Table of Contents

1 Objectives....2
2 Presenter....2
3 What is Cryptography....3
3.1 Symmetric Key Encryption....3
3.2 Public Key Encryption....4
4 The Complete Picture....5
5 Database Roles and Privileges....7
5.1 Privileges Example....7
5.2 Display Access Privileges....13
5.3 Displaying privileges in more readable manner....14
5.4 System Provided Functions for ACLs....15
6 Row Level Security....16
6.1 RLS Example....16
6.2 pg_dump and RLS....24

1 Objectives

A) Learn database roles, privileges and row level security.

2 Presenter

My name is Abbas, I have a Masters in Computer Engineering. I have spent most of my career in product development. I work as a Senior Architect at EnterpriseDB. My work highlights are as follows:

- EDB Replication Server (xDB)
- Migration Portal for online schema migration from Oracle to PostgreSQL
- Schema Cloning with support for parallelism using Background Workers
- Distributed Transactions (XA) Compliance for PostgreSQL using PgBouncer
- Oracle Compatible Packages for IBM DB2 :
  - UTL_ENCODE
  - UTL_TCP
  - UTL_SMTP
  - UTL_MAIL
- HDFS_FDW, Mongo_FDW, MySQL FDW
- Postgres-XC

Email : abbas.butt@enterprisedb.com
Linkedin : https://pk.linkedin.com/in/abbasbutt
Blog : https://abbas-technical.blogspot.com

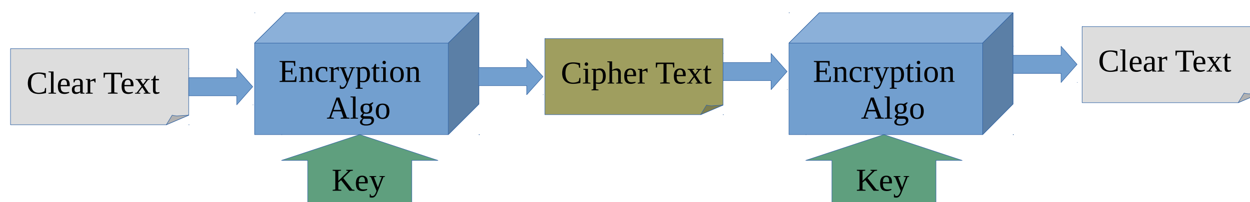
3 What is Cryptography

Cryptography algorithms can be classified into two main categories:

Symmetric Key Encryption and Asymmetric Key Encryption which is also called Public Key Encryption.

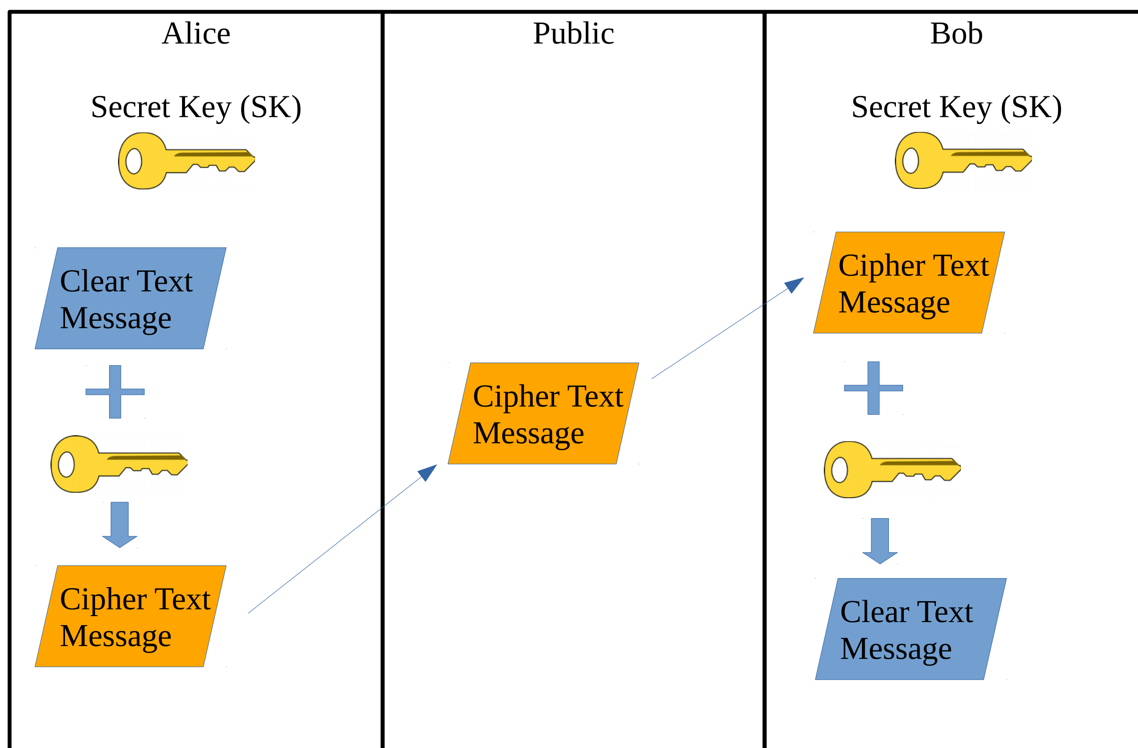
3.1 Symmetric Key Encryption

Symmetric key algorithms encrypt and decrypt data using a single key.



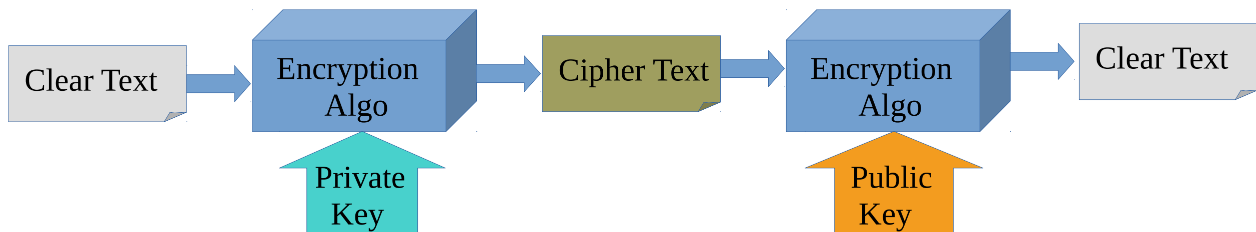
The key in symmetric key algorithms must be kept secret. Exchanging key between the sender and the receiver can be difficult. The same communication channel cannot be used and sending keys in clear is not a very good idea. Security is related to the key length, the longer the better.

Popular symmetric key algorithms are Triple DES, AES & Blowfish. Triple DES uses 112 bit key, AES supports key lengths of 128 bit or more.



3.2 Public Key Encryption

Public Key Encryption uses two keys: one that must remain secret is the private key and the one that has to be freely distributed is the public key. The public and the private key pair are related to each other in such a manner that a message encrypted by the public key can be decrypted only by its private key pair and vice versa. Hence there is no issue of key distribution.



Public keys are distributed with a bunch of supporting information called a certificate. Certificates are validated by trusted third parties called certification authority. A certification authority (CA) certifies that the owner of the public key is the one who is the named subject of the certificate.

Alice

Alice Private Key

Clear Text Message

Cipher Text Message

Hash

Message Digest

Signature

Public

Alice Public Key

Bob Public Key

Cipher Text Message

Signature

Bob

Bob Private Key

Cipher Text Message

Hash

Message Digest

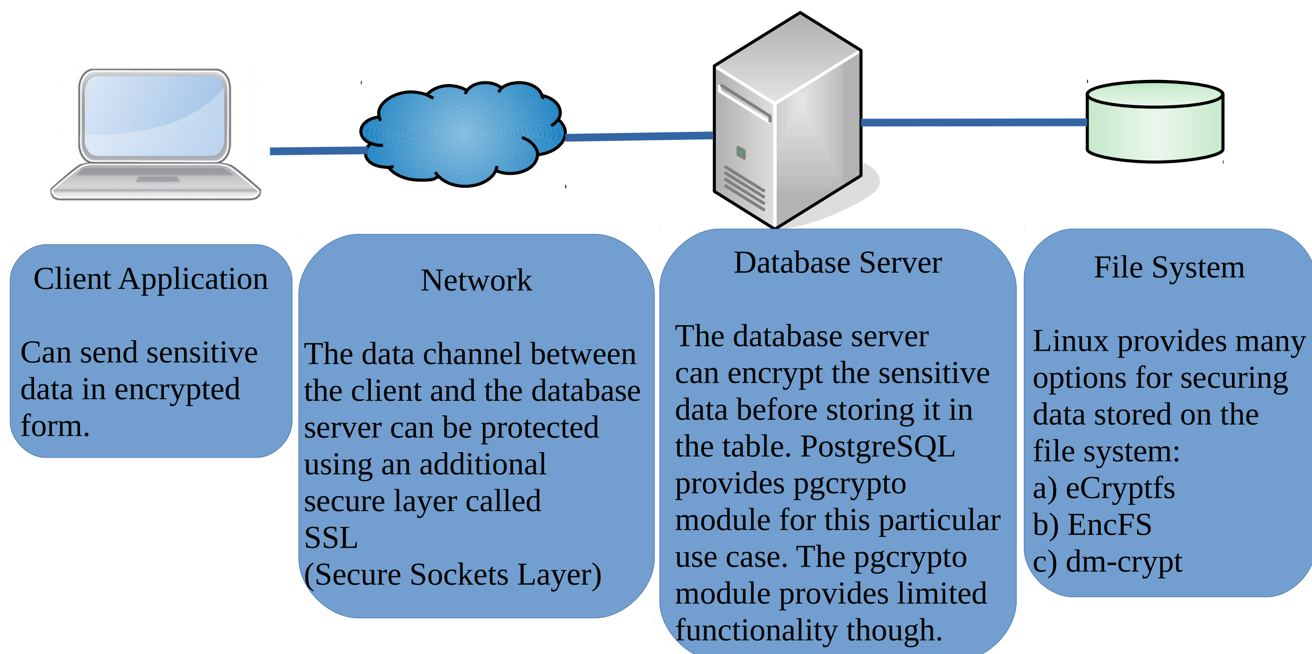
Clear Text Message

Signature

Message Digest

Equal

4 The Complete Picture



The PostgreSQL documentation lacks sample applications for securing data at different levels. For sample applications please check the following links:

https://github.com/gabbasb/SMEA

A Java based sample client application that encrypts sensitive messages before sending them over to the PostgreSQL server for storage.

https://github.com/gabbasb/PMEA

A C based sample client application that encrypts sensitive messages before sending them over to the PostgreSQL server for storage.

https://github.com/gabbasb/CMEA

A Java based sample application that uses pgcrypto for encrypting secret messages before they are stored in PostgreSQL.

5 Database Roles and Privileges

Once a user gets authenticated, authorization decides what that particular user can do. In PostgreSQL authorization is controlled by creating roles and assigning privileges to them.

5.1 Privileges Example

5.1.1 Setup

```
./psql -p 5432 -U abbas postgres
CREATE DATABASE src_db;
./psql -p 5432 -U abbas src_db
CREATE SCHEMA ms;
CREATE ROLE alice with NOSUPERUSER LOGIN;
CREATE TABLE ms.t1(a INT PRIMARY KEY, b VARCHAR(255));
INSERT INTO ms.t1 VALUES(1, 'One');
INSERT INTO ms.t1 VALUES(2, 'Two');
INSERT INTO ms.t1 VALUES(3, 'Three');

CREATE FUNCTION ms.ibv(val int, inc int) RETURNS integer AS
$$
BEGIN
  RETURN val + inc;
END;
$$
LANGUAGE PLPGSQL;
```

5.1.2 Revoke all privileges

All roles inherit privileges from PUBLIC role.

```
REVOKE ALL on DATABASE src_db FROM PUBLIC;
REVOKE ALL on SCHEMA ms FROM PUBLIC;
REVOKE ALL on TABLE ms.t1 FROM PUBLIC;
REVOKE ALL on FUNCTION ms.ibv FROM PUBLIC;
REVOKE ALL on LANGUAGE PLPGSQL FROM PUBLIC;
```

5.1.3 CONNECT

```
./psql -p 5432 -U alice src_db

psql: error: could not connect to server: FATAL:  permission denied for
database "src_db"

DETAIL:  User does not have CONNECT privilege.

./psql -p 5432 -U abbas src_db

src_db=# GRANT CONNECT on DATABASE src_db TO alice;

GRANT

./psql -p 5432 -U alice src_db

src_db=> \c

You are now connected to database "src_db" as user "alice".
```

5.1.4 USAGE

```
./psql -p 5432 -U alice src_db

src_db=> SELECT * FROM ms.t1;

ERROR:  permission denied for schema ms

./psql -p 5432 -U abbas src_db

src_db=# GRANT USAGE ON SCHEMA ms TO alice;

GRANT
```

5.1.5 SELECT

```
./psql -p 5432 -U alice src_db

src_db=> SELECT * FROM ms.t1;

ERROR:  permission denied for table t1

./psql -p 5432 -U abbas src_db

src_db=# GRANT SELECT ON TABLE ms.t1 TO alice;

GRANT

src_db=# \q

./psql -p 5432 -U alice src_db

src_db=> SELECT * FROM ms.t1;

 a |   b

---+-------

 1 | One

 2 | Two

 3 | Three

(3 rows)
```

5.1.6 EXECUTE

```
./psql -p 5432 -U alice src_db
src_db=> SELECT ms.ibv(10,20);
ERROR:  permission denied for function ibv
./psql -p 5432 -U abbas src_db
src_db=# GRANT EXECUTE ON FUNCTION ms.ibv TO alice;
GRANT
./psql -p 5432 -U alice src_db
src_db=> SELECT ms.ibv(10,20);
 ibv
-----
  30
```

5.1.7 INSERT

```
./psql -p 5432 -U alice src_db
src_db=> INSERT INTO ms.t1 VALUES(4,'Four');
ERROR:  permission denied for table t1
./psql -p 5432 -U abbas src_db
src_db=# GRANT INSERT ON TABLE ms.t1 TO alice;
GRANT
./psql -p 5432 -U alice src_db
src_db=> INSERT INTO ms.t1 VALUES(4,'Four');
INSERT 0 1
```

5.1.8 UPDATE

```
./psql -p 5432 -U alice src_db
src_db=> UPDATE ms.t1 SET b='Threee' where a = 3;
ERROR:  permission denied for table t1
./psql -p 5432 -U abbas src_db
src_db=# GRANT UPDATE ON TABLE ms.t1 TO alice;
GRANT
./psql -p 5432 -U alice src_db
src_db=> UPDATE ms.t1 SET b='Threee' where a = 3;
UPDATE 1
```

5.1.9 DELETE

```
./psql -p 5432 -U alice src_db
src_db=> DELETE FROM ms.t1 WHERE a = 4;
ERROR:  permission denied for table t1
./psql -p 5432 -U abbas src_db
src_db=# GRANT DELETE ON TABLE ms.t1 TO alice;
GRANT
./psql -p 5432 -U alice src_db
src_db=> DELETE FROM ms.t1 WHERE a = 4;
DELETE 1
```

5.1.10 TRUNCATE

```
./psql -p 5432 -U alice src_db
src_db=> TRUNCATE TABLE ms.t1;
ERROR:  permission denied for table t1
./psql -p 5432 -U abbas src_db
src_db=# GRANT TRUNCATE ON TABLE ms.t1 TO alice;
GRANT
./psql -p 5432 -U alice src_db
src_db=> TRUNCATE TABLE ms.t1;
TRUNCATE TABLE
```

5.1.11 CREATE

```
./psql -p 5432 -U alice src_db
src_db=> CREATE TABLE ms.t2(c INT PRIMARY KEY, d INT REFERENCES ms.t1(a));
ERROR:  permission denied for schema ms
LINE 1: CREATE TABLE ms.t2(c INT PRIMARY KEY, d INT REFERENCES ms.t1...
./psql -p 5432 -U abbas src_db
src_db=# GRANT CREATE ON SCHEMA ms TO alice;
GRANT
./psql -p 5432 -U alice src_db
src_db=> CREATE TABLE ms.t2(c INT PRIMARY KEY, d INT REFERENCES ms.t1(a));
ERROR:  permission denied for table t1
```

5.1.12 REFERENCES

```
./psql -p 5432 -U abbas src_db
src_db=# GRANT REFERENCES ON TABLE ms.t1 TO alice;
GRANT
./psql -p 5432 -U alice src_db
src_db=> CREATE TABLE ms.t2(c INT PRIMARY KEY, d INT REFERENCES ms.t1(a));
CREATE TABLE
```

5.1.13 USAGE

```
./psql -p 5432 -U alice src_db
src_db=> CREATE OR REPLACE FUNCTION ms.for_test() RETURNS trigger AS
src_db-> $$
src_db$> BEGIN
src_db$>    RETURN NEW;
src_db$> END;
src_db$> $$
src_db-> LANGUAGE PLPGSQL;
ERROR:  permission denied for language plpgsql
./psql -p 5432 -U abbas src_db
src_db=# GRANT USAGE ON LANGUAGE PLPGSQL TO alice;
GRANT
./psql -p 5432 -U alice src_db
src_db=> CREATE OR REPLACE FUNCTION ms.for_test() RETURNS trigger AS
$$
BEGIN
   RETURN NEW;
END;
$$
LANGUAGE PLPGSQL;
CREATE FUNCTION
```

5.1.14 TRIGGER

```
./psql -p 5432 -U alice src_db

src_db=> CREATE TRIGGER trig_test BEFORE INSERT ON ms.t1 FOR EACH ROW
EXECUTE PROCEDURE ms.for_test();

ERROR:  permission denied for table t1

./psql -p 5432 -U abbas src_db

src_db=# GRANT TRIGGER ON TABLE ms.t1 TO alice;

GRANT

./psql -p 5432 -U alice src_db

src_db=> CREATE TRIGGER trig_test BEFORE INSERT ON ms.t1 FOR EACH ROW
EXECUTE PROCEDURE ms.for_test();

CREATE TRIGGER
```

5.1.15 TEMPORARY

```
./psql -p 5432 -U alice src_db

src_db=> CREATE TEMPORARY TABLE temp_tab(id INT, fname VARCHAR(80) ) ON
COMMIT DELETE ROWS;

ERROR:  permission denied to create temporary tables in database "src_db"

LINE 1: CREATE TEMPORARY TABLE temp_tab(id INT, fname VARCHAR(80) ) ...

./psql -p 5432 -U abbas src_db

src_db=# GRANT TEMPORARY ON DATABASE src_db TO alice;

GRANT

./psql -p 5432 -U alice src_db

src_db=> CREATE TEMPORARY TABLE temp_tab(id INT, fname VARCHAR(80) ) ON
COMMIT DELETE ROWS;

CREATE TABLE
```

5.2 Display Access Privileges

Privileges are displayed by default as grantee=privileges/granter

Privileges are displayed using the following shorthand notation:

```
r -- SELECT ("read")
w -- UPDATE ("write")
a -- INSERT ("append")
d -- DELETE
D -- TRUNCATE
x -- REFERENCES
t -- TRIGGER
X -- EXECUTE
U -- USAGE
C -- CREATE
c -- CONNECT
T -- TEMPORARY
```

```
src_db=# \l src_db
                             List of databases
  Name   | Owner | Access privileges
--------+-------+-------------------
 src_db | abbas | abbas=CTc/abbas  +
                | alice=Tc/abbas
src_db=# \dn+ ms
                List of schemas
 Name | Owner | Access privileges | Description
------+-------+-------------------+-------------
 ms   | abbas | abbas=UC/abbas   +|
      |       | alice=UC/abbas    |
src_db=# \dp ms.t1;
                             Access privileges
 Schema | Name | Type  |  Access privileges
--------+------+-------+---------------------+
 ms     | t1   | table | abbas=arwdDxt/abbas+|
        |      |       | alice=arwdDxt/abbas |
src_db=# \df+ ms.ibv
               List of functions
 Schema | Name | Access privileges | Language
--------+------+-------------------+--------
 ms     | ibv  | abbas=X/abbas    +| plpgsql
               | alice=X/abbas     |
```

```
src_db=# \dL+ PLPGSQL
                 List of languages
  Name    | Owner | Access privileges |          Description
---------+-------+-------------------+------------------------------
 plpgsql | abbas | abbas=U/abbas    +| PL/pgSQL procedural language
         |       | alice=U/abbas     |
```

5.3 Displaying privileges in more readable manner

Instead of the default syntax that uses short hand for privileges: **grantee=privileges/granter**

we can use the function aclexplode to display them as:

granter oid, grantee oid, granted privilege, is privilege grant-able

```
select relname, relacl from pg_class where relname like 't1';
 relname |                  relacl
---------+-------------------------------------------
 t1      | {abbas=arwdDxt/abbas,alice=arwdDxt/abbas}
select relname, aclexplode(relacl) from pg_class where relname like 't1';
 relname |       aclexplode
---------+------------------------
 t1      | (10,10,INSERT,f)
 t1      | (10,10,SELECT,f)
 t1      | (10,10,UPDATE,f)
 t1      | (10,10,DELETE,f)
 t1      | (10,10,TRUNCATE,f)
 t1      | (10,10,REFERENCES,f)
 t1      | (10,10,TRIGGER,f)
 t1      | (10,16438,INSERT,f)
 t1      | (10,16438,SELECT,f)
 t1      | (10,16438,UPDATE,f)
 t1      | (10,16438,DELETE,f)
 t1      | (10,16438,TRUNCATE,f)
 t1      | (10,16438,REFERENCES,f)
 t1      | (10,16438,TRIGGER,f)
```

```
select proname, proacl from pg_proc where proname like 'ibv';
 proname |             proacl
---------+-------------------------------
 ibv     | {abbas=X/abbas,alice=X/abbas}
select proname, aclexplode(proacl) from pg_proc where proname like 'ibv';
 proname |      aclexplode
---------+----------------------
 ibv     | (10,10,EXECUTE,f)
 ibv     | (10,16438,EXECUTE,f)
```

5.4 System Provided Functions for ACLs

PostgreSQL provides many function to check whether a user has a certain privilege on a certain database object. For Example:

```
select has_table_privilege('alice', 'ms.t1', 'INSERT');
 has_table_privilege
---------------------
 t
select has_schema_privilege('alice', 'ms', 'USAGE');
 has_schema_privilege
----------------------
 t
```

6 Row Level Security

In PostgreSQL it is possible to restrict access to tables rows, for a certain role, while using SELECT, INSERT, UPDATE or DELETE.

6.1 RLS Example

We will create a sample message exchange application. Users will be able to store messages in the database for their friends. Friends will be able to read the messages that have been sent to them. The database has only two tables. One table stores users and their friends and the other table stores messages. The following policies will be applied using RLS.

1. Users can only see their own friends. Users will not be able to see their other user's own friends.
2. Users can only add their own friends. Users cannot add friends for any other user.
3. A user can send messages to his/her friends only.
4. A user can read only the messages that were sent to him/her.

RLS policies can be created by either using WITH CHECK clause or USING clause.

The main difference between WITH CHECK and USING clause is that USING clause does not apply to INSERTs whereas WITH CHECK does not apply to DELETEs.

When a policy created using WITH CHECK clause is violated an error message is thrown by the database server, no error is thrown in case of USING clause but the query fails in both cases.

We will use RLS to apply some policies in our sample message sending application as follows:

1. Admin user and a database is created.

```
./psql -p 5432 -U abbas postgres
CREATE DATABASE my_db;
CREATE ROLE admin with NOSUPERUSER LOGIN CREATEROLE;
GRANT CREATE ON DATABASE my_db TO admin;
```

2. Admin creates roles and message exchange application tables, admin then inserts all roles as users

```
./psql -p 5432 -U admin my_db
CREATE SCHEMA mea;
```

```
CREATE ROLE tina with NOSUPERUSER LOGIN;
CREATE ROLE eve with NOSUPERUSER LOGIN;
CREATE ROLE tom with NOSUPERUSER LOGIN;
CREATE ROLE harry with NOSUPERUSER LOGIN;

CREATE TABLE mea.tbl_users(
u_id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
u_name varchar(255) NOT NULL UNIQUE,
u_friends INT[]);

CREATE TABLE mea.tbl_messages(
m_id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
m_from_u_id INT REFERENCES mea.tbl_users(u_id),
m_to_u_id INT REFERENCES mea.tbl_users(u_id),
m_message VARCHAR NOT NULL,
m_sent_on timestamp with time zone NOT NULL);

INSERT INTO mea.tbl_users (u_name, u_friends) VALUES('tina', '{}');
INSERT INTO mea.tbl_users (u_name, u_friends) VALUES('eve', '{}');
INSERT INTO mea.tbl_users (u_name, u_friends) VALUES('tom', '{}');
INSERT INTO mea.tbl_users (u_name, u_friends) VALUES('harry', '{}');
```

3. Admin revokes all privileges

```
REVOKE ALL on SCHEMA mea FROM PUBLIC;
REVOKE ALL on TABLE mea.tbl_users FROM PUBLIC;
REVOKE ALL on TABLE mea.tbl_messages FROM PUBLIC;
```

4. Admin explicitly grants required privileges to roles

```
GRANT USAGE ON SCHEMA mea TO tina;
GRANT USAGE ON SCHEMA mea TO eve;
GRANT USAGE ON SCHEMA mea TO tom;
GRANT USAGE ON SCHEMA mea TO harry;
```

```
GRANT SELECT ON TABLE mea.tbl_users TO tina;
GRANT SELECT ON TABLE mea.tbl_users TO eve;
GRANT SELECT ON TABLE mea.tbl_users TO tom;
GRANT SELECT ON TABLE mea.tbl_users TO harry;

GRANT UPDATE ON TABLE mea.tbl_users TO tina;
GRANT UPDATE ON TABLE mea.tbl_users TO eve;
GRANT UPDATE ON TABLE mea.tbl_users TO tom;
GRANT UPDATE ON TABLE mea.tbl_users TO harry;

GRANT INSERT ON TABLE mea.tbl_messages TO tina;
GRANT INSERT ON TABLE mea.tbl_messages TO eve;
GRANT INSERT ON TABLE mea.tbl_messages TO tom;
GRANT INSERT ON TABLE mea.tbl_messages TO harry;

GRANT UPDATE ON TABLE mea.tbl_messages TO tina;
GRANT UPDATE ON TABLE mea.tbl_messages TO eve;
GRANT UPDATE ON TABLE mea.tbl_messages TO tom;
GRANT UPDATE ON TABLE mea.tbl_messages TO harry;

GRANT SELECT ON TABLE mea.tbl_messages TO tina;
GRANT SELECT ON TABLE mea.tbl_messages TO eve;
GRANT SELECT ON TABLE mea.tbl_messages TO tom;
GRANT SELECT ON TABLE mea.tbl_messages TO harry;
```

5. Users can only update their own friends. Users cannot update any other user's friends. Also users can only see their own friends, they cannot see other user's friends.

```
ALTER TABLE mea.tbl_users ENABLE ROW LEVEL SECURITY;

CREATE POLICY add_friends_s ON mea.tbl_users AS PERMISSIVE FOR SELECT TO
tina, eve, tom, harry USING (u_name = current_user);

CREATE POLICY add_friends_u ON mea.tbl_users AS PERMISSIVE FOR UPDATE TO
tina, eve, tom, harry USING (u_name = current_user);

Note : Policies are not created in any schema
```

6. Add some friends

```
./psql -p 5432 -U tina my_db
my_db=> explain verbose select * from mea.tbl_users;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on mea.tbl_users  (cost=0.00..12.10 rows=1 width=552)
   Output: u_id, u_name, u_friends
   Filter: ((tbl_users.u_name)::text = CURRENT_USER)
(3 rows)
my_db=> select * from mea.tbl_users;
 u_id | u_name | u_friends
------+--------+-----------
    1 | tina   | {}
(1 row)

my_db=> explain verbose UPDATE mea.tbl_users SET u_friends = '{2,3,4}'
WHERE u_id = 1;
                      QUERY PLAN
-------------------------------------------------------------------------
 Update on mea.tbl_users  (cost=0.14..8.17 rows=1 width=558)
   ->  Index Scan using tbl_users_pkey on mea.tbl_users  (cost=0.14..8.17
rows=1 width=558)
         Output: u_id, u_name, '{2,3,4}'::integer[], ctid
         Index Cond: (tbl_users.u_id = 1)
         Filter: ((tbl_users.u_name)::text = CURRENT_USER)
(5 rows)
```

Note that the condition used in the USING clause appears in the Filter node of the plan. This means that the USING condition is evaluated as a qual. This is not true for the condition used in the WITH CHECK clause.

```
my_db=> UPDATE mea.tbl_users SET u_friends = '{2,3,4}' WHERE u_id = 1;
UPDATE 1
```

7. Try adding friends of some other user

```
my_db=> UPDATE mea.tbl_users SET u_friends = '{1,3,4}' WHERE u_id = 2;
UPDATE 0
No error is thrown because the policy was created using the USING clause.
If we had used WITH CHECK then an error would have occurred.
```

8. Add some more friends

```
./psql -p 5432 -U eve my_db
my_db=> select * from mea.tbl_users;
 u_id | u_name | u_friends
------+--------+-----------
    2 | eve    | {}
(1 row)
my_db=> UPDATE mea.tbl_users SET u_friends = '{1,3,4}' WHERE u_id = 2;
UPDATE 1
./psql -p 5432 -U tom my_db
my_db=> select * from mea.tbl_users;
 u_id | u_name | u_friends
------+--------+-----------
    3 | tom    | {}
(1 row)
my_db=> UPDATE mea.tbl_users SET u_friends = '{1,2}' WHERE u_id = 3;
UPDATE 1
```

9. A user can send (insert/update) message to a another user only if u_id of friend is already in u_friends array

```
./psql -p 5432 -U admin my_db
CREATE OR REPLACE FUNCTION mea.isFriend(p_u_id integer, p_f_id integer)
RETURNS boolean AS $$
  DECLARE ret BOOLEAN;
BEGIN
  SELECT p_f_id = ANY(u_friends) INTO ret FROM mea.tbl_users
  WHERE u_id = p_u_id;
  RETURN ret;
END; $$
LANGUAGE PLPGSQL;
```

```
my_db=> SELECT * FROM mea.tbl_users;
 u_id | u_name | u_friends
------+--------+-----------
    4 | harry  | {}
    1 | tina   | {2,3,4}
    2 | eve    | {1,3,4}
    3 | tom    | {1,2}
(4 rows)
my_db=> SELECT mea.isFriend(3,4);
 isfriend
----------
 f
(1 row)
my_db=> SELECT mea.isFriend(3,1);
 isfriend
----------
 t
(1 row)
ALTER TABLE mea.tbl_messages ENABLE ROW LEVEL SECURITY;

CREATE POLICY send_messages ON mea.tbl_messages AS PERMISSIVE
FOR INSERT TO tina, eve, tom, harry
WITH CHECK (mea.isFriend(m_from_u_id, m_to_u_id));

CREATE POLICY update_messages ON mea.tbl_messages AS PERMISSIVE
FOR UPDATE TO tina, eve, tom, harry
WITH CHECK (mea.isFriend(m_from_u_id, m_to_u_id));
```

10. Send some messages

```
./psql -p 5432 -U tina my_db

my_db=> explain verbose INSERT INTO mea.tbl_messages
        (m_from_u_id, m_to_u_id, m_message, m_sent_on) VALUES
         (1, 2, 'Hello this is first message', now());
                QUERY PLAN
-----------------------------------------------------------------
 Insert on mea.tbl_messages  (cost=0.00..0.01 rows=1 width=52)
   ->  Result  (cost=0.00..0.01 rows=1 width=52)
         Output: nextval('mea.tbl_messages_m_id_seq'), 1, 2, 'Hello this
is first message'::character varying, now()
(3 rows)
Note there is no qual in the plan.


my_db=> INSERT INTO mea.tbl_messages
        (m_from_u_id, m_to_u_id, m_message, m_sent_on) VALUES
        (1, 2, 'Hello this is first message', now());
INSERT 0 1
my_db=> INSERT INTO mea.tbl_messages
        (m_from_u_id, m_to_u_id, m_message, m_sent_on) VALUES
        (1, 3, 'Hello this is second message', now());
INSERT 0 1
./psql -p 5432 -U tom my_db
my_db=> SELECT * FROM mea.tbl_users;
 u_id | u_name | u_friends
------+--------+-----------
    3 | tom    | {1,2}
(1 row)
my_db=> INSERT INTO mea.tbl_messages
        (m_from_u_id, m_to_u_id, m_message, m_sent_on) VALUES
        (3, 1, 'Hello this is third message', now());
INSERT 0 1
```

```
my_db=> INSERT INTO mea.tbl_messages
        (m_from_u_id, m_to_u_id, m_message, m_sent_on) VALUES
        (3, 4, 'Hello this is third message', now());
ERROR:  new row violates row-level security policy for table
"tbl_messages"
```

Note that an error is thrown for policies created WITH CHECK option.

11. A user can read (SELECT) messages only if they were sent to him

```
./psql -p 5432 -U admin my_db
CREATE OR REPLACE FUNCTION mea.getUserID()
RETURNS INTEGER AS $$
  DECLARE c_uid INTEGER;
BEGIN
  IF NOT EXISTS (SELECT u_id FROM mea.tbl_users WHERE u_name =
current_user) THEN
    RETURN 0;
  END IF;
  SELECT u_id INTO c_uid FROM mea.tbl_users WHERE u_name = current_user;
  RETURN c_uid;
END; $$
LANGUAGE PLPGSQL;

CREATE POLICY read_messages ON mea.tbl_messages AS PERMISSIVE
FOR SELECT TO tina, eve, tom, harry
USING (m_to_u_id = mea.getUserID());
```

12. Read Messages

```
./psql -p 5432 -U eve my_db
my_db=> EXPLAIN VERBOSE select * from mea.tbl_messages;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on mea.tbl_messages  (cost=0.00..277.75 rows=5 width=52)
   Output: m_id, m_from_u_id, m_to_u_id, m_message, m_sent_on
   Filter: (tbl_messages.m_to_u_id = mea.getuserid())
(3 rows)
```

```
my_db=> select m_id as id, m_from_u_id as frm, m_to_u_id as to, m_message
from mea.tbl_messages;
 id | frm | to |           m_message
----+-----+----+----------------------------
  1 |   1 |  2 | Hello this is first message
(1 row)


./psql -p 5432 -U tom my_db
my_db=> select m_id as id, m_from_u_id as frm, m_to_u_id as to, m_message
from mea.tbl_messages;
 id | frm | to |           m_message
----+-----+----+-----------------------------
  2 |   1 |  3 | Hello this is second message
(1 row)


./psql -p 5432 -U harry my_db
my_db=> select * from mea.tbl_messages;
 m_id | m_from_u_id | m_to_u_id | m_message | m_sent_on
------+-------------+-----------+-----------+-----------
(0 rows)
```

13. List policies defined

```
./psql -p 5432 -U admin my_db
my_db=> select tablename, cmd, qual, with_check from pg_policies;
 tablename   | cmd  |             qual             |             with_check
------------+------+------------------------------+----------------------------------
tbl_users   |SELECT|((u_name)::text = CURRENT_USER)|
tbl_users   |UPDATE|((u_name)::text = CURRENT_USER)|
tbl_messages|INSERT|                              |mea.isfriend(m_from_u_id, m_to_u_id)
tbl_messages|UPDATE|                              |mea.isfriend(m_from_u_id, m_to_u_id)
tbl_messages|SELECT|(m_to_u_id = mea.getuserid())  |
(5 rows)
```

6.2 pg_dump and RLS

6.2.1 row_security

When this configuration parameter is set to off, an error is thrown if any query's results would get filtered by a policy. The reason for the error can then be investigated and fixed. This is required to make sure pg_dump does not silently omit rows from the backup.

6.2.2 Enable-row-security

If pg_dump is used along with –enable-row-security, then only that content will be dumped to which the user has access to.