PostgreSQL Row Level Security: Easy Guide with Practical Examples

PostgreSQL's Row Level Security (RLS) is a great security feature that allows you to control which rows users can access at the database level. Instead of handling security logic in your application, you can define policies directly in the database that automatically filter data based on the current user's identity and roles. This approach provides a centralized security model that's hard to bypass and easy to audit.

What is Row Level Security?

Row Level Security enables you to define policies that restrict which rows a user can see, insert, update, or delete. These policies are enforced at the database level, making them transparent to applications while providing fine-grained access control.

Think of RLS as adding a WHERE clause to every query automatically based on who's running it. The beauty is that this happens transparently – applications don't need to know about the security policies.

Basic RLS Setup

Enabling Row Level Security

Let's start with a practical example using a multi-tenant application:

```
-- Create a sample table for a multi-tenant SaaS application

CREATE TABLE documents (
    id SERIAL PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    content TEXT,
    tenant_id INTEGER NOT NULL,
    author VARCHAR(100) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    is_public BOOLEAN DEFAULT FALSE
);

-- Insert sample data

INSERT INTO documents (title, content, tenant_id, author, is_public) VALUES
    ('Company Policy', 'Internal company policies...', 1, 'admin@company1.com', FALSE),
    ('Product Manual', 'How to use our product...', 1, 'docs@company1.com', TRUE),
    ('Financial Report', 'Q4 financial results...', 2, 'cfo@company2.com', FALSE),
    ('User Guide', 'Step by step guide...', 2, 'support@company2.com', TRUE),
    ('Secret Project', 'Top secret information...', 1, 'admin@company1.com', FALSE);

-- Enable Row Level Security on the table

ALTER TABLE documents ENABLE ROW LEVEL SECURITY;
```

At this point, RLS is enabled but no policies exist, so only table owners and superusers can access the data.

Creating Basic Policies

```
-- Policy: Users can only see documents from their tenant

CREATE POLICY tenant_policy ON documents
   FOR ALL
   TO public
   USING (tenant_id = current_setting('app.tenant_id')::INTEGER);
```

Explaination:-

- **CREATE POLICY tenant_policy ON documents**

Defines a new policy named tenant_policy on the table documents.

- F**OR ALL**

The policy applies to all commands: SELECT, INSERT, UPDATE, and DELETE.

(You could also use FOR SELECT, FOR INSERT, etc. if you wanted finer control.)

- **TO public**

The policy applies to all roles (users).

If you wanted to restrict it, you could specify a particular role.

- **USING (tenant_id = current_setting('app.tenant_id')::INTEGER)**

1. Defines the condition that must be true for rows to be visible or modifiable.

2. Only rows where tenant_id matches the value of the session variable app.tenant_id are accessible.

3. current_setting('app.tenant_id') fetches the tenant ID set in the current session (e.g., using SET app.tenant_id = 123;).

4. Casting with ::INTEGER ensures proper type matching.

```
-- Test the policy by setting a tenant context
SET app.tenant_id = '1';
SELECT * FROM documents; -- Only shows documents for tenant 1

SET app.tenant_id = '2';
SELECT * FROM documents; -- Only shows documents for tenant 2
```

Advanced Policy Examples

Role-Based Access Control

Let's create a more sophisticated example with different user roles:

```
-- Create roles
CREATE ROLE tenant_admin;
CREATE ROLE tenant_user;
CREATE ROLE tenant_readonly;

-- Create users and assign roles
CREATE USER admin1 WITH PASSWORD 'secure_password';
CREATE USER user1 WITH PASSWORD 'user_password';
CREATE USER readonly1 WITH PASSWORD 'readonly_password';

GRANT tenant_admin TO admin1;
GRANT tenant_user TO user1;
GRANT tenant_readonly TO readonly1;

-- Grant basic permissions
GRANT SELECT, INSERT, UPDATE, DELETE ON documents TO tenant_admin, tenant_user;
GRANT SELECT ON documents TO tenant_readonly;
GRANT USAGE ON SEQUENCE documents_id_seq TO tenant_admin, tenant_user;
```

Comprehensive Policy Set

```
-- Policy 1: Tenant isolation (applies to all operations)

CREATE POLICY tenant_isolation ON documents
    FOR ALL
    TO public
    USING (tenant_id = current_setting('app.tenant_id')::INTEGER);

-- Policy 2: Admins can see everything in their tenant

CREATE POLICY admin_all_access ON documents
    FOR ALL
    TO tenant_admin
    USING (tenant_id = current_setting('app.tenant_id')::INTEGER);

-- Policy 3: Users can see public documents and their own documents

CREATE POLICY user_access ON documents
    FOR SELECT
    TO tenant_user
    USING (
        tenant_id = current_setting('app.tenant_id')::INTEGER
        AND (is_public = TRUE OR author = current_user)
    );
```

```
-- Policy 4: Users can only modify their own documents

CREATE POLICY user_modify_own ON documents
    FOR UPDATE
    TO tenant_user
    USING (
        tenant_id = current_setting('app.tenant_id')::INTEGER
        AND author = current_user
    );

-- Policy 5: Users can only insert documents in their tenant with themselves as author

CREATE POLICY user_insert ON documents
    FOR INSERT
    TO tenant_user
    WITH CHECK (
        tenant_id = current_setting('app.tenant_id')::INTEGER
        AND author = current_user
    );

-- Policy 6: Readonly users can only see public documents

CREATE POLICY readonly_access ON documents
    FOR SELECT
    TO tenant_readonly
    USING (
        tenant_id = current_setting('app.tenant_id')::INTEGER
        AND is_public = TRUE
    );
```

Testing the Policies

Let's test our policies with different users:

```
-- Test as admin user
SET ROLE admin1;
SET app.tenant_id = '1';
SELECT title, author, is_public FROM documents;
-- Should see all documents for tenant 1

-- Test as regular user
SET ROLE user1;
SET app.tenant_id = '1';
SELECT title, author, is_public FROM documents;
-- Should only see public documents and documents authored by user1

-- Try to insert a document
INSERT INTO documents (title, content, tenant_id, author)
VALUES ('My Document', 'Content here', 1, 'user1');
-- Should succeed

-- Try to insert with different author
INSERT INTO documents (title, content, tenant_id, author)
```

```
VALUES ('Fake Document', 'Content here', 1, 'someone_else');

-- Should fail due to WITH CHECK constraint

-- Test as readonly user
SET ROLE readonly1;
SET app.tenant_id = '1';
SELECT title, author, is_public FROM documents WHERE is_public = TRUE;
-- Should only see public documents

-- Try to insert (should fail)
INSERT INTO documents (title, content, tenant_id, author)
VALUES ('Test', 'Content', 1, 'readonly1');

-- ERROR: permission denied
```

Dynamic Policies with Functions

You can create more sophisticated policies using functions:

```
-- Create a function to check if user has access to specific documents

CREATE OR REPLACE FUNCTION user_has_document_access(doc_tenant_id INTEGER, doc_author
VARCHAR)
RETURNS BOOLEAN AS $$
BEGIN
  -- Superusers and table owners always have access
  IF pg_has_role(current_user, 'pg_database_owner', 'MEMBER') THEN
    RETURN TRUE;
  END IF;

  -- Check if user is in the same tenant
  IF doc_tenant_id != current_setting('app.tenant_id')::INTEGER THEN
    RETURN FALSE;
  END IF;

  -- Admins can access everything in their tenant
  IF pg_has_role(current_user, 'tenant_admin', 'MEMBER') THEN
    RETURN TRUE;
  END IF;

  -- Users can access their own documents
  IF doc_author = current_user THEN
    RETURN TRUE;
  END IF;

  -- Department-based access (assuming user has department setting)
  IF current_setting('app.user_department', TRUE) IS NOT NULL THEN
    IF current_setting('app.user_department') = 'management' THEN
      RETURN TRUE;
    END IF;
  END IF;
```

```
    RETURN FALSE;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

-- Create policy using the function

CREATE POLICY dynamic_access ON documents
    FOR ALL
    TO public
    USING (user_has_document_access(tenant_id, author));
```

Time-Based Access Control

```
-- Add columns for time-based access
ALTER TABLE documents
ADD COLUMN valid_from TIMESTAMP DEFAULT NOW(),
ADD COLUMN valid_until TIMESTAMP DEFAULT NULL;

-- Policy for time-based access
CREATE POLICY time_based_access ON documents
    FOR SELECT
    TO public
    USING (
        NOW() >= valid_from
        AND (valid_until IS NULL OR NOW() <= valid_until)
        AND tenant_id = current_setting('app.tenant_id')::INTEGER
    );

-- Test with time-restricted document
INSERT INTO documents (title, content, tenant_id, author, valid_from, valid_until)
VALUES (
    'Limited Time Offer',
    'Special promotion details...',
    1,
    'marketing@company1.com',
    NOW(),
    NOW() + INTERVAL '7 days'
);
```

Application Integration

Setting Context in Applications

Here's how you'd integrate RLS with different application frameworks:

Python (using psycopg2):

```
import psycopg2

def set_user_context(conn, tenant_id, user_role=None, department=None):
    """Set the application context for RLS policies"""
```

```
    with conn.cursor() as cur:
        cur.execute("SET app.tenant_id = %s", (tenant_id,))
        if user_role:
            cur.execute("SET ROLE %s", (user_role,))
        if department:
            cur.execute("SET app.user_department = %s", (department,))

# Usage example
conn = psycopg2.connect("postgresql://user:pass@localhost/mydb")

# Set context for user from tenant 1
set_user_context(conn, tenant_id=1, user_role='tenant_user')

# Now all queries will be filtered by RLS policies
with conn.cursor() as cur:
    cur.execute("SELECT * FROM documents")
    results = cur.fetchall()  # Only documents user can access
```

Node.js (using pg library):

```
const { Pool } = require('pg');

class SecureDatabase {
    constructor(config) {
        this.pool = new Pool(config);
    }

    async setUserContext(client, tenantId, userRole = null) {
        await client.query('SET app.tenant_id = $1', [tenantId]);
        if (userRole) {
            await client.query('SET ROLE $1', [userRole]);
        }
    }

    async executeWithContext(tenantId, userRole, queryText, params = []) {
        const client = await this.pool.connect();
        try {
            await this.setUserContext(client, tenantId, userRole);
            const result = await client.query(queryText, params);
            return result.rows;
        } finally {
            client.release();
        }
    }
}

// Usage
const db = new SecureDatabase({
    user: 'app_user',
    host: 'localhost',
    database: 'myapp',
    password: 'password',
    port: 5432,
});

// Query with automatic RLS filtering
```

```
const documents = await db.executeWithContext(
  1, // tenant_id
  'tenant_user', // role
  'SELECT * FROM documents WHERE title ILIKE $1',
  ['%policy%']
);
```

Monitoring and Debugging RLS

Viewing Current Policies

```
-- See all policies on a table
SELECT schemaname, tablename, policyname, roles, cmd, qual, with_check
FROM pg_policies
WHERE tablename = 'documents';

-- Check if RLS is enabled
SELECT schemaname, tablename, rowsecurity, forcerowsecurity
FROM pg_tables
WHERE tablename = 'documents';
```

Debugging Policy Issues

```
-- Create a debug function to show what RLS sees
CREATE OR REPLACE FUNCTION debug_rls_context()
RETURNS TABLE(
  current_user_name TEXT,
  current_role TEXT,
  tenant_id TEXT,
  user_department TEXT
) AS $$
BEGIN
  RETURN QUERY SELECT
    current_user::TEXT,
    current_setting('role')::TEXT,
    current_setting('app.tenant_id', TRUE)::TEXT,
    current_setting('app.user_department', TRUE)::TEXT;
END;
$$ LANGUAGE plpgsql;

-- Use it to debug
SELECT * FROM debug_rls_context();
```

Performance Monitoring

```
-
- Check query plans to see RLS impact
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM documents WHERE title ILIKE '%report%';

-- Monitor RLS-related slow queries
SELECT query, mean_exec_time, calls
FROM pg_stat_statements
WHERE query LIKE '%documents%'
ORDER BY mean_exec_time DESC;
```

Best Practices and Considerations

1. Policy Performance

```
-- BAD: Expensive function call in policy
CREATE POLICY slow_policy ON documents
   FOR ALL USING (expensive_function(tenant_id));

-- GOOD: Use indexed columns and simple expressions
CREATE POLICY fast_policy ON documents
   FOR ALL USING (tenant_id = current_setting('app.tenant_id')::INTEGER);

-- Ensure proper indexing
CREATE INDEX idx_documents_tenant_author ON documents (tenant_id, author);
```

2. Policy Ordering and Combination

```
-- Policies are combined with OR logic
-- If ANY policy allows access, the row is visible

-- This means you need to be careful about overly permissive policies
CREATE POLICY restrictive_policy ON documents
   FOR SELECT USING (author = current_user);

CREATE POLICY permissive_policy ON documents
   FOR SELECT USING (is_public = TRUE);

-- Result: User sees their own docs OR public docs (combined with OR)
```

3. Bypass RLS When Necessary

```
-- Sometimes you need to bypass RLS (admin operations, reporting, etc.)
SET row_security = OFF;
SELECT * FROM documents; -- Sees all data
SET row_security = ON;

-- Or create a policy that allows bypass for specific roles
CREATE POLICY admin_bypass ON documents
   FOR ALL
   TO admin_role
   USING (TRUE); -- Always true = no restrictions
```

4. Testing and Validation

```
-- Create a comprehensive test suite
CREATE OR REPLACE FUNCTION test_rls_policies()
RETURNS TABLE(test_name TEXT, passed BOOLEAN, message TEXT) AS $$
DECLARE
   doc_count INTEGER;
BEGIN
   -- Test 1: Tenant isolation
   SET app.tenant_id = '1';
   SET ROLE tenant_user;
   SELECT COUNT(*) INTO doc_count FROM documents;

   RETURN QUERY SELECT
```

```
    'tenant_isolation'::TEXT,
    doc_count > 0 AND doc_count < (SELECT COUNT(*) FROM documents WHERE tenant_id = 1),
    format('User sees %s documents for tenant 1', doc_count)::TEXT;

  -- Add more tests...
  RESET ROLE;
END;
$$ LANGUAGE plpgsql;

-- Run tests
SELECT * FROM test_rls_policies();
```

Common Pitfalls and Solutions

1. Forgetting to Set Context

```
-- Problem: Queries fail because context isn't set
SELECT * FROM documents; -- ERROR or no results

-- Solution: Always set context in application connection setup
-- Use connection pooling that maintains context per session
```

2. Policy Conflicts

```
-- Problem: Conflicting policies can cause unexpected behavior
-- Solution: Use descriptive names and document policy interactions

-- Drop conflicting policies
DROP POLICY IF EXISTS old_policy ON documents;

-- Create new comprehensive policy
CREATE POLICY comprehensive_access ON documents
  FOR ALL
  USING (
    tenant_id = current_setting('app.tenant_id')::INTEGER
    AND (
      pg_has_role('tenant_admin', 'MEMBER')
      OR author = current_user
      OR (is_public = TRUE AND pg_has_role('tenant_user', 'MEMBER'))
    )
  );
```

3. Performance Issues

```
-- Problem: RLS policies causing slow queries
-- Solution: Optimize policies and add proper indexes

-- Check policy performance
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM documents;

-- Add indexes to support policy conditions
CREATE INDEX CONCURRENTLY idx_documents_rls
ON documents (tenant_id, author, is_public);
```

Advanced Use Cases

Multi-Level Security

```
-- Create a classification-based security system
ALTER TABLE documents ADD COLUMN security_level INTEGER DEFAULT 1;

-- Users have clearance levels
CREATE TABLE user_clearance (
   username VARCHAR(100) PRIMARY KEY,
   clearance_level INTEGER NOT NULL DEFAULT 1
);

-- Policy based on security clearance
CREATE POLICY security_clearance ON documents
   FOR SELECT
   USING (
      tenant_id = current_setting('app.tenant_id')::INTEGER
      AND security_level <= (
         SELECT clearance_level
         FROM user_clearance
         WHERE username = current_user
      )
   );
```

Audit Trail Integration

```
-- Create audit table
CREATE TABLE document_access_log (
   id SERIAL PRIMARY KEY,
   document_id INTEGER,
   accessed_by VARCHAR(100),
   access_time TIMESTAMP DEFAULT NOW(),
   access_type VARCHAR(20)
);

-- Function to log access
CREATE OR REPLACE FUNCTION log_document_access()
RETURNS TRIGGER AS $$
BEGIN
   INSERT INTO document_access_log (document_id, accessed_by, access_type)
   VALUES (NEW.id, current_user, TG_OP);
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger for access logging
CREATE TRIGGER document_access_trigger
   AFTER SELECT OR UPDATE ON documents
   FOR EACH ROW
   EXECUTE FUNCTION log_document_access();
```

PostgreSQL's Row Level Security provides a robust, database-level solution for fine-grained access control. When implemented correctly, RLS offers several key advantages:

- Centralized Security: All access control logic lives in the database
- Application Transparency: Applications don't need to handle security logic
- Audit-Friendly: Easy to audit and verify security policies
- Performance: Database-level filtering is often faster than application-level

RLS is just one part of a security strategy. Combine it with proper authentication, network security, and application-level safeguards for a complete security posture.