SCHARP

at FRED HUTCH

Viewing data at the intersection between roles

By Lloyd Albin

SCHARP at FRED HUTCH

Where to find this presentation:

- Blog
  - http://lloyd.thealbins.com/intersection%20between%20roles
- Presentation (with PowerPoint & SQL Files)
  - http://lloyd.thealbins.com/intersection%20between%20roles%20presentation

Additional notes in the note section of the slide when you see this icon (🗒) in the upper right corner of the slide.

SCHARP at FRED HUTCH

What this presentation covers:

- We ran across a use case where we needed to restrict people's access to the data by requiring them to belong to 2 or more groups. PostgreSQL by default does not support this.
- The role permissions in Postgres only supports OR. This use case needs to support AND.
- The Blog was my response to the developers on how to do this and which method was the best and/or fastest method.
- In our use case, we wanted the user to SELECT on a table and return an error if the user did not have the correct rights instead of returning an empty table.
- This is because, if you have a table containing adverse events, it could jeopardize human safety if users thought there were no adverse events, when, in fact, it's just that they didn't have permission to see the adverse events.

Symbol Key:

- These symbols will be used on many of the future slides.

Icon	Meaning
▦	Table
🔍	View
$f(x)$	Function
✅	Normal Table Owner
⛔	Table must be owned by Superuser
❌	No Data
🗋	Can view empty data set
🗐	Can view data set with data
🙂	Happy with Results
☹	Un-Happy with Results

SCHARP at FRED HUTCH

Criteria:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐		

- This is what was wished for, but is not possible. Now we need to figure out what we can accomplish and how it affects performance.
- Any Ideas?

Check users Groups

What we need to find out is, does this user belong to these roles/groups. As normal, there are several ways to ask this question. While style 1 works, the more groups that you need to check against the uglier the code becomes. While I am told two groups right now, I can see a possibility of more than two in the future.

Style 2 is cleaner code and handles more than two roles/groups easily.

I run into some issues along the way, such as using the WHERE clause in the views/rules using the functions. This does not work, because the WHERE clause does not get evaluated when the source table is empty. This is why I moved the function to act as a source table within the views/rules so that it would get evaluated every time and throw errors when there was partial permissions.

```
-- Check Group Permissions – Style 1
SELECT CASE WHEN
   pg_has_role(current_user, 'group_a', 'MEMBER') IS TRUE
   AND pg_has_role(current_user, 'group_b', 'MEMBER') IS TRUE
   THEN TRUE
   ELSE FALSE
   END AS check;

-- Check Group Permissions – Style 2
SELECT (array_agg(role_name::text) @>
   ARRAY['group_a', 'group_b']) AS check
      FROM information_schema.applicable_roles
      WHERE grantee = current_user;
```

Setup for all examples

When I show each sample set in this presentation, you will need to start with an empty database and run this generic setup for each of the examples.

```sql
-- Setup for all examples
-- Create a new database and then login as the superuser
-- This allows us to switch roles easily for testing
-- Create the accounts needed for this test
CREATE ROLE group_a;
CREATE ROLE group_b;
CREATE ROLE user_a WITH INHERIT IN ROLE group_a, group_b;
CREATE ROLE user_b WITH INHERIT IN ROLE group_a;
CREATE ROLE user_c;
CREATE ROLE table_owner;

-- Change to the table owner to create the table and insert
the data
SET ROLE table_owner;

CREATE TABLE public.test (
   id INTEGER,
   string TEXT
);

INSERT INTO public.test VALUES
    (1, 'testing’),
    (2, 'more testing’),
    (3, 'even more testing');
```

TABLE

Setup for Table

The problem with the plain TABLE is that if you only belong to one of the two group, you can view the data. We want to enforce that you must belong to both groups. This means that a plain table with normal permissions will not work for what we want.

The con's:

- Can view the data if you have partial permissions.

```
-- In an empty database run the "Setup for all examples"
-- first then run the code below

-- Grant both groups SELECT permissions on the table
GRANT SELECT ON public.test TO group_a;
GRANT SELECT ON public.test TO group_b;

SET ROLE user_a; -- Privileged User
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 31 ms)

SET ROLE user_b; -- Partial Privileged User
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 0 ms)

SET ROLE user_c; -- Un-Privileged User
SELECT * FROM public.test;
-- ERROR: permission denied for relation test

SET ROLE group_a; -- 1st Group
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 0 ms)

SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 16 ms)

SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 16 ms)

SET ROLE postgres; -- Superuser
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 15 ms)
```

Results:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐		
Feature Chart							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
TABLE	▦	✓	✗	🗋 🗐	🗋 🗐	🗋 🗐	🗋 🗐

- With the Table, user_b who belongs to only one of the two groups can read the table. This is a security issue.

SCHARP at FRED HUTCH

VIEW

SCHARP
at FRED HUTCH

Setup for View

For this method, no SELECT permissions are granted on the table. Instead, there's a view to the table and each role is granted separately. The role intersection check is implemented in a LEFT JOIN. The problem with the VIEW is that if you only belong to one of the two groups, you see an empty dataset instead of knowing that you did not have the correct permissions.

The con's:

- Empty table returned when you don't have all the correct permissions instead of an error message.

```sql
-- In an empty database run the "Setup for all examples"
-- first then run the code below

SET ROLE table_owner;

CREATE OR REPLACE VIEW public.test_view AS
SELECT a.*
FROM public.test a
LEFT JOIN (
   SELECT (array_agg(role_name::text) @>
      ARRAY['group_a', 'group_b']) AS check
         FROM information_schema.applicable_roles
         WHERE grantee = current_user
) b ON (TRUE)
WHERE b.check = TRUE;
GRANT SELECT ON public.test_view TO group_a;
GRANT SELECT ON public.test_view TO group_b;
```

Check with Dataset

☺
```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test_view;
-- 3 rows returned (execution time: 0 ms; total time: 16 ms)
```

☺
```
SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 32 ms)
```

☺
```
SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test_view;
-- ERROR: permission denied for relation test
```

☹
```
SET ROLE group_a; -- 1st Group
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 16 ms)
```

☹
```
SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 16 ms)
```

☺
```
SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 16 ms)
```

☺
```
SET ROLE postgres; -- Superuser
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 15 ms)
```

Check with Empty Data Set

```
-- Check to see what happens with an empty dataset
SET ROLE table_owner;
TRUNCATE TABLE public.test;
```

☺
```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 16 ms)
```

☹
```
SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 16 ms)
```

☺
```
SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test_view;
-- ERROR: permission denied for relation test
```

☹
```
SET ROLE group_a; -- 1st Group
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

☹
```
SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

☹
```
SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

☹
```
SET ROLE postgres; -- Superuser
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

SCHARP at FRED HUTCH

Results:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	📄 🗐		
Feature Chart							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
VIEW	🔍	✓	✗	📄	📄 🗐	📄	📄

- With the View, user_b who belongs to only one of the two groups see's a blank data set assuming there are no records. This is actually wrong and needs to be prevented. It is also a security issue.

SCHARP at FRED HUTCH

FUNCTION

Setup for Function

For this method, no SELECT permissions are granted on the table. Instead, there's a function that reads table and each role is granted separately on the function. The role intersection check is implemented inside the function. While a function may seem like a simple fix, it is not. Normally you would set the function to be owned by the table_owner and run as security definer. The problem with this is that the table_owner can't see the results from information_schema.applicable_roles for any other user but table_owner. This means that the function must be owned by a superuser and run as security definer for it to work properly. This is normally a bad practice, so I do not recommend this style.

- Function must be owned by a super user with execution as security definer

```
-- In an empty database run the "Setup for all examples"
-- first then run the code below

SET ROLE postgres;

CREATE OR REPLACE FUNCTION public.test ()
RETURNS SETOF public.test AS
$body$
DECLARE
   r RECORD;
   check_roles NAME[];
BEGIN
   check_roles = ARRAY['group_a'::name, 'group_b'::name];
   -- Check to see if the current_user is a direct member of the
check_roles
   SELECT ((SELECT array_agg(role_name::name)
      FROM information_schema.applicable_roles
      WHERE grantee = session_user) @> check_roles)
      AS security_check
      INTO r;
   IF r.security_check = TRUE THEN
      -- current_user was found to be a member of all check_roles
      RETURN QUERY SELECT * FROM public.test;
   ELSE
      -- current_user was NOT found to be a member of all check_roles
      RAISE EXCEPTION 'test(): User: % is required to be a member of
all of these groups: %', session_user, check_roles;
   END IF;
END;
$body$
LANGUAGE 'plpgsql'
STABLE
CALLED ON NULL INPUT
SECURITY DEFINER;
```

Check with Dataset

```
GRANT EXECUTE ON FUNCTION public.test() TO group_a, group_b;
REVOKE EXECUTE ON FUNCTION public.test() FROM PUBLIC;

SET SESSION AUTHORIZATION user_a; -- Privilaged User
SELECT * FROM public.test();
-- 3 rows returned (execution time: 0 ms; total time: 15 ms)

SET SESSION AUTHORIZATION user_b; -- Partial Privilaged User
SELECT * FROM public.test();
-- ERROR: test(): User: user_b is required to be a member of all of these groups: {group_a,group_b}

SET SESSION AUTHORIZATION user_c; -- Un-Privilaged User
SELECT * FROM public.test();
-- ERROR: permission denied for function test

SET SESSION AUTHORIZATION group_a; -- 1st Group
SELECT * FROM public.test();
-- ERROR: test(): User: group_a is required to be a member of all of these groups: {group_a,group_b}

SET SESSION AUTHORIZATION group_b; -- 2nd Group
SELECT * FROM public.test();
-- ERROR: test(): User: group_b is required to be a member of all of these groups: {group_a,group_b}

SET SESSION AUTHORIZATION table_owner; -- Table Owner
SELECT * FROM public.test();
-- ERROR: permission denied for function test

SET SESSION AUTHORIZATION postgres; -- Superuser
SELECT * FROM public.test();
-- ERROR: test(): User: postgres is required to be a member of all of these groups: {group_a,group_b}
```

Check with Empty Data Set

```
-- Check to see what happens with an empty dataset
SET ROLE table_owner;
TRUNCATE TABLE public.test;

SET SESSION AUTHORIZATION user_a; -- Privilaged User
SELECT * FROM public.test();
-- Empty set (execution time: 0 ms; total time: 0 ms

SET SESSION AUTHORIZATION user_b; -- Partial Privilaged User
SELECT * FROM public.test();
-- ERROR: test(): User: user_b is required to be a member of all of these groups: {group_a,group_b}

SET SESSION AUTHORIZATION user_c; -- Un-Privilaged User
SELECT * FROM public.test();
-- ERROR: permission denied for relation test

SET SESSION AUTHORIZATION group_a; -- 1st Group
SELECT * FROM public.test();
-- ERROR: test(): User: group_a is required to be a member of all of these groups: {group_a,group_b}

SET SESSION AUTHORIZATION group_b; -- 2nd Group
SELECT * FROM public.test();
-- EERROR: test(): User: group_b is required to be a member of all of these groups: {group_a,group_b}

SET SESSION AUTHORIZATION table_owner; -- Table Owner
SELECT * FROM public.test();
-- ERROR: permission denied for function test

SET SESSION AUTHORIZATION postgres; -- Superuser
SELECT * FROM public.test();
-- ERROR: test(): User: postgres is required to be a member of all of these groups: {group_a,group_b}
```

SCHARP at FRED HUTCH

Results:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐		
Feature Chart							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
FUNCTION	f(x)	⛔	✗	✗	🗋 🗐	✗	✗

- With the Function, we have what we want, but I don't like the function having to run as a super user and there would be one function per table.

TABLE with POLICY

Setup for Table with Policy

For this method, each role is granted SELECT permissions on the table. The role intersection check is implemented in the policy. This is sort of an unconventional way of using Row Level Security. We can perform the test we want using a Policy which requires PostgreSQL 9.5+. The problem with this method is that it is a per row evaluation. This means that if the table is empty, the evaluation does not happen.

The con's:

- Empty table returned when you don't have all the correct permissions instead of an error message.

```
-- In an empty database run the "Setup for all examples"
-- first then run the code below

-- Grant both groups SELECT permissions on the table
GRANT SELECT ON public.test TO group_a;
GRANT SELECT ON public.test TO group_b;

CREATE POLICY test_policy ON public.test
   USING ((SELECT array_agg(role_name::text)
      FROM information_schema.applicable_roles
      WHERE grantee = current_user) @> ARRAY['group_a', 'group_b']);

ALTER TABLE public.test
   ENABLE ROW LEVEL SECURITY;
```

Check with Dataset

🙂
```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test;
-- 3 rows returned (execution time: 15 ms; total time: 31 ms)
```

🙁
```
SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test;
-- ERROR: permission denied for relation test
```

🙁
```
SET ROLE group_a; -- 1st Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 16 ms)
```

🙁
```
SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 15 ms)
```

🙂
```
SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE postgres; -- Superuser
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 16 ms)
```

Check with Empty Data Set

```
-- Check to see what happens with an empty dataset
SET ROLE table_owner;
TRUNCATE TABLE public.test;
```

🙂
```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙁
```
SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test;
-- ERROR: permission denied for relation test
```

🙁
```
SET ROLE group_a; -- 1st Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙁
```
SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE postgres; -- Superuser
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

Results:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐		
Feature Chart							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
TABLE with POLICY 9.5+	▦	✓	✗	🗋	🗋 🗐	🗋 🗐	🗋 🗐

SCHARP at FRED HUTCH

TABLE with POLICY (FORCE’d)

SCHARP at FRED HUTCH

Setup for Table with Policy using forced Row Level Security

For this method, each role is granted SELECT permissions on the table. The role intersection check is implemented in the policy and also enforced against the table owner. This is sort of an unconventional way of using Row Level Security. We can perform the test we want using a Policy which requires PostgreSQL 9.5+ and enforce the policy against the table owner.

The con's:

- Empty table returned when you don't have all the correct permissions instead of an error message.
- Table owner, does not have the rights to read, write, update, or delete the data.

```
-- In an empty database run the "Setup for all examples"
-- first then run the code below

-- Grant both groups SELECT permissions on the table
GRANT SELECT ON public.test TO group_a;
GRANT SELECT ON public.test TO group_b;

CREATE POLICY test_policy ON public.test
   USING ((SELECT array_agg(role_name::text)
      FROM information_schema.applicable_roles
      WHERE grantee = current_user) @> ARRAY['group_a', 'group_b']);

ALTER TABLE public.test
   ENABLE ROW LEVEL SECURITY;

ALTER TABLE public.test
   FORCE ROW LEVEL SECURITY;
```

Check with Dataset

```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 0 ms)

SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test;
-- ERROR: permission denied for relation test

SET ROLE group_a; -- 1st Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE postgres; -- Superuser
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 0 ms)
```

Check with Empty Data Set

```
-- Check to see what happens with an empty dataset
SET ROLE table_owner;
TRUNCATE TABLE public.test;

SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test;
-- ERROR: permission denied for relation test

SET ROLE group_a; -- 1st Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE postgres; -- Superuser
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

SCHARP at FRED HUTCH

Results:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐		

Feature Chart							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
TABLE with POLICY 9.5+	▦	✓	✗	🗋	🗋 🗐	🗋 🗐	🗋

SCHARP at FRED HUTCH

TABLE with RULE

SCHARP
at FRED HUTCH

Setup for Table with Rule

For this method, no SELECT permissions are granted on the table. Instead, there's a rule/view to the table and each role is granted separately. The role intersection check is implemented in a WHERE clause of the rule. The problem with the RULE is that if you only belong to one of the two group, you get an empty dataset instead of an error message. Rules are how PostgreSQL implements views internally. Per the PostgreSQL Docs: "It is considered better style to write a CREATE VIEW command than to create a real table and define an ON SELECT rule for it."

The con's:

- Empty table returned when you don't have all the correct permissions instead of an error message.
- Must update the TABLE and RULE every time you change the underlying table structure.
- Once the RULE is applied, the TABLE turns into a VIEW.
- Must use DROP VIEW to get rid of the TABLE with RULE.

```
-- In an empty database run the "Setup for all examples"
-- first then run the code below

SET ROLE table_owner;
CREATE TABLE public.test_rule (LIKE public.test);
CREATE RULE "_RETURN" AS ON SELECT TO public.test_rule DO INSTEAD
   SELECT * FROM public.test
      WHERE (SELECT array_agg(role_name::text)
            FROM information_schema.applicable_roles
            WHERE grantee = current_user)
         @> ARRAY['group_a', 'group_b'];

GRANT SELECT ON public.test_rule TO group_a;
GRANT SELECT ON public.test_rule TO group_b;
```

SCHARP at FRED HUTCH

Check with Dataset

🙂
```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test_rule;
-- 3 rows returned (execution time: 0 ms; total time: 0 ms)
```

☹
```
SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 16 ms)
```

🙂
```
SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test_rule;
-- ERROR: permission denied for relation test
```

☹
```
SET ROLE group_a; -- 1st Group
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

☹
```
SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE postgres; -- Superuser
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

Check with Empty Data Set

```
-- Check to see what happens with an empty dataset
SET ROLE table_owner;
TRUNCATE TABLE public.test;
```

🙂
```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 15 ms)
```

☹
```
SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 15 ms)
```

🙂
```
SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test_rule;
-- ERROR: permission denied for relation test
```

☹
```
SET ROLE group_a; -- 1st Group
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

☹
```
SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE postgres; -- Superuser
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

Results:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐		
Feature Chart							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
TABLE with RULE	▦	✓	✗	🗋	🗋 🗐	🗋	🗋

VIEW with FUNCTION

Setup for View with Function

For this method, no SELECT permissions are granted on the table. Instead, there's a view to the table and each role is granted separately. The role intersection check is implemented just before the LEFT JOIN. This causes the security_check function to always be checked even if the table is empty. This does enforce most of what we want. Just wish it appeared as a table.

The con's:

- Empty table returned when you don't have all the correct permissions instead of an error message.
- Table owner, does not have the rights to read, write, update, or delete the data.

```
-- In an empty database run the "Setup for all examples"
-- first then run the code below

SET ROLE table_owner;
CREATE OR REPLACE FUNCTION public.security_check ( check_roles name [] )
RETURNS boolean AS $body$
DECLARE
  r RECORD;
BEGIN
  -- Checks to see if NULL was passed
  IF array_length(check_roles,1) IS NULL THEN
    RAISE EXCEPTION 'security_check(): Must specify roles to be checked.';
  END IF;
  -- Checks to see if any value in the array is NULL
  IF check_roles @> ARRAY[NULL::name] THEN
    RAISE EXCEPTION 'security_check(): NULL roles are not allowed in check_roles: %', check_roles;
  END IF;
  -- Check to see if the current_user is a direct member of the check_roles
  SELECT ((SELECT array_agg(role_name::name) FROM information_schema.applicable_roles WHERE
grantee = current_user) @> check_roles) AS security_check INTO r;
  IF r.security_check = TRUE THEN
    -- current_user was found to be a member of all check_roles
    RETURN TRUE;
  ELSE
    -- current_user was NOT found to be a member of all check_roles
    RAISE EXCEPTION 'security_check(): User: % is required to be a member of all of these groups: %',
current_user, check_roles;
  END IF;
END; $body$ LANGUAGE 'plpgsql'
STABLE
CALLED ON NULL INPUT
SECURITY INVOKER;
```

Setup for View with Function

For this method, no SELECT permissions are granted on the table. Instead, there's a view to the table and each role is granted separately. The role intersection check is implemented just before the LEFT JOIN. This causes the security_check function to always be checked even if the table is empty. This does enforce most of what we want. Just wish it appeared as a table.

The con's:

- Empty table returned when you don't have all the correct permissions instead of an error message.
- Table owner, does not have the rights to read, write, update, or delete the data.

```
GRANT EXECUTE ON FUNCTION public.security_check(check_roles name []) TO PUBLIC;

CREATE OR REPLACE VIEW public.test_view AS
  SELECT a.*
  FROM public.security_check(ARRAY['group_a', 'group_b'])
  LEFT JOIN public.test a ON (TRUE)
  WHERE a.id IS NOT NULL;

GRANT SELECT ON public.test_view TO group_a;
GRANT SELECT ON public.test_view TO group_b;
```

Check with Dataset

```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test_view;
-- 3 rows returned (execution time: 0 ms; total time: 0 ms)

SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: user_b is required to be a member of all of these groups: {group_a,group_b}

SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test_view;
-- ERROR: permission denied for relation test

SET ROLE group_a; -- 1st Group
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: group_a is required to be a member of all of these groups: {group_a,group_b}

SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: group_b is required to be a member of all of these groups: {group_a,group_b}

SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: table_owner is required to be a member of all of these groups:
{group_a,group_b}

SET ROLE postgres; -- Superuser
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: postgres is required to be a member of all of these groups: {group_a,group_b}
```

Check with Empty Data Set

```
-- Check to see what happens with an empty dataset
SET ROLE table_owner;
TRUNCATE TABLE public.test;

SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test_view;
-- Empty set (execution time: 0 ms; total time: 0 ms)

SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: user_b is required to be a member of all of these groups: {group_a,group_b}

SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test_view;
-- ERROR: permission denied for relation test

SET ROLE group_a; -- 1st Group
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: group_a is required to be a member of all of these groups: {group_a,group_b}

SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: group_b is required to be a member of all of these groups: {group_a,group_b}

SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: table_owner is required to be a member of all of these groups:
{group_a,group_b}

SET ROLE postgres; -- Superuser
SELECT * FROM public.test_view;
-- ERROR: security_check(): User: postgres is required to be a member of all of these groups: {group_a,group_b}
```

Results:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐		

Feature Chart							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
VIEW with FUNCTION	🔍	✓	✗	✗	🗋 🗐	✗	✗

TABLE with POLICY and FUNCTION

SCHARP
at FRED HUTCH

Setup for Table with Policy and Function

For this method, each role is granted SELECT permissions on the table. The role intersection check is implemented in the policy as a call to the security_check function. This is better than just having a table and policy because this method can throw errors for partial privileged users if there is data.

The con's:

- Empty table returned when you don't have all the correct permissions instead of an error message.
- Table owner, does not have the rights to read, write, update, or delete the data.

```
-- In an empty database run the "Setup for all examples"
-- first then run the code below

SET ROLE table_owner;
CREATE OR REPLACE FUNCTION public.security_check ( check_roles name [] )
RETURNS boolean AS $body$
DECLARE
  r RECORD;
BEGIN
  -- Checks to see if NULL was passed
  IF array_length(check_roles,1) IS NULL THEN
    RAISE EXCEPTION 'security_check(): Must specify roles to be checked.’;
  END IF;
  -- Checks to see if any value in the array is NULL
  IF check_roles @> ARRAY[NULL::name] THEN
    RAISE EXCEPTION 'security_check(): NULL roles are not allowed in check_roles: %', check_roles;
  END IF;
  -- Check to see if the current_user is a direct member of the check_roles
  SELECT ((SELECT array_agg(role_name::name)
    FROM information_schema.applicable_roles
    WHERE grantee = current_user) @> check_roles) AS security_check INTO r;
  IF r.security_check = TRUE THEN
    -- current_user was found to be a member of all check_roles
    RETURN TRUE;
  ELSE
    -- current_user was NOT found to be a member of all check_roles
    RAISE EXCEPTION 'security_check(): User: % is required to be a member of all of these groups: %',
current_user, check_roles;
  END IF;
END;
$body$
LANGUAGE 'plpgsql’
STABLE
CALLED ON NULL INPUT
SECURITY INVOKER;
```

Setup for Table with Policy and Function

```
GRANT EXECUTE ON FUNCTION public.security_check(check_roles name []) TO PUBLIC;

-- Grant both groups SELECT permissions on the table
GRANT SELECT ON public.test TO group_a;
GRANT SELECT ON public.test TO group_b;

CREATE POLICY test_policy ON public.test
  USING (public.security_check(ARRAY['group_a'::name, 'group_b'::name]));

ALTER TABLE public.test ENABLE ROW LEVEL SECURITY;
```

Check with Dataset

🙂
```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 15 ms)
```

🙂
```
SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test;
-- ERROR: security_check(): User: user_b is required to be a member of all of these groups: {group_a,group_b}
```

🙂
```
SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test;
-- ERROR: permission denied for relation test
```

🙂
```
SET ROLE group_a; -- 1st Group
SELECT * FROM public.test;
-- ERROR: security_check(): User: group_a is required to be a member of all of these groups: {group_a,group_b}
```

🙂
```
SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test;
-- ERROR: security_check(): User: group_b is required to be a member of all of these groups: {group_a,group_b}
```

🙂
```
SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE postgres; -- Superuser
SELECT * FROM public.test;
-- 3 rows returned (execution time: 0 ms; total time: 16 ms)
```

Check with Empty Data Set

```
-- Check to see what happens with an empty dataset
SET ROLE table_owner;
TRUNCATE TABLE public.test;
```

🙂
```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙁
```
SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test;
-- ERROR: permission denied for relation test
```

🙁
```
SET ROLE group_a; -- 1st Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙁
```
SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

🙂
```
SET ROLE postgres; -- Superuser
SELECT * FROM public.test;
-- Empty set (execution time: 0 ms; total time: 0 ms)
```

Results:

Our Criteria							
Style	User SELECT’s on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐	✗	
Feature Chart							
Style	User SELECT’s on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
TABLE with POLICY and FUNCTION	▦	✓	✗	🗋 ✗	🗋 🗐	🗋 ✗	🗋 ✗

SCHARP at FRED HUTCH

Speed Test

The table policy is a per row item, you will want to understand how this affects performance. In this test we will add 10 millions rows of data and then test with and without using a policy several times. We can that the row level security adds about 3-5 seconds for 10 million rows. Because the function is set as STABLE, this also means that it will only be evaluated once, thereby saving some time over having to be run for every row.

```
-- Continuing from the previous test.

SET ROLE table_owner;
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 10000000
)
INSERT INTO public.test
SELECT n, 'testing' FROM t;

SET ROLE user_a;
CREATE TABLE test2 AS SELECT * FROM public.test;
-- Query OK, 10000000 rows affected (execution time: 11.014 sec; total time: 11.014 sec)

SET ROLE table_owner;
ALTER TABLE public.test DISABLE ROW LEVEL SECURITY;
SET ROLE user_a;
CREATE TABLE test3 AS SELECT * FROM public.test;
-- Query OK, 10000000 rows affected (execution time: 8.159 sec; total time: 8.159 sec)

SET ROLE table_owner;
ALTER TABLE public.test ENABLE ROW LEVEL SECURITY;
SET ROLE user_a;
CREATE TABLE test4 AS SELECT * FROM public.test;
-- Query OK, 10000000 rows affected (execution time: 13.104 sec; total time: 13.104 sec)

SET ROLE table_owner;
ALTER TABLE public.test DISABLE ROW LEVEL SECURITY;
SET ROLE user_a;
CREATE TABLE test5 AS SELECT * FROM public.test;
-- Query OK, 10000000 rows affected (execution time: 8.315 sec; total time: 8.315 sec)
```

SCHARP at FRED HUTCH

TABLE with RULE and FUNCTION

SCHARP
at FRED HUTCH

Setup for Table with Rule and Function

For this method, no SELECT permissions are granted on the table. Instead, there's a view to the table and each role is granted separately. The role intersection check is implemented just before the LEFT JOIN. This causes the security_check function to always be checked even in the table is empty. The problem with this method is that it is a per row evaluation.

The con's:

- Table owner, does not have the rights to read, write, update, or delete the data.
- Once the RULE is applied, the TABLE turns into a VIEW.
- Must use DROP VIEW to get rid of the TABLE with RULE.

```
-- In an empty database run the "Setup for all examples"
-- first then run the code below

SET ROLE table_owner;
CREATE OR REPLACE FUNCTION public.security_check ( check_roles name [] )
RETURNS boolean AS $body$
DECLARE
  r RECORD;
BEGIN
  -- Checks to see if NULL was passed
  IF array_length(check_roles,1) IS NULL THEN
    RAISE EXCEPTION 'security_check(): Must specify roles to be checked.’;
  END IF;
  -- Checks to see if any value in the array is NULL
  IF check_roles @> ARRAY[NULL::name] THEN
    RAISE EXCEPTION 'security_check(): NULL roles are not allowed in check_roles: %', check_roles;
  END IF;
  -- Check to see if the current_user is a direct member of the check_roles
  SELECT ((SELECT array_agg(role_name::name) FROM
    information_schema.applicable_roles WHERE
    grantee = current_user) @> check_roles) AS security_check INTO r;
  IF r.security_check = TRUE THEN
    -- current_user was found to be a member of all check_roles
    RETURN TRUE;
  ELSE
    -- current_user was NOT found to be a member of all check_roles
    RAISE EXCEPTION 'security_check(): User: % is required to be a member of all of these groups: %',
current_user, check_roles;
  END IF;
END;
$body$
LANGUAGE 'plpgsql’
STABLE
CALLED ON NULL INPUT
SECURITY INVOKER;
```

Setup for Table with Rule and Function

For this method, no SELECT permissions are granted on the table. Instead, there's a view to the table and each role is granted separately. The role intersection check is implemented just before the LEFT JOIN. This causes the security_check function to always be checked even in the table is empty. The problem with this method is that it is a per row evaluation.

The con's:

- Table owner, does not have the rights to read, write, update, or delete the data.
- Once the RULE is applied, the TABLE turns into a VIEW.
- Must use DROP VIEW to get rid of the TABLE with RULE.

```
GRANT EXECUTE ON FUNCTION public.security_check(check_roles name []) TO PUBLIC;

DROP VIEW public.test_rule;
CREATE TABLE public.test_rule (LIKE public.test);
SELECT relname, relkind FROM pg_catalog.pg_class WHERE relname = 'test_rule';
/*
"relname" "relkind"
"test_rule" "r"
*/
CREATE RULE "_RETURN" AS ON SELECT TO public.test_rule DO INSTEAD
  SELECT test.*
  FROM public.security_check(ARRAY['group_a'::name, 'group_b'::name])
  LEFT JOIN public.test ON (TRUE)
  -- Where clause needed to prevent 1 row from being returned when test contains 0 rows
  WHERE test.id IS NOT NULL;

SELECT relname, relkind FROM pg_catalog.pg_class WHERE relname = 'test_rule';
/*
"relname" "relkind"
"test_rule" "v"
*/

GRANT SELECT ON public.test_rule TO group_a;
GRANT SELECT ON public.test_rule TO group_b;
```

SCHARP at FRED HUTCH

Check with Dataset

```
SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test_rule;
-- 3 rows returned (execution time: 0 ms; total time: 15 ms)

SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: user_b is required to be a member of all of these groups: {group_a,group_b}

SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test_rule;
-- ERROR: permission denied for relation test

SET ROLE group_a; -- 1st Group
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: group_a is required to be a member of all of these groups: {group_a,group_b}

SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: group_b is required to be a member of all of these groups: {group_a,group_b}

SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: table_owner is required to be a member of all of these groups:
{group_a,group_b}

SET ROLE postgres; -- Superuser
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: postgres is required to be a member of all of these groups: {group_a,group_b}
```

Check with Empty Data Set

```
-- Check to see what happens with an empty dataset
SET ROLE table_owner;
TRUNCATE TABLE public.test;

SET ROLE user_a; -- Privilaged User
SELECT * FROM public.test_rule;
-- Empty set (execution time: 0 ms; total time: 16 ms)

SET ROLE user_b; -- Partial Privilaged User
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: user_b is required to be a member of all of these groups: {group_a,group_b}

SET ROLE user_c; -- Un-Privilaged User
SELECT * FROM public.test_rule;
-- ERROR: permission denied for relation test

SET ROLE group_a; -- 1st Group
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: group_a is required to be a member of all of these groups: {group_a,group_b}

SET ROLE group_b; -- 2nd Group
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: group_b is required to be a member of all of these groups: {group_a,group_b}

SET ROLE table_owner; -- Table Owner
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: table_owner is required to be a member of all of these groups:
{group_a,group_b}

SET ROLE postgres; -- Superuser
SELECT * FROM public.test_rule;
-- ERROR: security_check(): User: postgres is required to be a member of all of these groups: {group_a,group_b}
```

SCHARP at FRED HUTCH

Results:

Our Criteria							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner
CRITERIA	▦	✓	✗	✗	🗋 🗐		
Feature Chart							
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data				
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner
TABLE with RULE and FUNCTION	🔍	✓	✗	✗	🗋 🗐	✗	✗

SCHARP at FRED HUTCH

Upgrading the Function

While we are only dealing with users that are directly granted access to the group, you may wish to deal with inherited roles. We can create a simple recursive function to lookup this information. Because the user is a member of these group, we can use the set role function to switch to that group to look up what groups that group belongs to, we just need to set the user back afterwards.

```
CREATE OR REPLACE FUNCTION public.applicable_roles ( lookup name )
RETURNS SETOF information_schema.applicable_roles AS
$body$
DECLARE
  original_user NAME;
  r RECORD;
BEGIN
  original_user = current_user;
  EXECUTE 'SET ROLE ' || quote_ident(lookup);
  FOR r IN SELECT * FROM information_schema.applicable_roles WHERE grantee = current_user LOOP
    RETURN NEXT r;
    RETURN QUERY SELECT * FROM public.applicable_roles(r.role_name);
  END LOOP;
  EXECUTE 'SET ROLE ' || quote_ident(original_user);
END;
$body$
LANGUAGE 'plpgsql’
VOLATILE
RETURNS NULL ON NULL INPUT
SECURITY INVOKER;
```

Upgrading the Function

We want to have the same permission denied errors that an unprivileged user would receive on a normal table. This can be achieved by passing in the relation, aka table or view name, into the function and then updating our error messages.

```
SET ROLE table_owner;
CREATE OR REPLACE FUNCTION public.security_check
( check_roles name [], relation name )
RETURNS boolean AS $body$
  DECLARE r RECORD;
BEGIN
  -- Checks to see if NULL was passed
  IF array_length(check_roles,1) IS NULL THEN
    RAISE EXCEPTION 'check_roles argument of security_check() is null' USING ERRCODE =
'null_value_not_allowed', HINT = 'security_check(): Must specify roles to be checked.’;
  END IF;
  IF relation IS NULL THEN
    RAISE EXCEPTION 'relation argument of security_check() is null' USING ERRCODE =
'null_value_not_allowed', HINT = 'security_check(): Must specify relation being checked.’;
  END IF;
  -- Check to see if the current_user is a direct member of the check_roles
  -- SELECT ((SELECT array_agg(role_name::name) FROM information_schema.applicable_roles WHERE
grantee = current_user)
  SELECT ((SELECT array_agg(role_name::name)
  FROM public.applicable_roles(current_user)) @> check_roles) AS security_check
  INTO r;
  IF r.security_check = TRUE THEN
    -- current_user was found to be a member of all check_roles
    RETURN TRUE;
  ELSE
    -- current_user was NOT found to be a member of all check_roles
    RAISE EXCEPTION 'permission denied for relation %', relation USING ERRCODE =
'insufficient_privilege', HINT = 'security_check(): User: ' || current_user || ' is required to be a
member of all of these groups: {' || array_to_string(check_roles, ',', 'NULL') || '}’;
  END IF;
END; $body$ LANGUAGE 'plpgsql' STABLE CALLED ON NULL INPUT SECURITY INVOKER;
```

Upgrading the Function

```
GRANT EXECUTE ON FUNCTION public.security_check(check_roles name [], relation name) TO PUBLIC;

SET ROLE user_b;
SELECT public.security_check(null, null);
-- ERROR: check_roles argument of security_check() is null
-- HINT: security_check(): Must specify roles to be checked.

SELECT public.security_check(ARRAY['group_a'::name, 'group_b'::name], null);
-- ERROR: relation argument of security_check() is null
-- HINT: security_check(): Must specify roles to be checked.

SELECT public.security_check(ARRAY[null::name], 'test_rule'::name);
-- ERROR: permission denied for relation test_rule
-- HINT: security_check(): User: user_b is required to be a member of all of these groups: {NULL}

SELECT public.security_check(ARRAY['group_a'::name, 'group_b'::name], 'test_rule'::name)
-- ERROR: permission denied for relation test_rule
-- HINT: security_check(): User: user_b is required to be a member of all of these groups:
{group_a,group_b}
```

SCHARP at FRED HUTCH

Speed Testing

SCHARP at FRED HUTCH

Speed Testing

Thanks to Jeremy Schneider, Amazon AWS, for doing speed testing using PostgreSQL build version 10.4 from yum.postgresql.org.

Re-reading your blog, your code for speed testing does the trick and is a little nicer than my code. :) i could have just used that, tweaking for wider rows. I prefer this, and it's easy enough to do 'testing'||repeat('X',58).

```
insert into public.test
select random()*9000000+100000, md5(random()::text)||repeat('X',58)
from generate_series(1,10000) g1
cross join generate_series(1,100) g2;

insert into public.test select * from public.test;
insert into public.test select * from public.test;

select count(*) from test;
count
---------
4,000,000
(1 row)

Time: 358.332 ms

select pg_relation_size('test');
pg_relation_size
------------------
504,127,488
(1 row)
```

Results & Speed Test:

Our Criteria								
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data					With Data
			Un-Privileged User	Partial Privileged User	Privileged User	Superuser	Table Owner	Speed Test
CRITERIA	▦	✓	❌	❌	🗋 🗐			

Feature Chart								
Style	User SELECT's on	Object owned by table_owner	Empty Table / With Data					With Data
			Un-Privileged user_c	Partial Privileged user_b	Privileged user_a	Superuser	Table Owner	Speed Test
TABLE	▦	✓	❌	🗋 🗐	🗋 🗐	🗋 🗐	🗋 🗐	990.815 ms
VIEW	🔍	✓	❌	🗋	🗋 🗐	🗋	🗋	1,081.979 ms
FUNCTION	f(x)	⛔	❌	❌	🗋 🗐	❌	❌	2,093.991 ms
TABLE with POLICY 9.5+	▦	✓	❌	🗋	🗋 🗐	🗋 🗐	🗋 🗐	2,023.267 ms
TABLE with POLICY (FORCE'd) 9.5+	▦	✓	❌	🗋	🗋 🗐	🗋 🗐	🗋	2,028.994 ms
TABLE with RULE	🔍	✓	❌	🗋	🗋 🗐	🗋	🗋	1,064.975 ms
VIEW with FUNCTION	🔍	✓	❌	❌	🗋 🗐	❌	❌	1,255.351 ms ~1.25 s
TABLE with POLICY and FUNCTION *	▦	✓	❌	🗋 ❌	🗋 🗐	🗋 ❌	🗋 ❌	129,706.047 ms ~2.16 m
TABLE with RULE and FUNCTION	🔍	✓	❌	❌	🗋 🗐	❌	❌	1,198.303 ms

Conclusion

Take Away

- There is nothing that meets our full criteria.
- We ended up going with the View with Function due to the tighter security over the Table with Policy and Function.
- When querying as a partially privileged user, aka user_b, the Table with Policy and Function, we can get different results depending on the PostgreSQL version.
- Setting the volatility of the function to stable in PostgreSQL 9.6 will throw a permission denied error where as in PostgreSQL 10 you will get an empty table returned. My writing and testing was done in Postgres 9.4.

THANK YOU

SCHARP
at FRED HUTCH