EDB Postgres™ Advanced Server

Release 13

Security Features Guide

Oct 20, 2020

Contents

CHAPTER 1

Introduction

This guide describes features that provide added security to EDB Postgres Advanced Server installations. It is not a comprehensive guide to the security functionality provided by PostgreSQL that is built into Advanced Server.

- *SQL/Protect* provides protection from SQL injection attacks.
- *Virtual Private Database* provides fine-grained access control for sensitive data.
- *sslutils* is a Postgres extension that allows you to generate SSL certificates.
- *Data redaction* functionality allows you to dynamically mask portions of data.

For information about Postgres authentication and security features, consult the PostgreSQL core documentation, available at:

https://www.enterprisedb.com/edb-docs/p/postgresql

CHAPTER 2

Protecting Against SQL Injection Attacks

Advanced Server provides protection against SQL injection attacks. A *SQL injection attack* is an attempt to compromise a database by running SQL statements whose results provide clues to the attacker as to the content, structure, or security of that database.

Preventing a SQL injection attack is normally the responsibility of the application developer. The database administrator typically has little or no control over the potential threat. The difficulty for database administrators is that the application must have access to the data to function properly.

SQL/Protect is a module that allows a database administrator to protect a database from SQL injection attacks. SQL/Protect provides a layer of security in addition to the normal database security policies by examining incoming queries for common SQL injection profiles.

SQL/Protect gives the control back to the database administrator by alerting the administrator to potentially dangerous queries and by blocking these queries.

2.1 SQL/Protect Overview

This section contains an introduction to the different types of SQL injection attacks and describes how SQL/Protect guards against them.

2.1.1 Types of SQL Injection Attacks

There are a number of different techniques used to perpetrate SQL injection attacks. Each technique is characterized by a certain `signature`. SQL/Protect examines queries for the following signatures:

Unauthorized Relations

While Advanced Server allows administrators to restrict access to relations (tables, views, etc.), many administrators do not perform this tedious task. SQL/Protect provides a `learn` mode that tracks the relations a user accesses.

This allows administrators to examine the workload of an application, and for SQL/Protect to learn which relations an application should be allowed to access for a given user or group of users in a role.

When SQL/Protect is switched to either `passive` or `active` mode, the incoming queries are checked against the list of learned relations.

Utility Commands

A common technique used in SQL injection attacks is to run utility commands, which are typically SQL Data Definition Language (DDL) statements. An example is creating a user-defined function that has the ability to access other system resources.

SQL/Protect can block the running of all utility commands, which are not normally needed during standard application processing.

SQL Tautology

The most frequent technique used in SQL injection attacks is issuing a tautological `WHERE` clause condition (that is, using a condition that is always true).

The following is an example:

```
WHERE password = 'x' OR 'x'='x'
```

Attackers will usually start identifying security weaknesses using this technique. SQL/Protect can block queries that use a tautological conditional clause.

Unbounded DML Statements

A dangerous action taken during SQL injection attacks is the running of unbounded DML statements. These are `UPDATE` and `DELETE` statements with no `WHERE` clause. For example, an attacker may update all users' passwords to a known value or initiate a denial of service attack by deleting all of the data in a key table.

2.1.2 Monitoring SQL Injection Attacks

This section describes how SQL/Protect monitors and reports on SQL injection attacks.

2.1.2.1 Protected Roles

Monitoring for SQL injection attacks involves analyzing SQL statements originating in database sessions where the current user of the session is a protected role. A protected role is an Advanced Server user or group that the database administrator has chosen to monitor using SQL/Protect. (In Advanced Server, users and groups are collectively referred to as roles.)

Each protected role can be customized for the types of SQL injection attacks for which it is to be monitored, thus providing different levels of protection by role and significantly reducing the user maintenance load for DBAs.

A role with the superuser privilege cannot be made a protected role. If a protected non-superuser role is subsequently altered to become a superuser, certain behaviors are exhibited whenever an attempt is made by that superuser to issue any command:

- A warning message is issued by SQL/Protect on every command issued by the protected superuser.
- The statistic in column superusers of edb_sql_protect_stats is incremented with every command issued by the protected superuser. See Attack Attempt Statistics for information on the edb_sql_protect_stats view.
- When SQL/Protect is in active mode, all commands issued by the protected superuser are prevented from running.

A protected role that has the superuser privilege should either be altered so that it is no longer a superuser, or it should be reverted back to an unprotected role.

2.1.2.2 Attack Attempt Statistics

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded. Statistics are collected by type of SQL injection attack as discussed in Types of SQL Injection Attacks.

These statistics are accessible from view edb_sql_protect_stats that can be easily monitored to identify the start of a potential attack.

The columns in edb_sql_protect_stats monitor the following:

- **username.** Name of the protected role.
- **superusers.** Number of SQL statements issued when the protected role is a superuser. In effect, any SQL statement issued by a protected superuser increases this statistic. See Protected Roles for information on protected superusers.
- **relations.** Number of SQL statements issued referencing relations that were not learned by a protected role. (That is, relations that are not in a role's protected relations list.)
- **commands.** Number of DDL statements issued by a protected role.

- **tautology.** Number of SQL statements issued by a protected role that contained a tautological condition.
- **dml.** Number of `UPDATE` and `DELETE` statements issued by a protected role that did not contain a `WHERE` clause.

This gives database administrators the opportunity to react proactively in preventing theft of valuable data or other malicious actions.

If a role is protected in more than one database, the role's statistics for attacks in each database are maintained separately and are viewable only when connected to the respective database.

Note: SQL/Protect statistics are maintained in memory while the database server is running. When the database server is shut down, the statistics are saved to a binary file named `edb_sqlprotect.stat` in the `data/global` subdirectory of the Advanced Server home directory.

2.1.2.3 Attack Attempt Queries

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded in the `edb_sql_protect_queries` view.

The `edb_sql_protect_queries` view contains the following columns:

- **username.** Database user name of the attacker used to log into the database server.
- **ip_address.** IP address of the machine from which the attack was initiated.
- **port.** Port number from which the attack originated.
- **machine_name.** Name of the machine, if known, from which the attack originated.
- **date_time.** Date and time at which the query was received by the database server. The time is stored to the precision of a minute.
- **query.** The query string sent by the attacker.

The maximum number of offending queries that are saved in `edb_sql_protect_queries` is controlled by the `edb_sql_protect.max_queries_to_save` configuration parameter.

If a role is protected in more than one database, the role's queries for attacks in each database are maintained separately and are viewable only when connected to the respective database.

2.2 Configuring SQL/Protect

Ensure the following prerequisites are met before configuring SQL/Protect:

- The library file (`sqlprotect.so` on Linux, `sqlprotect.dll` on Windows) necessary to run `SQL/Protect` should be installed in the `lib` subdirectory of your Advanced Server home directory. For Windows, this should be done by the Advanced Server installer. For Linux, install the `edb-asxx-server-sqlprotect` RPM package where `xx` is the Advanced Server version number.
- You will also need the SQL script file `sqlprotect.sql` located in the `share/contrib` subdirectory of your Advanced Server home directory.
- You must configure the database server to use `SQL/Protect`, and you must configure each database that you want `SQL/Protect` to monitor:
  - The database server configuration file, `postgresql.conf`, must be modified by adding and enabling configuration parameters used by `SQL/Protect`.
  - Database objects used by `SQL/Protect` must be installed in each database that you want `SQL/Protect` to monitor.

Step 1: Edit the following configuration parameters in the `postgresql.conf` file located in the `data` subdirectory of your Advanced Server home directory.

- **shared_preload_libraries.** Add `$libdir/sqlprotect` to the list of libraries.
- **edb_sql_protect.enabled.** Controls whether or not `SQL/Protect` is actively monitoring protected roles by analyzing SQL statements issued by those roles and reacting according to the setting of `edb_sql_protect.level`. When you are ready to begin monitoring with `SQL/Protect` set this parameter to `on`. If this parameter is omitted, the default is `off`.
- **edb_sql_protect.level.** Sets the action taken by `SQL/Protect` when a SQL statement is issued by a protected role. If this parameter is omitted, the default behavior is `passive`. Initially, set this parameter to `learn`.

  See *Setting the Protection Level* for more information.
- **edb_sql_protect.max_protected_roles.** Sets the maximum number of roles that can be protected. If this parameter is omitted, the default setting is `64`.
- **edb_sql_protect.max_protected_relations.** Sets the maximum number of relations that can be protected per role. If this parameter is omitted, the default setting is `1024`.

  Please note that the total number of protected relations for the server will be the number of protected relations times the number of protected roles. Every protected relation consumes space in shared memory. The space for the maximum possible protected relations is reserved during database server startup.
- **edb_sql_protect.max_queries_to_save.** Sets the maximum number of offending queries to save in the `edb_sql_protect_queries` view. If this parameter is omitted, the default setting is `5000`. If the number of offending queries reaches the limit, additional queries are not saved in the view, but are accessible in the database server log file.

Please note that the minimum valid value for this parameter is `100`. If a value less than `100` is specified, the database server starts using the default setting of `5000`. A warning message is recorded in the database server log file.

The following example shows the settings of these parameters in the `postgresql.conf` file:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/
sqlprotect'
                                        # (change requires restart)
                                      .
                                      .
                                      .
edb_sql_protect.enabled = off
edb_sql_protect.level = learn
edb_sql_protect.max_protected_roles = 64
edb_sql_protect.max_protected_relations = 1024
edb_sql_protect.max_queries_to_save = 5000
```

Step 2: Restart the database server after you have modified the `postgresql.conf` file.

On Linux: Invoke the Advanced Server service script with the `restart` option.

On a Redhat or CentOS 6.x installation, use the command:

```
/etc/init.d/edb-as-13 restart
```

On a Redhat or CentOS 7.x installation, use the command:

```
systemctl restart edb-as-13
```

On Windows: Use the Windows Services applet to restart the service named `edb-as-13`.

Step 3: For each database that you want to protect from SQL injection attacks, connect to the database as a superuser (either `enterprisedb` or `postgres`, depending upon your installation options) and run the script `sqlprotect.sql` located in the `share/contrib` subdirectory of your Advanced Server home directory. The script creates the SQL/Protect database objects in a schema named `sqlprotect`.

The following example shows this process to set up protection for a database named `edb`:

```
$ /usr/edb/as13/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (13.0.0, server 13.0.0)
Type "help" for help.

edb=# \i /usr/edb/as13/share/contrib/sqlprotect.sql
CREATE SCHEMA
GRANT
SET
CREATE TABLE
GRANT
CREATE TABLE
GRANT
CREATE FUNCTION
```

(continues on next page)

(continued from previous page)

```
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
DO
CREATE FUNCTION
CREATE FUNCTION
DO
CREATE VIEW
GRANT
DO
CREATE VIEW
GRANT
CREATE VIEW
GRANT
CREATE FUNCTION
CREATE FUNCTION
SET
```

2.2.1 Selecting Roles to Protect

After the SQL/Protect database objects have been created in a database, you can select the roles for which SQL queries are to be monitored for protection, and the level of protection that will be assigned to each role.

2.2.1.1 Setting the Protected Roles List

For each database that you want to protect, you must determine the roles you want to monitor and then add those roles to the *protected roles list* of that database.

Step 1: Connect as a superuser to a database that you wish to protect with either `psql` or Postgres Enterprise Manager Client:

```
$ /usr/edb/as13/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (13.0.0, server 13.0.0)
Type "help" for help.

edb=#
```

Step 2: Since the SQL/Protect tables, functions, and views are built under the `sqlprotect` schema, use the `SET search_path` command to include the `sqlprotect` schema in your search path. This eliminates the need to schema-qualify any operation or query involving SQL/Protect database objects:

```
edb=# SET search_path TO sqlprotect;
SET
```

Step 3: Each role that you wish to protect must be added to the protected roles list. This list is maintained in the table `edb_sql_protect`.

To add a role, use the function `protect_role('rolename')`. The following example protects a role named `appuser`:

```
edb=# SELECT protect_role('appuser');
 protect_role
--------------

(1 row)
```

You can list the roles that have been added to the protected roles list by issuing the following query:

```
edb=# SELECT * FROM edb_sql_protect;
 dbid  | roleid | protect_relations | allow_utility_cmds | allow_tautology |
 allow_empty_dml
-------+--------+-------------------+--------------------+-----------------+--
-------------
 13917 |  16671 | t                 | f                  | f               | f
(1 row)
```

A view is also provided that gives the same information using the object names instead of the Object Identification numbers (OIDs):

```
edb=# \x
Expanded display is on.
edb=# SELECT * FROM list_protected_users;
-[ RECORD 1 ]------+--------
dbname             | edb
username           | appuser
protect_relations  | t
allow_utility_cmds | f
allow_tautology    | f
allow_empty_dml    | f
```

2.2.1.2 Setting the Protection Level

The `edb_sql_protect.level` configuration parameter sets the protection level, which defines the behavior of SQL/Protect when a protected role issues a SQL statement. The defined behavior applies to all roles in the protected roles lists of all databases configured with SQL/Protect in the database server.

The `edb_sql_protect.level` configuration parameter (in the `postgresql.conf` file) can be set to one of the following values to use either `learn` mode, `passive` mode, or `active` mode:

- **learn.** Tracks the activities of protected roles and records the relations used by the roles. This is used when initially configuring SQL/Protect so the expected behaviors of the protected applications are learned.
- **passive.** Issues warnings if protected roles are breaking the defined rules, but does not stop any SQL statements from executing. This is the next step after SQL/Protect has learned the expected

behavior of the protected roles. This essentially behaves in intrusion detection mode and can be run in production when properly monitored.

- **active.** Stops all invalid statements for a protected role. This behaves as a SQL firewall preventing dangerous queries from running. This is particularly effective against early penetration testing when the attacker is trying to determine the vulnerability point and the type of database behind the application. Not only does SQL/Protect close those vulnerability points, but it tracks the blocked queries allowing administrators to be alerted before the attacker finds an alternate method of penetrating the system.

If the `edb_sql_protect.level` parameter is not set or is omitted from the configuration file, the default behavior of `SQL/Protect` is `passive`.

If you are using `SQL/Protect` for the first time, set `edb_sql_protect.level` to `learn`.

2.2.2 Monitoring Protected Roles

Once you have configured SQL/Protect in a database, added roles to the protected roles list, and set the desired protection level, you can then activate SQL/Protect in either `learn` mode, `passive` mode, or `active` mode. You can then start running your applications.

With a new SQL/Protect installation, the first step is to determine the relations that protected roles should be permitted to access during normal operation. Learn mode allows a role to run applications during which time SQL/Protect is recording the relations that are accessed. These are added to the role's `protected relations list` stored in table `edb_sql_protect_rel`.

Monitoring for protection against attack begins when SQL/Protect is run in passive or active mode. In passive and active modes, the role is permitted to access the relations in its protected relations list as these were determined to be the relations the role should be able to access during typical usage.

However, if a role attempts to access a relation that is not in its protected relations list, a `WARNING` or `ERROR` severity level message is returned by SQL/Protect. The role's attempted action on the relation may or may not be carried out depending upon whether the mode is passive or active.

2.2.2.1 Learn Mode

Step 1: To activate SQL/Protect in learn mode, set the parameters in the `postgresql.conf` file as shown below:

```
edb_sql_protect.enabled = on
edb_sql_protect.level = learn
```

Step 2: Reload the `postgresql.conf` file.

Choose `Expert Configuration`, then `Reload Configuration` from the Advanced Server application menu.

For an alternative method of reloading the configuration file, use the `pg_reload_conf` function. Be sure you are connected to a database as a superuser and execute `function pg_reload_conf` as shown by the following example:

```
edb=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Step 3: Allow the protected roles to run their applications.

As an example the following queries are issued in the `psql` application by protected role `appuser`:

```
edb=> SELECT * FROM dept;
NOTICE:  SQLPROTECT: Learned relation: 16384
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)

edb=> SELECT empno, ename, job FROM emp WHERE deptno = 10;
NOTICE:  SQLPROTECT: Learned relation: 16391
 empno | ename  |    job
-------+--------+-----------
  7782 | CLARK  | MANAGER
  7839 | KING   | PRESIDENT
  7934 | MILLER | CLERK
(3 rows)
```

SQL/Protect generates a `NOTICE` severity level message indicating the relation has been added to the role's protected relations list.

In SQL/Protect learn mode, SQL statements that are cause for suspicion are not prevented from executing, but a message is issued to alert the user to potentially dangerous statements as shown by the following example:

```
edb=> CREATE TABLE appuser_tab (f1 INTEGER);
NOTICE: SQLPROTECT: This command type is illegal for this user
CREATE TABLE
edb=> DELETE FROM appuser_tab;
NOTICE: SQLPROTECT: Learned relation: 16672
NOTICE: SQLPROTECT: Illegal Query: empty DML
DELETE 0
```

Step 4: As a protected role runs applications, the SQL/Protect tables can be queried to observe the addition of relations to the role's protected relations list.

Connect as a superuser to the database you are monitoring and set the search path to include the `sqlprotect` schema:

```
edb=# SET search_path TO sqlprotect;
SET
```

Query the edb_sql_protect_rel table to see the relations added to the protected relations list:

```
edb=# SELECT * FROM edb_sql_protect_rel;
 dbid   | roleid | relid
--------+--------+-------
 13917  | 16671  | 16384
 13917  | 16671  | 16391
 13917  | 16671  | 16672
(3 rows)
```

The list_protected_rels view provides more comprehensive information along with the object names instead of the OIDs:

```
edb=# SELECT * FROM list_protected_rels;
 Database | Protected User | Schema |    Name     | Type  |    Owner
----------+----------------+--------+-------------+-------+------------
 edb      | appuser        | public | dept        | Table | enterprisedb
 edb      | appuser        | public | emp         | Table | enterprisedb
 edb      | appuser        | public | appuser_tab | Table | appuser
(3 rows)
```

2.2.2.2 Passive Mode

Once you have determined that a role's applications have accessed all relations they will need, you can now change the protection level so that SQL/Protect can actively monitor the incoming SQL queries and protect against SQL injection attacks.

Passive mode is the less restrictive of the two protection modes, passive and active.

Step 1: To activate SQL/Protect in passive mode, set the following parameters in the postgresql.conf file as shown below:

```
edb_sql_protect.enabled = on
edb_sql_protect.level = passive
```

Step 2: Reload the configuration file as shown in Step 2 of the *Learn Mode* section.

Now SQL/Protect is in passive mode. For relations that have been learned such as the dept and emp tables of the prior examples, SQL statements are permitted with no special notification to the client by SQL/Protect as shown by the following queries run by user appuser:

```
edb=> SELECT * FROM dept;
 deptno |   dname    |   loc
--------+------------+----------
    10  | ACCOUNTING | NEW YORK
    20  | RESEARCH   | DALLAS
    30  | SALES      | CHICAGO
    40  | OPERATIONS | BOSTON
(4 rows)

edb=> SELECT empno, ename, job FROM emp WHERE deptno = 10;
```

(continues on next page)

(continued from previous page)

```
 empno | ename  |    job
-------+--------+-----------
  7782 | CLARK  | MANAGER
  7839 | KING   | PRESIDENT
  7934 | MILLER | CLERK
(3 rows)
```

SQL/Protect does not prevent any SQL statement from executing, but issues a message of `WARNING` severity level for SQL statements executed against relations that were not learned, or for SQL statements that contain a prohibited signature as shown in the following example:

```
edb=> CREATE TABLE appuser_tab_2 (f1 INTEGER);
WARNING: SQLPROTECT: This command type is illegal for this user
CREATE TABLE
edb=> INSERT INTO appuser_tab_2 VALUES (1);
WARNING: SQLPROTECT: Illegal Query: relations
INSERT 0 1
edb=> INSERT INTO appuser_tab_2 VALUES (2);
WARNING: SQLPROTECT: Illegal Query: relations
INSERT 0 1
edb=> SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
WARNING: SQLPROTECT: Illegal Query: relations
WARNING: SQLPROTECT: Illegal Query: tautology
f1
----
  1
  2
(2 rows)
```

Step 3: Monitor the statistics for suspicious activity.

By querying the view `edb_sql_protect_stats`, you can see the number of times SQL statements were executed that referenced relations that were not in a role's protected relations list, or contained SQL injection attack signatures. See `Attack Attempt Statistics` for more information on view `edb_sql_protect_stats`.

The following is a query on `edb_sql_protect_stats`:

```
edb=# SET search_path TO sqlprotect;
SET
edb=# SELECT * FROM edb_sql_protect_stats;
 username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
 appuser   |     0      |     3     |    1     |     1     | 0
(1 row)
```

Step 4: View information on specific attacks.

By querying the `edb_sql_protect_queries` view, you can see the SQL statements that were executed that referenced relations that were not in a role's protected relations list, or contained SQL injection attack signatures. See `Attack Attempt Queries` for more information on view `edb_sql_protect_queries`.

The following code sample shows a query on edb_sql_protect_queries:

```
edb=# SELECT * FROM edb_sql_protect_queries;
-[ RECORD 1 ]+--------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 2 ]+--------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 3 ]+--------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 4 ]+--------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

Note: The ip_address and port columns do not return any information if the attack originated on the same host as the database server using the Unix-domain socket (that is, pg_hba.conf connection type local).

2.2.2.3 Active Mode

In active mode, disallowed SQL statements are prevented from executing. Also, the message issued by SQL/Protect has a higher severity level of ERROR instead of WARNING.

Step 1: To activate SQL/Protect in active mode, set the following parameters in the postgresql.conf file as shown below:

```
edb_sql_protect.enabled = on
edb_sql_protect.level = active
```

Step 2: Reload the configuration file as shown in Step 2 of the *Learn Mode* section.

The following example illustrates SQL statements similar to those given in the examples of Step 2 in

Passive Mode, but executed by user appuser when edb_sql_protect.level is set to active:

```
edb=> CREATE TABLE appuser_tab_3 (f1 INTEGER);
ERROR: SQLPROTECT: This command type is illegal for this user
edb=> INSERT INTO appuser_tab_2 VALUES (1);
ERROR: SQLPROTECT: Illegal Query: relations
edb=> SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
ERROR: SQLPROTECT: Illegal Query: relations
```

The following shows the resulting statistics:

```
edb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
 appuser   |      0     |     5     |     2    |     1     | 0
(1 row)
```

The following is a query on edb_sql_protect_queries:

```
edb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
-[ RECORD 1 ]+-------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 2 ]+-------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 3 ]+-------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | CREATE TABLE appuser_tab_3 (f1 INTEGER);
-[ RECORD 4 ]+-------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 5 ]+-------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
```

(continues on next page)

(continued from previous page)

```
date_time     | 20-JUN-14 13:39:00 -04:00
query         | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

2.3 Common Maintenance Operations

The following describes how to perform other common operations.

You must be connected as a superuser to perform these operations and have included the `sqlprotect` schema in your search path.

2.3.1 Adding a Role to the Protected Roles List

To add a role to the protected roles list run `protect_role('rolename')` as shown in the following example:

```
edb=# SELECT protect_role('newuser');
 protect_role
--------------
(1 row)
```

2.3.2 Removing a Role From the Protected Roles List

To remove a role from the protected roles list, use either of the following functions:

```
unprotect_role('rolename')
```

```
unprotect_role(roleoid)
```

The variation of the function using the `OID` is useful if you remove the role using the `DROP ROLE` or `DROP USER` SQL statement before removing the role from the protected roles list. If a query on a SQL/Protect relation returns a value such as `unknown (OID=16458)` for the user name, use the `unprotect_role(roleoid)` form of the function to remove the entry for the deleted role from the protected roles list.

Removing a role using these functions also removes the role's protected relations list.

The statistics for a role that has been removed are not deleted until you use the *drop_stats function*.

The offending queries for a role that has been removed are not deleted until you use the *drop_queries function*.

The following is an example of the `unprotect_role` function:

```
edb=# SELECT unprotect_role('newuser');
 unprotect_role
----------------
(1 row)
```

Alternatively, the role could be removed by giving its OID of `16693`:

```
edb=# SELECT unprotect_role(16693);
 unprotect_role
----------------
(1 row)
```

2.3.3 Setting the Types of Protection for a Role

You can change whether or not a role is protected from a certain type of SQL injection attack.

Change the Boolean value for the column in `edb_sql_protect` corresponding to the type of SQL injection attack for which protection of a role is to be disabled or enabled.

Be sure to qualify the following columns in your `WHERE` clause of the statement that updates `edb_sql_protect`:

- **dbid.** OID of the database for which you are making the change
- **roleid.** OID of the role for which you are changing the Boolean settings

For example, to allow a given role to issue utility commands, update the `allow_utility_cmds` column as follows:

```
UPDATE edb_sql_protect SET allow_utility_cmds = TRUE WHERE dbid =
13917 AND roleid = 16671;
```

You can verify the change was made by querying `edb_sql_protect` or `list_protected_users`. In the following query note that column `allow_utility_cmds` now contains `t`:

```
edb=# SELECT dbid, roleid, allow_utility_cmds FROM edb_sql_protect;
 dbid   | roleid | allow_utility_cmds
--------+--------+--------------------
 13917  | 16671  | t
(1 row)
```

The updated rules take effect on new sessions started by the role since the change was made.

2.3.4 Removing a Relation From the Protected Relations List

If SQL/Protect has learned that a given relation is accessible for a given role, you can subsequently remove that relation from the role's protected relations list.

Delete its entry from the `edb_sql_protect_rel` table using any of the following functions:

```
unprotect_rel('rolename', 'relname')  unprotect_rel('rolename',
'schema', 'relname') unprotect_rel(roleoid, reloid)
```

If the relation given by `relname` is not in your current search path, specify the relation's schema using the second function format.

The third function format allows you to specify the OIDs of the role and relation, respectively, instead of their text names.

The following example illustrates the removal of the `public.emp` relation from the protected relations list of the role `appuser`:

```
edb=# SELECT unprotect_rel('appuser', 'public', 'emp');
 unprotect_rel
---------------
(1 row)
```

The following query shows there is no longer an entry for the emp relation:

```
edb=# SELECT * FROM list_protected_rels;
 Database | Protected User | Schema |    Name     | Type  |    Owner
----------+----------------+--------+-------------+-------+-------------
 edb      | appuser        | public | dept        | Table | enterprisedb
 edb      | appuser        | public | appuser_tab | Table | appuser
(2 rows)
```

SQL/Protect will now issue a warning or completely block access (depending upon the setting of edb_sql_protect.level) whenever the role attempts to utilize that relation.

2.3.5 Deleting Statistics

You can delete statistics from view edb_sql_protect_stats using either of the two following functions:

```
drop_stats('rolename')
```

```
drop_stats(roleoid)
```

The variation of the function using the OID is useful if you remove the role using the DROP ROLE or DROP USER SQL statement before deleting the role's statistics using drop_stats('rolename'). If a query on edb_sql_protect_stats returns a value such as unknown (OID=16458) for the user name, use the drop_stats(roleoid) form of the function to remove the deleted role's statistics from edb_sql_protect_stats.

The following is an example of the drop_stats function:

```
edb=# SELECT drop_stats('appuser');
 drop_stats
------------
(1 row)

edb=# SELECT * FROM edb_sql_protect_stats;
 username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
(0 rows)
```

The following is an example of using the drop_stats(roleoid) form of the function when a role is dropped before deleting its statistics:

```
edb=# SELECT * FROM edb_sql_protect_stats;
      username       | superusers | relations | commands | tautology | dml
---------------------+------------+-----------+----------+-----------+-----
 unknown (OID=16693) |          0 |         5 |        3 |         1 |   0
 appuser             |          0 |         5 |        2 |         1 |   0
(2 rows)

edb=# SELECT drop_stats(16693);
 drop_stats
------------
```

(continues on next page)

(continued from previous page)

```
(1 row)

edb=# SELECT * FROM edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
 appuser  |          0 |         5 |        2 |         1 |   0
(1 row)
```

2.3.6 Deleting Offending Queries

You can delete offending queries from `view edb_sql_protect_queries` using either of the two following functions:

```
drop_queries('rolename')

drop_queries(roleoid)
```

The variation of the function using the OID is useful if you remove the role using the `DROP ROLE` or `DROP USER` SQL statement before deleting the role's offending queries using `drop_queries('rolename')`. If a query on `edb_sql_protect_queries` returns a value such as `unknown (OID=16454)` for the user name, use the `drop_queries(roleoid)` form of the function to remove the deleted role's offending queries from `edb_sql_protect_queries`.

The following is an example of the `drop_queries` function:

```
edb=# SELECT drop_queries('appuser');
 drop_queries
--------------
            5
(1 row)

edb=# SELECT * FROM edb_sql_protect_queries;
 username  | ip_address | port | machine_name | date_time | query
-----------+------------+------+--------------+-----------+-------
(0 rows)
```

The following is an example of using the `drop_queries(roleoid)` form of the function when a role is dropped before deleting its queries:

```
edb=# SELECT username, query FROM edb_sql_protect_queries;
      username       |                    query
---------------------+-----------------------------------------------
 unknown (OID=16454) | CREATE TABLE appuser_tab_2 (f1 INTEGER);
 unknown (OID=16454) | INSERT INTO appuser_tab_2 VALUES (2);
 unknown (OID=16454) | CREATE TABLE appuser_tab_3 (f1 INTEGER);
 unknown (OID=16454) | INSERT INTO appuser_tab_2 VALUES (1);
 unknown (OID=16454) | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
(5 rows)
```

(continues on next page)

(continued from previous page)

```
edb=# SELECT drop_queries(16454);
 drop_queries
--------------
            5
(1 row)

edb=# SELECT * FROM edb_sql_protect_queries;
 username | ip_address | port | machine_name | date_time | query
----------+------------+------+--------------+-----------+-------
(0 rows)
```

2.3.7 Disabling and Enabling Monitoring

If you wish to turn off SQL/Protect monitoring, modify the `postgresql.conf` file, setting the `edb_sql_protect.enabled` parameter to `off`. After saving the file, reload the server configuration to apply the settings.

If you wish to turn on SQL/Protect monitoring, modify the `postgresql.conf` file, setting the `edb_sql_protect.enabled` parameter to `on`. After saving the file, reload the server configuration to apply the settings.

2.4 Backing Up and Restoring a SQL/Protect Database

Backing up a database that is configured with SQL/Protect, and then restoring the backup file to a new database requires additional considerations to what is normally associated with backup and restore procedures. This is primarily due to the use of Object Identification numbers (OIDs) in the SQL/Protect tables as explained in this section.

Note: This section is applicable if your backup and restore procedures result in the re-creation of database objects in the new database with new OIDs such as is the case when using the `pg_dump` backup program.

If you are backing up your Advanced Server database server by simply using the operating system's copy utility to create a binary image of the Advanced Server data files (file system backup method), then this section does not apply.

2.4.1 Object Identification Numbers in SQL/Protect Tables

SQL/Protect uses two tables (`edb_sql_protect` and `edb_sql_protect_rel`) to store information on database objects such as databases, roles, and relations. References to these database objects in these tables are done using the objects' OIDs, and not the objects' text names. The OID is a numeric data type used by Advanced Server to uniquely identify each database object.

When a database object is created, Advanced Server assigns an OID to the object, which is then used whenever a reference is needed to the object in the database catalogs. If you create the same database object in two databases, such as a table with the same `CREATE TABLE` statement, each table is assigned a different OID in each database.

In a backup and restore operation that results in the re-creation of the backed up database objects, the restored objects end up with different OIDs in the new database than what they were assigned in the original database. As a result, the OIDs referencing databases, roles, and relations stored in the `edb_sql_protect` and `edb_sql_protect_rel` tables are no longer valid when these tables are simply dumped to a backup file and then restored to a new database.

The following sections describe two functions, `export_sqlprotect` and `import_sqlprotect`, that are used specifically for backing up and restoring SQL/Protect tables in order to ensure the OIDs in the SQL/Protect tables reference the correct database objects after the tables are restored.

2.4.2 Backing Up the Database

The following steps back up a database that has been configured with SQL/Protect.

Step 1: Create a backup file using `pg_dump`.

The following example shows a plain-text backup file named `/tmp/edb.dmp` created from database `edb` using the `pg_dump` utility program:

```
$ cd /usr/edb/as13/bin
$ ./pg_dump -U enterprisedb -Fp -f /tmp/edb.dmp edb
Password:
$
```

Step 2: Connect to the database as a superuser and export the SQL/Protect data using the `export_sqlprotect('sqlprotect_file')` function (where `sqlprotect_file` is the fully qualified path to a file where the `SQL/Protect` data is to be saved).

The `enterprisedb` operating system account (`postgres` if you installed Advanced Server in PostgreSQL compatibility mode) must have read and write access to the directory specified in `sqlprotect_file`.

```
edb=# SELECT sqlprotect.export_sqlprotect('/tmp/sqlprotect.dmp');
 export_sqlprotect
-------------------
(1 row)
```

The files `/tmp/edb.dmp` and `/tmp/sqlprotect.dmp` comprise your total database backup.

2.4.3 Restoring From the Backup Files

Step 1: Restore the backup file to the new database.

The following example uses the `psql` utility program to restore the plain-text backup file `/tmp/edb.dmp` to a newly created database named `newdb`:

```
$ /usr/edb/as13/bin/psql -d newdb -U enterprisedb -f /tmp/edb.dmp
Password for user enterprisedb:
SET
SET
SET
SET
SET
COMMENT
CREATE SCHEMA
    .
    .
    .
```

Step 2: Connect to the new database as a superuser and delete all rows from the `edb_sql_protect_rel` table.

This step removes any existing rows in the edb_sql_protect_rel table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored:

```
$ /usr/edb/as13/bin/psql -d newdb -U enterprisedb
Password for user enterprisedb:
psql.bin (13.0.0, server 13.0.0)
Type "help" for help.

newdb=# DELETE FROM sqlprotect.edb_sql_protect_rel;
DELETE 2
```

Step 3: Delete all rows from the edb_sql_protect table.

This step removes any existing rows in the edb_sql_protect table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored:

```
newdb=# DELETE FROM sqlprotect.edb_sql_protect;
DELETE 1
```

Step 4: Delete any statistics that may exist for the database.

This step removes any existing statistics that may exist for the database to which you are restoring the backup. The following query displays any existing statistics:

```
newdb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
(0 rows)
```

For each row that appears in the preceding query, use the drop_stats function specifying the role name of the entry.

For example, if a row appeared with appuser in the username column, issue the following command to remove it:

```
newdb=# SELECT sqlprotect.drop_stats('appuser');
 drop_stats
------------
(1 row)
```

Step 5: Delete any offending queries that may exist for the database.

This step removes any existing queries that may exist for the database to which you are restoring the backup. The following query displays any existing queries:

```
edb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
 username  | ip_address | port | machine_name | date_time | query
-----------+------------+------+--------------+-----------+-------
(0 rows)
```

For each row that appears in the preceding query, use the `drop_queries` function specifying the role name of the entry.

For example, if a row appeared with `appuser` in the `username` column, issue the following command to remove it:

```
edb=# SELECT sqlprotect.drop_queries('appuser');
 drop_queries
--------------
(1 row)
```

Step 6: Make sure the role names that were protected by SQL/Protect in the original database exist in the database server where the new database resides.

If the original and new databases reside in the same database server, then nothing needs to be done assuming you have not deleted any of these roles from the database server.

Step 7: Run the function `import_sqlprotect('sqlprotect_file')` where `sqlprotect_file` is the fully qualified path to the file you created in `Step 2` of `Backing Up the Database`.

```
newdb=# SELECT sqlprotect.import_sqlprotect('/tmp/sqlprotect.dmp');
 import_sqlprotect
-------------------
(1 row)
```

Tables `edb_sql_protect` and `edb_sql_protect_rel` are now populated with entries containing the OIDs of the database objects as assigned in the new database. The statistics view `edb_sql_protect_stats` also now displays the statistics imported from the original database.

The SQL/Protect tables and statistics are now properly restored for this database. This is verified by the following queries on the Advanced Server system catalogs:

```
newdb=# SELECT datname, oid FROM pg_database;
  datname  |  oid
-----------+-------
 template1 |     1
 template0 | 13909
 edb       | 13917
 newdb     | 16679
(4 rows)

newdb=# SELECT rolname, oid FROM pg_roles;
   rolname    |  oid
--------------+-------
 enterprisedb |    10
 appuser      | 16671
 newuser      | 16678
(3 rows)

newdb=# SELECT relname, oid FROM pg_class WHERE relname IN
('dept','emp','appuser_tab');
   relname   |  oid
```

(continues on next page)

(continued from previous page)

```
-------------+-------
 appuser_tab | 16803
 dept        | 16809
 emp         | 16812
(3 rows)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect;
 dbid  | roleid | protect_relations | allow_utility_cmds | allow_tautology |
 allow_empty_dml
-------+--------+-------------------+--------------------+-----------------+--
---------------
 16679 |  16671 | t                 | t                  | f               | f
(1 row)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect_rel;
 dbid  | roleid | relid
-------+--------+-------
 16679 |  16671 | 16809
 16679 |  16671 | 16803
(2 rows)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
 appuser  |          0 |         5 |        2 |         1 |   0
(1 row)

newedb=# \x
Expanded display is on.
nwedb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
-[ RECORD 1 ]+-----------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 2 ]+-----------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 3 ]+-----------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | CREATE TABLE appuser_tab_3 (f1 INTEGER);
-[ RECORD 4 ]+-----------------------------------------
```

(continues on next page)

(continued from previous page)

```
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 5 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

Note the following about the columns in tables `edb_sql_protect` and `edb_sql_protect_rel`:

- **dbid.** Matches the value in the `oid` column from `pg_database` for `newdb`
- **roleid.** Matches the value in the `oid` column from `pg_roles` for `appuser`

Also note that in table `edb_sql_protect_rel`, the values in the `relid` column match the values in the `oid` column of `pg_class` for relations `dept` and `appuser_tab`.

Step 8: Verify that the SQL/Protect configuration parameters are set as desired in the `postgresql.conf` file for the database server running the new database. Restart the database server or reload the configuration file as appropriate.

You can now monitor the database using SQL/Protect.

CHAPTER 3

Virtual Private Database

`Virtual Private Database` is a type of fine-grained access control using security policies. `Fine-grained access control` means that access to data can be controlled down to specific rows as defined by the security policy.

The rules that encode a security policy are defined in a *policy function*, which is an SPL function with certain input parameters and return value. The *security policy* is the named association of the policy function to a particular database object, typically a table.

In Advanced Server, the policy function can be written in any language supported by Advanced Server such as SQL and PL/pgSQL in addition to SPL.

Note: The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

- Provides a fine-grained level of security. Database object level privileges given by the `GRANT` command determine access privileges to the entire instance of a database object, while Virtual Private Database provides access control for the individual rows of a database object instance.
- A different security policy can be applied depending upon the type of SQL command (`INSERT, UPDATE, DELETE, or SELECT`).
- The security policy can vary dynamically for each applicable SQL command affecting the database object depending upon factors such as the session user of the application accessing the database object.
- Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.

- Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
- Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

Note: The only way security policies can be circumvented is if the `EXEMPT ACCESS POLICY` system privilege has been granted to a user. The `EXEMPT ACCESS POLICY` privilege should be granted with extreme care as a user with this privilege is exempted from all policies in the database.

The `DBMS_RLS` package provides procedures to create policies, remove policies, enable policies, and disable policies.

CHAPTER 4

sslutils

sslutils is a Postgres extension that provides SSL certificate generation functions to Advanced Server for use by the EDB Postgres Enterprise Manager server. sslutils is installed by using the edb-asxx-server-sslutils RPM package where xx is the Advanced Server version number.

The sslutils package provides the functions shown in the following sections.

In these sections, each parameter in the function's parameter list is described by parameter n under the Parameters subsection where n refers to the nth ordinal position (for example, first, second, third, etc.) within the function's parameter list.

4.1 openssl_rsa_generate_key

The openssl_rsa_generate_key function generates an RSA private key. The function signature is:

```
openssl_rsa_generate_key(<integer>) RETURNS <text>
```

When invoking the function, pass the number of bits as an integer value; the function returns the generated key.

4.2 openssl_rsa_key_to_csr

The openssl_rsa_key_to_csr function generates a certificate signing request (CSR). The signature is:

```
openssl_rsa_key_to_csr(<text>, <text>, <text>, <text>, <text>, <text>,
<text>) RETURNS text
```

The function generates and returns the certificate signing request.

Parameters

parameter 1

The name of the RSA key file.

parameter 2

The common name (e.g., agentN) of the agent that will use the signing request.

parameter 3

The name of the country in which the server resides.

parameter 4

The name of the state in which the server resides.

parameter 5

The location (city) within the state in which the server resides.

parameter 6

The name of the organization unit requesting the certificate.

parameter 7

The email address of the user requesting the certificate.

4.3 openssl_csr_to_crt

The openssl_csr_to_crt function generates a self-signed certificate or a certificate authority certificate. The signature is:

```
openssl_csr_to_crt(<text>, <text>, <text>) RETURNS <text>
```

The function returns the self-signed certificate or certificate authority certificate.

Parameters

parameter 1

The name of the certificate signing the request.

parameter 2

The path to the certificate authority certificate, or NULL if generating a certificate authority certificate.

parameter 3

The path to the certificate authority's private key or (if argument 2 is NULL) the path to a private key.

4.4 openssl_rsa_generate_crl

The openssl_rsa_generate_crl function generates a default certificate revocation list. The signature is:

```
openssl_rsa_generate_crl(<text>, <text>) RETURNS <text>
```

The function returns the certificate revocation list.

Parameters

parameter 1

The path to the certificate authority certificate.

parameter 2

The path to the certificate authority private key.

CHAPTER 5

Data Redaction

Data redaction limits sensitive data exposure by dynamically changing data as it is displayed for certain users.

For example, a social security number (SSN) is stored as `021-23-9567`. Privileged users can see the full SSN, while other users only see the last four digits `xxx-xx-9567`.

Data redaction is implemented by defining a function for each field to which redaction is to be applied. The function returns the value that should be displayed to the users subject to the data redaction.

So for example, for the SSN field, the redaction function would return `xxx-xx-9567` for an input SSN of `021-23-9567`.

For a salary field, a redaction function would always return `$0.00` regardless of the input salary value.

These functions are then incorporated into a redaction policy by using the `CREATE REDACTION POLICY` command. This command specifies the table on which the policy applies, the table columns to be affected by the specified redaction functions, expressions to determine which session users are to be affected, and other options.

The `edb_data_redaction` parameter in the `postgresql.conf` file then determines whether or not data redaction is to be applied.

By default, the parameter is enabled so the redaction policy is in effect and the following occurs:

- Superusers and the table owner bypass data redaction and see the original data.
- All other users get the redaction policy applied and see the reformatted data.

If the parameter is disabled by having it set to `FALSE` during the session, then the following occurs:

- Superusers and the table owner bypass data redaction and see the original data.
- All other users get will get an error.

A redaction policy can be changed by using the `ALTER REDACTION POLICY` command, or it can be eliminated using the `DROP REDACTION POLICY` command.

The redaction policy commands are described in more detail in the subsequent sections.

5.1 CREATE REDACTION POLICY

`CREATE REDACTION POLICY` defines a new data redaction policy for a table.

Synopsis

```
CREATE REDACTION POLICY <name> ON <table_name>
  [ FOR ( <expression> ) ]
  [ ADD [ COLUMN ] <column_name> USING <funcname_clause>
    [ WITH OPTIONS ( [ <redaction_option> ]
      [, <redaction_option> ] )
    ]
  ] [, ...]
```

where `redaction_option` is:

```
{ SCOPE <scope_value> |
  EXCEPTION <exception_value> }
```

Description

The `CREATE REDACTION POLICY` command defines a new column-level security policy for a table by redacting column data using redaction function. A newly created data redaction policy will be enabled by default. The policy can be disabled using `ALTER REDACTION POLICY ... DISABLE`.

`FOR ( expression )`

This form adds a redaction policy expression.

`ADD [ COLUMN ]`

This optional form adds a column of the table to the data redaction policy. The `USING` specifies a redaction function expression. Multiple `ADD [ COLUMN ]` form can be used, if you want to add multiple columns of the table to the data redaction policy being created. The optional `WITH OPTIONS ( ... )` clause specifies a scope and/or an exception to the data redaction policy to be applied. If the scope and/or exception are not specified, the default values for scope and exception will be `query` and `none` respectively.

Parameters

`name`

The name of the data redaction policy to be created. This must be distinct from the name of any other existing data redaction policy for the table.

`table_name`

The name (optionally schema-qualified) of the table the data redaction policy applies to.

`expression`

The data redaction policy expression. No redaction will be applied if this expression evaluates to false.

`column_name`

Name of the existing column of the table on which the data redaction policy being created.

`funcname_clause`

The data redaction function which decides how to compute the redacted column value. Return type of the redaction function should be same as the column type on which data redaction policy being added.

`scope_value`

The scope identified the query part where redaction to be applied for the column. Scope value could be `query, top_tlist` or `top_tlist_or_error`. If the scope is `query` then, the redaction applied on the column irrespective of where it appears in the query. If the scope is `top_tlist` then, the redaction applied on the column only when it appears in the query's top target list. If the scope is `top_tlist_or_error` the behavior will be same as the `top_tlist`, but throws an errors when the column appears anywhere else in the query.

`exception_value`

The exception identified the query part where redaction to be exempted. Exception value could be `none, equal` or `leakproof`. If exception is `none` then there is no exemption. If exception is `equal`, then the column is not redacted when used in an equality test. If exception is `leakproof`, the column will is not redacted when a leakproof function is applied to it.

Notes:

You must be the owner of a table to create or change data redaction policies for it.

The superuser and the table owner are exempt from the data redaction policy.

Examples

Below is an example of how this feature can be used in production environments. Create the components for a data redaction policy on the `employees` table:

```
CREATE TABLE employees (
 id          integer GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
 name        varchar(40) NOT NULL,
 ssn         varchar(11) NOT NULL,
 phone       varchar(10),
 birthday    date,
 salary      money,
 email       varchar(100)
);

-- Insert some data
INSERT INTO employees (name, ssn, phone, birthday, salary, email)
VALUES
( 'Sally Sample', '020-78-9345', '5081234567', '1961-02-02', 51234.34,
'sally.sample@enterprisedb.com'),
( 'Jane Doe', '123-33-9345', '6171234567', '1963-02-14', 62500.00,
'jane.doe@gmail.com'),
( 'Bill Foo', '123-89-9345', '9781234567','1963-02-14', 45350,
'william.foe@hotmail.com');
```

(continues on next page)

(continued from previous page)

```
-- Create a user hr who can see all the data in employees
CREATE USER hr;
-- Create a normal user
CREATE USER alice;
GRANT ALL ON employees TO hr, alice;

-- Create redaction function in which actual redaction logic resides
CREATE OR REPLACE FUNCTION redact_ssn (ssn varchar(11)) RETURN varchar(11) IS
BEGIN
   /* replaces 020-12-9876 with xxx-xx-9876 */
   return overlay (ssn placing 'xxx-xx' from 1) ;
END;

CREATE OR REPLACE FUNCTION redact_salary () RETURN money IS BEGIN return
0::money;
END;
```

Now create a data redaction policy on `employees` to redact column `ssn` which should be accessible in equality condition and `salary` with default scope and exception. The redaction policy will be exempt for the `hr` user.

```
CREATE REDACTION POLICY redact_policy_personal_info ON employees FOR
(session_user != 'hr')
ADD COLUMN ssn USING redact_ssn(ssn) WITH OPTIONS (SCOPE query,
EXCEPTION equal),
ADD COLUMN salary USING redact_salary();
```

The visible data for the `hr` user will be:

```
-- hr can view all columns data
edb=# \c edb hr
edb=> SELECT * FROM employees;
 id | name          | ssn         | phone      | birthday           |
 salary     | email
----+--------------+-------------+------------+--------------------+---
--+---------------------
  1 | Sally Sample | 020-78-9345 | 5081234567 | 02-FEB-61 00:00:00 |
  $51,234.34 | sally.sample@enterprisedb.com
  2 | Jane Doe     | 123-33-9345 | 6171234567 | 14-FEB-63 00:00:00 |
  $62,500.00 | jane.doe@gmail.com
  3 | Bill Foo     | 123-89-9345 | 9781234567 | 14-FEB-63 00:00:00 |
  $45,350.00 | william.foe@hotmail.com
(3 rows)
```

The visible data for the normal user `alice` will be:

```
-- Normal user cannot see salary and ssn number.
edb=> \c edb alice
edb=> SELECT * FROM employees;
id  | name          | ssn         | phone      | birthday           | salary |
email
```

(continues on next page)

(continued from previous page)

```
----+--------------+-------------+------------+--------------------+--------+-
-----------------------------
  1 | Sally Sample | xxx-xx-9345 | 5081234567 | 02-FEB-61 00:00:00 |  $0.00 |
  sally.sample@enterprisedb.com
  2 | Jane Doe     | xxx-xx-9345 | 6171234567 | 14-FEB-63 00:00:00 |  $0.00 |
  jane.doe@gmail.com
  3 | Bill Foo     | xxx-xx-9345 | 9781234567 | 14-FEB-63 00:00:00 |  $0.00 |
  william.foe@hotmail.com
(3 rows)
```

But `ssn` data is accessible when it used for equality check due to `exception_value` setting.

```
-- Get ssn number starting from 123
edb=> SELECT * FROM employees WHERE substring(ssn from 0 for 4) = '123';
 id | name     | ssn         | phone      | birthday            | salary |
 email
----+----------+-------------+------------+---------------------+--------+-----
--------------------
  2 | Jane Doe | xxx-xx-9345 | 6171234567 | 14-FEB-63 00:00:00 |  $0.00 |
  jane.doe@gmail.com
  3 | Bill Foo | xxx-xx-9345 | 9781234567 | 14-FEB-63 00:00:00 |  $0.00 |
  william.foe@hotmail.com
(2 rows)
```

Caveats

1. The data redaction policy created on inheritance hierarchies will not be cascaded. For example, if the data redaction policy is created for a parent, it will not be applied to the child table, which inherits it and vice versa. Someone who has access to these child tables can see the non-redacted data. For information about inheritance hierarchies, see `Inheritance` in the PostgreSQL Core Documentation available at:

   https://www.postgresql.org/docs/current/static/ddl-inherit.html

2. If the superuser or the table owner has created any materialized view on the table and has provided the access rights `GRANT SELECT` on the table and the materialized view to any non-superuser, then the non-superuser will be able to access the non-redacted data through the materialized view.

3. The objects accessed in the redaction function body should be schema qualified otherwise `pg_dump` might fail.

Compatibility

`CREATE REDACTION POLICY` is an EDB extension.

See Also

`ALTER REDACTION POLICY, DROP REDACTION POLICY`

5.2 ALTER REDACTION POLICY

`ALTER REDACTION POLICY` changes the definition of data redaction policy for a table.

Synopsis

```
ALTER REDACTION POLICY <name> ON <table_name> RENAME TO <new_name>

ALTER REDACTION POLICY <name> ON <table_name> FOR ( <expression> )

ALTER REDACTION POLICY <name> ON <table_name> { ENABLE | DISABLE}

ALTER REDACTION POLICY <name> ON <table_name>
  ADD [ COLUMN ] <column_name> USING <funcname_clause>
    [ WITH OPTIONS ( [ <redaction_option> ]
      [, <redaction_option> ] )
    ]

ALTER REDACTION POLICY <name> ON <table_name>
  MODIFY [ COLUMN ] <column_name>
  {
    [ USING <funcname_clause> ]
  |
    [ WITH OPTIONS ( [ <redaction_option> ]
      [, <redaction_option> ] )
    ]
  }

ALTER REDACTION POLICY <name> ON <table_name>
  DROP [ COLUMN ] <column_name>
```

where `redaction_option` is:

```
{ SCOPE <scope_value> |
  EXCEPTION <exception_value> }
```

Description

`ALTER REDACTION POLICY` changes the definition of an existing data redaction policy.

To use `ALTER REDACTION POLICY`, you must own the table that the data redaction policy applies to.

`FOR ( expression )`

This form adds or replaces the data redaction policy expression.

`ENABLE`

Enables the previously disabled data redaction policy for a table.

`DISABLE`

Disables the data redaction policy for a table.

`ADD [ COLUMN ]`

This form adds a column of the table to the existing redaction policy. See CREATE REDACTION POLICY for the details.

MODIFY [COLUMN]

This form modifies the data redaction policy on the column of the table. You can update the redaction function clause and/or the redaction options for the column. The USING clause specifies the redaction function expression to be updated and the WITH OPTIONS (...) clause specifies the scope and/or the exception. For more details on the redaction function clause, the redaction scope and the redaction exception, see CREATE REDACTION POLICY.

DROP [COLUMN]

This form removes the column of the table from the data redaction policy.

Parameters

name

The name of an existing data redaction policy to alter.

table_name

The name (optionally schema-qualified) of the table that the data redaction policy is on.

new_name

The new name for the data redaction policy. This must be distinct from the name of any other existing data redaction policy for the table.

expression

The data redaction policy expression.

column_name

Name of existing column of the table on which the data redaction policy being altered or dropped.

funcname_clause

The data redaction function expression for the column. See CREATE REDACTION POLICY for details.

scope_value

The scope identified the query part where redaction to be applied for the column. See CREATE REDACTION POLICY for the details.

exception_value

The exception identified the query part where redaction to be exempted. See CREATE REDACTION POLICY for the details.

Examples

Update data redaction policy called redact_policy_personal_info on the table named employees:

```
ALTER REDACTION POLICY redact_policy_personal_info ON employees
FOR (session_user != 'hr' AND session_user != 'manager');
```

And to update data redaction function for the column `ssn` in the same policy:

```
ALTER REDACTION POLICY redact_policy_personal_info ON employees
MODIFY COLUMN ssn USING redact_ssn_new(ssn);
```

Compatibility

`ALTER REDACTION POLICY` is an EDB extension.

See Also

`CREATE REDACTION POLICY, DROP REDACTION POLICY`

5.3 DROP REDACTION POLICY

`DROP REDACTION POLICY` removes a data redaction policy from a table.

Synopsis

```
DROP REDACTION POLICY [ IF EXISTS ] <name> ON <table_name>
  [ CASCADE | RESTRICT ]
```

Description

`DROP REDACTION POLICY` removes the specified data redaction policy from the table.

To use `DROP REDACTION POLICY`, you must own the table that the redaction policy applies to.

Parameters

`IF EXISTS`

Do not throw an error if the data redaction policy does not exist. A notice is issued in this case.

`name`

The name of the data redaction policy to drop.

`table_name`

The name (optionally schema-qualified) of the table that the data redaction policy is on.

`CASCADE`

`RESTRICT`

These keywords do not have any effect, since there are no dependencies on the data redaction policies.

Examples

To drop the data redaction policy called `redact_policy_personal_info` on the table named `employees`:

```
DROP REDACTION POLICY redact_policy_personal_info ON employees;
```

Compatibilities

`DROP REDACTION POLICY` is an EDB extension.

See Also

`CREATE REDACTION POLICY, ALTER REDACTION POLICY`

5.4 System Catalogs

This section describes the system catalogs that store the redaction policy information.

5.4.1 edb_redaction_column

The `edb_redaction_column` system catalog stores information about the data redaction policy attached to the columns of a table.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute, must be explicitly selected)
rdpolicyid	oid	edb_redaction_policy.oid	The data redaction policy applies to the described column
rdrelid	oid	pg_class.oid	The table that the described column belongs to
rdattnum	int2	pg_attribute.attnum	The number of the described column
rdscope	int2		The redaction scope: 1 = query, 2 = top_tlist, 4 = top_tlist_or_error
rdexception	int2		The redaction exception: 8 = none, 16 = equal, 32 = leakproof
rdfuncexpr	pg_node_tree		Data redaction function expression

Note: The described column will be redacted if the redaction policy `edb_redaction_column.rdpolicyid` on the table is enabled and the redaction policy expression `edb_redaction_policy.rdexpr` evaluates to `true`.

5.4.2 edb_redaction_policy

The catalog `edb_redaction_policy` stores information about the redaction policies for tables.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute, must be explicitly selected)
rdname	name		The name of the data redaction policy
rdrelid	oid	pg_class.oid	The table to which the data redaction policy applies
rdenable	boolean		Is the data redaction policy enabled?
rdexpr	pg_node_tree		The data redaction policy expression

Note: The data redaction policy applies for the table if it is enabled and the expression ever evaluated true.

CHAPTER 6

Conclusion

EDB Postgres™ Advanced Server Security Features Guide

Copyright © 2007 - 2020 EnterpriseDB Corporation.

All rights reserved.

EnterpriseDB® Corporation

34 Crosby Drive, Suite 201, Bedford, MA 01730, USA

T +1 781 357 3390 F +1 978 467 1307 E

info@enterprisedb.com

www.enterprisedb.com

- EnterpriseDB and Postgres Enterprise Manager are registered trademarks of EnterpriseDB Corporation. EDB and EDB Postgres are trademarks of EnterpriseDB Corporation. Oracle is a registered trademark of Oracle, Inc. Other trademarks may be trademarks of their respective owners.

- EDB designs, establishes coding best practices, reviews, and verifies input validation for the logon UI for EDB products where present. EDB follows the same approach for additional input components, however the nature of the product may require that it accepts freeform SQL, WMI or other strings to be entered and submitted by trusted users for which limited validation is possible. In such cases it is not possible to prevent users from entering incorrect or otherwise dangerous inputs.
- EDB reserves the right to add features to products that accept freeform SQL, WMI or other potentially dangerous inputs from authenticated, trusted users in the future, but will ensure all such features are designed and tested to ensure they provide the minimum possible risk, and where possible, require superuser or equivalent privileges.
- EDB does not warrant that we can or will anticipate all potential threats and therefore our process cannot fully guarantee that all potential vulnerabilities have been addressed or considered.

Index