The Security-Enhanced PostgreSQL Security Guide

(English edition)

SE-PostgreSQL development team

Change Logs	
2007/07/01	English Edition (beta edition) is published
2007/09/03	The first SE-PostgreSQL Security Guide (English edition) is published
2007/02/14	Chapter.5 Labeled Network related descriptons are updated

Contents

Copyright and License

This is the official document by the Security-Enhanced PostgreSQL development team. It includes useful examples that illustrate the security features provided by the current version of SE-PostgreSQL.

1 Introduction

Security-Enhanced PostgreSQL, or SE-PostgreSQL, provides access control mechanisms based on the SELinux architecture built in PostgreSQL, a popular open-source relational database system. The result is an innovation in database security.

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, which is enforced even on a privileged database user. SE-PostgreSQL allows database systems to be built into an information flow control scheme integrated with the operating system, and protects from threats like compromise, manipulation, or destruction of information properties.

1.1 Security Enhanced PostgreSQL

While traditional DBMS access controls are independent from OS-based access controls, SE-PostgreSQL provides unified access control with the OS security policy in conjunction with SELinux. This design provides fine-grained DBMS mandatory access controls.

Mandatory Access Control

PostgreSQL has the concept of a privileged DB user, who can bypass all native PostgreSQL access controls. On the contrary, SE-PostgreSQL enforces access controls on every client without any exception even if the client is a privileged DB user.

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

Fine-grained Access Control

DBMS access controls allow or deny access to DB objects based on the client's privileges. Which DB object is subject to access control depends on the DBMS product, however, SE-PostgreSQL provides row and column-level access controls. Only a few commercial DBMS products provide such functionality.

Because rows and columns are the smallest DB object units in relational databases, SE-PostgreSQL allows the DBA to apply the most flexible access controls.

Consistent Client Authority

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

If TCP/IP is used to connect to SE-PostgreSQL, additional configuration is required. See section "5.3 Labeled IPSec".

Consistent Security Policy

SE-PostgreSQL can attach security contexts explicitly or implicitly based on the SELinux security policy and the database objects under supervision.

It controls accesses based on the security contexts of the client and the database object. Because control is

based on the SELinux security policy, it becomes possible to provide consistent access control throughout the whole system. Information flow control can be built into databases, which becomes an effective method to avoid information compromises.

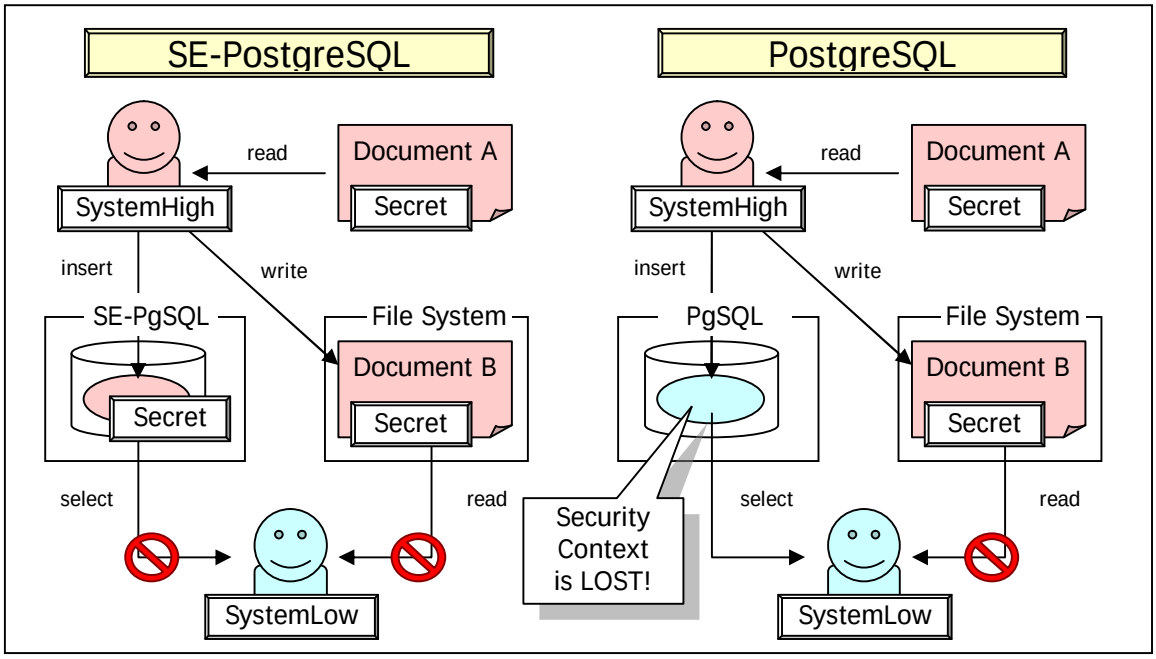


Figure. 1 Information flow control

When a process can read a file labeled "`Secret`" (`SystemHigh` process), and the process creates a file, the file is also given the "`Secret`" label. That is why a low-authority process (`SystemLow` process) can not read it.

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

2 Access Control of SE-PostgreSQL

This chapter describes how SE-PostgreSQL access control work.

2.1 Collaboration with SELinux

SE-PostgreSQL is a reference monitor built in PostgreSQL. It checks every SQL query provided by clients, and prevents access violations to DB objects. This is achieved by consulting the SELinux security policy stored in the Linux kernel. The policy includes all the SELinux access control rules.

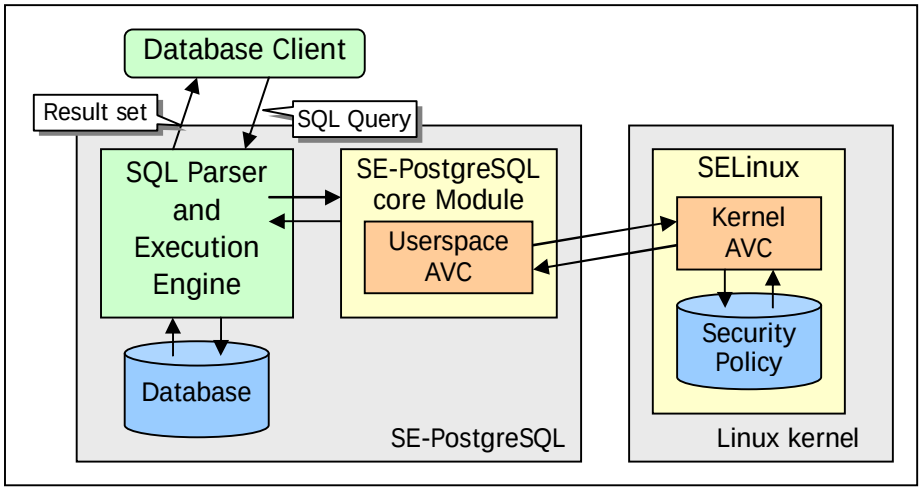


Figure. 2 SE-PostgreSQL architecture

Access controls in SELinux are done with white lists. The policy must define “who” (Subject) “does” (Action) “to what” (Object) explicitly, or the requested action will be denied. For more details about access control methods, see “4.Security Design of SELinux”.

SE-PostgreSQL applies the same TE, MLS, or MCS access controls as the SELinux-enabled OS. The SELinux policy can be described simply as “the object classes are under supervision of the OS".

System-calls to the SELinux policy are high-cost operations, but SE-PostgreSQL minimizes performance loss by caching a part of the security policy in an Access Vector Cache (AVC).

2.2 Security Context

The output when `id -z` is executed on an SELinux enabled host, is shown below:

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

These are “Security Context” sequences, used by SELinux to identify the security property of a resource. In this example, the security context of a user shell process is shown.

All resources, not just processes, that SELinux can observe, such as files, sockets, or IPC objects, have security contexts associated with them.

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 uses to determine whether required accesses are allowed, or not. All SELinux access controls are done based on them.

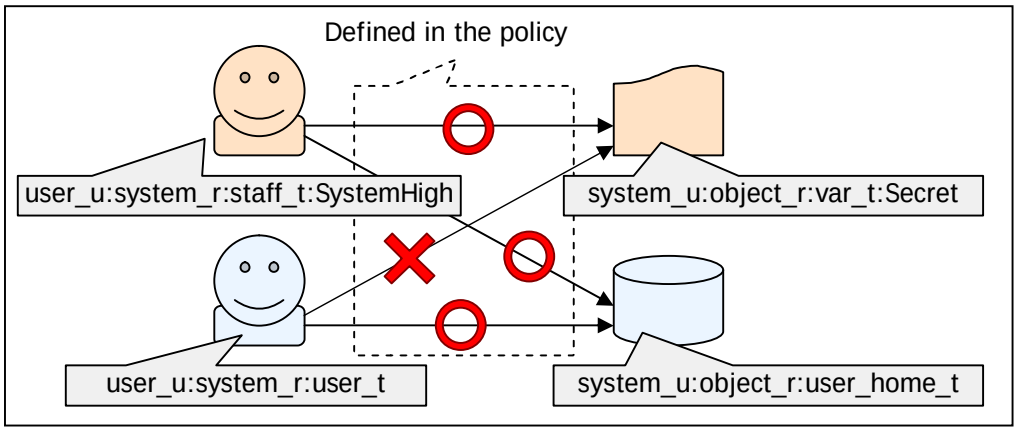


Figure. 3 Security Context and Access Control

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

SE-PostgreSQL enforces access controls on DB objects. Security contexts need to be associated with DB objects when using the SELinux access control mechanism.

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

```
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, table metadata is stored in a tuple contained in `pg_class`, and column metadata is contained in `pg_attribute`. Security Contexts associated with these tuples are seen 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, client Security Contexts are handled in a different way from the DB objects. Clients are the Subjects taking action on the DB objects.

SELinux has an API to obtain Security Contexts from a stream socket peer. SE-PostgreSQL obtains the

Security Context of the remote process using that API, and applies it as the client's Security Context. In other words, SE-PostgreSQL can enforce the same access controls even on remote clients.

While special configuration is not required when connecting via UNIX domain sockets, note that Labeled IPSec Networking is required when connecting via TCP/IP. For more details, see "5.3 Labeled IPSec".

2.3 Mandatory Access Control by SE-PostgreSQL

Several steps are required for PostgreSQL to processes client SQL queries, to access the DB, and to return result sets.

SE-PostgreSQL checks all SQL queries during the query process, and protects DB objects from access violations.

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 the Query analyzer phase, SE-PostgreSQL controls access at the next phase of Query rewriter.

This difference has some interesting implications on SE-PostgreSQL access control.

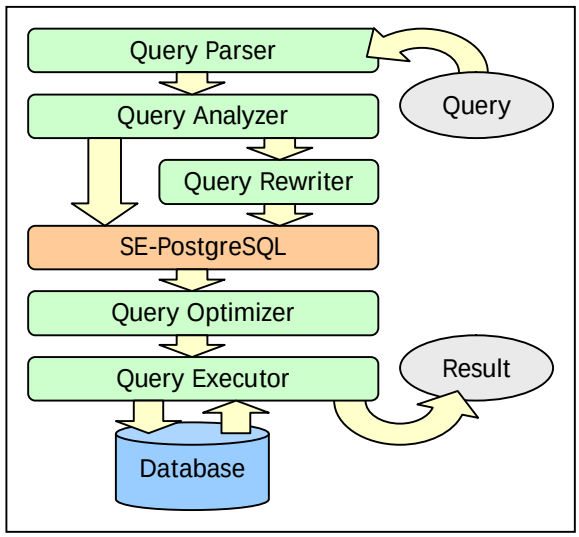


Figure. 4 Query Processing Flow in SE-PostgreSQL

At the Query rewriter phase, a reference via view will be rewritten to a reference to the actual table. When SQL query #2 executes, using `my_view` as defined by SQL query #1, The query rewriter modifies SQL query #2 as SQL query #3, 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 the Query rewriter and performs Mandatory Access Control. In other words, only the Security Contexts of the tables to be accessed determine if a SQL query is executable or not, even if the query has passed through a view.

Native PostgreSQL restricts DB users with ACLs on views. This means that access paths to a certain table cannot be described clearly, because multiple views to a table may be defined. Moreover, an ACL on a table is not evaluated when accessed from a view. Thus, a DB user who has the authority to define a view can access any information in the database.

Because SE-PostgreSQL tests queries after query expansions, a one-to-one access control rule is applied to

a table. Thus, the consistency of an Access Control Policy can be improved with this feature.

2.4 SQL Query Checks

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

A simple SQL query 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 `db_table:select` permission for product and `db_column:select` for `name`, `price`, and `weight`.

See "3.Database Objects and Permissions" for a list of object classes and permissions used by SE-PostgreSQL.

The next example is a little bit more complicated, including calculation and `WHERE`:

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

This query accesses the table `product` and the column `name`, following accesses to the `price` column during calculation and the `weight` column by a `WHERE` clause. As a result, the client is required to have `db_table:{use select}` permission on the `product` table, `db_column:select` permission on the `name` and `price` columns, and `column:use` permission on 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 those functions. Thus, the client must have `db_procedure:execute` permission for these functions.

An sample query that includes `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, access requirements vary with each column.

For instance, the `price` column is subject to update, and is also used for value calculation. This means the client is required to have `db_column:{select update}` permission on the `price` column. The `name` column is just subject to update, so `db_column:update` permission is required. While the `id` column is not subject to update, `db_column:use` permission is needed because it is referenced by a `WHERE` clause.

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

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

This SQL query deletes all tuples in the `product` table, and returns deleted tuples to the client. Since '*' is specified, every column in the `product` table will be returned.

The `product` table is the target for deletion, as well as for reference by a `RETURNING` clause. That requires the client to have `db_table:{select delete}` permission on the `product` table. Furthermore, specifying '*' is equivalent to specifying all columns in the table. The client must have `db_column:{select}` permission for all columns in the `product` table.

Query inspection is also executed when using DDL statements like the DML statements. Below is an example of renaming a table, which updates table metadata.

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

The client must have `db_table:setattr` permission on the table. Moreover, `db_table:setattr` permission is required to update this metadata because PostgreSQL generates a table type, which corresponds to a table inside the system.

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

Below is a simple table definition statement. The client needs `db_table:create` permission for the `simple_tbl` table and `db_column:create` permission for any column contained in.

Newly created tables and columns are attached their security context implicitly based on the security policy..SE-PostgreSQL access control checks are based on these security contexts.

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

The client is required to have sufficient authority to every relevant DB object: TOAST tables are generated when using the TEXT type; indexes are automatically generated when using primary key constraints.

2.5 Rewriting SQL Query

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

SE-PostgreSQL rewrites part of a SQL query to provide row-level access control. SE-PostgreSQL adds a 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 is applied to WHERE clauses by the 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 or not the client has requisite permissions to tuples. It returns true if it has permission, or returns false if it does not have permission.

This compulsory filtering removes violated tuples from the client's result set.

Result set filtering is also enforced when accessing another table with a JOIN clause or subquery, allowing the client to refer only to the subset of tuples included in the table.

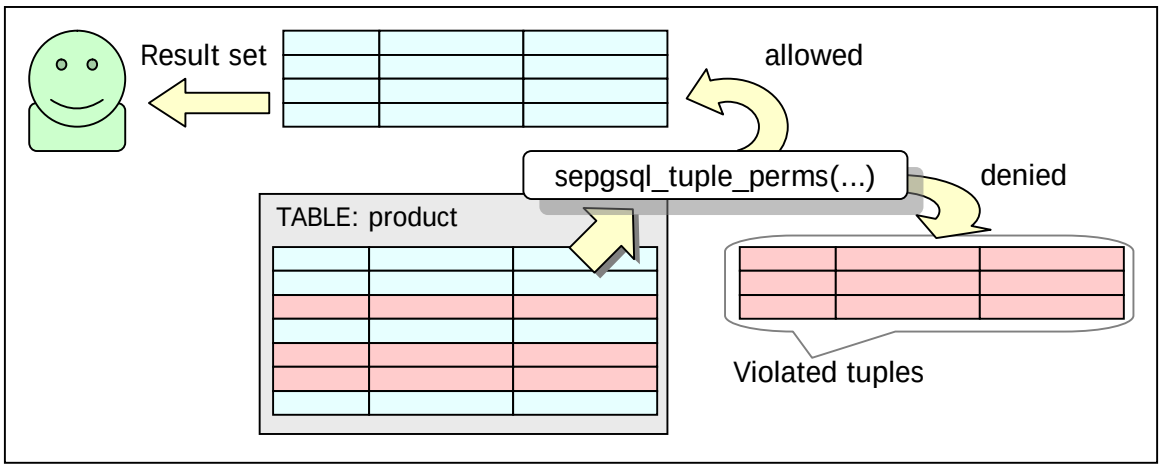


Figure. 5 Filtering violated tuples

Violated tuples are filtered not only on reference, but also on update and deletion. To avoid updating a client's violated tuples, SE-PostgreSQL rewrites the 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 the UNIQUE constraint prevents two or more different tuples from having the same value set.

With the SE-PostgreSQL row-level access control, any tuple on which the client doesn't have sufficient authority will be filtered from the result set. At first glance, it may appear that a newly inserted/updated value meets the UNIQUE constraint.

Nevertheless, the UNIQUE constraint ensures the uniqueness of tuples in the actual table, independently from the row-level access control of SE-PostgreSQL. As a result, if a tuple invisible to the client and a newly inserted/updated value are duplicates, then the operation violates the UNIQUE constraint.

Theoretically, repeating these operations enables a client to guess the contents of invisible tuples.

This is a limitation of the current version of SE-PostgreSQL. To improve it, there is a plan to support polyinstantiation in a future version.

2.5.2 Special case: Foreign Key Constraint

Foreign key constraints define referential integrity between two tables. In PostgreSQL, a referenced value must be a primary key.

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

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

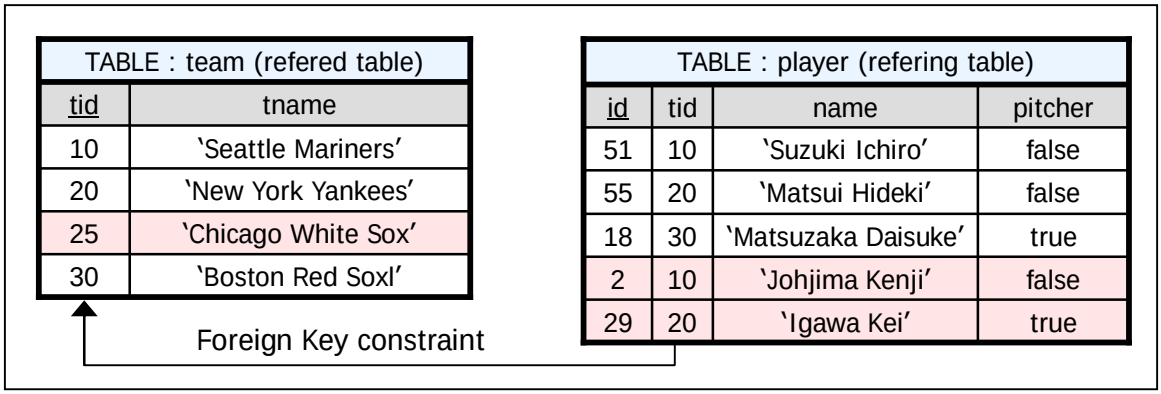


Figure. 6 An Example of Foreign Key Constraint

Figure. 6 is an example of foreign key constraint. A foreign key constraint of `tid` in the `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 cannot access the tuples shown in pink, inserts the tuple shown below into the `player` table.

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

The `tid` value inserted into the `player` table must exist as a `tid` in the `team` table. Indeed, there is a tuple that has `tid`=25 in `team` table, however, the client does not have authority to access this tuple. Therefore, the new tuple insertion will fail.

In contrast, when a privileged client, which can access any tuple, inserts the same tuple into the `player` table, the new tuple insertion will succeed because the client can access a tuple with `tid`=25 from the `team` table.

The PostgreSQL foreign key constraint can update or delete a dependent foreign key when a referenced primary key value is updated or deleted. This option is activated ON UPDATE or ON DELETE of a foreign key constraint. This keeps a foreign key value consistent upon update or deletion of a primary key value with the `CASCADE/SET NULL/SET DEFAULT` action.

For more details about foreign key constraints, see the "5.3.5 Foreign Key" section of the PostgreSQL manual.

SE-PostgreSQL independently checks each, individual access permission needed to update or delete referenced primary key values. Similarly, SE-PostgreSQL independently checks each, individual access permission needed to update or delete dependent foreign key values.

When a foreign key is updated or deleted, as a result of primary key value update or deletion, the client must have proper authority to every tuple subject to update or deletion. If the 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 designed to preserve referential consistency. When a foreign key value is updated or deleted, SE-PostgreSQL prevents referential consistency constraint violations, not by excluding violated tuples, but by aborting the whole transaction.

2.5.3 Special case: OUTER JOIN

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

A result set of LEFT OUTER JOIN contains at least one tuple from the left table. This is not restricted by a join condition with an ON clause. In other words, a tuple in the left table cannot be filtered out from a result set by a simple condition clause replacement. It works the same way for RIGHT OUTER JOIN and FULL OUTER JOIN.

SE-PostgreSQL rewrites table references within an OUTER JOIN clause as a subquery clause with a conditional clause for row-level access control, to prevent tuple access violations.

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

The simple LEFT OUTER JOIN shown above will be rewritten by SE-PostgreSQL as shown below. A reference to the left table is transformed into a subquery with a conditional clause, which prevents a tuple that the client does not have proper access permission to from being subject to 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 consistency, SE-PostgreSQL rewrites SQL queries to handle the case when system columns within OUTER JOINs are referenced.

2.6 Trusted Procedure

SE-PostgreSQL may perform a client domain transition with function execution. The entry point function for a domain transition is known as a Trusted Procedure.
Domain transition is an SELinux concept, see section “4.2.1 Domain Transition” for more details.
The security context of the client is changed based on security policy during Trusted Procedure execution. This allows access to DB objects that are not reachable directly, from inside Trusted Procedures.

This concept is similar to the OS SetUID or the Security Invoker of SQL, but domain transition is not related to the owner of the DB object. Instead, domain transition is defined by the Security Policy.

Here is an example that uses a Trusted Procedure. A customer table is shown below. The contents of the `password` column are used for authentication, but direct access to the `password` column by the client should be prohibited.

customer table

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, define a SQL function that returns an authentication result using the `password` column. Designate the function a Trusted Procedure, and clients may authenticate using the `password` column without direct access to that `password` column.

Steps on creating a Trusted Procedure that authenticates with `customer.password` follow.

First, define SQL function `check_customer_password` that takes `customer.id` and an authentication token as arguments and returns a boolean that indicates if the customer's password matches.

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

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

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

Sample SQL queries that include Trusted Procedures are shown below. Note that the client gets enough information to authenticate, but it is still not able to read the `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
```

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

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

If the `password` column is no longer subject to SELECT, then a result is given. Note that 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)
```

If the Trusted Procedure is called an authentication result is returned to the client by the `check_customer_password` function, while the contents of the `password` column itself are not leaked out.

3 Database Objects and Permissions

This chapter explains how SE-PostgreSQL controls access to each DB object variant. General information about SELinux access control is introduced in section “4.Security Design of SELinux”. Please read them altogether.

3.1 Object Classes and Permissions

Just as access methods differ between tables and functions, other DB objects have unique requirements. SELinux describes these requirements as an object class and a group of permissions (access vector) associated with that object class.

The security policy for SE-PostgreSQL describes 6 object classes and the fifty-eight permissions associated with those object classes. The following table lists each of them.

db_database	db_table	db_procedure	db_column	db_blob	db_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, each kind of DB object should have a unique object class. In PostgreSQL, however, there are DB objects that do not have unique object classes, such as triggers or database roles.

Access controls to those objects are handled as operations on special purpose tables called the system catalog, and are evaluated using `db_table`, `db_column` and `db_tuple` permissions.

3.2 Common access control for each Object class

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

Of these operations, only changing security attributes is defined for tuples. This is because `create` and `drop` are equivalent to `insert` and `delete`, and the tuple class does not have its own metadata.

Some common permissions of these DB objects are explained in this section.

create permission

When a CREATE statement like CREATE TABLE creates a DB object, the new DB object is associated with a security context implicitly. The client is required to have `create` permission on 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 the `lo_create()` function, and create permission is evaluated at that time.

drop permission

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

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

getattr permission

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

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

setattr permission

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

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

relabelfrom/relabelto permission

To change the security context of DB objects, use the extended ALTER statement or UPDATE the `security_context` column. `relabelfrom` and `relabelto` permission is evaluated at that time, and the client must have `relabelfrom` permission to the security context before the change to the DB object, and `relabelto` permission to the security context after the change to the DB object.

3.3 Access Control for Database

The `db_database` object class contains the permission set for the database itself and five unique permissions.

Implicit Security Context

When a new database is created with the CREATE DATABASE statement, the new database inherits its type from the domain of the SE-PostgreSQL server process. In addition, we can describe type transition rules for `db_database` class, targeting with the domain of the SE-PostgreSQL server process which manages the newly created database.

The default type transition rule in the default security policy gives the `sepgsql_db_t` type to the database itself.

access permission

The client must have `access` permission to the destination database when connecting. This is the minimum permission requirement to connect to SE-PostgreSQL.

install_module permission

To install a Dynamic Link Library(DLL), the client must have `install_module` permission both on the database and on the DLL file.

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

For example, when a client working in the `staff_t` domain connects to a `sepgsql_db_t` type database, loading a `lib_t` type DLL requires the security policy below:

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

load_module permission

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

This permission differs from the `install_module` permission because `load_module` permission evaluation may be done several times, whenever a DLL is loaded into databases.

The following is a security policy sample granting this permission:

```
allow sepgsql_db_t lib_t : db_database load_module;
```

get_param permission

To reference run-time parameters with the SHOW statement, the client must have `get_param` permission for the connected database.

This permission is evaluated on the database so that each run-time parameter cannot be set to permit or prohibit.

set_param permission

To set run-time parameters with the SET statement, the client must have `set_param` permission for the connected database.

This permission is evaluated on the database so that each run-time parameter cannot be set to permit or prohibit.

3.4 Access Control for Table

The `db_table` object class contains six unique permissions.

Implicit Security Context

When a new table is created with the CREATE TABLE statement, the new table's type is inherited from the database type. In addition, we can describe type transition rules for `db_table` class, targeting with the type of the database which contains the newly created table.

The default security policy gives the `sepgsql_table_t` type to new tables with a type transition rule.

use permission

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

For example, an UPDATE statement, like the one below, requires not only `update` permission on the `drink` table, but also `use` permission. That is because the `id` column of the `drink` table is referenced from the WHERE clause.

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

select permission

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

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

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

`use` permission differs from `select` permission because `select` permission is evaluated when the client reads table contents and the database returns the content to the client. The database evaluates `use` permission when table content is not returned to the client such as in a conditional clause or in an ORDER BY clause.

update permission

The client must have `update` permission on the targeted table when updating it with an UPDATE statement.

insert permission

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

delete permission

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

lock permission

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

3.5 Access Control for Column

The `db_column` object class contains four unique permissions for columns.

Implicit Security Context

A new column inherits the type of the table containing the columns when they are created with a CREATE TABLE or an ALTER TABLE statement. In addition, we can describe type transition rules for `db_column` class, targeting with the type of the table which contains the newly created column.

use permission

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

For example, the SELECT statement below requires not only `select` permission on the `id` and `name` columns, but also `use` permission on the `alcohol` and `price` columns. That is because the `alcohol` column is referenced by the WHERE clause and the `price` column is referenced by the ORDER BY clause.

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

select permission

The client must have `select` permission on the targeted column when referencing columns.

Typically, there is a column or a formula including a column in the target list of the SELECT statement. Note that the client will be checked for `select` permission to the targeted column when:

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

The example SQL query below requires not only `select` permission on the `uid` and `uname` columns, but also requires `select` permission on the `birthday` column.

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

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

update permission

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

insert permission

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

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

3.6 Access Control for Tuple

The `db_tuple` object class contains seven unique permissions for tuples.

Implicit Security Context

When inserted with an INSERT statement or a COPY FROM statement, new tuples inherit their type from the table they are created in. In addition, we can describe type transition rules for `db_tuple` class, targeting with the type of the table which contains the newly created tuple.

Note that the database does not treat an INSERT into a part of the System Catalog, like `pg_class` or

pg_proc, as a db_tuple class operation. That is because tuple insertion to those System Catalogs is equivalent to the creation of tables or functions. Thus, this case is treated simply as the creation of that object class.

relabelfrom/relabelto permission

The security context of a tuple can be changed with an UPDATE to the security_context column at which time relabelfrom and relabelto permission is evaluated. The client must have relabelfrom permission to the security context before the tuple is changed, and relabelto permission to the security context after the tuple is changed.

use permission

The client must have use permission on tables including columns referenced by conditional SQL statement (WHERE, JOIN~ON, or HAVING clauses), the GROUP BY clause, or the ORDER BY clause.

Tuples that the client does not have use permission on will be filtered from the result set and from being subject to update /deletion.

select permission

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

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

update permission

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

Tuples that the client does not have update permission on don't be updated.

insert permission

The client must have insert permission on the targeted tuple when inserting a new tuple with an INSERT statement, or restoring tables with a COPY FROM statement.

The new tuple's security context is set implicitly, however, the security context can be set explicitly using an INSERT statement with a value set for the security_context column. Tuples don't be inserted in any way without insert permission.

delete permission

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

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

3.7 Access Control for Function

The db_procedure object class has two unique permissions.

Implicit Security Context

A new function inherits its type from the database type when the new function is created with a CREATE FUNCTION statement. In addition, we can describe type transition rules for `db_procedure` class, targeting with the type of the database which contains the newly created function.

The default security policy gives newly created functions the `sepgsql_proc_t` type.

execute permission

A client must have `execute` permission for functions executed with SQL queries. PostgreSQL implements operators as SQL functions. It executes the `int4eq` function to compare four byte integers, for example. A client must have `execute` permission for the function.

Meanwhile, if 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 any functions defined as Trusted Procedures. For more details about Trusted Procedures, see section "2.6 Trusted Procedure".

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

A description of Security Policy required for Trusted Procedures follows:

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

The client has the ability to execute the Trusted Procedure with (1). (2) allows the client to transition to <new domain>. Actual domain transitions occur when Trusted Procedures are executed with (3).

3.8 Access Control for Binary Large Object

The `db_blob` object class has four unique permissions.

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

Implicit Security Context

New binary large objects inherit their type from the database type when they are created with the `lo_create()` function. In addition, we can describe type transition rules for `db_blob` class, targeting with the type of the database which contains the newly created blob.

The database gives the `sepgsql_blob_t` type to 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 the `loread()` function. This is the same as referring to the `pg_largeobject` system catalog directly.

write permission

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

import/export permission

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

These two functions do not read/write data directly between the client and the DB object. It is only a SE-PostgreSQL server process that reads/writes actual files on the OS. SELinux controls these read/write operations at the kernel level.

A client can start these operations on the server with `lo_import()` or `lo_export()`. This is the relationship between the client and the `import`/`export` server process of SE-PostgreSQL.

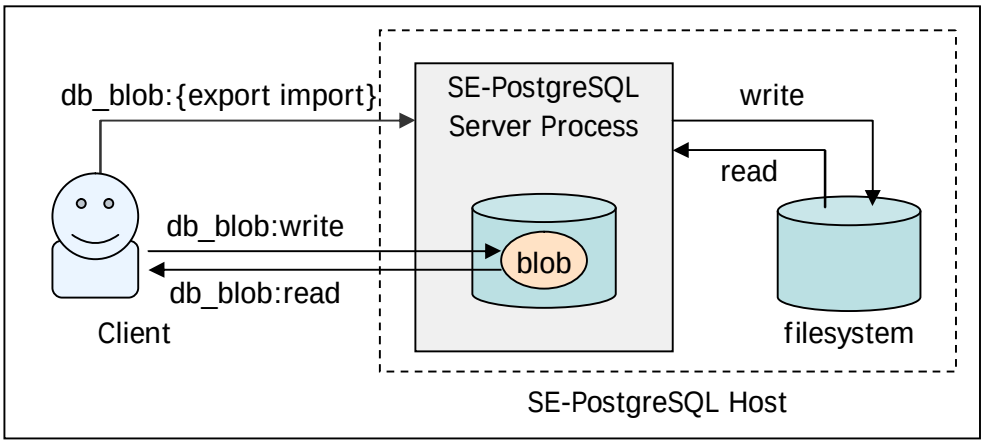


Figure. 7 import/export of binary large object

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

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

4 Security Design of SELinux

This chapter provides a generic explanation of SELinux access control, that is not limited to SE-PostgreSQL.

4.1 Security Contexts

A security context is a character string that 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 are separated into 4 parts with ":". These fields signify the "user", "role", "type (domain)" and the "MLS label".

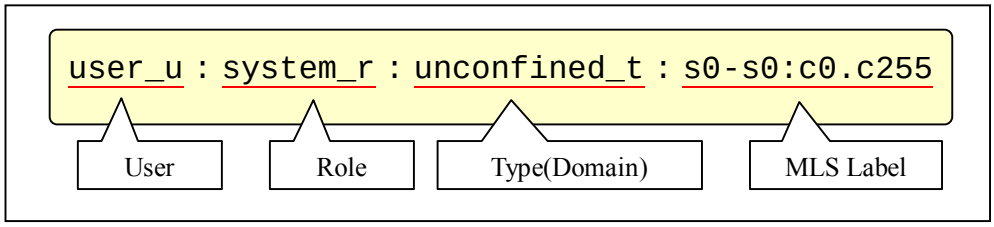


Figure. 8 Security Context

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, which is described below. Role has meaning only in the security context of a process, and otherwise is uniformly set to "`object_r`".

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

The MLS label is an identifier used in MLS/MCS, which is described below. The mcstrans service displays the MLS label as a meaningful alias such as "`SystemHigh`", etc.

SELinux provides access control using TE, MLS/MCS and RBAC, and these are all implemented 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, the action, and the type of the object accessed by the process. This is the most important access control mechanism in SELinux.

The security policy defines which domains can execute what kind of actions on what types. All actions that have not been explicitly allowed are denied.

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`". 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, when sending a signal, the action can become "`process:kill`".

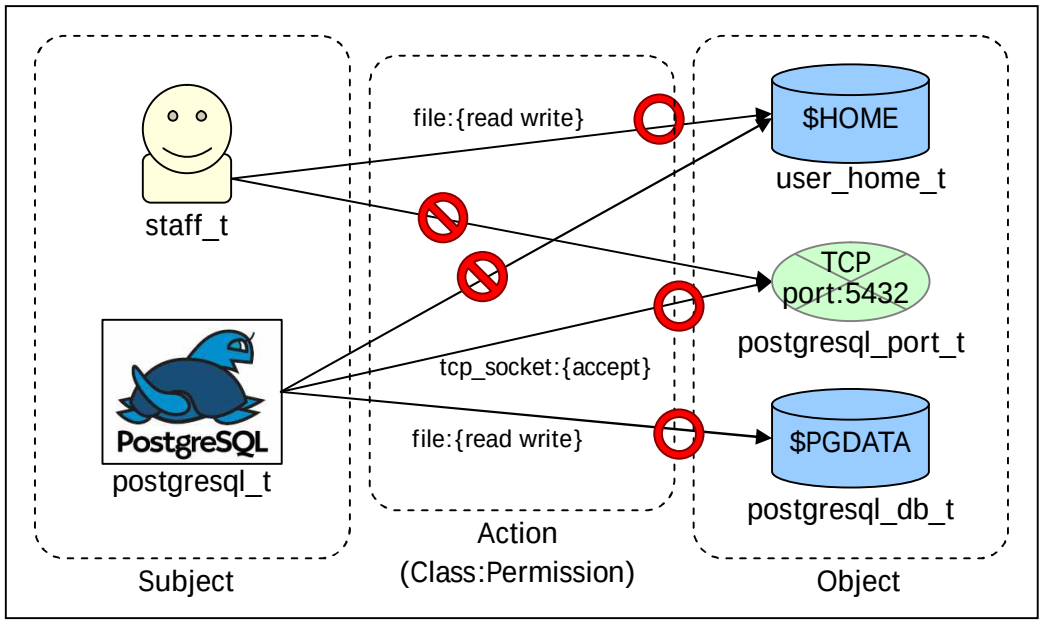


Figure. 9 Type Enforcement

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 each allowed action is described line by line, it makes description of the security policy complicated and maintenance becomes difficult. Therefore, a set of modular, high-level macros comes with the Reference Policy provided by the SELinux community. Using the macros provided in the Reference Policy, the security policy writer can develop a highly readable, modular security policy.

4.2.1 Domain Transition

A process inherits the domain of its parent process, unless a domain transition occurs.

That is, if a process operating in the `unconfined_t` domain starts a child process, the child process also operates in the `unconfined_t` domain. Similarly, 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 would operate in exactly the same domain, and it would be impossible to perform meaningful access control. Domain transition is used to resolve this problem.

Domain transition is a mechanism which determines the domain of a new process 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 ;
```

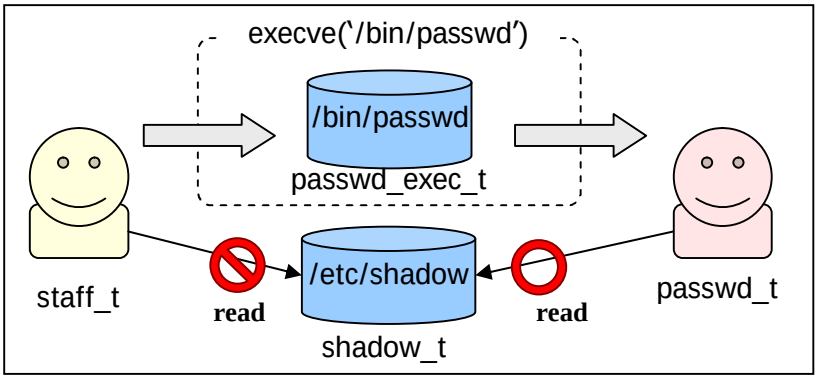


Figure. 10 Domain Transition

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 with `/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 controls the domain transitions described above.
An arbitrary number of domains can be associated 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.

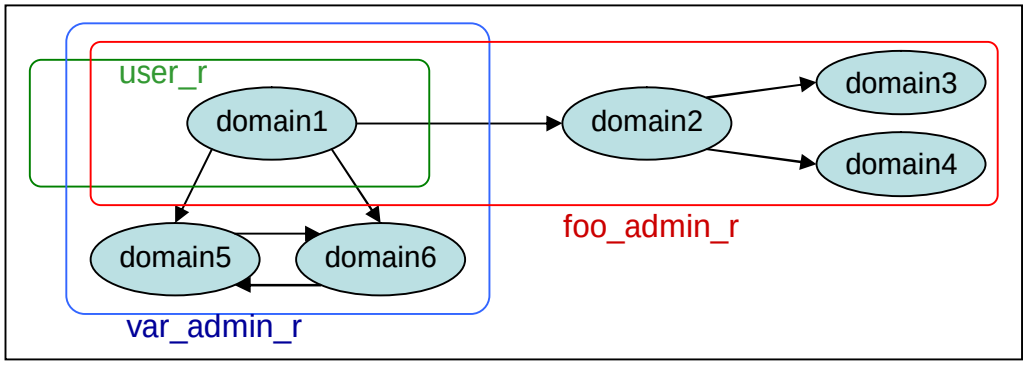


Figure. 11 Conceptual diagram of RBAC

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.

With these sorts of constraints, it is possible to create administrators who are limited only to services or parts of the system. However, for that reason, the domains must be partitioned with 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.

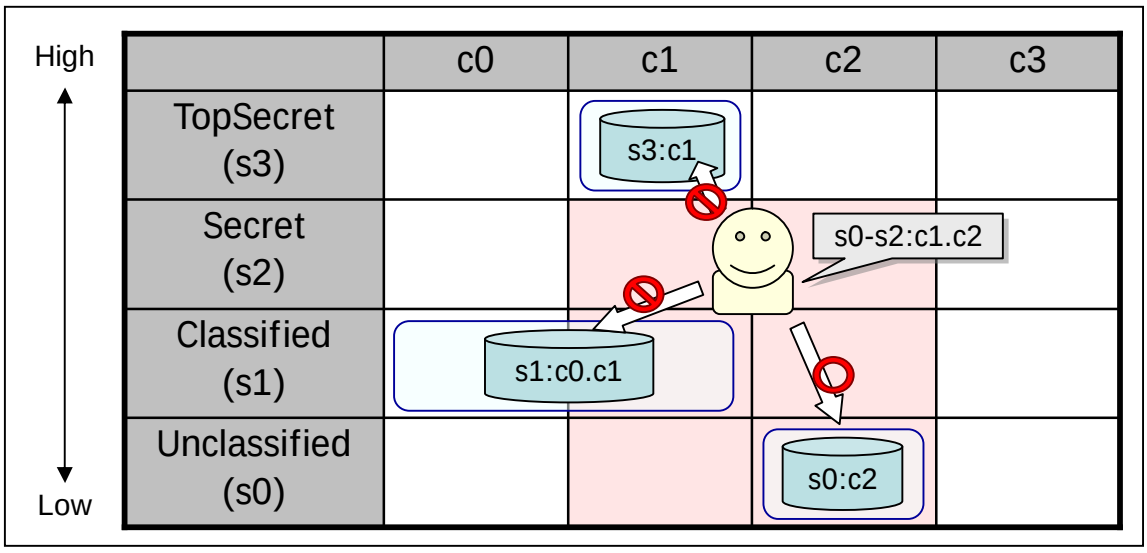


Figure. 12 Conceptual diagram of MLS

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.

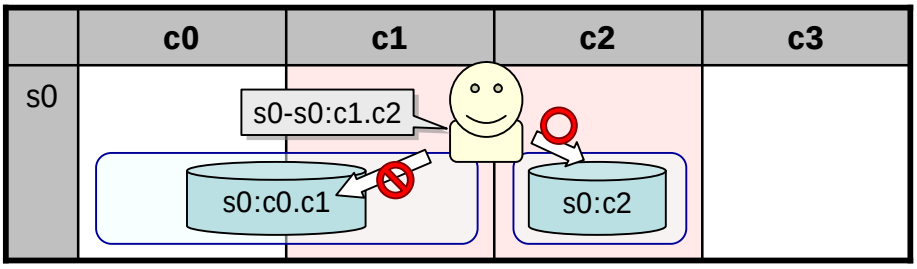


Figure. 13 Conceptual diagram of MCS

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 creates an access denied log that can be used to debug the security policy.

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 variables, it is possible to enable/disable specific points in the SELinux security policy of a running system. Boolean variables are either “on” or “off”, and can be set using the `setsebool` command. Use the `getsebool` command to acquire a list of condition variables and to check their values.

How the individual condition variables customize the security policy depends on the definitions in the security policy. See section “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 semanage manpage.

5 SE-PostgreSQL System Administration

SE-PostgreSQL specific administrative tools, policies, and configurations are described in this chapter.

5.1 Backup and Restore in SE-PostgreSQL

`pg_dump` and `pg_dumpall` are utilities to backup PostgreSQL databases. These utilities have been extended to enable backups with security contexts.

The DBA must have authority to refer to all target database objects in the database being backed up. If the DBA does not have authority on those objects, the generated dump image will be incomplete or the backup process will abort.

In contrast, the DBA must have the authority to insert and to update any database object during restoration. The restore process will fail if the DBA authority is not sufficient.

When SE-PostgreSQL is deployed via RPM package, `pg_dump` and `pg_dumpall` are respectively renamed to `sepg_dump` and `sepg_dumpall`, to avoid a confliction with the native postgresql package.

pg_dump command

`--enable-selinux` option enables the backup of a specified database with security contexts. The database objects dumped with security contexts are tables, columns, SQL functions, binary large objects and tuples.

pg_dumpall command

`--enable-selinux` option enables the backup of a whole database cluster with security contexts. The database objects dumped with security contexts are databases, tables, columns, SQL functions, binary large objects and tuples.

pg_restore command

`pg_restore` restores the backup image with the security contexts generated by `pg_dump` or `pg_dumpall` with `--enable-selinux` option.

5.2 The default security policy

The default security policy is configured to provide maximum compatibility with a non-SELinux environment. The intent is to avoid breaking traditional applications. To apply mandatory access control more efficiently, special purpose security contexts can be assigned to database objects like tables and procedures.

There are two kinds of security policy in SELinux: Targeted and Strict. The `unconfined_t` domain in the Targeted policy and `sysadm_t` domain in the Strict policy have wider authority than any other domain. These domains are allowed to execute DDL statements and to access confidential database objects, for example. These domains are called an “administrative domain” as a matter of convenience.

Other domains are called a “generic domain” for the same reason. Generic domains are restricted from executing DDL statements and accessing 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 mentioned above, the `unconfined_t` domain in the Targeted policy and the `sysadm_t` domain in the Strict policy have wider authorities than any other domain which can connect to SE-PostgreSQL.
It is necessary to setup, backup and restore databases from those administrative domains.

5.2.2 Types for Table/Column/Type

Any column and tuple implicitly inherits the type of the table in which it is contained. Thus, the security context of columns and tuples are the same as the table’s security context, if they are not changed explicitly.
The following five types are already defined in the default security policy.

sepgsql_table_t

This is the default type which is attached implicitly to newly created tables in the default security policy. Any client is allowed all operations except for relabeling the table, column or tuple with `sepgsql_table_t`. In other words, any client can access them as with native PostgreSQL.

sepgsql_secret_table_t

All operations are denied on tables, columns and tuples on which this type is attached, except for the administrative domains and accesses via trusted procedure.
Use the `sepgsql_secret_t` type to store confidential information like password phrases or credit card numbers.

sepgsql_ro_table_t

Only the SELECT statement is allowed on tables, columns and tuples on which this type is attached, except for the administrative domains and accesses via trusted procedure.
Use the `sepgsql_ro_table_t` type to protect read-only information from manipulation, like a master table of commodities for example.

sepgsql_fixed_table_t

Only SELECT and INSERT statements are allowed on tables, columns and tuples on which this type is attached, except for the administrative domains and accesses via trusted procedure.
Use the `sepgsql_fixed_table_t` type to guarantee that once inserted, information is not manipulated or destroyed. For example, this may be useful to store documents which must be kept for a certain period for legal reasons.

sepgsql_sysobj_t

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

itself.

`sepgsql_enable_users_ddl` switches whether or not a generic domain can modify tuples within system catalogs.

5.2.3 Types for Procedure

sepgsql_proc_t

In the default security policy, this is the default type for a 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, this is the default type for procedures created in the generic domain.

In the generic domain it is possible to execute functions with `sepgsql_user_proc_t` type, but, conversely, it is not possible in the administrative domain.

Executing an "untrusted procedure", defined by someone with unconfined authority often gives a dangerous result. The administrative domain cannot execute the user defined procedure until it has been checked and its security context has been changed to `sepgsql_proc_t`.

sepgsql_trusted_proc_t

A function with this type is called a Trusted-Procedure.

Any client can execute `sepgsql_trusted_proc_t` typed functions, which will cause a domain transition. Any database objects that the Trusted-Procedure provides a way to access can be accessed, as if the client is in the administrative domain.

A Trusted-Procedure can be used to restrict access to database objects which contain confidential information, but be careful not to create a Trusted-Procedure that circumvents the security functionality of SE-PostgreSQL. Understand this risk when configuring Trusted-Procedures.

5.2.4 Types for binary large object

sepgsql_blob_t

This is the default type attached implicitly to newly created binary large objects in the default security policy.

Any client can do any operation on binary large objects with `sepgsql_blob_t`, as with native PostgreSQL.

sepgsql_ro_blob_t

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

sepgsql_secret_blob_t

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

For example, use the `sepgsql_secret_blob_t` type to store confidential information like a binary formatted secret document.

5.2.5 Types for database

sepgsql_db_t

This is the only type attached to the database in the default security policy.
No other type can be associated with databases.

5.2.6 boolean variables

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

sepgsql_enable_unconfined

This boolean enables or disables the administrative domain. When this boolean is “off”, the client’s authority is restricted to the generic domain, even if it is an administrative domain.
This boolean defaults to “on”.

sepgsql_enable_users_ddl

This boolean toggles whether or not a generic domain can execute DDL statements. When this boolean is “off”, the execution of DDL statements like “CREATE TABLE” is denied.
This boolean defaults to “on”.

sepgsql_enable_auditallow

This boolean toggles whether or not allowed-audit logs are generated. When this boolean is “on”, allowed-audit logs are generated. Allowed-audit logs can be used to determine what actions are required on what database objects, and what access controls are in place.
This boolean defaults to “off”.

sepgsql_enable_auditdeny

This boolean toggles whether or not denied-audit logs are generated. When this boolean is “on”, denied-audit logs are generated. Denied-audit logs record database access violations.
This boolean defaults to “on”.

sepgsql_enable_audittuple

This boolean toggles whether or not allowed or denied audit logs for tuples are generated.
Audit log generation for tuples depends on `sepgsql_enable_audittuple`, even if `sepgsql_enable_auditallow` or `sepgsql_enable_auditdeny` are enabled.
These booleans are separated to avoid a flood of audit logs when tables are scanned containing a large number of tuples.
This boolean defaults to “off”.

5.3 Labeled IPSec

Labeled IPsec is an excellent technology to obtain the security context of a peer process that communicates over IPsec networks. When a client connects to SE-PostgreSQL via a TCP/IP connection, Labeled IPsec is set up to determine the security context of the client.
This section describes how to set up Labeled IPSec.

Confirming required packages

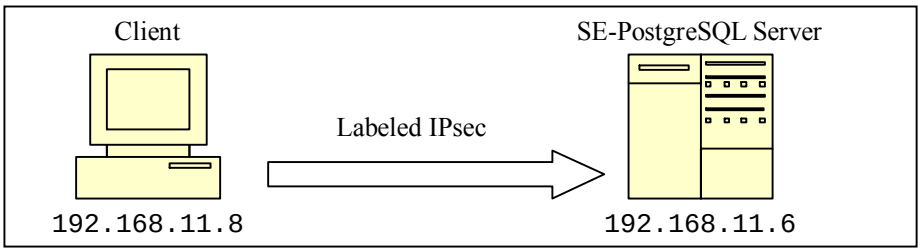
Confirm that the following packages are installed:

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

Labeled IPsec enabled kernels are built with CONFIG_SECURITY_NETWORK_XFRM=y.
This is enabled on Red Hat Enterprise Linux 5, Cent OS 5, Fedora 7 and on an updated Fedora Core 6.
A key exchange process is extended to deliver the security context of server and client to its peer.
These extensions are added to `raccoon`, which is a key exchange server contained in `ipsec-tools`.

Sample environment

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



Adding a policy to SPD(Security Policy Database)

Use the following configuration examples to setup encrypted communication between client and server and to deliver each peer's security context in the key exchange process.

1. Make SPD definition files for each peer. Note that the order of IP addresses is reversed for the client.

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

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

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

Configuration of /etc/racoon/racoon.conf

Edit `/etc/racoon/racoon.conf` to setup raccoon, which is the key exchange server. Add the lines indicated in bold characters. Note that the order of IP addresses is reversed on the client side, because it is a server side (192.168.11.6) configuration file.

```
[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 ;
        }
}
```

Configuration of /etc/racoon/psk.txt

This simple, example configuration encrypts the communication pathway by PSK(Pre Shared Key). `/etc/racoon/psk.txt` is used to store the peer host's PSK.

The following configuration is a sample for the server side(192.168.11.6). Note that the order of IP addresses is reversed on 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 configuration above.

```
# service racoon start
Starting racoon: [  OK  ]
```

Racoon is required 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 when the connection is established.

5.4 Static Network Labels

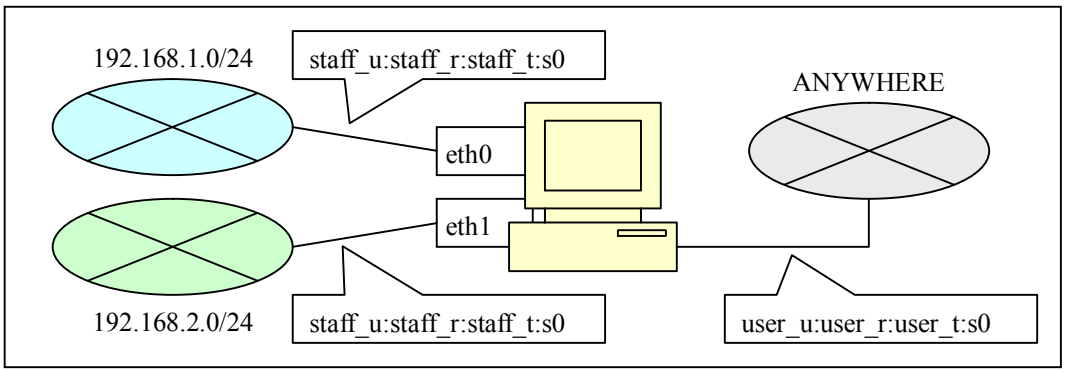
Labeled IPsec system exchanges the security context of both peers before establishing a connection. It enables servers (like, SE-PostgreSQL) to obtain the security context of its clients.

Static Network Labels is a functionality to associate an alternative security context of the peer based on its using network interfaces and source IP addresses in the case when we cannot apply Labeled IPsec. This feature is available on the Linux kernel 2.6.25 or later.

You can use `netlabelctl` to config this feature. It is containd in `netlabel_tools` package in the Fedora system, so you should install it preliminarily.

We explain Static Network Labels configuration with the following example:

```
connection from 192.168.1.0/24, via eth0        ... staff_u:staff_r:staff_t:s0
connection from 192.168.2.0/24, via eth1        ... staff_u:staff_r:staff_t:s0:c0
<ANYWHERE>                                      ... user_u:user_r:user_t:s0
```



You can run `netlabelctl` command as follows, to associate an alternative security context with these network interfaces and source IP addresses.

This configuration enables a connection come from 192.168.1.100 via eth0 to act as if it has a security context of "`staff_u:staff_r:staff_t:s0`".

```
# netlabelctl unlbl add interface:eth0 address:192.168.1.0/24  ¥
            label:staff_u:staff_r:staff_t:s0
# netlabelctl unlbl add interface:eth1 address:192.168.2.0/24  ¥
            label:staff_u:staff_r:staff_t:s0:c0
# netlabelctl unlbl add default address:0.0.0.0/0  ¥
            label:user_u:user_r:user_t:s0
```

You should pay mention that Labeled IPsec has higher priority than Static Network Labels for the connection configured with Labeled IPsec.

6 Appendix

6.1 Extended SQL statement

To leverage the features of SE-PostgreSQL, some SQL statements are extended.
These extended statements are described in this section.

6.1.1 CREATE DATABASE statement

Format

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

Description

The enhanced `CREATE DATABASE` statement enables explicit security context specification at database creation.

The client must have `db_database:{create}` permission to explicitly specify the security context.

Example

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

6.1.2 ALTER DATABASE statement

Format

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

Description

The enhanced `ALTER DATABASE` statement enables changing the security context of a database.
The client must have `db_database:{setattr relabelfrom}` permission on the target database, and `db_database:{relabelto}` permission for the newly attached security context.

Example

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

Remarks

If the security context of the database is not changed with this statement, `db_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 enhanced CREATE TABLE statement enables table creation with explicitly specified table and column security contexts.

The client must have `db_table:{create}` and/or `db_column:{create}` permission for 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 enhanced ALTER TABLE statement enables changing the security context of a table or a column.

The client must have `db_table:{setattr relabelfrom}` permission for target table, and `db_table:{relabelto}` permission for the newly attached security context.

In the same manner, the client must have `db_column:{setattr relabelfrom}` permission for the target column, and `db_column:{relabelto}` permission for the newly attached security context.

Example

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

Remarks

If the security context of the table is not changed, `db_table:{relabelfrom relabelto}` permissions are not evaluated. Similarly, `db_column:{relabelfrom relabelto}` permissions

are not evaluated, if the security context of the column is unchanged.

6.1.5 CREATE FUNCTION statement

Format

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

Description

The enhanced `CREATE FUNCTION` statement enables function creation with its security context specified explicitly.

The client must have `db_procedure:{create}` permission for 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 enhanced `ALTER FUNCTON` statement enables changing the security context of a function.

The client must have `db_procedure:{setattr relabelfrom}` permission for the target function, and `db_procedure:{relabelto}` permission for 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

If the security context of the function is not changed, `db_procedure:{relabelfrom relabelto}` permissions are not evaluated.

6.2 Extended SQL function

Several new functions are provided to leverage SE-PostgreSQL's security features.
These extended SQL functions are described in this section.

6.2.1 sepgsel_getcon() function

Definition

```
sepgsql_getcon() returns security_label
```

Description

A function is provided to obtain a connected client's security context.
`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

A function is provided 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 `db_blob:{getattr}` permission for the target binary large object.

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

A function is provided to change the security context of a binary large object. `lo_set_security()` changes the security context identified by `loid`, and returns TRUE on success.
The client must have `db_blob:{setattr relabelfrom}` permissions for the target binary large object, and `db_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)
```