The security guide of Security-Enhanced PostgreSQL

(English edition)

SE-PostgreSQL development team

Change Logs	
2007/07/01	English Edition (beta edition) is published

Contents

1 Introduction

Security-Enhanced PostgreSQL, or SE-PostgreSQL, provides access control mechanisms based on SELinux architecture built in PostgreSQL, a popular open-source relational database system. It improves database security innovatively.

The core ideas of SE-PostgreSQL are; integration with SELinux, which is known as the most widely used secure OS, fine-grained access control including row- and column-level control, and mandatory access control, even which a privileged database user, cannot be avoided. SE-PostgreSQL allows database systems to be built into information flow control scheme integrated with operating system, and protects from threats like compromise, manipulation, or destruction of information properties.

1.1 Security Enhanced PostgreSQL

While traditional access controls of DBMS are independent from OS-based access controls, SE-PostgreSQL provides a unified access control with the OS security policy in collaboration with SELinux. That realizes fine-grained mandatory access controls.

Mandatory Access Control

PostgreSQL has the concept of privileged DB user, which enables to skip all native PostgreSQL access controls. On the contrary, SE-PostgreSQL forces access controls to every client without any exception even if privileged DB user.

This is as alike as a relationship between Discretionary Access Control (DAC) and Mandatory Access Control (MAC) on the operating system. Only when both access controls of native PostgreSQL and SE-PostgreSQL allow an access, a client could access to the DB object.

Fine-grained Access Control

Access control features on DBMS are to allow or to deny accessing to the DB object based on the client authorities. It depends on DBMS products that which DB object is subject to access control, however, SE-PostgreSQL provides row- and column-level access controls. Only few commercial DBMS products have been serving that facility.

Row and column are the smallest DB object units in relational databases, so that the DBA can apply the most flexible configuration.

Consistent Client Authority

SE-PostgreSQL uses security contexts of the process connecting to as client's authorities. Native PostgreSQL database authentication does not affect SE-PostgreSQL access controls.

If you use TCP/IP to connect SE-PostgreSQL, it requires additional settings. See section "5.3 Labeled IPsec".

Consistent Security Policy

SE-PostgreSQL could correlate security contexts by an explicit or an implicit way based on the SELinux security policy with the database objects under supervision.

It controls accesses by security contexts of the client and the database object. Because any control is

based on the SELinux security policy, the system administrator, or you, could conduct consistent access controls throughout the whole system. It means the availability of building databases into information flow control and being an effective method to avoid information compromises.

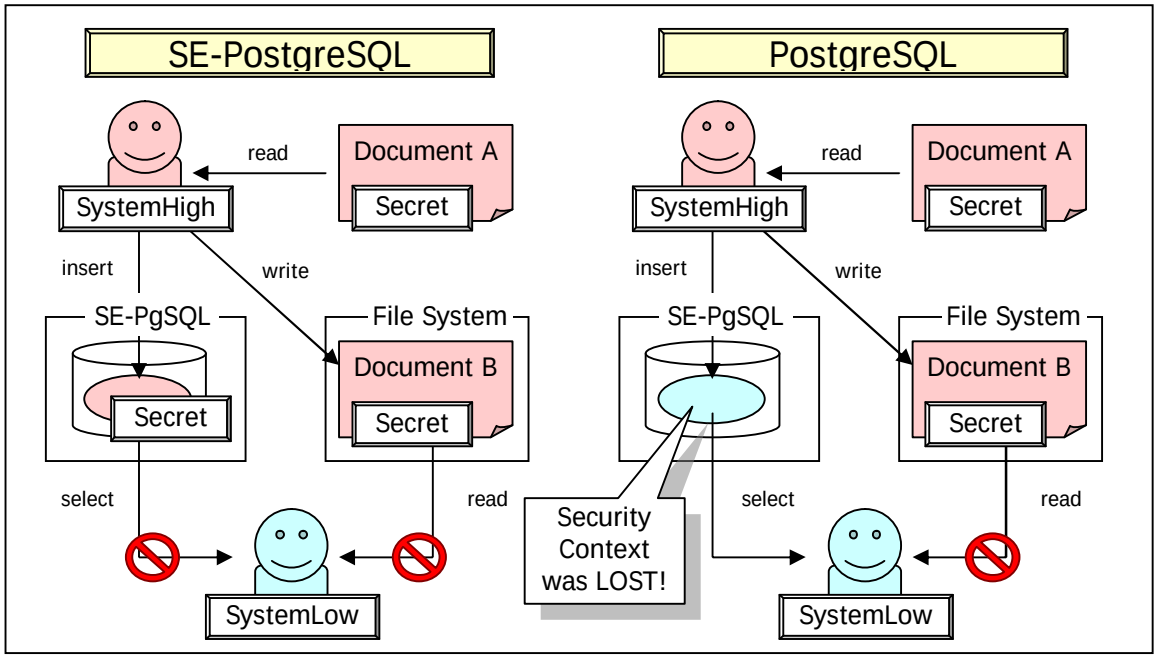


Figure. 1 Information flow control

When there is a process which is able to read a file labeled "`Secret`" (`SystemHigh` process), and the process create a file, the file is also given "`Secret`" label. That is why a low-authority process (`SystemLow` process) could not read it.

On the other hand, if a `SystemHigh` process injects this information into the database, a classic RDBMS will lose this security context. It allows a `SystemLow` process to read it as a result. Because SE-PostgreSQL also add an injected data the "`Secret`" label, however, a `SystemLow` process could not reference "`Secret`" data even if via an RDBMS.

1.2 Copyright and License

SE-PostgreSQL is implemented as an extension of the original PostgreSQL. Copyright of the original PostgreSQL is retained by the PostgreSQL global development group. The SE-PostgreSQL development team retains that of an extension part of SE-PostgreSQL, including PGACE.

Since SE-PostgreSQL is a derivative of PostgreSQL, we hope our product to be merged into PostgreSQL in the future. Thus, we chose the BSD license as the PostgreSQL team did. See the original license shown below.

```
BSD License: http://www.postgresql.org/about/licence
```

2 Access Control of SE-PostgreSQL

In this chapter, we describe how SE-PostgreSQL access control works.

2.1 Collaboration with SELinux

SE-PostgreSQL is a reference monitor built in PostgreSQL. It will check every SQL query provided by clients, and prevent violated accesses to DB objects. This is achieved with the SELinux security policy stored in the Linux kernel. The policy includes the all access control rules in SELinux.

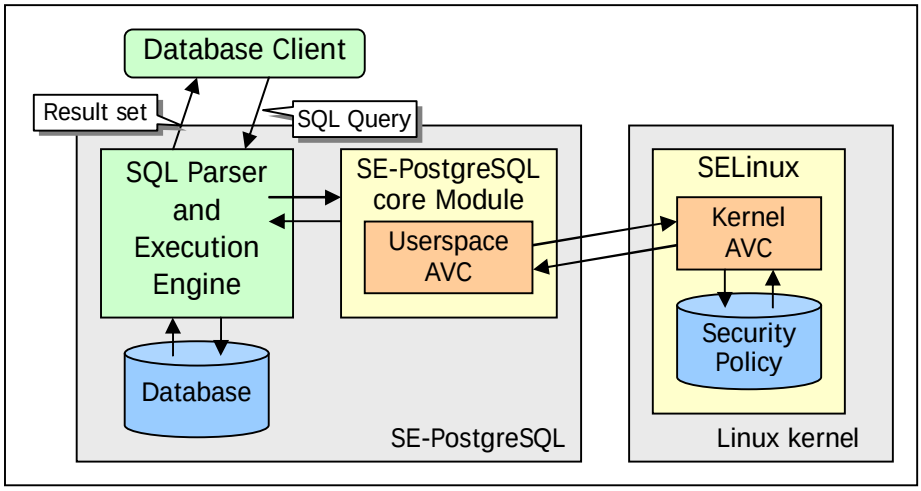


Figure. 2 SE-PostgreSQL architecture

Access controls in SELinux is done with the white list. You must define that "who" (Subject) "does" (Action) "to what" (Object) implicitly, or requested action will be declined. For more details about access control methods, see "4.Security Design of SELinux".

SE-PostgreSQL applies access controls such as TE, MLS, or MCS as well as the SELinux-enabled OS. You can describe the SELinux policy just as the object classes are under supervision of the OS.

System-call is a high-cost treatment usually, but SE-PostgreSQL prevents performance loss to minimize by caching a part of the security policy on Access Vector Cache (AVC).

2.2 Security Context

When you execute `id -z` on the SELinux enabled host, you will see as below:

```
[root@masu ~]# id -Z
root:system_r:unconfined_t:SystemLow-SystemHigh
```

These are "Security Context" sequences, given for SELinux to identify security property of the resource. In this example, security contexts of a process are shown.

Not only processes, but also all resources SELinux could observe, such as files, sockets, or IPC objects, are subjects to be associated.

To show file security contexts, execute `ls -z`.

```
 [root@masu ~]# ls -Z /etc/passwd
-rw-r--r--  root root system_u:object_r:etc_t        /etc/passwd
```

See “4.1 Security Contexts” for more specific information about security contexts.

A security context contains all needed information that SELinux identifies any resources and determines whether required accesses are allowed, or not. All SELinux access controls are done based on them.

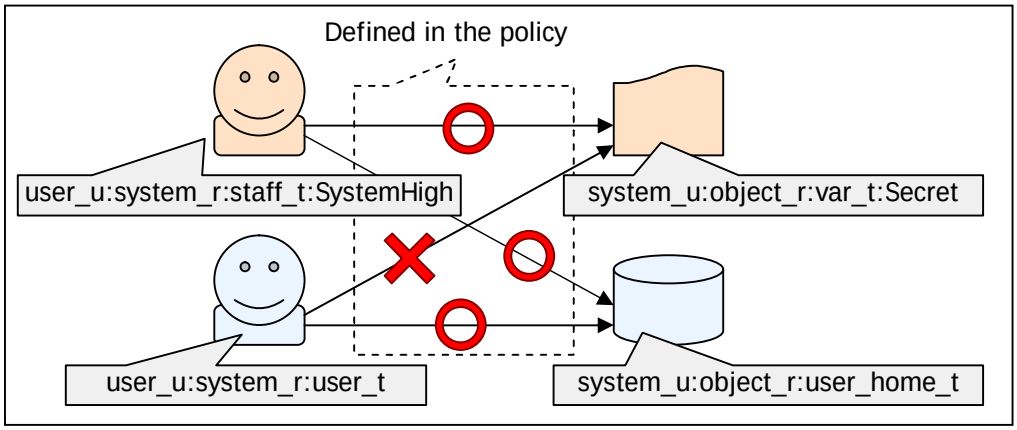


Figure. 3 Security Context and Access Control

Figure. 3 shows a model of access control. Processes and files have each security contexts, and permitted actions are described in the security policy between each security context.

SE-PostgreSQL enforces access controls onto DB objects. You need to associate security contexts with DB objects when you use SELinux access control mechanism.

SE-PostgreSQL associates security contexts with every tuple (row). To confirm this attribute, check the system column of `security_context`.

```
kaigai=# select security_context, * from drink;
           security_context                | id |  name  | price | alcohol
-------------------------------------------+----+--------+-------+--------
 user_u:object_r:sepgsql_table_t           |  1 | coffee |   120 | f
 user_u:object_r:sepgsql_table_t           |  2 | tea    |   120 | f
 user_u:object_r:sepgsql_table_t           |  5 | water  |   110 | f
 user_u:object_r:sepgsql_table_t           |  6 | coke   |   110 | f
 user_u:object_r:sepgsql_table_t:Secret    |  3 | wine   |   360 | t
 user_u:object_r:sepgsql_table_t:Secret    |  4 | beer   |   240 | t
(6 rows)
```

It handles DB objects such as tables and columns as tuples inside PostgreSQL. These DB objects are represented as tuples stored in the special table called System Catalog.

For example, metadata of a table is stored in a tuple contained in `pg_class`, and that of a column is `pg_attribute`. Security Contexts associated with these tuples are seemed as that of tables or columns. SE-PostgreSQL utilizes those Security Contexts for access controls.

2.2.1 Client Security Context

On the other hand, Security Contexts of clients, the actor of accesses to DB objects, is associated with different way from DB objects their selves.

SELinux has an API to obtain Security Contexts of the peer of stream socket. SE-PostgreSQL will

obtain Security Contexts of the remote process using the API to apply it as Security Context of the client. The client is under access control of SE-PostgreSQL just as if it is controlled by the OS, in other words.

While you do not need special configuration when connecting via UNIX domain sockets, note that Labeled Network of IPSec is required when connecting via TCP/IP. See more details in “5.3 Labeled ”.

2.3 Mandatory Access Control by SE-PostgreSQL

It will take some steps while PostgreSQL is processing SQL queries handed by the client, accessing to the DB, and returning results sets.

SE-PostgreSQL checks all SQL queries during query process, and protects DB objects from violated accesses.

This check is mandatory even if a privileged DB user executes the query. This Mandatory Access Control is one of the important features of SE-PostgreSQL.

While native PostgreSQL controls accesses in a Query analyzer phase, but SE-PostgreSQL does it at the next phase of Query rewriter.

This different presents you an interesting nature of SE-PostgreSQL access control.

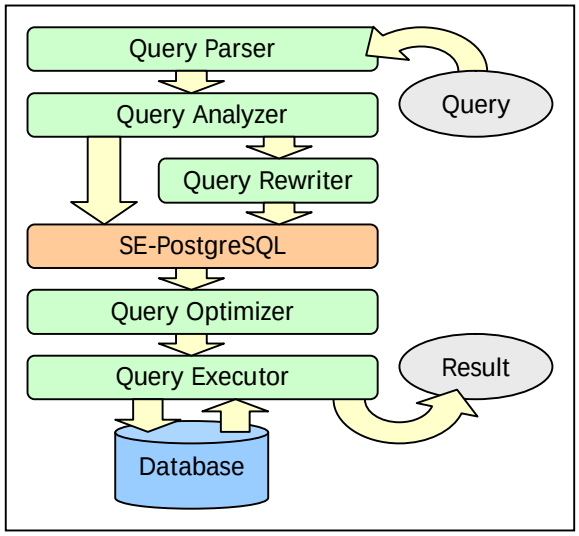


Figure. 4 Query Processing Flow in SE-PostgreSQL

At the Query rewriter phase, a reference via view will be rewritten to a reference to the actual table. When you execute SQL query #2, using `my_view` that is defined by SQL query #1, Query rewriter modifies SQL query #2 as if the SQL query #3 is input, for example

```
(1)  CREATE VIEW my_view AS SELECT a, b + c AS x FROM my_table;
(2)  SELECT a, x * x FROM my_view;
(3)  SELECT a, (b + c) * (b + c) FROM my_table;
```

SE-PostgreSQL checks rewritten queries by Query rewriter and executes Mandatory Access Control. In other words, only Security Contexts of a table themselves to be accessed determine that SQL query is executable or not, even the query has passed through any view.

Native PostgreSQL restricts available DB users due to ACL of a view. It means that you cannot describe access paths to a certain table clearly, because you may define multiple views to a table. Moreover, an ACL of a table is not evaluated when accessed from view. That is a reason why a DB user who has an authority to define a view has a powerful authority to refer any information in the database.

Because of this interesting nature of SE-PostgreSQL to test queries after query expansions, one-to-one

access control rule is applied to a table. You may improve the consistency of Access Control Policy with this feature.

2.4 SQL Query Checks

SE-PostgreSQL scans a given SQL query and checks whether the client has a proper authority to the DB object to be accessed. It stops SQL query execution and aborts the current transaction immediately if it is violated.

Suppose the SQL query as follows.

```
SELECT name, price, weight FROM product;
```

This simple query accesses a table `product` and columns `name`, `price`, and `weight`. Because SE-PostgreSQL checks that the client has proper authorities to these objects, the client has to have a `table:select` permission to the product and `column:select` to the `name`, `price`, and `weight`.

Go "3.Database Object and Permissions" and see the list object classes and permissions to be referred from SE-PostgreSQL.

Next example is a little bit complicated, including calculation and `WHERE`.

```
SELECT name, 1.05 * price FROM product WHERE weight > 500;
```

This query will access the table `product` and the column `name`, following accesses to the `price` column during calculation and the `weight` column by `WHERE` clause. As a result, the client is required to have a `table:{use select}` permission to the `product` table, a `column:select` permission to `name` and `price` column, and `column:use` permission to the `weight` column.

In addition, this query includes '*' and '>' operators. Because these operators are implemented as functions inside PostgreSQL, SE-PostgreSQL checks authorities to execute to those functions. Thus, the client must have `procedure:execute` permission to these functions.

See another example. This query comes with update.

```
UPDATE product SET price = 1.20 * price, name = 'rice' WHERE id = 51;
```

This query accesses a table `product` and columns `price`, `name`, and `id`, however, aims of accesses to these columns varies with each.

For instance, the `price` column is subject to update, as well as used for value calculation. It means the client is required to have `column:{select update}` permission to the `price` column. The `name` column is just a subject to update, `column:update` permission is required. While the `id` column is not subject to update, you need `column:select` permission because referenced by a `WHERE` clause.

Note that this SQL query does referring even it has an `UPDATE` statement, because of referring to the `price` and `id` column. The client is needed to have `table:{use select update}` authority to the `product` table.

```
DELETE FROM product RETURNING *;
```

This SQL query will delete all tuples in the `product` table, and return deleted tuples to the client. Since

‘*’ is indicated, every column in the `product` table will be returned.

The `product` table is the target for deletion, as well as for reference by `RETURNING` clause. That requires the client to have a `table:{select delete}` permission for the `product` table. Besides that, designating ‘*’ is equivalent to designating all columns in the table. The client must have `column:{select}` permission for all columns in the `product` table.

Query inspection is also executed when using DDL statements like as these DML syntaxes. Below is an example of renaming a table, showing an update of table metadata.

```
ALTER TABLE product RENAME TO old_product;
```

The client has to have a `table:setattr` permission to the table. Moreover, it will be required a `table:setattr` permission to update this metadata because PostgreSQL generates a table type, which corresponds a table inside the system.

Things will be a little bit more complicated when creating a lot of DB objects simultaneously such as using `CREATE TABLE` statement.

Here is one of the simplest table definition syntax. The client needs `table:create` permission for the `simple_tbl` table and `column:create` permission for columns `x` and `y`.

It is implicitly given based on the security policy that the security context of newly created tables and columns. SE-PostgreSQL access controls are executed based on these security contexts.

```
CREATE TABLE simple_tbl (
   x  integer,
   y  varchar(32)
);
```

The client is required to have any sufficient authority to every DB object to be subject: TOAST table is to be generated when using TEXT type, an index would be automatically generated when primary key constraint given, or so.

2.5 Rewriting SQL Query

In the inspection phase of SQL queries, it will check an authority to the DB objects, which are accessed by that query. But you cannot check whether access permission to the tuples is sufficient or not until the query has been executed. That is because you cannot determine which tuple is accessed by the query before the query is actually done.

SE-PostgreSQL rewrites a part of a SQL query to provide row-level access control. It means that SE-PostgreSQL adds conditional phrase to filter violated tuples, which the client does not have requisite authority, from result sets or from the scope of UPDATE/DELETE.

See examples below. An additional condition as applied to WHERE clause by SQL query rewriting of SE-PostgreSQL.

```
BEFORE:
   SELECT * FROM product WHERE price > 500;
ALTER:
   SELECT * FROM product WHERE price > 500 AND sepsql_tuple_perms(...);
```

`sepgsql_tuple_perms()` is a function to check whether the client has requisite permissions to

tuples, or not. It returns true if it has, or returns false if not.
Non-accessible tuples for the client will be filtered from a result set compulsory.

Result set filtering is also enforced when accessing to any other table, like use of JOIN clause or subquery, making the client refer only subsets of tuples included the table.

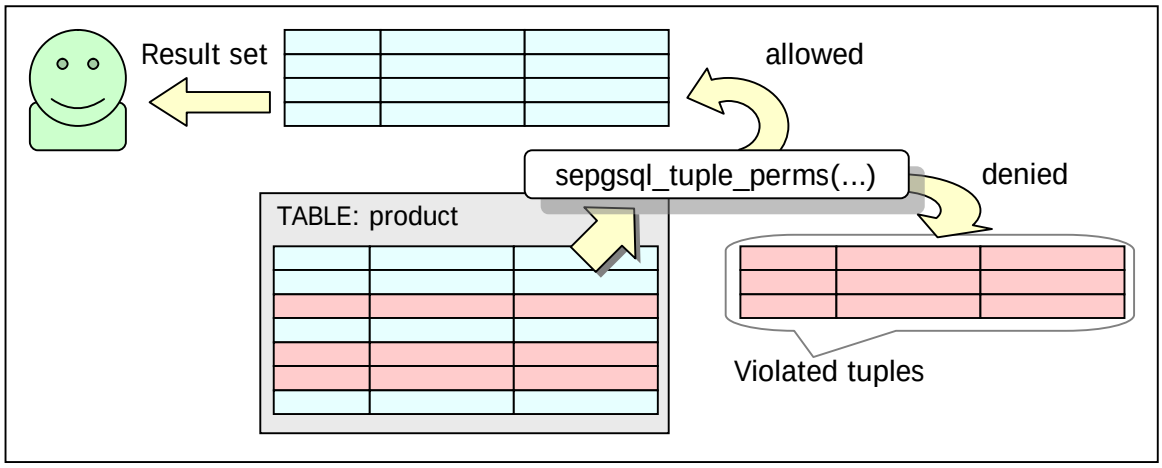


Figure. 5 Filtering violated tuples

Not only reference, but also update or deletion, violated tuples are filtered. To avoid update operation to the non-accessible tuples for the client, SE-PostgreSQL rewrites UPDATE statement as below:

```
BEFORE:
    UPDATE product SET price = 1.05 * price;
AFTER:
    UPDATE product SET price = 1.05 * price WHERE sepgsql_tuple_perms(...);
```

2.5.1 Special case: UNIQUE constraint

A pair of columns with UNIQUE constraint does not allow two or more different tuples have the same value set.
With row-level access control of SE-PostgreSQL, any tuple on which the client doesn't have sufficient authorities will be filtered from the result set. On first glance, the client may seem newly inserted/updated values meets UNIQUE constraint.
Nevertheless, UNIQUE constraint works to meet uniqueness to tuples in the actual table, independently from row-level access control of SE-PostgreSQL. As a result, invisible tuple for the client and newly inserted/updated values may be duplicated, then the operation will be violated to UNIQUE constraint. Theoretically, repeating these operations enables to estimate the contents of invisible tuples.
This is a limitation of the current version of SE-PostgreSQL. To improve it, we have a plan to support Polyinstantiation table in the future version.

2.5.2 Special case: Foreign Key Constraint

Foreign key constraint defines a referential integrity between two tables. In PostgreSQL, a referenced side must be the primary key.

There are two points to remember when using foreign keys under row-level access control of SE-PostgreSQL. One is a visibility point of view about primary key value of the referenced when foreign key inserted or updated. The other is a point of view about action to foreign key value when primary key updated or deleted.

When foreign key is inserted or updated, primary key of the referenced table must be accessible from the client. It means the client must have not only an insertion or update permission to the table with foreign key, but also a reference permission to the table with primary key.

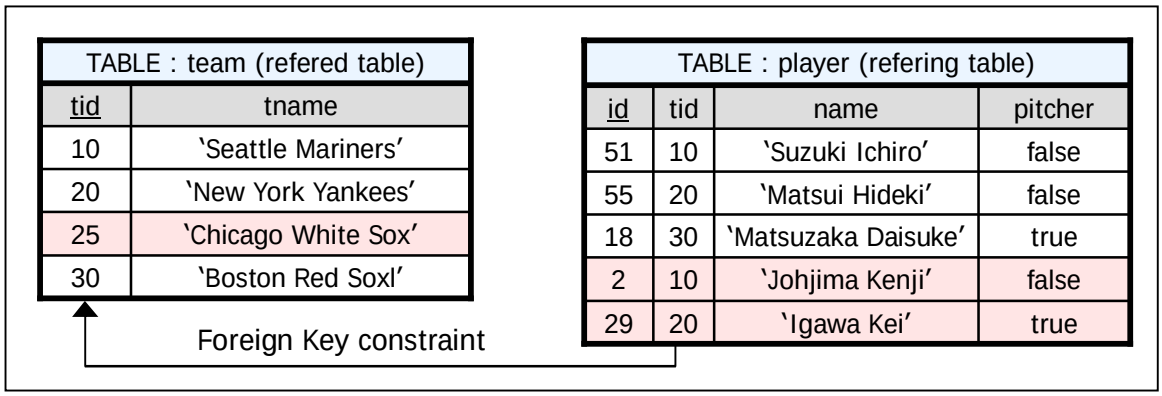


Figure. 6 An Example of Foreign Key Constraint

Figure. 6 is an example of foreign key constraint. You see here that foreign key constraint of `tid` of `player` table is set to `tid` of `team` table, which is the primary key of the table.

In this situation, suppose that a client, which is not accessible to tuples shown in pink, inserted a tuple shown below into the `player` table.

```
{id=15, tid=25, name='Iguchi Tadahito', pitcher=false}
```

Inserted value to `tid` of `player` table must exist in `tid` of `team` table. There is a tuple that has `tid`=25 in `team` table indeed, however, the client does not have an authority to access this tuple. Therefore, new tuple insertion will fail.

In contrast, when a mighty client, which is accessible to any tuple, inserted the same way to `player` table, new tuple insertion will succeed because the client can access a tuple with `tid`=25 of `team` table.

Foreign key constraint of PostgreSQL has a feature to update or delete referrer foreign key when referred primary key value is updated or deleted. This is an option activated ON UPDATE or ON DELETE of foreign key constraint. It keeps a foreign key value properly corresponding update or deletion of a primary key value with `CASCADE/SET NULL/SET DEFAULT` action.

For more details about options of foreign key constraint, see "5.3.5 Foreign Key" section of a proper version of PostgreSQL document.

SE-PostgreSQL independently and individually checks access permission needed for update or deletion of referred primary key value, and access permission needed for update or deletion, associated with the former one, of referrer foreign key value.

When foreign key is updated or deleted, associated with primary key value update or deletion, the client must have a proper authority to every tuple, which is subject for update or deletion. If update or deletion of one or more tuple is denied, SE-PostgreSQL will stop query execution immediately, and abort the current transaction.

This is a special case to preserve reference consistency. When foreign key value is updated or deleted, SE-PostgreSQL prevents from not being compliant to reference consistency constraint, not by excluding a violated tuple, but by aborting a whole transaction.

2.5.3 Special case: OUTER JOIN

SE-PostgreSQL rewrites SQL query specially when joining tables with any OUTER JOIN.

A result set of LEFT OUTER JOIN contains at least one tuple belonging left table. This is not restricted by join condition with ON clause. In other words, you cannot filter out a tuple in left table from a result set by simple condition clause replacement. You can say the same way for RIGHT OUTER JOIN and FULL OUTER JOIN.

SE-PostgreSQL rewrites a reference to table within OUTER JOIN clause by subquery clause with a conditional clause for row-level access control, to prevent from violated accesses to tuples.

```
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.id = t2.id;
```

A simplest LEFT OUTER JOIN shown above will be rewritten by SE-PostgreSQL as below. A reference to left table is transformed subquery with conditional clause, and a tuple that the client does not have proper access permission is not subject for JOIN.

```
SELECT * FROM
       (SELECT * FROM t1 WHERE sepgsql_tuple_perms(...)) AS t1
       LEFT OUTER JOIN t2
       ON t1.id = t2.id and sepgsql_tuple_perms(...);
```

For consistent access, SE-PostgreSQL rewrites SQL query to handle the case when system columns within OUTER JOINed relation are referred.

2.6 Trusted Procedure

SE-PostgreSQL may cause a client domain transition with function execution. We call the entry point function of domain transition as Trusted Procedure.

Domain transition is one of the concept of SELinux. See 4.2.1 Domain Transition for more details. The security context of the client is changed based on security policy during execution of Trusted Procedure. That allows access to DB object which is not reachable directly, if it is from inside Trusted Procedure.

This concept looks like SetUID of OS or Security Invoker of SQL. Domain transition is not related with the owner of DB object, however, and is defined by the Security Policy.

Let us introduce you an example of leveraging Trusted Procedure. Here is a customer table shown below. Contents of `password` column are used for authentication, while we would like to prohibit direct access to the `password` column by the client.

customer テーブル

id	name	email	password
11	'KaiGai'	kaigai@example.com	'aaa'
12	'tak'	tak@example.net	'bbb'
13	'ymj'	ymj@example.org	'ccc'

In this situation, defining a SQL function, which returns authentication result using `password` column, and setting it Trusted Procedure, you may authenticate using `password` column without direct access to the `password` column by the client.

You can do a Trusted Procedure setting with authentication by `customer.password` as follows.

First, define SQL function `check_customer_password` function takes `customer.id` and authentication token as arguments and returns boolean whether those are matched or not.

```
CREATE or REPLACE FUNCTION check_customer_password (integer, text)
        RETURNS bool LANGUAGE 'sql'
        as 'SELECT password = $2 FROM customer WHERE uid=$1';
```

Second, change Security Context of Trusted Procedure and set it entry point.
`sepgsql_trusted_proc_t` is set as an entry point of Trusted Procedure by default.

```
ALTER FUNCTION check_customer_password (integer, text)
        CONTEXT = 'user_u:object_r:sepgsql_trusted_proc_t';
```

When you execute SQL queries including Trusted Procedure, you will get results as below. Note that the client gets enough information to authenticate, while it is still not able to read `password` column.

```
kaigai=# SELECT * FROM customer WHERE id = 11;
ERROR:  SELinux: denied { select } scontext=user_u:system_r:initrc_t
tcontext=user_u:object_r:sepgsql_secret_table_t tclass=column name=password
```

First, the client is denied to access by SE-PostgreSQL, trying to access to every column including `password` column.

```
kaigai=# SELECT id, name, email FROM customer WHERE id = 11;
 id |  name  |       email
----+--------+-------------------
 11 | KaiGai | kaigai@example.com
(1 row)
```

Second, it removed `password` column from subject to SELECT, and got a result set. There is no `password` column.

```
kaigai=# SELECT id, name, email, check_customer_password(id, 'bbb') FROM customer;
 id |  name  |       email        | check_customer_password
----+--------+--------------------+-------------------------
```

```
 11 | KaiGai | kaigai@example.com | f
 12 | tak    | tak@example.net    | t
 13 | ymj    | ymj@example.org    | f
 14 | erina  | erina@example.mil  | f
(4 rows)
```

Third, it called Trusted Procedure. Authentication result is returned to the client by `check_customer_password` function, while the content in itself of `password` column is not leaked out.

3 Database Object and Permissions

In this chapter, we explain how SE-PostgreSQL controls accesses to each DB object variant. General information of SELinux access control is introduced in “4.Security Design of SELinux”. Please read them altogether.

3.1 Object Classes and Permissions

As access methods differ from a table and a function, a DB object has each unique character. SELinux describes this character as an object class and a group of permissions (access vector) associated with that object class.

To describe security policy of SE-PostgreSQL, 6 object classes and fifty-eight permissions, associated with them, are prepared. This table is the list.

database	table	procedure	column	blob	tuple
create	create	create	create	create	relabelfrom
drop	drop	drop	drop	drop	relabelto
getattr	getattr	getattr	getattr	getattr	use
setattr	setattr	setattr	setattr	setattr	select
relabelfrom	relabelfrom	relabelfrom	relabelfrom	relabelfrom	update
relabelto	relabelto	relabelto	relabelto	relabelto	insert
access	use	execute	use	read	delete
install_module	select	entrypoint	select	write	
load_module	update		update	import	
get_param	insert		insert	export	
set_param	delete				
	lock				

List. 1 Object Classes and Permissions

Ideally speaking, any kind of DB object should have each unique object class. In PostgreSQL, however, there are DB objects that do not have unique object classes, such as trigger or database role.

Access controls to those objects are handled as operations on special purpose tables called as system catalog, and are evaluated using table, column and tuple permissions.

3.2 Common access control for each Object classes

DB objects except tuple class have some common operations: creating and deleting DB object (`create` and `drop`), getting and setting metadata (`getattr` and `setattr`), and changing security attributes (`relabelfrom` and `relabelto`).

With tuple, only changing security attribute is defined. That is come from that create and drop are equivalent insert and delete, and it does not have its own metadata. We explain common permissions of

these DB objects in this section.

create permission

When CREATE statement like CREATE TABLE creates a DB object, new DB object is associated with security context implicitly. The client is required to have `create` permission to this new DB object.

Note that one SQL query may create one or more DB objects. A CREATE TABLE statement creates a table and multiple columns, for instance. A large object is created by `lo_create()` function, evaluating create permission in this time.

drop permission

When DROP statement like DROP TABLE deletes a DB object, the client is required to have a `drop` permission to the target DB object.

Note that one SQL query may delete one or more DB objects. A DROP TABLE statement deletes not only a table but also any column inside that, for instance. A large object is deleted by `lo_unlink()` function, evaluating drop permission.

getattr permission

As mentioned above, DB objects except tuple are expressed as tuples stored in the special table called System Catalog. You may refer metadata of DB objects by executing SELECT to the System Catalog.

The client must have getattr permission to the referred DB object at that time.

setattr permission

The client must have setattr permission to the target DB object when updating metadata of DB object with ALTER statement such as ALTER TABLE.

Note that even ALTER statement may be handled as creation/deletion of DB objects. ALTER TABLE tblname ADD colname ... ; statement means creating new columns, for instance.

relabelfrom/relabelto permission

You may change security context of DB object with extended ALTER statement or UPDATE to `security_context` column. Evaluated `relabelfrom` and `relabelto` permission then, the client must have `relabelfrom` permission to the security context before DB object changed, and `relabelto` permission to the security context after DB object changed.

3.3 Access Control to Database

Database object class contains a permission set for database itself and five unique ones.

Implicit Security Context

New database type inherits the domain of SE-PostgreSQL server process, when creating a new database with CREATE DATABASE statement. In addition, you may describe type transition rules targeted a domain of SE-PostgreSQL server process.

It gives `sepgsql_db_t` type to database itself, due to type transition rule in default security policy.

access permission

The client must have `access` permission to the destination database when connecting. This is

minimum required permission requirement to connect SE-PostgreSQL.

install_module permission

When installing Dynamic Link Library(DLL), the client must have `install_module` permission both to the database and to the DLL file.

Note that DLL is not only explicitly loaded by LOAD statement, but also implicitly loaded by defining function implemented in the DLL.

When the client working in `staff_t` domain is connecting to `sepgsql_db_t` type database, loading `lib_t` type DLL, it requires security policy as below, for example.

```
allow staff_t sepgsql_db_t : database install_module;
allow staff_t lib_t : database install_module;
```

load_module permission

When Dynamic Link Library is loaded into database, the database must have `load_module` permission for DLL file. It is the only permission which client is not dealt as a subject.

The difference from `install_module` permission is that evaluation may be done several times whenever DLL is loaded into databases.

The following is a sample of security policy.

```
allow sepgsql_db_t lib_t : database load_module;
```

get_param permission

To refer on-time parameters by SHOW statement, the client must have `get_param` permission to the connected database.

This permission is evaluated on the database so that you cannot set to permit or prohibit each on-time parameters.

set_param permission

To set on-time parameters by SET statement, the client must have `set_param` permission to the connected database.

This permission is evaluated on the database so that you cannot set to permit or prohibit each on-time parameters.

3.4 Access Control for Table

Table object class contains an permission set for table and six unique ones.

Implicit Security Context

When you create new table with CREATE TABLE statement, type of new table is inherited from that of the database.

It gives `sepgsql_table_t` type to new table with type transition rule by default security policy.

use permission

The client must have `use` permission to tables including columns referred from conditional clause of SQL statement (WHERE, JOIN~ON, or HAVING clauses), GROUP BY clause, or ORDER BY clause.

In UPDATE statement as below, for example, it requires not only update permission to `drink` table,

but also `use` permission. That is because `id` column of `drink` table is referred from WHERE clause.

```
UPDATE drink SET price = 120 WHERE id = 4;
```

select permission

To refer a table with SELECT statement or to dump table contents with COPY TO statement, the client must have select permission to the targeted table.

Moreover, it evaluates this permission when; using a formula including reference to a table in order to calculate updated value with UPDATE statement, or returning execution results of updating SQL statement with RETURNING clause.

```
DELETE FROM drink RETURNING *;
```

Difference from `use` permission is `select` permission is evaluated, when the client reads table contents somehow and it returns the content to the client. It evaluates `use` permission when the table content is not returned to the client such as conditional clause or ORDER BY clause.

update permission

The client must have `update` permission to the targeted table when updating a table with UPDATE statement.

insert permission

The client must have `insert` permission to the targeted table when inserting tuple into a table with INSERT statement, or restore it with COPY FROM statement.

delete permission

The client must have `delete` permission to the targeted table when deleting tuples from the table with DELETE statement, or remove the table contents with TRUNCATE statement.

lock permission

The client must have `lock` permission to the client when acquiring table lock explicitly with LOCK statement.

3.5 Access Control for Column

Column object class contains an permission set to columns, including four unique ones.

Implicit Security Context

New column type inherits the type of table where columns belong. When creating a new column with CREATE TABLE or ALTER TABLE statement. In addition, you may describe type transition rules of column class targeting table type.

use permission

The client must have `use` permission to tables including columns referred conditional clause of SQL statement (WHERE, JOIN-ON, or HAVING clauses), GROUP BY clause, or ORDER BY clause.

In SELECT statement as below, for example, it requires not only select permission to `id` and `name` columns, but also `use` permission to `alcohol` and `price` column. That is because `alcohol` column by WHERE clause and `price` column by ORDER BY clause are referred each.

```
SELECT id,name FROM drink WHERE alcohol = true ORDER BY price;
```

select permission

The client must have `select` permission to the targeted column when referring columns.

It is the most typical case that there is a column or a formula including a column in the target list of SELECT statement. Note that the client will be evaluated for select permission to the targeted column with;

- Specifying as RETURNING clause of INSERT, UPDATE, or DELETE statement
- Dumping table with COPY TO statement
- Specifying columns as function arguments

Suppose executing SQL query as below. It requires not only select permission to `uid` and `uname` columns, but also select permissions to `birthday` column.

```
SELECT uid, uname, age(birthday) FROM person;
```

Difference from `use` permission is `select` permission is evaluated, when the client reads column contents somehow and it returns the content to the client. It evaluates `use` permission when a column is used by conditional clause or GROUP/ORDER BY clause. The table content is not returned to the client with those clauses.

update permission

The client must have `update` permission to the targeted column when updating a table with UPDATE statement. It does not evaluate columns that are not subject for update.

insert permission

The client must have `insert` permission to columns which is specified the value explicitly, when inserting a new tuple with INSERT statement.

It does not evaluate `insert` permission to columns which is not specified the value in the new tuple. NULL or default value is set to these fields.

3.6 Access Control for Tuple

Tuple object class contains an permission set to tuples and seven unique ones.

Implicit Security Context

New tuple type inherits the type of table where tuples belong, when you insert a new tuple with INSERT statement or COPY FROM statement. In addition, you may describe type transition rules of tuple class targeting table type.

Note that it does not treat INSERT into a part of System Catalog like `pg_class` or `pg_proc` as an operation to tuple class. That is because tuple insertion to those System Catalogs is equivalent creation of tables or functions. These operations are treated as create of counterpart object classes.

relabelfrom/relabelto permission

You may change security context of a tuple with UPDATE to `security_context` column. Evaluated `relabelfrom` and `relabelto` permission then, the client must have `relabelfrom` permission to the security context before tuple changed, and `relabelto` permission to the security context after tuple changed.

use permission

The client must have `use` permission to tables including columns referred conditional clause of SQL statement (WHERE, JOIN~ON, or HAVING clauses), GROUP BY clause, or ORDER BY clause.

Tuples that the client does not have `use` permission will be filtered from result set or subject for update /deletion.

select permission

A client must have `select` permission to the targeted tuple when; reading tuples with SELECT statement, returning tuples that are subjects for updating queries with RETURNING clause, or dumping tables with COPY TO statement.

Tuples that the client does not have `select` permission will be filtered from the result set.

update permission

The client must have update permission to the targeted tuple when updating a tuple with UPDATE statement.

Tuples that the client does not have select permission will be filtered from the result set.

insert permission

The client must have `select` permission to the targeted tuple when inserting a new tuple with INSERT statement, or restoring tables with COPY FROM statement.

It gives security context of new tuple implicitly, however, you can specify security context explicitly using INSERT statement with value-set `security_context` column. You may not insert tuples without `insert` permission, in anyway.

delete permission

A client must have `delete` permission to the targeted tuple when deleting tuples with DELETE statement or TRUNCATE statement.

Note the fact that TRUNCATE statement has no benefit in SE-PostgreSQL. TRUNCATE statement will be replaced as unconditional DELETE statement internally, keeping tuples without `delete` permission inside the table.

3.7 Access Control for Function

Procedure object class has an permission set for functions and two unique ones.

Implicit Security Context

New function inherits the type of database when creating a new function with CREATE FUNCTION statement. In addition, you may describe type transition rules of `procedure` class targeting `database` type.

It gives newly created function `sepgsql_proc_t` type by the default security policy.

execute permission

A client must have `execute` permission for function when executing it contained with SQL queries. PostgreSQL implements operators as SQL functions. It executes int4eq function to operate comparison between four bytes integers, for example. A client must have execute permission for the function

though.

Meanwhile, PostgreSQL calls SQL functions for its internal processing. It does not evaluate execute permission for the function at that time.

entrypoint permission

A client must have `entrypoint` permission for functions when executing functions defined as Trusted Procedure. For more details about Trusted Procedure, see "2.6 Trusted Procedure".

Trusted Procedure execution causes a domain transition. Therefore, the client must have `transition` permission of `process` object class for the destination domain of transition.

Here is the description of Security Policy to set Trusted Procedure.

```
allow <client domain> <procedure type>
            : procedure { execute entrypoint };       ... (1)
allow <client domain> <new domain}
            : process { transition };                 ... (2)
type_transition <client domain> <procedure type>
            : process <new domain> ;                  ... (3)
```

The client has ability to execute Trusted Procedure with (1). (2) allows the client to transit for <new domain>. Actual domain transition occurs when Trusted Procedure executed with (3).

3.8 Access Control for Binary Large Object

Blob object class has a permission set for binary large objects and four unique ones.

Since native PostgreSQL does not provide any access control for them, SE-PostgreSQL is the only provider for that.

Implicit Security Context

The type of new binary large object inherits database type when creating new large object with `lo_create()` function. You may describe type transition rules of `blob` class targeting `database` type.

It gives `sepgsql_blob_t` type for newly created binary large object in the default Security Policy.

read permission

A client must have `read` permission for a binary large object when reading it with `loread()` function. It is just same as referring `pg_largeobject` system catalog directly.

write permission

A client must have `write` permission for a binary large object when writing it with `lowrite()` function. It is just same as updating `pg_largeobject` system catalog directly.

import/export permission

`lo_import()` function reads a specified file from OS filesystem and stores as a binary large object. `lo_export()` function write a specified binary large object on OS filesystem with specified name. Note that the argument file name is not on the client OS, but on the server OS.

With these two functions, it does not read/write data directly between the client and the DB object. It is nothing but server process of SE-PostgreSQL reads/writes actual files on the OS. SELinux controls these read/write operations with that on the kernel level.

A client can start these operations to the server process with `lo_import()` or `lo_export()`. This is the matter between the client and the server process of SE-PostgreSQL.

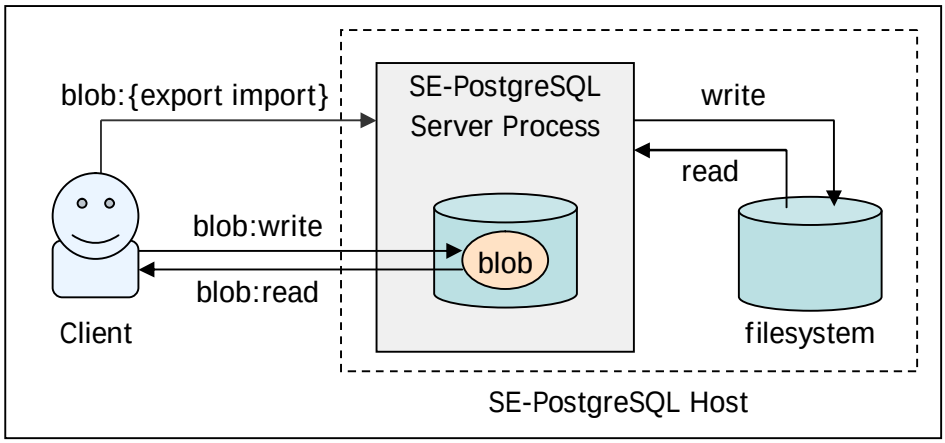


Figure. 7 import/export of binary large object

`import`/`export` permission works as follows.

A client must have `import` permission for server process of SE-PostgreSQL when importing a file into binary large object calling `lo_import()` function. Also, a client must have `export` permission for server process of SE-PostgreSQL when exporting binary large object calling `lo_export()` function.

4 Security Design of SELinux

In this chapter, we describe a generic explanation of the access control of SELinux, not limited to SE-PostgreSQL.

4.1 Security Contexts

A security context is a character string expressed in the following form, and includes all of the information required by SELinux for access control. In a system configuration with SELinux, all objects such as processes, files and sockets are assigned a security context. The same is also true for DB objects under SE-PostgreSQL control.

The different fields of a security context can be separated into 4 parts using ":". These fields signify the "user", "role", "type (domain)" and "MLS label".

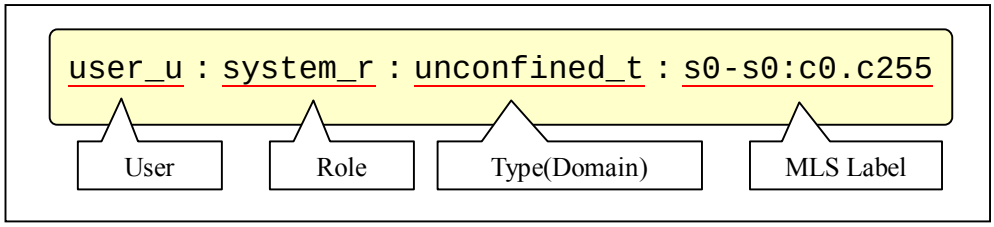


Figure. 8 Security Context

User corresponds to a user of the OS. "`user_u`" is used as a substitute when a user has not been defined in the security policy.

Role is an identifier used in RBAC, described below. It has meaning only in the security context for a process, and otherwise is uniformly set to "`object_r`".

Type is an identifier used in TE, described below. A type assigned to a process is specially called a domain, and is distinguished from other objects.

The MLS label is an identifier used in MLS/MCS, described below. When the mcstrans service has started, it is possible to display the MLS label using a meaningful alias such as "`SystemHigh`" etc.

SELinux provides access control using TE, MLS/MCS and RBAC, and these are all performed based on the security contexts of processes, and the security contexts of the objects accessed by them. The following section describes each access control mechanism of SELinux.

4.2 TE(Type Enforcement)

Type enforcement is an access control mechanism comprised of three factors: the process domain and action, and the type of the object accessed by the process. This is the most important access control mechanism in SELinux.

The security policy lists which domains can execute, what kind of actions on what types, and all execution of actions which have not been explicitly permitted is refused.

Here the term action refers to the combination of an object class and access vectors. For example, in the case of a read action on a file, this becomes "`file:read`" and, if a TCP socket is waiting for a

connection, it becomes "`tcp_socket:accept`". The access vector type varies depending on the applicable object class. Sometimes a process becomes the object of an action. For example, for sending a signal, it becomes "`process:kill`".

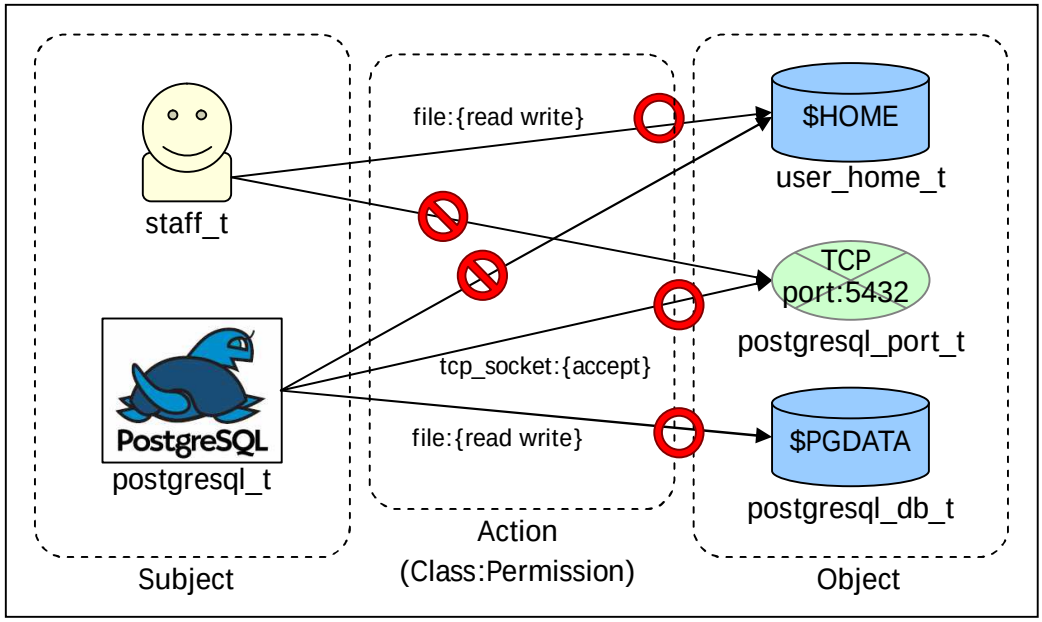


Figure. 9 Type Enforcement

In the security policy, access control rules based on TE are described as follows.

```
allow postgresql_t postgresql_db_t : file { read getattr };
```

This sort of access control policy must be described beforehand for all items required by an application.

However, if this sort of description is done point-by-point, it makes description of the security policy complicated and maintenance becomes difficult. Therefore, a set of macros modularized at a high level and called the Reference Policy is provided by the SELinux community. Using this, the developer of a security policy can develop a highly readable, modularized security policy.

4.2.1 Domain Transition

If special setting is not done, a process inherits the domain of its parent process.

That is, if a process operating in the `unconfined_t` domain starts a child process, the child process too will operate in the `unconfined_t` domain, and if a process operating in the `staff_t` domain starts a child process, the child process will operate in the `staff_t` domain.

However, with this approach alone, all processes will operate with exactly the same authorities, and it will be impossible to perform meaningful access control. Domain transition is used to solve the matter.

Domain transition is a mechanism which determines the domain where a new process will operate based on the process domain and type of executable binary file. Using this, it becomes possible for a child process to operate in a domain different than the parent process.

In the security policy, a domain transition can be described as follows.

```
TYPE_TRANSITION <source domain> <target type> : process <dest domain> ;
```

For example, if a system init script starts Apache, and Apache will operate in the `httpd_t` domain, then a description like the following will be necessary.

```
TYPE_TRANSITION initrc_t httpd_exec_t : process httpd_t ;
```

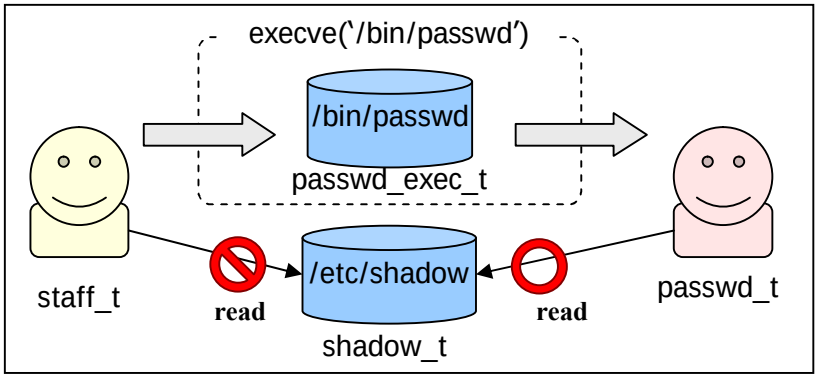


Figure. 10 Domain Transition

Figure. 10 shows a conceptual diagram of a typical domain transition. It is impossible to directly access the `/etc/shadow` file from the user shell, but by executing the `/bin/passwd` command, a transition is made to the `passwd_t` domain, and it becomes possible to access `/etc/shadow`.

This is based on the approach of permitting access to the password file only via a safe procedure, i.e. the `/bin/passwd` command. This idea resembles SetUID in traditional UNIX, but there are differences, such as the fact that it is possible to obtain different results depending on the domain transitioned from (source domain), and adopt more constrained privileges for the domain transitioned to (destination domain).

In Linux, all processes are generated taking `/sbin/init` as the root, but these processes are executed with the proper privileges via repeated domain transitions based on the security policy.

4.3 RBAC(Role Based Access Control)

In SELinux, RBAC is for controlling the domain transitions described above.
We can associate an arbitrary number of domains with a role in the security policy. The ability of a process to transition via a domain transition is constrained only to domains in the scope related to the role of a process. Note that the role of a process is invariant before and after a domain transition.

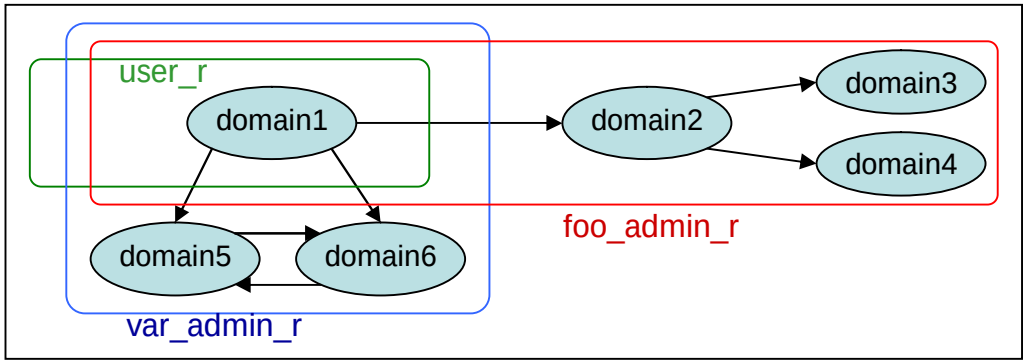


Figure. 11 Conceptual diagram of RBAC

Figure. 11 is a conceptual diagram of RBAC. From `domain1`, domain transition is possible to `domain2`, `domain5` and `domain6`, but if the process belongs to the `user_r` domain, domain transition cannot be

done. Also, if it belongs to `foo_admin_r`, domain transition to `domain5` and `domain6` cannot be done, and if it belongs to `var_admin_r`, domain transition to `domain2` cannot be done.

By making these sorts of constraints, it is possible to establish administrators who are limited only to services in part of the system. However, for that reason, the domains must be partitioned with a sufficiently fine grain for each service provided by the system.

4.4 MLS(Multi Level Security) & MCS(Multi Category Security)

MLS is a mechanism which provides access control based on the traditional Bell-La-Padulla model.

Processes and objects such as files are assigned a sensitivity based on vertical relationships, and categories based on inclusion. Access control is performed, based on the following rules, using the sensitivity and categories of the process and object.

1. When a process reads information from an object, the sensitivity of the process must be the same or higher than the sensitivity of the object.
2. When a process reads information from an object, the categories of the process must completely subsume the categories of the object.
3. When a process writes information to an object, the sensitivity and category of the process must be the same as those of the object.

By applying these rules, it is possible to forbid the release of high-sensitivity information to low-sensitivity domains and domains with unrelated categories.

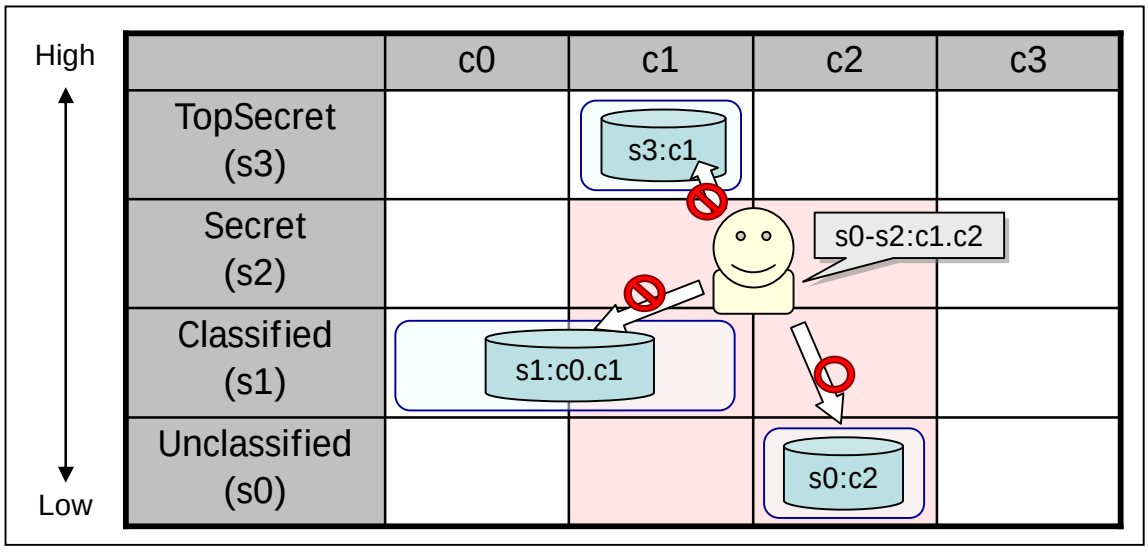


Figure. 12 Conceptual diagram of MLS

Figure. 12 is a conceptual diagram of access using MLS. When the MLS label of a process is "`s0-s2:c1.c2`", the process dominates the sensitivities and categories indicated in pink.

The process can access a "`s0:c2`" file. However, it cannot access an "`s3:c1`" file because its sensitivity is higher than that of the process. Similarly, the process cannot access an "`s1:c0.c1`" file because it is not completely subsumed by the category of the process.

MCS is a simplification of MLS, and it uses only a single sensitivity. Therefore, access control is not

performed depending on sensitivity, and is only performed using the inclusion relationship of categories.

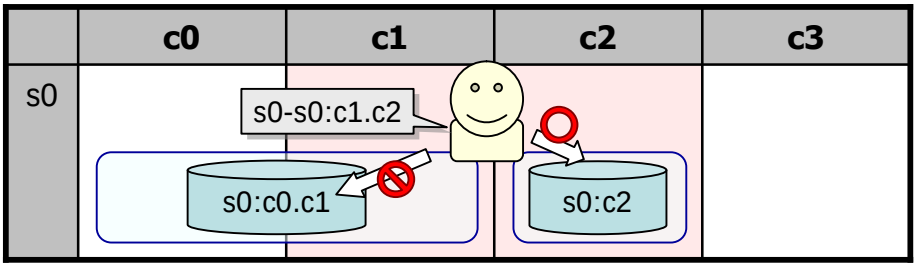


Figure. 13 Conceptual diagram of MCS

Figure. 13 is a conceptual diagram of access control using MCS. If the MLS label of the process is "`s0-s0:c1.c2`", the process dominates the categories indicated in pink.

The process can access the "`s0:c2`" file. However, the process cannot access the "`s0:c0.c1`" file because it is not completely subsumed by the category of the process.

4.5 SELinux Customization

4.5.1 Enforcing mode and Permissive mode

SELinux has two operation modes: enforcing mode and permissive mode.

Mandatory access control based on the security policy is only performed when SELinux is set to enforcing mode. Security policy evaluation is performed even in permissive mode, but those results are not applied to access control. This feature is used to debug the security policy by checking the access denied log.

The `setenforce` command is used to switch between enforcing mode and permissive mode. The mode at system startup can be set using `/etc/selinux/config`.

4.5.2 Boolean Variables

Using boolean variable, it is possible to enable/disable specific points in the SELinux security policy during system working. Boolean variables take two values, on or off, and can be set using the `setsebool` command. Use the `getsebool` command to acquire a list of condition variables and check their values.

How the individual condition variables customize the security policy depends on the definitions in the security policy. See “5.2.6 boolean variable” for the boolean variables relating to SE-PostgreSQL.

4.5.3 Semanage Command

Using the semanage command, it is possible to set things such as the MLS labels related to the user shell at login, and the MLS label alias when the `mcstrans` service is enabled.

For details, see the manpage on the semanage command.

5 SE-PostgreSQL System Administration

We describe some features to leverage SE-PostgreSQL in this chapter.

5.1 Backup and Restore in SE-PostgreSQL

`pg_dump` and `pg_dumpall` are utilities to backup databases managed by PostgreSQL. Those are extended to enable to backup with security context.

DBA must have authority to refer any target database objects within the specified database under backup process. If they don't has enough authority on those objects, generated dump image will be incomplete or backup process will abort.

In contrast, DBA must have authority to insert and to update any database object which will be restored. Restoring process will be failed, if their authority is not sufficient.

pg_dump command

`--enable-security` option enables to backup the specified database with security context. The kind of database objects that we can dump them with security context are tables, columns, SQL functions, binary large objects and tuples.

pg_dumpall command

`--enable-security` option enables to backup the whole database cluster with security context. The kind of database objects that we can dump them with security context are databases, tables, columns, SQL functions, binary large objects and tuples.

pg_restore command

`pg_restore` enables to restore the backuped image with security context which is generated `pg_dump` or `pg_dumpall` with –`enable-security` option.

5.2 The default security policy

In this section, we describe the default security policy.

The default security policy is configured to provide maximized compatibility with none-SELinux environment. It intend to avoid that traditional applications cannot work correctly. But you can explicitly assign special purpose security context onto database objects like table and procedure, to apply mandatory access control more efficiently.

There are two kinds of security policy in SELinux, called Targeted and Strict. The `unconfined_t` domain in the Targeted policy and `sysadm_t` domain in the Strict policy has wider authority than any other domain. It is allowed to execute DDL statement and to access confidential database objects, for example. We call them "administrative domain" as a matter of convenience.

We call any other domains "generic domain" in same reason. It is restricted to execute DDL statement and to access confidential database objects.

Some types are already defined in the default security policy. An administrative domain can change the security context of database objects to apply more appropriate access control for characteristics of the target objects.

5.2.1 Domains for Client

As we mentioned above, `unconfined_t` domain in the Targeted policy and `sysadm_t` domain in the Strict policy has wider authorities than any other domain which can connect to SE-PostgreSQL.
We have to setup, backup and restore databases from those administrative domains.

5.2.2 Typs for Table/Column/Type

Any column and tuple implicitly inherits the type of table in which they are contained. Thus, the security context of columns and tuples are same as table's one, if those are not changed explicitly.
The following five types are already defined in the default security policy. You can select the one of them for your purpose.

sepgsql_table_t

It is the default type which is attached implicitly for newly created tables in the default security policy. Any client is allowed any operations except for relabeling onto table, column and tuple with `sepgsql_table_t`. In other word, any client can access them as we use native PostgreSQL.

sepgsql_secret_table_t

Any operations are denied onto table, column and tuple on which this type is attached, except for administrative domain and accesses via trusted procedure.
We can leverage `sepgsql_secret_t` type to store so confidential information like password phrase, credit card number.

sepgsql_ro_table_t

Only SELECT statement is allowed onto tables, columns and tuples on which this type is attached, except for administrative domain and accesses via trusted procedure.
We can leverage `sepgsql_ro_table_t` to protect from manipulating read-only information, like a master table of commodities for example.

sepgsql_fixed_table_t

Only SELECT and INSERT statement are allowed onto tables, columns and tuples on which this type is attached, except for administrative domain and accesses via trusted procedure.
We can leverage `sepgsql_fixed_table_t` to guarantee that once inserted information is not manipulated or destructed. It is useful to store a kind of document which must be kept for a certain period by legal reason, for example.

sepgsql_sysobj_t

In PostgreSQL, meta-information to manage any DB object is stored in special tables called as System Catalog. This type is the default type attached to tuples, columns within them, and system catalog itself.

`sepgsql_enable_users ddl` enables to switch whether generic domain can modify tuples within system catalogs, or not.

5.2.3 Types for Procedure

sepgsql_proc_t

In the default security policy, it is the default type for newly created procedure by administrative domains, when `sepgsql_enable_unconfined` is "on".

Any client can execute procedures with `sepgsql_proc_t`. It does not cause a domain transition.

sepgsql_user_proc_t

In the default security policy, it is the default type for newly created procedure by generic domain.

Generic domain is possible to execute functions with `sepgsql_user_proc_t` type, but administrative domain is NOT possible contrastively.

It means that executing an "untrusted procedure", defined by someone, with unconfined authority often gives us a dangerous result. Administrative domain cannot execute it until he has checked the user defined procedure and changed its security context into `sepgsql_proc_t`.

sepgsql_trusted_proc_t

The functions with this type are called as Trusted-Procedure.

Any client can execute `sepgsql_trusted_proc_t` typed functions, and it cause a domain transition. We can access any database objects in which the Trusted-Procedure provides a way to access, as if we are working with administrative domain.

Trusted-Procedure is a fine feature to restrict the way to access database objects which contains confidential information. But there is a risk to make the security functionality of SE-PostgreSQL nonsense. You should configure Trusted-Procedure with understanding the risk.

5.2.4 Types for binary large object

sepgsql_blob_t

It is the default type which is attached implicitly for newly created binary large object in the default security policy.

Any client can do any operation onto binary large objects with `sepgsql_blob_t`, as if they works with native PostgreSQL.

sepgsql_ro_t

Any write operation are denied onto binary large objects on which this type is attached, except for administrative domain and accesses via Trusted-Procedures.

sepgsql_secret_blob_t

Any operation are denied onto binary large object on which this type is attached, except for administrative domain and accesses via Trusted-Procedures.

We can leverage `sepgsql_secret_blob_t` to store so confidential information like a binary formatted secret document for example.

5.2.5 Types for database

sepgsql_db_t

The only type which is attached onto database in the default security policy.
We cannot associate any other type with databases.

5.2.6 boolean variable

The following four booleans are used to customize the default security policy related to SE-PostgreSQL.

sepgsql_enable_unconfined

We can switch whether an administrative domain is enabled, or disabled. In the case when this boolean is “off”, their authority is restricted to same as generic domain, even if it is an administrative domain.
The default of the boolean is “on”.

sepgsql_enable_users_ddl

We can switch whether an generic domain can execute any DDL statement, or not. In the case when this boolean is “off”, executing DDL statement like “CREATE TABLE” is denied.
The default of the boolean is “on”.

sepgsql_enable_auditallow

We can switch whether allowed-audit logs are generated, or not. In the case when this boolean is “on”, allowed-audit logs are generated. We can confirm what actions are required on what database objects, and what access controls are done, using them.
The default of the boolean is “off”

sepgsql_enable_auditdeny

We can switch whether denied-audit logs are generated, or not. In the case when this boolean is “on”, denied-audit logs are generated. We can confirm any violated database access.
The default of the boolean is “on”.

sepgsql_enable_audittuple

We can switch whether allowed or denied audit logs for tuples are generated, or not.
Generating audit logs for any tuples are depend on `sepgsql_enable_audittuple`, even if `sepgsql_enable_auditallow` or `sepgsql_enable_auditdeny` are enabled.
The reason why those booleans are separated is to avoid a flood of audit logs when we scan a table containing so much number of tuples.
The default of the boolean is “off”.

5.3 Labeled IPsec

Labeled IPsec is an excellent technology to obtain security context of the peer process which communicates over IPsec networks. In the case when a client connect to SE-PostgreSQL via TCP/IP connection, we have to set up Labeled IPsec to determine the security context of the client.
In this section, we describe the way to set up it.

Confirming required packages

You should confirm the following packages are installed:

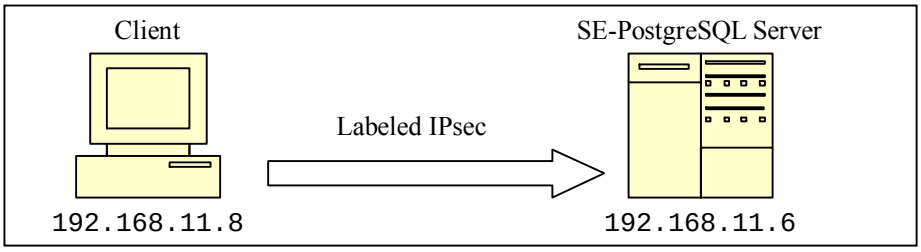
- `kernel`(Labeled IPsec enabled)
- `ipsec-tools-0.6.5-6`, or later

Labeled IPsec enabled kernel have to be built with CONFIG_SECURITY_NETWORK_XFRM=y. It is enabled on Red Hat Enterprise Linux 5, Cent OS 5, Fedora 7 and recent updated Fedora core 6.

A key exchange process is extended to deliver the security context of server and client to its peer. Those extensions are added with `racoon` which is a key exchange server contained in `ipsec-tools`.

Assumption in the sample environment

In the following configuration example, 192.168.11.6 is used for the server side IP address on which SE-PostgreSQL works, and 192.168.11.8 is used for the client side IP address on which a client connects to the server.



Adding a policy to SPD(Security Policy Database)

You have to set up the following configuration to encrypt the communication between client and server and to deliver each peer's security context in a key exchanging process.

1. Make SPD definition files for each side. You should pay mention that the description of IP addresses are reversed in the client side.

```
[Server side (192.168.11.6) configuration]
spdadd 192.168.11.6 192.168.11.8 any
-ctx 1 1 "system_u:object_r:unlabeled_t:s0"
-P out ipsec
esp/transport//require;

spdadd 192.168.11.8 192.168.11.6 any
-ctx 1 1 "system_u:object_r:unlabeled_t:s0"
-P in ipsec
esp/transport//require;
```

2. Run `setkey` command to load the contains of the SPD definition file.

```
# setkey -f <SPD definition file>
```

Configuration for /etc/racoon/racoon.conf

Edit `/etc/racoon/racoon.conf` to setup racoon which is the key exchange server. You have to add some configurations remarked with bold characters. You should pay mention that the description of

IP addresses are reversed in the client side, because it is a server side (192.168.11.6) configuration.

```
[Server side (192.168.11.6) configuration]
# Racoon IKE daemon configuration file.
# See 'man racoon.conf' for a description of the format and entries.

path include "/etc/racoon";
path pre_shared_key "/etc/racoon/psk.txt";
path certificate "/etc/racoon/certs";
sainfo anonymous
{
        pfs_group 2;
        lifetime time 1 hour ;
        encryption_algorithm 3des, blowfish 448, rijndael ;
        authentication_algorithm hmac_sha1, hmac_md5 ;
        compression_algorithm deflate ;
}

remote 192.168.11.8
{
        exchange_mode aggressive, main;
        my_identifier address;
        proposal {
                encryption_algorithm 3des;
                hash_algorithm sha1;
                authentication_method pre_shared_key;
                dh_group 2 ;
        }
}
```

Configurations for /etc/racoon/psk.txt

In this case, we explain the way to encrypt communication pathway by PSK(Pre Shared Key) to simplify the configuration. `/etc/racoon/psk.txt` is used to store PSK of the peer host.
The following configuration is a sample for the server side(192.168.11.6). You should pay mention that the description of IP addresses are reversed in the client side.

```
[Server side (192.168.11.6) configuration]
# file for pre-shared keys used for IKE authentication
# format is:  'identifier' 'key'
# For example:
#
#  10.1.1.1             flibbertigibbet
#  www.example.com      12345
#  foo@www.example.com  micropachycephalosaurus
192.168.11.8            somethingsecrettext
```

Start racoon

start `racoon` with the above configurations.

```
# racoon
```

It enables to encrypt the communication pathway between the server side (192.168.11.6) and the client side (192.168.11.6), and to exchange the security context of both peers on establishing the connection.

6 Appendix

6.1 Extended SQL statement

To leverage the features of SE-PostgreSQL, some SQL statements are extended.
In this section, we will describe them.

6.1.1 CREATE DATABASE statement

Format

```
CREATE DATABASE dbname CONTEXT = 'context'
```

Description

The enhancement of `CREATE DATABASE` statement enables to create database with its security context specified explicitly.

The client must have `database:{create}` permission against to the explicitly specified security context.

Example

```
kaigai=# CREATE DATABASE testdb
     CONTEXT = 'system_u:object_r:sepgsql_db_t';
CREATE DATABASE
kaigai=#
```

6.1.2 ALTER DATABSE statement

Format

```
ALTER DATABASE dbname CONTEXT = 'context'
```

Description

The enhancement of `ALTER DATABASE` statement enables to change security context of database.
The client must have `database:{setattr relabelfrom}` permission onto the target database, and `database:{relabelto}` permission against to the newly attached security context.

Example

```
kaigai=# ALTER DATABASE testdb
     CONTEXT = 'user_u:object_r:sepgsql_db_t:s0:c0';
ALTER DATABASE
kaigai=#
```

Remarks

In the case when the security contexts of the database is not changed with this statement, `database{relabelfrom relabelto}` permissions are not evaluated.

6.1.3 CREATE TABLE statement

Format

```
CREATE TABLE tblname (
     colname <TYPE> [<CONSTRAINT>] CONTEXT = 'column context',
       :
) [<table options>] CONTEXT = 'table context'
```

Description

The enhancement of CREATE TABLE statement enables to create a table with explicitly specified security context of table and column.

The client must have `table:{create}` or/and `column:{create}` permission against to the explicitly specified security context.

Example

```
kaigai=# create table tbl1 (
   id integer primary key
      context = 'system_u:object_r:sepgsql_table_t',
   body text
      context = 'system_u:object_r:sepgsql_secret_table_t'
) context = 'system_u:object_r:sepgsql_table_t:s0:c0';
CREATE TABLE
kaigai=#
```

6.1.4 ALTER TABLE statement

Format

```
ALTER TABLE tblname [ALTER colname] CONTEXT = 'context'
```

Description

The enhancement of ALTER TABLE statement enables to change security context of a table or a column.

The client must have table:{setattr relabelfrom} permission onto the target table, and table:{relabelto} permission against to the newly attached security context.

In the same manner, the client must have column:{setattr relabelfrom} permission onto the target column, and column:{relabelto} permission against to the newly attached security context.

Example

```
kaigai=# ALTER TABLE drink
       CONTEXT = 'user_u:object_r:sepgsql_secret_table_t';
ALTER TABLE
kaigai=#
```

Remarks

In the case when the security context of the table is not changes with the statement, `table:{relabelfrom relabelto}` permissions are not evaluated. In the same manner,

`column:{relabelfrom relabelto}` permissions are not evaluated, if the security context of column is unchanged.

6.1.5 CREATE FUNCTION statement

Format

```
CREATE [OR REPLACE] FUNCTION (<type>,...)
    RETURNS <type> [<options> ...] CONTEXT = 'context'
    [AS '<definition>']
```

Description

The enhancement of `CREATE FUNCTION` statement enables to create a function with its security context specified explicitly.

The client must have `procedure:{create}` permission against to the explicitly specified security context.

Example

```
kaigai=# create or replace function less_than (integer, integer)
       returns bool language 'sql'
       context = 'system_u:object_r:sepgsql_trusted_proc_t'
       ss 'select $1 < $2';
CREATE FUNCTION
kaigai=#
```

6.1.6 ALTER FUNCTION statement

Format

```
ALTER FUNCTION funcname CONTEXT = 'context'
```

Description

The enhancement of `ALTER FUNCTON` statement enables to change the security context of a function.

The client must have `procedure:{setattr relabelfrom}` permission onto the target function, and `procedure:{relabelto}` permission against to the newly attached security context.

Example

```
kaigai=# alter function check_person_passwd(integer, text)
     context = 'user_u:object_r:sepgsql_trusted_proc_t';
ALTER FUNCTION
kaigai=#
```

Remarks

In the case when the security contexts of the function is not changed with this statement, `procedure:{relabelfrom relabelto}` permissions are not evaluated.

6.2 Extended SQL function

Several new functions are provided to leverage SE-PostgreSQL's security features.
In this section, we describe those extended SQL functions.

6.2.1 sepgsel_getcon() function

Definition

```
sepgsql_getcon() returns security_label
```

Description

We provide the function to obtain the security context of the client which connects to SE-PostgreSQL.
`sepgsql_getcon()` returns the current security context. In the case when it is called from inside of Trusted Procedure, it returns the domain-translated security context.

Example

```
kaigai=# select sepgsql_getcon();
              sepgsql_getcon
-----------------------------------------------
 root:system_r:unconfined_t:SystemLow-SystemHigh
(1 row)
```

6.2.2 lo_get_security() function

Definition

```
lo_get_security(Oid loid) returns security_label
```

Description

We provide the function to obtain the security context of a binary large object.
`lo_get_security()` returns the security context of the binary large object identified by loid.
The client must have `blob:{getattr}` permission onto the target one.

Example

```
kaigai=# select lo_get_security(16410);
       lo_get_security
-------------------------------
 user_u:object_r:sepgsql_blob_t
(1 row)
```

6.2.3 lo_set_security() function

Definition

```
lo_set_security(Oid loid, security_label context) returns bool
```

Description

We provide the function to change the security context of a binary large object.

`lo_set_security()` changes the security context identified by `loid`, and returns TRUE, if succeeded.

The client must have `blob:{setattr relabelfrom}` permissions onto the target binary large object, and `blob:{relabelto}` permission against to the explicitly specified security context.

Example

```
kaigai=# select
lo_set_security(16410,'user_u:object_r:sepgsql_secret_blob_t');
 lo_set_security
-----------------
 t
(1 row)
```