Transparent Data Encryption

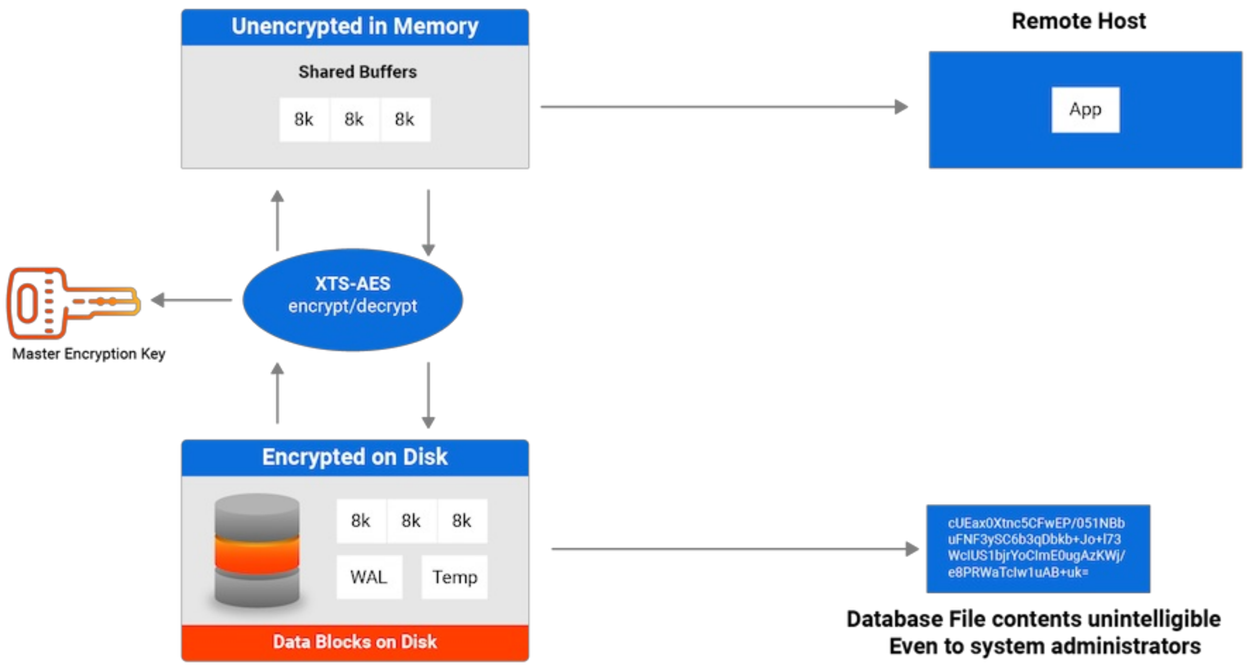
Version 15

Built at 2024-11-20T17:56:19

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.


Unencrypted in Memory
Shared Buffers
8k 8k 8k
Remote Host
App
XTS-AES encrypt/decrypt
Master Encryption Key
Encrypted on Disk
8k 8k 8k
WAL
Temp
Data Blocks on Disk
cUEax0Xtnc5CFwEP/051NBb uFNF3ySC6b3qDbkb+Jo+I73 WcIUS1bjrYoClmE0ugAzKWj/ e8PRWaTclw1uAB+uk=
Database File contents unintelligible
Even to system administrators

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.

2 About TDE

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

It encrypts user data stored in the database system.

2.1 How does TDE encrypt data?

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?

EDB TDE uses OpenSSL to encrypt data files with the AES encryption algorithm. In Windows systems, TDE uses OpenSSL 3. In Linux systems, TDE uses the OpenSSL version installed in the host operating system. To check the installed version, run `openssl version`. For more information, see the OpenSSL documentation. If you're using a custom build not provided by the OpenSSL community, consult your vendor's documentation.

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
```

2.2 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.)

2.3 Why should you use TDE?

TDE encryption ensures that user data remains protected from unauthorized access.

When configured with a data encryption key securing mechanism, data stored on the database server and in backup is only accessible by users and processes with decryption keys.

Some use cases include:

- **Protection of sensitive personal data:** Industries like finance, e-commerce, healthcare, and government organizations often deal with personally identifiable information that must be protected to comply with data privacy regulations such as GDPR, HIPPA, PCI DSS.

- **Compliance with government standards:** Government institutions must comply with information processing standards like FIPS to ensure computer security and interoperability.

- **Protection of transactional data:** Financial institutions deal with transaction, account, and payment data that must be protected to prevent fraud and financial losses.

- **Protection of intellectual property:** Organizations safeguard proprietary information, designs, and plans to keep their competitive advantage, support brand value, and foster innovation.

- **Protection of data in cloud-based deployments and public web applications:** Encrypting a database's data provides an additional layer of security when infrastructure is shared, or when vulnerabilities could potentially infiltrate in an application's API.

When your data is encrypted, it becomes unintelligible if it's stolen or misplaced.

3 TDE compatibility

You can create TDE-enabled database servers on several database distributions and technologies.

Supported database distributions

You can create TDE-enabled databases on:

- EDB Postgres Advanced Server 15 and later versions.
- EDB Postgres Extended Server 15 and later versions.

Note

EDB doesn't support creating TDE-enabled servers on PostgreSQL. TDE relies on a mechanism developed by EDB for EDB Postgres Advanced and Extended Servers.

Supported technologies

You can create TDE-enabled servers in the following environments:

- EDB Postgres Distributed (PGD)
- EDB Postgres Distributed for Kubernetes
- EDB Postgres for Kubernetes
- EDB Postgres AI Cloud Service

4 Overview

If you want to start using Transparent Data Encryption (TDE) on your database, you either want to create a new TDE-enabled database server, or you want to migrate an existing database server to a TDE-enabled environment. It is not possible to enable TDE on existing instances.

Regardless of whether you are creating a database server from scratch, or creating a new instance to migrate an existing database server, you have to create a TDE-enabled database by initializing a database cluster using initdb.

Before you begin

- Choose a method to secure the data encryption key generated by TDE.

  You can protect they key with a passphrase, a wrapping key from a key store or choose not protect the key for testing purposes.

- Review the initdb TDE options to ensure you have all information required for the initialization of a TDE-enabled database cluster.

- Review the Limitations and TDE-specific options to understand limitations and changes in the handling of PostgreSQL utilities when you enable TDE.

- If you plan on migrating data from an existing database server, ensure you perform a backup of the source database server.

Initializing a server

1. Export the wrapping and unwrapping commands to secure the encryption key. Use the wrapping method you chose during the planning phase.

   Alternatively, you can provide the wrapping and unwrapping commands when initializing the server with the command line arguments.

   See Providing the wrapping and unwrapping commands for examples.

2. Initialize a database server with `--data-encryption` enabled on the target directory. Include other TDE options as required.

3. Start the database cluster and verify that TDE is enabled.

See Tutorials for detailed initialization examples.

Migrating data (for existing instances)

If you want to migrate data and objects from an existing database server, use `pg_upgrade` to copy data from an exiting instance:

1. Stop both the source and new server.

2. Use `pg_upgrade` with `--copy-by-block` option to copy data from the source server to the new server. Include other TDE pg_upgrade options as required.

3. Start the new encrypted database server.

4. Connect to the encrypted database server and ensure the data was transfered.

5. Perform any required cleanup operations.

> **Note**
>
> See TDE pg_upgrade use cases for an overview of the supported enablement and migration use cases.

See Tutorials for detailed migration examples.

Tutorials

- Creating a TDE-enabled database server.
- Enabling TDE on an existing EDB Postgres Advanced Server database cluster.
- Upgrading a PostgreSQL database server to EDB Postgres Extended Server while enabling TDE.

5 Securing the data encryption key

Data encryption key

The key for transparent data encryption (the data key) is 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. The data key is a single sequence of random bytes in the file.

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.

Choosing a mechanism to protect the data encryption key

To secure the data encryption key, you must specify a wrap and a unwrap command that provides TDE with a data encryption protection mechanism. You can set this parameter only at server start.

With the wrap and unwrap commands you can:

- Protect the data key with a passphrase. A wrapping key is derived from the passphrase and used to encrypt the data key.
- Protect the data key with a wrapping key stored in a Key Management System or key store. This second key is also called the *key-wrapping key* or *master key*.
- You can also choose to Disable the data encryption key wrapping. Do this only for test environments, as it leaves the TDE key unprotected.

Configuring a wrapping and unwrapping command

After you have chosen a method to protect your key, you can create the wrapping/unwrapping commands. The configuration of these commands is left to the user, which allows tailoring the setup to local requirements and integrating with existing key management software or similar.

The Using a passphrase topic provides an example for wrapping/unwrapping commands using OpenSSL and a passphrase to secure the TDE data key. The Using a key store topic provides an example for wrapping/unwrapping commands using an external key store key to protect the TDE data key.

When you initialize a server with TDE, the `initdb` command adds the `data_encryption_key_unwrap_command` parameter in the `postgresql.conf` configuration file. The string specified in `data_encryption_key_unwrap_command` is then able to unwrap (decrypt) the data encryption key.

The commands 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.

Providing the wrapping and unwrapping commands to TDE

You must make the commands available to the TDE database server so it can wrap and unwrap the data encryption key. You have the following options:

- You can configure the wrapping and unwrapping commands as environment variables before creating the database, so TDE can fall back on those variables when initializing a server.

  Example

  ```
  PGDATAKEYWRAPCMD='openssl enc -e -aes128-wrap -pbkdf2 -out "%p"'
  PGDATAKEYUNWRAPCMD='openssl enc -d -aes128-wrap -pbkdf2 -in "%p"'
  export PGDATAKEYWRAPCMD PGDATAKEYUNWRAPCMD
  #After these variables are set, you can initialize the server:
  initdb --data-encryption -D /var/lib/edb/as16/data
  ```

- You can provide the wrapping and unwrapping commands directly in the command line when initializing a server with the `--key-wrap-command=<command>` and `--key-unwrap-command=<command>` options.

  Example

  ```
  initdb --data-encryption -D /var/lib/edb/as16/data --key-wrap-command='openssl enc -e -aes128-wrap -pbkdf2 -out "%p"' --key-unwrap-command='openssl enc -d -aes128-wrap -pbkdf2 -in "%p"'
  ```

- You can disable the protection of your data encryption key with `--no-key-wrap`. This leaves your key unprotected and is recommended for testing proposes only.

  Example

  ```
  initdb --data-encryption -D /var/lib/edb/as16/data --no-key-wrap
  ```

5.1 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 with the AES-128-CBC (AESKW) algorithm. The encryption uses a key derived from a passphrase with 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 initdb utility replaces `%p` with the name of the file that stores 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
```

5.2 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)
- Microsoft Azure Key Vault
- Google Cloud - Cloud Key Management Service
- HashiCorp Vault (KMIP Secrets Engine and Transit Secrets Engine)
- Thales CipherTrust Manager
- Fortanix Data Security Manager
- Entrust KeyControl

To use one of the available key stores, see the configuration examples.

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.

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

```
PGDATAKEYWRAPCMD='az keyvault key encrypt --name pg-tde-master-1 --vault-name pg-tde --algorithm
A256GCM --value @- --data-type plaintext --only-show-errors --output json | jq -r .result > "%p"'
PGDATAKEYUNWRAPCMD='az keyvault key decrypt --name pg-tde-master-1 --vault-name pg-tde --algorithm
A256GCM --value @"%p" --data-type plaintext --only-show-errors --output json | jq -r .result'
```

> **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.

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 `gcloud kms` 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'
```

HashiCorp Vault Transit Secrets Engine example

Enable transit with HashiCorp Vault Transit Secrets Engine:

```
vault secrets enable transit
```

Create a key and give it a name:

```
vault write -f transit/keys/pg-tde-master-1
```

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

```
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'
```

Thales CipherTrust Manager example

See the Using section of the Implementing Thales CipherTrust Manager documentation for instructions how to wrap the data encryption key with a key from the Thales key store.

Fortanix Data Security Manager example

See Using Fortanix Data Security Manager with EDB Postgres for TDE for a step-by-step configuration tutorial.

Entrust KeyControl integration guide

See the EDB Postgres and Entrust KeyControl integration guide for installation, configuration and usage instructions (including key rotation).

5.3 Disabling the key wrapping

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 leaves your data encryption key unsecured, and results in a fatal error when running an affected utility program.

5.4 Rotating the data encryption key

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.

Rotating the passphrase

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.

Rotating the key store wrapping key

When using a key store, 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.

6 Using TDE initialization options

Initializing a TDE-enabled server requires two mandatory settings: one enables TDE and the other protects the data encryption key.

To enable TDE

To create a TDE-enabled database server, you must use the `--data-encryption` option, which creates a data encryption key to encrypt your server.

If you want to copy a key from an existing cluster when preparing a new cluster as a target for pg_upgrade, additionally use the `--copy-key-from=<file>` option.

To protect the data encryption key

When creating a TDE-enabled database, TDE generates a data encryption key that is transparent to the user.

An additional protection mechanism in the form or a wrapping and an unwrapping command is required to wrap this key, which you must make available to the database server.

See Providing the wrapping and unwrapping commands to TDE for an overview of the available protection mechanism, and examples of how to provide this configuration to `initdb`.

Options reference

See initdb TDE options for an overview of all mandatory and elective options and supported values.

7 Working with encrypted files

Certain Postgres utilities and operations require access to files to read data and deliver the required output.

In a TDE-enabled database server, data is encrypted and therefore not accesible without a decryption mechanism. For these utilities to be able to perform read operations on encrypted files, you must provide a decryption mechanism.

- Backup files provides guidance on how to work with and troubleshoot any issues with backup files.

- WAL files provides guidance on how to work with and troubleshoot any issues with WAL files.

7.1 Working with encrypted backup files

When TDE is enabled, backup files are encrypted. If you want to perform operations on the encrypted backup files, you need to allow the operations to decrypt the file.

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. SeeSecuring 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.

7.2 Working with encrypted WAL files

When TDE is enabled, WAL files are encrypted. If you want to perform operations on the encrypted WAL files, you need to allow the operations to decrypt the file.

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

Dumping a TDE-encrypted WAL file

To work with an encrypted WAL file, you need to ensure the pg_waldump utility can access the unwrap key and decrypt it. For this purpose, the utility requires awareness of three values.

Pass these values using the following options to the `pg_waldump` command. Be sure to use the same values you used when initializing the TDE-enabled cluster.

`--data-encryption`

Specify this option if the WAL files were encrypted by transparent data encryption.

The `--data-encryption` or `-y` option ensures the command is aware of the encryption. Otherwise, `pg_waldump` can't detect whether WAL files are encrypted.

Provide the same encryption configuration you used when initializing the TDE-enabled database cluster. For example, if you specified an AES key length during the cluster creation, you must specify it here as well. Otherwise, run the flag with no values. See Using initdb TDE options for more information.

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

Use the `--key-file-name=<file>` option to reference the file that contains the data encryption key required to decrypt the WAL file. Provide the location of the `pg_encryption/key.bin` file. This file is generated when you initialize a cluster with encryption enabled.

The command can then load the data encryption key from the provided location.

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

For the `--key-unwrap-command=<command>` option, provide the decryption command you specified to unwrap (decrypt) the data encryption key when initializing the TDE cluster. See Using initdb TDE options for more information.

Alternatively, you can set the `PGDATAKEYUNWRAPCMD` environment variable before running the `pg_waldump` command. If the `--key-unwrap-command=<command>` option isn't specified, `pg_waldump` falls back on `PGDATAKEYUNWRAPCMD`. This cluster initialization example shows how to export an environment variable.

Example

This example uses `pg_waldump` to display the WAL log of an encrypted cluster that uses `openssl` to wrap the data encryption key:

```
pg_waldump --data-encryption --key-file-name=pg_encryption/key.bin --key-unwrap-command='openssl enc -d
-aes-128-cbc -pass pass:<passphrase> -in %p'
```

Resetting a corrupt TDE-encrypted WAL file

To reset a corrupt encrypted WAL file, you must ensure the pg_resetwal command can access the unwrap key and decrypt it. 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>`

For the `--key-unwrap-command=<command>` option, provide the decryption command you specified to unwrap (decrypt) the data encryption key when initializing the TDE cluster. See Using initdb TDE options for more information.

Alternatively, you can set the `PGDATAKEYUNWRAPCMD` environment variable before running the `pg_resetwal` command. If the `--key-unwrap-command=<command>` option isn't specified, `pg_resetwal` falls back on `PGDATAKEYUNWRAPCMD`. This cluster initialization example shows how to export an environment variable.

Example

This example uses `pg_resetwal` to reset a corrupt encrypted WAL log of an encrypted cluster that uses `openssl` to wrap the data encryption key:

```
pg_resetwal --key-unwrap-command='openssl enc -d -aes-128-cbc -pass pass:<passphrase> -in %p'
```

8 Upgrading a TDE-enabled database

You can use the pg_upgrade with additional TDE-arguments to perform a number of upgrading and migrating operations. This table provides an overview of supported use cases.

Use case	Source unencrypted server	Target encrypted server
Perform a minor upgrade and add encryption	Unencrypted EDB Postgres Extended Server 16.1	Encrypted EDB Postgres Extended Server 16.2
Change the Postgres distribution and add encryption	Unencrypted PostgreSQL 16	Encrypted EDB Postgres Advanced Server 16
Maintain the Postgres distribution and add encryption	Unencrypted EDB Postgres Advanced Server 15	Encrypted EDB Postgres Advanced Server 15
Maintain the Postgres distribution and rotate encryption keys	Encrypted EDB Postgres Advanced Server 15	Encrypted EDB Postgres Advanced Server 15 with new encryption keys

> **Important**
>
> Both source and target servers must be in the same Postgres major version. `pg_upgrade` only supports upgrades between minor versions.

Tutorials

See Tutorials for a list of step-by-step tutorials that highlight different migration scenarios.

9 Tutorials

Create a TDE-enabled database server using `initdb`.

Or migrate an existing database instance by creating a TDE-enabled database server with `initdb` and then migrating data with `pg_upgrade`.

9.1 Creating a database with TDE

Create a new EDB Postgres Advanced Server cluster with TDE enabled.

- Set the environment variables to export the `wrap` and `unwrap` commands for encryption.
- Initialize a server with encryption enabled.
- Start the database server.
- Verify TDE is enabled.

Worked example

This example uses EDB Postgres Advanced Server 15 running on a Linux platform. It uses openssl to define the passphrase 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.
   - Ensure you replace `ok` with the passphrase you want to use to wrap the data encryption key.

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 verify the setting of `data_encryption_key_unwrap_command`:

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

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

5. Verify that data encryption is enabled.

9.2 Enabling TDE on an existing EDB Postgres Advanced Server

Create a new EDB Postgres Advanced Server cluster with TDE enabled and use `pg_upgrade` to transfer data from the existing source cluster to the new encrypted cluster.

- Prepare your upgrade by performing a backup of the existing instance.
- Create a new database server
  - Create an empty directory for the new server and ensure `enterprisedb` owns it.
  - Set the environment variables to export the `wrap` and `unwrap` commands for encryption.
  - Initialize a server with encryption enabled.
  - Change the default port, so the new server is available at another port.
  - Start the database server.
  - Connect to the database server and ensure it is functioning.
- Upgrade to the encrypted server
  - Stop both the source and the new server.
  - Use `pg_upgrade` with `--copy-by-block` option to copy data from the source server to the new server. Specify the source and target bin and data directories.
  - Start the new encrypted databaser server.
  - Connect to the encrypted database server and ensure the data was transfered.
- Clean up and delete the source server
  - Clean up the database and its statistics.
  - Remove the source EDB Postgres Advanced Server cluster with the script provided by `pg_upgrade`.

Worked example

This example enables Transparent Data Encryption on an EDB Postgres Advanced Server version 16 running on an Ubuntu 22.04 machine.

A similar workflow applies to other versions of EDB Postgres Advanced Server and EDB Postgres Extended Server. Note that the location of the BIN and CONFIG directories differs depending on your operating system and the Postgres version.

Preparing your upgrade

Use pg_dumpall, pgBackRest, or Barman to create a backup of your unencrypted source server.

Creating an encrypted server

1. Create an empty directory for the new server. In this example, the directory name is `TDE`.

   ```
   mkdir /var/lib/edb-as/16/TDE
   ```

2. Ensure the `enterprisedb` user owns the directory:

   ```
   sudo chown enterprisedb /var/lib/edb-as/16/TDE
   sudo chgrp /var/lib/edb-as/16/TDE
   ```

3. Set environment variables to export the `wrap` and `unwrap` commands:

   ```
   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**
   >
   > Alternatively, use the `--key-unwrap-command=<command>` and `--key-wrap-command=<command>` arguments when initializing the encrypted server to include the `wrap` and `unwrap` commands.
   >
   > See Using initdb TDE options for more information on possible configurations.

4. Initialize the new server with encryption:

   ```
   /usr/lib/edb-as/16/bin/initdb --data-encryption -D /var/lib/edb-as/16/TDE
   ```

   This command initializes a CONFIG directory with all configuration files for the encrypted server.

5. Modify the port number in the configuration file of the encrypted instance. Uncomment the line with `#port` and change the port number. For example:

   ```
   port                    5590
   ```

6. Start the encrypted server:

   ```
   /usr/lib/edb-as/16/bin/pg_ctl -D /var/lib/edb-as/16/TDE -l logfile start
   ```

7. Connect to the server:

   ```
   /usr/lib/edb-as/16/bin/psql -p 5590 edb
   ```

   > **Note**
   >
   > If you're using two different Postgres versions, use the psql utility of the encrypted server. Otherwise, the system will attempt to use psql from the previous instance.

8. To ensure the new server is encrypted, check for TDE presence.

Upgrading to the encrypted server

1. Stop both servers:

   ```
   /usr/lib/edb-as/16/bin/pg_ctl -D /var/lib/edb-as/16/non-TDE stop
   /usr/lib/edb-as/16/bin/pg_ctl -D /var/lib/edb-as/16/TDE stop
   ```

2. To test for incompatibilities, run the `pg_upgrade` command in check mode.

   With `-b` and `-B`, specify the source and target BIN directories. With `-d` and `-D`, specify the source and target CONFIG directories. Include the `--copy-by-block` option.

   ```
   /usr/lib/edb-as/16/bin/pg_upgrade -b /usr/lib/edb-as/16/bin -B /usr/lib/edb-as/16/bin -d
   /var/lib/edb-as/16/non-TDE -D /var/lib/edb-as/16/TDE --copy-by-block --check
   ```

   > **Note**
   >
   > The `--check` mode performs preliminary checks without executing the command.

3. To copy data from the source server to the target server, run the `pg_upgrade` command in normal mode:

   ```
   /usr/lib/edb-as/16/bin/pg_upgrade -b /usr/lib/edb-as/16/bin -B /usr/lib/edb-as/16/bin -d
   /var/lib/edb-as/16/non-TDE -D /var/lib/edb-as/16/TDE --copy-by-block
   ```

4. Restart the encrypted server:

   ```
   /usr/lib/edb-as/16/bin/pg_ctl -D /var/lib/edb-as/16/TDE start
   ```

5. Connect to the encrypted database server:

   ```
   /usr/lib/edb-as/16/bin/psql -p 5590 edb
   ```

6. Perform a spot check to ensure the databases, tables, schemas, and resources you had in the unencrypted server are available in the new server. For example, list all databases:

   ```
   \l
   ```

   Connect to a database, for example `hr`, and search for existing tables:

   ```
   \c hr
   SELECT * FROM dept;
   ```

Cleaning up after upgrade

After you verify that `pg_upgrade` encrypted the data successfully, perform a cleanup.

1. Clean up the database and its statistics:

   ```
   /usr/lib/edb-as/16/bin/vacuumdb --all --analyze-in-stages
   ```

2. Remove all data files of the unencrypted server with the script generated by `pg_upgrade`:

   ```
   ./delete_old_cluster.sh
   ```

9.3 Upgrading PostgreSQL to EDB Postgres Extended Server while enabling TDE

Create a new EDB Postgres Extended Server cluster with TDE enabled and use pg_upgrade to transfer data from the existing PostgreSQL cluster to the new encrypted cluster.

- Prepare your upgrade by performing a backup of the existing instance.
- Create a new database server.
  - Create an empty directory for the new server and ensure the postgres user owns it.
  - Set the environment variables to export the `wrap` and `unwrap` commands for encryption.
  - Initialize a server with encryption enabled.
  - Change the default port so the new server is available at another port.
  - Start the database server.
  - Connect to the database server and ensure it's functioning.
- Upgrade to the encrypted server.
  - Stop both the source and the new server.
  - Use pg_upgrade with the `--copy-by-block` option to copy data from the source server to the new server. Specify the source and target bin and data directories.
  - Start the new encrypted database server.
  - Connect to the encrypted database server and ensure the data was transferred.
- Clean up and delete the source server.
  - Clean up the database and its statistics.
  - Remove the source PostgreSQL cluster with the script provided by pg_upgrade.

Worked example

This example upgrades a PostgreSQL 16 instance to EDB Postgres Extended Server 16 while enabling Transparent Data Encryption on an Ubuntu 22.04 machine. The location of the bin and config directories differs depending on your operating system and Postgres versions.

Preparing your upgrade

- Install EDB Postgres Extended Server from the EDB repository. Ensure the version you install has the same major version as the source server. pg_upgrade supports upgrades between minor and patch versions but not between different major versions.
- Use pg_dumpall, pgBackRest, or Barman to create a backup of your unencrypted source server.

Creating an encrypted server

1. Create an empty directory for the new server. In this example, the directory name is `TDE`.

   ```
   mkdir /var/lib/edb-pge/16/TDE
   ```

2. Ensure the postgres user owns the directory:

   ```
    sudo chown postgres /var/lib/edb-pge/16/TDE
   sudo chgrp /var/lib/edb-pge/16/TDE
   ```

3. Set environment variables to export the `wrap` and `unwrap` commands:

   ```
   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**
   >
   > Alternatively, use the `--key-unwrap-command=<command>` and `--key-wrap-command=<command>` arguments when initializing the encrypted server to include the `wrap` and `unwrap` commands.
   >
   > See Using initdb TDE options for more information on possible configurations.

4. Initialize the new server with encryption:

   ```
   /usr/lib/edb-pge/16/bin/initdb --data-encryption -D /var/lib/edb-pge/16/TDE
   ```

   This command initializes a config directory with all configuration files for the encrypted server.

5. Modify the default port number in the configuration file of the encrypted instance. Uncomment the line with `#port` and change the port number. For example:

   ```
   port                    5590
   ```

6. Start the encrypted server:

   ```
   /usr/lib/edb-pge/16/bin/pg_ctl -D /var/lib/edb-pge/16/TDE start
   ```

7. Connect to the server:

   ```
   /usr/lib/edb-pge/16/bin/psql -p 5590
   ```

   > **Note**
   >
   > If you're using two different Postgres versions, use the psql utility of the encrypted server. Otherwise, the system attempts to use psql from the previous instance.

8. To ensure the new server is encrypted, check for TDE presence.

Upgrading to the encrypted server

1. Stop both servers:

   ```
   /usr/lib/postgresql/16/bin/pg_ctl -D /var/lib/postgresql/16/non-TDE stop
   /usr/lib/edb-pge/16/bin/pg_ctl -D /var/lib/edb-pge/16/TDE stop
   ```

2. To test for incompatibilities, run the `pg_upgrade` command in check mode.

   With `-b` and `-B`, specify the source and target BIN directories. With `-d` and `-D`, specify the source and target config directories. Include the `--copy-by-block` option.

   ```
   /usr/lib/edb-pge/16/bin/pg_upgrade -b /usr/lib/postgresql/16/bin -B /usr/lib/edb-pge/16/bin -d
   /var/lib/postgresql/16/non-TDE -D /var/lib/edb-pge/16/TDE --copy-by-block --check
   ```

   > **Note**
   >
   > The `--check` mode performs preliminary checks without executing the command.

3. To copy data from the source server to the target server, run the `pg_upgrade` command in normal mode:

   ```
   /usr/lib/edb-pge/16/bin/pg_upgrade -b /usr/lib/postgresql/16/bin -B /usr/lib/edb-pge/16/bin -d
   /var/lib/postgresql/16/non-TDE -D /var/lib/edb-pge/16/TDE --copy-by-block
   ```

4. Restart the encrypted server:

   ```
   /usr/lib/edb-pge/16/bin/pg_ctl -D /var/lib/edb-pge/16/TDE start
   ```

5. Connect to the encrypted database server:

   ```
   /usr/lib/edb-pge/16/bin/psql -p 5590
   ```

6. Perform a spot check to ensure the databases, tables, schemas, and resources you had in the unencrypted server are available in the new server. For example, list all databases:

   ```
   \l
   ```

   Connect to a database, for example `hr`, and search for existing tables:

   ```
   \c hr
   SELECT * FROM dept;
   ```

Cleaning up after upgrade

After you verify that pg_upgrade encrypted the data successfully, perform a cleanup.

1. As the postgres user, clean up the database and its statistics:

   ```
   /usr/lib/edb-pge/16/bin/vacuumdb --all --analyze-in-stages
   ```

2. Remove all data files of the unencrypted server with the script generated by pg_upgrade:

   ```
   ./delete_old_cluster.sh
   ```

9.4 Verifying TDE is enabled

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();
```

```
output
 data_encryption_version
-------------------------
                       1
(1 row)
```

10 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:

- initdb
- pg_waldump
- pg_resetwal
- pg_verifybackup
- pg_rewind
- pg_upgrade

11 initdb TDE options

To enable encryption, use the following options with the `initdb` command.

Option: `--data-encryption` or `-y`

Adds transparent data encryption when initializing a database server.

Supported values

You can optionally specify an AES key length. Valid values are 128 and 256. The default is 128.

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

Provides the wrapping/encryption command to protect the data encryption key.

Supported values

`<command>` is customizable, but it must contain the placeholder `%p` . See Wrapping commands for examples and information. See Securing the data encryption key for an overview of available wrapping mechanisms.

If not specified, TDE falls back on the environment variable `PGDATAKEYWRAPCMD`.

Use `-` if you don't want to apply a wrapping mechanism.

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

Provides the unwrapping/decryption command to access the data encryption key.

Supported values

`<command>` is customizable, but it must contain the placeholder `%p` . See Configuring wrapping commands for examples and information.

If not specified, TDE falls back on the environment variable `PGDATAKEYUNWRAPCMD`.

Use `-` if you didn't apply a wrapping mechanism.

Option: `--no-key-wrap`

Disables the 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 a key wrap and unwrap command. Otherwise, `initdb` terminates with an error.

Supported values

The `--no-key-wrap` option doesn't require specifying any values.

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

Copies an existing data encryption key from the provided location. For example, when reusing a key from an existing server during an upgrade with pg_upgrade.

Supported values

`<file>` is the directory to the existing key. Normally, encryption keys are stored in `pg_encryption/key.bin`.

12 pg_upgrade TDE options

These arguments to pg_upgrade help with upgrading encrypted clusters.

Option: `--copy-by-block`

Copies 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 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.

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` .

See Tutorials for `--copy-by-block` usage examples.

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

Specifies the command to unwrap (decrypt) the data encryption key and access the files to copy. It must be the same unwrap command you specified during the server initialization.

If you don't specify this option, pg_upgrade reads the environment variable `PGDATAKEYUNWRAPCMD` .

13 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.