PostgreSQL Procedure Privileges and Security

This document explains the behavior of PostgreSQL procedures and their privilege management, including the difference between SECURITY INVOKER and SECURITY DEFINER modes, and how they influence procedure execution rights. This will clarify why a user might be able to execute a procedure without explicitly granted EXECUTE privileges and how to restrict execution rights if necessary.

# 1. Rule about Procedure Privileges

By default, when a procedure is created in PostgreSQL, the following behavior occurs:  
  
1. The creator (e.g., user1) of the procedure is considered the owner of the procedure.  
2. No one except the owner or superusers can execute the procedure unless EXECUTE privilege is explicitly granted to other users.

# 2. What Happened in Your Scenario

Here’s a detailed explanation of what happened step by step:  
  
Step 1: Creating the Procedure  
You created the procedure `proc\_1` under user1. Since user1 is the owner of the procedure, only user1 has the EXECUTE privilege on it.  
  
Step 2: Creating user2  
You created a new user, user2, and registered a new server connection using this user. By default, user2 has no privileges on the database objects (including tables or procedures) unless explicitly granted.  
  
Step 3: Granting Privileges on the Table  
You logged back in as user1 and granted SELECT and INSERT privileges on table\_1 to user2:  
 GRANT SELECT, INSERT ON table\_1 TO user2;  
Now, user2 can directly access table\_1 using SELECT or INSERT commands.  
  
Step 4: Calling the Procedure  
You called the procedure `proc\_1` as user2. Despite not explicitly granting EXECUTE privilege on the procedure to user2, it worked.

# 3. Why Did the Procedure Work Without Explicit EXECUTE Privilege?

This is because PostgreSQL procedures default to SECURITY INVOKER behavior unless explicitly defined otherwise. Here's what happens:  
  
SECURITY INVOKER Behavior (Default)  
Procedures run with the privileges of the user who calls them (in this case, user2).  
When user2 called the procedure, PostgreSQL checked:  
- Does user2 have the EXECUTE privilege on the procedure?  
 Answer: No, but the database did not block execution because of SECURITY INVOKER.  
- Does user2 have the necessary privileges on the objects (like table\_1) that the procedure interacts with?  
 Answer: Yes, because you granted SELECT and INSERT privileges on table\_1.  
  
Since the procedure was executed with user2's privileges (default SECURITY INVOKER behavior), and user2 had sufficient table privileges, PostgreSQL allowed the procedure call to succeed.

# 4. To Restrict Procedure Execution

If you want the procedure to be executable only by user1 (or explicitly granted users), you need to use the EXECUTE privilege and potentially the SECURITY DEFINER mode:

Solution 1: Explicitly Control EXECUTE Privilege

Revoke public execution rights:  
 REVOKE EXECUTE ON PROCEDURE proc\_1 FROM PUBLIC;

Grant EXECUTE privilege to specific users (e.g., user2):  
 GRANT EXECUTE ON PROCEDURE proc\_1 TO user2;

Solution 2: Use SECURITY DEFINER

Modify the procedure to run with the privileges of the procedure owner (user1):  
 CREATE OR REPLACE PROCEDURE proc\_1()  
 LANGUAGE plpgsql  
 SECURITY DEFINER  
 AS $$  
 BEGIN  
 -- Procedure logic here  
 END;  
 $$;

With SECURITY DEFINER:  
Only EXECUTE privilege on the procedure is checked.  
Privileges on underlying objects (e.g., table\_1) are based on the owner’s (user1) privileges.

# 5. Summary

By default, a procedure is created with SECURITY INVOKER and without granting EXECUTE to others. However:  
  
- If a user has sufficient privileges on the underlying objects accessed by the procedure, the procedure can execute successfully because SECURITY INVOKER uses the caller's privileges.  
- To prevent this behavior:  
 1. Explicitly manage EXECUTE privileges.  
 2. Use SECURITY DEFINER to restrict execution rights and enforce ownership-based privilege checks.