

Packaging Permissions in Stored Procedures

Erland Sommarskog

SQL Server MVP



Erland Sommarskog

Independent consultant based in Stockholm

SQL Server MVP since 2001

<http://www.sommarskog.se>

esquel@sommarskog.se

Slides and all scripts are available on

<http://www.sommarskog.se/present>

What This is All About

We want a user to be able perform a specific action that requires permission X.

But we don't want to grant the user the permission directly. (Because that would permit the user to do a lot more.)

What if we could package the permission inside a stored procedure with full control of what the user can do?

“Hey, isn't that the way it always works?”

No, only in a special but common case.

Today, we will learn how to do it in all the other cases.

Agenda

- Ownership Chaining.
- Certificate Signing – on database and server level.
- The EXECUTE AS Clause – on db and server level.
- The Dangers of TRUSTWORTHY.
- Cross-database Access.

Ownership Chaining

When a stored procedure (or a view, function or trigger) accesses an object, with *the same owner*, permissions are not checked for:

- DML (SELECT, INSERT, UPDATE, DELETE & MERGE).
- Execution of stored procedures and functions.

Does not apply to:

- Access through dynamic SQL.
- Access to metadata about the object.
- Special permissions such as ALTER.

Certificate Signing

What you use when ownership chaining does not apply.

The recipe:

1. Create a certificate.
2. Sign the procedure with the certificate.
3. Create a user from the certificate.
4. Grant the certificate user the permissions needed (which could be role membership).

Certificates and Signatures

A certificate is an asymmetric key that consists of:

- A private key that you keep secret and protect.
- A public key that you can share with anyone.
- Some metadata, including a signature. For our purposes, self-signed certificates are sufficient.

You can sign a document (email etc) with your private key.

The receiver can use the public key to verify the signature.

Proves that document is from you and has not been altered.

Signatures and Stored Procedures

When a signed procedure is invoked, SQL Server checks if the signature is valid.

If so, SQL Server sees if there is a user tied to the certificate.

In that case, SQL Server adds the user to sys.user_token, and its permissions will apply inside the procedure.

The user is a special type of user – it cannot log in nor be impersonated.

You can only create one user per certificate.

Observations on Certificate Signing

Procedure must be signed after each change.

The token and thus the permissions of the certificate user are carried on to dynamic SQL and system procedures.

But they are **not** carried on to nested procedures, triggers or functions, that is user-written code you can sign yourself.

Keep in mind: DENY always trumps GRANT.

Automate it! – GrantPermsToSP

GrantPermsToSP, an SP to automate the recipe for signing.

- Parameters: procedure name and a TVP with permissions.
- Creates a certificate with name formed from the SP name.
 - Password is a random GUID.
 - Subject is the permissions to be granted.
- Signs the procedure with the certificate.
- Creates a user from the certificate, named from the SP.
- Grants the cert user the permissions in the TVP.
- On re-run, drops existing signature, user and certificate.

GrantPermsToSP, Example

```
DECLARE @perms Permission_list  
INSERT @perms (perm) VALUES ('SELECT ON PermTable')  
EXEC GrantPermsToSP N'TestSP', @perms, @debug = 1
```

Use @debug parameter to see the generated commands.

In many cases you will put a call like this in your deployment scripts.

Important principle: identify the minimum permission you need to package with the procedure.

Server-Level Permissions

A scenario:

- A multi-application instance.
- For each database there are power users with db_owner permissions, but no server permissions.
- They need to see which users that are connected to their database.
- This requires VIEW SERVER STATE – but with that permission they would see too much.
- Certificates to the rescue!

Server-Level Recipe

1. Create a certificate in the master database.
2. Sign the procedure with certificate (if in master).
3. Create a login from that certificate.
4. Grant the login the required permissions.

While called “login”, this login cannot log in – it exists only to connect permissions and certificate.

Tokens can be inspected in `sys.login_token`.

Server-Level Permission in User DB

1. Create a certificate in the user database.
2. Sign the procedure.
3. Drop the private key.

This step ensures that local power users cannot use the certificate, even if they would know the password.

4. Copy the certificate (i.e. the public key) to master.
5. Create a login from the certificate.
6. Grant the login the required permissions.

Script to Automate This

Script so that you easily can run it on different servers.

- You need to specify: database, procedure and permissions.
- The script creates a certificate in the user database.
- Signs the procedure and drops the private key.
- Copies the certificate to master.
- Creates a login from the certificate and grants permissions.
- If DB is in an Availability Group: Copies the certificate, the login and permissions to the other nodes in the AG over a temporary linked server.

The Beauty of it All

The server-level DBA reviews the code before signing it to add the extra permissions.

If the power user changes the code, signature and permissions disappear.

Thus, DBA must sign again – and can review the changes.

That is, power users cannot use this to elevate their permissions behind the back of the DBA.

What you see is what you sign: if the code calls other modules, they will not run with elevated permissions.

EXECUTE AS and DB Permissions

Proper version:

1. Create a proxy user WITHOUT LOGIN with the name derived from the procedure.
2. Grant the proxy user the required permissions.
3. Add the clause **WITH EXECUTE AS 'SPName\$Proxy'**.

User creation and granting are in the same file as the procedure.

Lazy version:

1. Use **WITH EXECUTE AS OWNER** and no proxy user.

EXECUTE AS, cont'd

A lot simpler than certificates. ...but!

- Lazy version breaks the principle of granting minimum permissions.
- Breaks schemes for row-level security and auditing based on SYSTEM_USER, USER etc.
- This can be mitigated by using original_login() or session_context (context_info) – requires planning ahead.
- If your system is not ready for EXECUTE AS – you can stop it with a DDL trigger.

Server-Level Permissions and EXEC AS

Create a proxy login, grant permissions, add EXECUTE AS clause?

[10_execasserver.sql](#)

EXECUTE AS in an SP is the same as EXECUTE AS USER.

When we impersonate a **database user**, we are sandboxed in the current database cannot access things outside it, unless two doors are opened:

1. The database is set TRUSTWORTHY.
2. The database owner has been granted the permission AUTHENTICATE SERVER. (True if owner = sa.)

[11_trustworthy1.sql](#)

TRUSTWORTHY is a Security Risk

With certificates, the DBA can review the code every time a power user wants to change an SP with server-level access.

EXECUTE AS + TRUSTWORTHY gives the power user carte blanche to change the SP to his/her own liking.

But that is not all. Danger alert!

[12_trustworthy2.sql](#)

Combined with AUTHENTICATE SERVER, a person with db_owner rights (or rights to create and impersonate users) can elevate to **sysadmin**.

TRUSTWORTHY, cont'd

Recommendation: the owner of each database should be a unique SQL login which exists solely to own that database.

Hopefully, you think twice before you grant that login `AUTHENTICATE SERVER` to open the second door.

It is OK to open both doors, **IF** everyone with permission to create users in the database already are sysadmin.

...and this will never change. (Think: consultants.)

Cross-Database Access

Say that in database A, you have a procedure P that accesses table T in database B.

By default there is no ownership chaining across databases.

Thus, if you do nothing at all, there will be a permission error.

What is the best solution depends on the situation.

Cross-Database Access, options

1. Just add users to B and grant them permissions on T.
Sometimes feasible, but far from always.
2. Enable Cross-DB ownership chaining for both databases.
Users must still be added to B.
Alas, has the same security problem as TRUSTWORTHY.
3. Certificate signing.
Perfect for occasional calls between different applications.
Create cert in B, create user and grant perm on T, copy to A and sign P.
You do **not** need to add users to B.

Cross-Database Access, options

4. EXECUTE AS + TRUSTWORTHY

For multi-database applications.

A better alternative than enabling cross-db chaining, *if done right*.

Databases must be owned by the same SQL login with no server-level permissions.

The same set of people must be db_owner in all databases.

If not: go for certificates.

My article discusses cross-database access in a lot more detail.

Recap: Ownership Chaining

What you use 95% of the time, for plain and simple DML in stored procedures.

- Does not work with dynamic SQL.
- Does not work with “advanced” permissions.
- Does not work with metadata.
- Does not work with server-level permissions.
- Disabled by default for cross-db access.

Recap: Certificate Signing

Permits you to package about any database or server permission in stored procedures in a fine-grained way. (But cannot overcome an explicit DENY.)

Easy to manage with GrantPermsToSP and the script for server-level permissions.

The preferred method for database permissions.

Always use certificates for server-level permissions!

Recap: EXECUTE AS

Good for the lazy and casual. :-)

Implications for auditing and row-level security that requires you to plan ahead.

Can be OK for database permissions if conditions are relaxed (or you need to overcome DENY).

Always use certificate signing for server-level permissions!

It's Getting Very Near the End...

Erland Sommarskog – esquel@sommarskog.se

Scripts and slides on <http://www.sommarskog.se/present>.

Packaging Permissions in Stored Procedures on the web:

<http://www.sommarskog.se/grantperm.html>

<http://www.sommarskog.se/grantperm-appendix.html>

...and beware of TRUSTWORTHY!

Clean-up script

[13_cleanupall.sql](#)