to supply the passphrases.

When using a key management system, you can connect the unwrap and wrap commands similarly, for example:

```
cd $PGDATA/pg_encryption/
crypt decrypt aws --in key.bin --region us-east-1 | crypt encrypt aws --out key.bin.new --region us-east-1 --kms alias/pg-tde-master-2
mv key.bin.new key.bin
```

> **Note**
>
> You can't change the data key (the key wrapped by the master key) on an existing data directory. If you need to do that, you need to run the data directory through an upgrade process using pg_dump, pg_upgrade, or logical replication.

3 Enabling TDE

You enable transparent data encryption when you initialize a database cluster using initdb.

Using initdb TDE options

To enable encryption, use the following options with the `initdb` command or their fallback environment variables:

`-y, --data-encryption`

Initialize the new database cluster with transparent data encryption. See Transparent Data Encryption for more information. Optionally specify an AES key length. Valid values are 128 and 256. The default is 128.

`--copy-key-from=<file>`

Copy the data encryption key from the given location. You can use this option to copy a key from an existing cluster when preparing a new cluster as a target for pg_upgrade.

`--key-wrap-command=<command>`

Specify a command to wrap (encrypt) the generated data encryption key. The command must include a placeholder `%p` that specifies the file to write the wrapped key to. The unwrapped key is provided to the command on its standard input. If you don't specify this option, the environment variable `PGDATAKEYWRAPCMD` is used.

Use the special value `-` if you don't want to apply any key wrapping command.

You must specify this option or the environment variable fallback if you're using data encryption. SeeSecuring the data encryption key for more information.

`--key-unwrap-command=<command>`

Specify a command to unwrap (decrypt) the data encryption key. The command must include a placeholder `%p` that specifies the file to read the wrapped key from. The command needs to write the unwrapped key to its standard output. If you don't specify this option, the environment variable `PGDATAKEYUNWRAPCMD` is used.

Use the special value `-` if you don't want to apply any key unwrapping command.

You must specify this option or the environment variable fallback if you're using data encryption. See Securing the data encryption key for more information.

`--no-key-wrap`

Disable key wrapping. The data encryption key is instead stored in plaintext in the data directory. (This option is a shortcut for setting both the wrap and the unwrap command to the special value `-`.)

> **Note**
>
> Using this option isn't secure. Use it only for testing purposes.

If you select data encryption and don't specify this option, then you must provide key wrap and unwrap commands. Otherwise, `initdb` terminates with an error.

Using environment variables

To simplify operations, you can set the key wrap and unwrap commands in the environment variables.

For example:

```
PGDATAKEYWRAPCMD='openssl enc -e -aes128-wrap -pbkdf2 -out "%p"'
PGDATAKEYUNWRAPCMD='openssl enc -d -aes128-wrap -pbkdf2 -in "%p"'
export PGDATAKEYWRAPCMD PGDATAKEYUNWRAPCMD
```

Setting the key parameter in postgresql.conf

When you enable TDE for a cluster, the `initdb` command initializes the `data_encryption_key_unwrap_command` parameter in the `postgresql.conf` configuration file. The string specified in `data_encryption_key_unwrap_command` unwraps (decrypts) the data encryption key.

The command must contain a placeholder `%p`, which is replaced with the name of the file containing the key to unwrap. The command must print the unwrapped (decrypted) key to its standard output.

If you don't specify this parameter, the environment variable `PGDATAKEYUNWRAPCMD` is used.

Use the special value `-` if you don't want to apply any key unwrapping command.

You must specify this parameter or the environment variable fallback if you're using data encryption. See Securing the data encryption key for more information.

You can set this parameter only at server start.

This parameter is normally initialized by `initdb`. Change it only if you change the key wrap method.

For more information on the configuration files, see PostgreSQL File Locations documentation.

Example

This example uses EDB Postgres Advanced Server 15 running on a Linux platform. It uses openssl to define the passkey to wrap and unwrap the generated data encryption key.

1. Set the data encryption key (wrap) and decryption (unwrap) environment variables:

```
export PGDATAKEYWRAPCMD='openssl enc -e -aes-128-cbc -pass pass:ok -out %p'
export PGDATAKEYUNWRAPCMD='openssl enc -d -aes-128-cbc -pass pass:ok -in %p'
```

> **Note**
>
> If you are on Windows you don't need the single quotes around the variable value.

2. Initialize the cluster using `initdb` with encryption enabled. This command sets the `data_encryption_key_unwrap_command` parameter in the postgresql.conf file.

```
/usr/edb/as15/bin/initdb --data-encryption -D /var/lib/edb/as15/data
```

3. Start the cluster:

```
/usr/edb/as15/bin/pg_ctl -D /var/lib/edb/as15/data start
```

4. Run grep on postgresql.conf to see the setting of `data_encryption_key_unwrap_command`:

```
grep data_encryption_key_unwrap_command /var/lib/edb/as15/data/postgresql.conf
```

```
data_encryption_key_unwrap_command = 'openssl enc -d -aes-128-cbc -pass pass:ok -in %p'
```

Checking for TDE presence using SQL

You can find out whether TDE is present on a server by querying the `data_encryption_version` column of the `pg_control_init` table.

A value of 0 means TDE isn't enabled. Any nonzero value reflects the version of TDE in use. Currently, when TDE is enabled, this value is 1.

```
# select data_encryption_version from pg_control_init();
 data_encryption_version
-------------------------
                       1
(1 row)
```

4 Limitations

`FILE_COPY`

If transparent data encryption is enabled, you can't use the `FILE_COPY` strategy in the `strategy` parameter with `CREATE DATABASE`.

See the PostgreSQL CREATE DATABASE documentation for more information.

5 Commands affected by TDE

When TDE is enabled, the following commands have TDE-specific options or read TDE settings in environment variables or configuration files:

- pg_waldump
- pg_resetwal
- pg_verifybackup
- pg_rewind
- pg_upgrade
- postgres

6 Troubleshooting with encrypted WAL files

You can encrypt WAL files. When troubleshooting with encrypted WAL falls, you can use WAL command options.

Dumping a TDE-encrypted WAL file

To work with an encrypted WAL file, the pg_waldump needs to be aware of the unwrap key. You can either pass the key for the unwrap command using the following options to the `pg_waldump` command or depend on the fallback environment variable:

`--data-encryption`

Consider the WAL files to encrypt, and decrypt them before processing them. You must specify this option if the WAL files were encrypted by transparent data encryption. `pg_waldump` can't automatically detect whether WAL files are encrypted. Optionally, specify an AES key length. Valid values are 128 and 256. The default is 128.

`--key-file-name=<file>`

Load the data encryption key from the given location.

`--key-unwrap-command=<command>`

Specifies a command to unwrap (decrypt) the data encryption key. The command must include a placeholder `%p` that specifies the file to read the wrapped key from. The command needs to write the unwrapped key to its standard output. If you don't specify this option, the environment variable `PGDATAKEYUNWRAPCMD` is used.

Use the special value `-` if you don't want to apply any key unwrapping command.

You must specify this option or the environment variable fallback if you're using data encryption. See Securing the data encryption key for more information.

Resetting a corrupt TDE-encrypted WAL file

To reset a corrupt encrypted WAL file, the pg_resetwal command needs to be aware of the unwrap key. You can either pass the key for the unwrap command using the following option to the `pg_resetwal` command or depend on the fallback environment variable:

```
--key-unwrap-command=<command>
```

Specifies a command to unwrap (decrypt) the data encryption key. The command must include a placeholder `%p` that specifies the file to read the wrapped key from. The command needs to write the unwrapped key to its standard output. If you don't specify this option, the environment variable `PGDATAKEYUNWRAPCMD` is used.

Use the special value `-` if you don't want to apply any key unwrapping command.

You must specify this option or the environment variable fallback if you're using data encryption. See Securing the data encryption key for more information.

7 Working with encrypted backup files

Verify a backup of a TDE system

To verify an encrypted backup file, the pg_verifybackup command needs to be aware of the unwrap key. You can either pass the key for the unwrap command using the following option to the `pg_verifybackup` command or depend on the fallback environment variable.

```
--key-unwrap-command=<command>
```

Specifies a command to unwrap (decrypt) the data encryption key. The command must include a placeholder `%p` that specifies the file to read the wrapped key from. The command needs to write the unwrapped key to its standard output. If you don't specify this option, the environment variable `PGDATAKEYUNWRAPCMD` is used.

Use the special value `-` if you don't want to apply any key unwrapping command.

You must specify this option or the environment variable fallback if you're using data encryption. See Securing the data encryption key for more information.

Resynchronize timelines in a TDE system

To resynchronize an encrypted cluster with its backup, the pg_rewind command needs to be aware of the unwrap key. You can either pass the key for the unwrap command using the following option to the `pg_rewind` command or depend on the fallback environment variable:

`--key-unwrap-command=<command>`

Specifies a command to unwrap (decrypt) the data encryption key. The command must include a placeholder `%p` that specifies the file to read the wrapped key from. The command needs to write the unwrapped key to its standard output. If you don't specify this option, the environment variable `PGDATAKEYUNWRAPCMD` is used.

Use the special value `-` if you don't want to apply any key unwrapping command.

You must specify this option or the environment variable fallback if you're using data encryption. SeeSecuring the data encryption key for more information.

8 Upgrading a TDE system

These options to pg_upgrade help with upgrading encrypted clusters.

`--copy-by-block`

Copy files to the new cluster block by block instead of the default, which is to copy the whole file at once. This option is the same as the default mode but somewhat slower. It does, however, support upgrades between clusters with different encryption settings.

You must use this option when upgrading between clusters with different encryption settings, that is, unencrypted to encrypted, encrypted to unencrypted, or both encrypted with different keys. While copying files to the new cluster, it decrypts them and reencrypts them with the keys and settings of the new cluster.

For added certainty, if the old cluster is encrypted and the new cluster was initialized as unencrypted, this option decrypts the data from the old cluster and copies it to the new cluster unencrypted. If the old cluster is unencrypted and the new cluster was initialized as encrypted, this option encrypts the data from the old cluster and places it into the new cluster encrypted.

See the description of the initdb --copy-key-from=<file> option for information on copying a key from an existing cluster when preparing a new cluster as a target for `pg_upgrade`.

`--key-unwrap-command=<command>`

Specifies a command to unwrap (decrypt) the data encryption key. The command must include a placeholder `%p` that specifies the file to read the wrapped key from. The command needs to write the unwrapped key to its standard output. If you don't specify this option, the environment variable `PGDATAKEYUNWRAPCMD` is used.

Use the special value `-` if you don't want to apply any key unwrapping command.

You must specify this option or the environment variable fallback if you're using data encryption. SeeSecuring the data encryption key for more information.

9 Single-user mode

If you invoke postgres in single-user mode with TDE enabled, the `postgres` command reads either:

- The `PGDATAKEYUNWRAPCMD` environment variable, if set
- The `data_encryption_key_unwrap_command` value in the `postgresql.conf` file

10 Testing a TDE configuration

To run the tests in single-user mode with transparent data encryption enabled, set the environment variable `PG_TEST_USE_DATA_ENCRYPTION`. For example:

```
make check PG_TEST_USE_DATA_ENCRYPTION=1
```

See also

- Enabling TDE
- PostgreSQL postgres command documentation
- PostgreSQL Running the Tests documentation