Application Authorization with SET ROLE

Aurynn Shaw, Command Prompt, Inc.
PGCon 2010

Hi

Hi

- Aurynn Shaw
  - DBA/Lead Dev/PM/etc @ Command Prompt

* Today we're talking about AuthZ in PG
* Benefits, drawbacks, and a quick implementation walkthrough
* Why you should even be doing this

And now I talk more

Permissions Systems

Permissions Systems

- Data I can access

Permissions Systems

* Postgres handles this through standard GRANT and REVOKE statements.

* Most app fabrics handle this away from the data.

- Data I can access
- Data I can't access

App-focused Design

* Easy enough to use the app to handle permissions
* Few-no restrictions on application powers.
* Permissions happen when the logic happens

Data Layer Disconnect

* App-focus development treats the DB as a dumb store
* Because the app embodies the AuthZ, the data fabric is at its most malleable.
* Nothing stops a malicious, or badly-written app from unlimited data modification
* Relying on a limited number of eyeballs to look for bugs

DB-focused Design

* Much tighter binding to the data layer
* We can put permissions into the database, GRANT and REVOKE! Not a problem!
* Creating a user, not a problem. Everything can just work!

AuthZ is closely coupled to your login

* Poolers, especially, have only a defined login
* Forced into the broadest permissions set available
* Can't attempt to restrict the data malleability - everything your app needs to do, your login has to be able to do, regardless of whether the user should.

Wait, I lied.

* Your login DOES embody your core permissions, but, there's this great permissions-swapping feature in PG.

SET ROLE TO STUN;

HEY, THIS IS IN THE TALK TITLE!

* I've seen this before! It's like a Unix system!
* So, SET ROLE is the funky mojo
* Similar to SET AUTHORIZATION
* Can be unwound - a very valuable aspect.

Can only switch to roles already in your tree

only allows you to become roles you would have been able to be already - you can't just become a superuser, unless you already are one.
* By default, all the roles you have are already part of your user

Why SET ROLE is interesting

* Can swap permissions dynamically, without compromising the base connection
* Vital in any pooled environment - long-lived connections don't need to be reset.
* Trusted apps can easily set the data fabric to just the permissions they need
* Can never exceed base fabric permissions

Transactional, too!

* Single transactions can be in their own permissions space
* Automatic, implicit RESET ROLE command on ROLLBACK

Transactional, too!

```
template1=# BEGIN;
BEGIN
template1=# SET ROLE test;
SET
template1=> ROLLBACK;
ROLLBACK
template1=#
```

A quick example.

Transactional, too!

```
test=> BEGIN;
BEGIN
test=> SET ROLE test;
SET
test=> SELECT * FROM test;
--
(0 rows)

test=> ROLLBACK;
ROLLBACK
test=> SELECT * FROM test;
ERROR:  permission denied for relation test
```

And another

Well, partly.

```
template1=# BEGIN;
BEGIN
template1=# SET ROLE test;
SET
template1=> COMMIT;
COMMIT
template1=>
```

So it doesn't quite work like you'd expect for a committed transaction.

So always RESET ROLE

```
template1=# BEGIN;
BEGIN
template1=# SET ROLE test;
SET
template1=> COMMIT;
COMMIT
template1=> RESET ROLE;
RESET
template1=#
```

So it doesn't quite work like you'd expect for a committed transaction.

Our Why

* Explored this to support a large Web application with very clear-cut access rules: A resource either is or isn't accessible.

* In-app frameworks were insufficient - and not useful when we needed external software- Rewriting perms is a pain.

Other Cool Whys

* Single definition of our permissions model, as close to the relevant data as possible.

* Don't Repeat Yourself

* Non-trusted clients can't manipulate your data fabric beyond your whim - you already have strong permissions on the data itself.

But there's all those other permissions systems...

* Lots, in a variety of languages
* Including that one you're working on right now
* And that other one YOU LOVE.
* Should you use them? They work, to a point
* Valuable aspect of the permissions setup
* Exclusive use ends up looking like THIS

This

- Data I can access
- Data I can't access
- **Data I shouldn't access, but can**

* Normal pooled application, single credentials relies on app to handle auth
* Never more than a strong warning about not using a resource, and some unfriendly language from your DBA.

Principle of Least Permission

Stolen from Steven Frost

* You should never have more ability than you need.
* Any time you do, Bad Things can happen.
* In-app permissions systems tend to violate this

Implementation

(it's easy)

So, let's look at how to go about implementing a SET ROLE-based system in your application framework.
It's surprisingly easy to do, too!
Let's begin.

GRANT AND REVOKE

First, a fairly core component is that you have to go through and GRANT, and REVOKE the various tables and views and suchly that make up your database.

REVOKE

```
test=# CREATE TABLE test ();
CREATE TABLE
test=# REVOKE ALL ON test FROM PUBLIC;
REVOKE
test=# SET ROLE TO test;
SET
test=> SELECT * FROM test;
ERROR:  permission denied f
test
test=>
```

A simple REVOKE example.

GRANT

```
test=> SET ROLE TO aurynn;
SET
test=# GRANT ALL ON test TO test;
GRANT
test=# SET ROLE TO test;
SET
test=> SELECT * FROM test;
--
(0 rows)
test=>
```

And a GRANT

A Permissions Tree

Next, a permissions tree.
This aspect of a SET ROLE design is really, really, really dependent on your application structure.
To really get the most benefit from a SET ROLE environment, you should spend some time laying out every single last permission that you want to have - as fine-grained as you can. This ends up being very valuable later, when you need to add less trustworthy clients.

A Permissions Tree

```
CREATE ROLE content_read NOLOGIN;
CREATE ROLE content_write NOLOGIN;
CREATE ROLE content_delete NOLOGIN;
```

A Permissions Tree

```
CREATE ROLE user_base NOLOGIN;
GRANT content_read TO user_base;
GRANT content_write TO user_base;
CREATE ROLE admin_base NOLOGIN;
GRANT content_delete TO admin_base;
GRANT user_base TO admin_base;
```

Your final node points

USER, ADMIN, MODERATOR, ETC.

Your final node points are the specific roles that a given user is going to be granted into - users, moderators, administrators, whatever. Your software would then issue SET ROLE TO your_user_role at the beginning of your transaction.

Caveat: Custom permissions are hard.

Permissions Endpoints

```
CREATE USER user NOINHERIT;
GRANT user_base TO user;
CREATE USER admin NOINHERIT;
GRANT admin_base TO admin;
```

NOINHERIT

The next piece is NOINHERIT. Right now, without this, you'd not exactly be restricting your permissions set - just granting the full set of useful permissions to a more limited, non-superuser user.
Pretty much exactly the same as before.

With NOINHERIT, we mark that those endpoint roles that we just defined aren't applied to our login role - we have to explicitly SET ROLE to grab those permissions.

A fully REVOKE'd, login user

* The credentials that the application/pooler/whatever uses to connect.
* This has pretty much every single possible permission, removed. All this role can do is SET ROLE to a different role, and pick up those permissions.
* By default, no connections can actually do anything useful.

Application Modifications

Lastly, modify your application. It's somewhat obvious, but it has to be said.

It's just that easy!

You've now successfully integrated a SET ROLE-based permissions system into your application.
It's just that easy.

I lied again. It's not quite that easy

Well, it's almost that easy. There are some bits that you do have to pay attention to, that you wouldn't otherwise

It's not quite that easy

At least in Python's psycopg2, permissions errors aren't mapped to something useful - you have to handle it yourself.

- You have to catch permissions errors

Plug, the shameless kind

- Exceptable, an exception-trapping library for Python

Plug, the shameless kind

- Exceptable, an exception-trapping library for Python
- Turns PG exceptions into smarter Python exceptions.

Plug, the shameless kind

- Exceptable, an exception-trapping library for Python
- Turns PG exceptions into smarter Python exceptions.
- We could use help with this - other language support &c.

It's not quite that easy

- You have to catch permissions er
- It's really coarsely grained

For one, this is fairly coarse-grained - you can restrict tables, but not individual
rows in those tables. For that, there's nothing to be done but write a stored procedure,
or a view that checks whether or not the user *can* read those roles.

The same applies for writes, obviously - but, that's a bit easier to solve with triggers to
verify per-row permissions, as opposed to the per-table permissions.

It's not quite that easy

* You need to add the SET ROLE mojo before you start running queries

- You have to catch permissions er
- It's really coarsely grained
- Requires modifications to the DB interface

It's not quite that easy

You actually have to send the SET ROLE and possibly RESET ROLE commands.

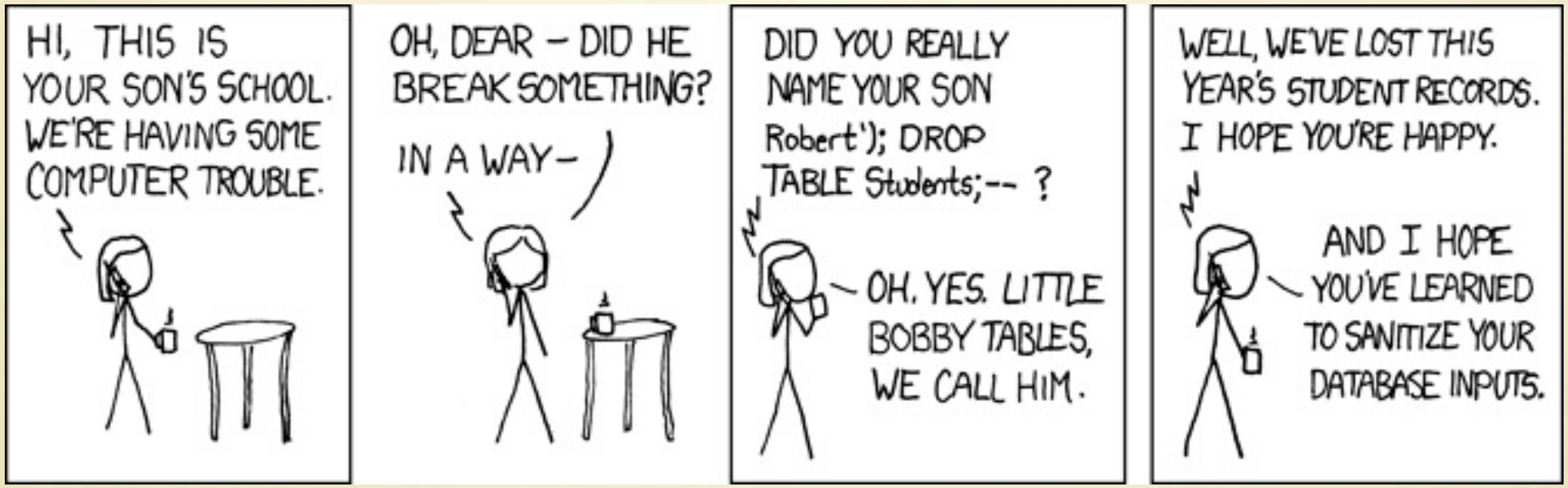
- You have to catch permissions er
- It's really coarsely grained
- Requires modifications to the DB interface
- Adds additional wire traffic

It's not quite that easy

- You have to catch permissions er
- It's really coarsely grained
- Requires modifications to the D
- Adds additional wire traffic
- Just as vulnerable to SQL injection as you were before

This doesn't give you any real additional protection against SQL injection attacks - it's pretty much security-by-obscurity at best, by requiring SET ROLE before your injection.
It does, however, grant you protection against random DELETE and DROP crap, which is good for something.

So always sanitize your inputs.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

It's just good data hygiene. Like brushing your teeth.

It's not quite that easy

- You have to catch permissions er
- It's really coarsely grained
- Requires modifications to the D
- Adds additional wire traffic
- Just as vulnerable to SQL injection as you were before
- Not entirely transactional

As I showed you before, it's not really transactional - you have to pay pretty close attention to your RESET ROLE statements.

set session_authorization

* The difference between SET ROLE and SET session_auth is a matter of semantics, mostly - both achieve the same effect.
* set session_authorization changes what roles are available to SET to, though

set session_authorization

```
test=# SET SESSION_AUTHORIZATION TO pgcon;
SET
test=> SET ROLE TO aurynn;
ERROR:  permission denied to set role
"aurynn"
test=> SET SESSION_AUTHORIZATION TO
aurynn;
SET
test=#
```

* Remarkably similar
* alters what roles are reachable from future SET ROLE requests.
* Useful from perspective of additonal layers of restriction over the connection

So that's it.
Any questions?

Thank you!

Slides will be available.