

# DON'T LET YOUR PERMISSIONS BE HIJACKED!

Erland Sommarskog

Data Platform MVP



# ERLAND SOMMARSKOG

Independent consultant based in Stockholm

SQL Server MVP since 2001

[esquel@sommarskog.se](mailto:esquel@sommarskog.se)

<http://www.sommarskog.se>

Slides and scripts are available on  
<http://www.sommarskog.se/present>



# WHAT THIS IS ABOUT

## Permission Hijacking:

A person with less permissions than you have makes you unknowingly run code that (ab)uses your permissions to perform actions that are advantageous to the attacker.

# PREAMBLE

## ISN'T THIS TOO FANTASTIC?

- When it comes to security, nothing is “too unlikely”.
- If an attack can be performed, it will be attempted sooner or later, some place.
  - You don't want that place be your workplace.
- It all depends on the stakes.
  - Perform advantageous updates.
  - Steal data.
  - Install ransomware.
  - Just cause a mess (e.g., a disgruntled employee).

# PREAMBLE

## NOT ALL SITES ARE EQUAL

- In some places: only sysadmin and plain users and nothing in between.
  - Not very susceptible to the attacks I will discuss. (But see below.)
- Elsewhere: sysadmin, db\_owner (application admin), db\_ddladmin or similar (devs).
  - This is where you need to watch out!
- Don't overlook application logins.
  - They could have elevated permissions and SQL-injection holes.
- What about developers whose code gets deployed on the server?



# THE EVIL DDL TRIGGER

[01\\_DDLTriggers.sql](#)

- When running index and statistics maintenance, an application admin could have installed a DDL trigger that performs malicious actions abusing sysadmin permissions.
  - Adding a user to sysadmin is just one example.
  - It could be data theft, data manipulation etc.
- Line of defence: PoLP, Principle of Least Privilege.
  - [Wikipedia](#). [Blog post from Andreas Wolter](#).
- In this case: `EXECUTE AS USER = 'dbo'`.
  - This way you sandbox yourself into the current database and renounce all your permissions on server level.

# A SMALL HICCUP

[02\\_dbo-hiccup.sql](#)

- `EXECUTE AS USER = 'dbo'` may produce error 15517:  
*Cannot execute as the database principal because the principal "dbo" does not exist, this type of principal cannot be impersonated, or you do not have permission.*
- Happens easily with databases restored from other servers.
  - Due to SID mismatch between sys.databases and the database itself.
- Routine: set database owner when you restore a database.
- My opinion: a database should be owned by an SQL login that exists solely to own that database.

# RUNNING DATABASE MAINTENANCE

## 1. Ola Hallengren's Maintenance Solution.

- IndexOptimize accepts the parameter `@ExecuteAsUser`, which you set to `'dbo'` (or another user of your choice).

## 2. The maintenance plans in SSMS – no provision for EXECUTE AS USER.

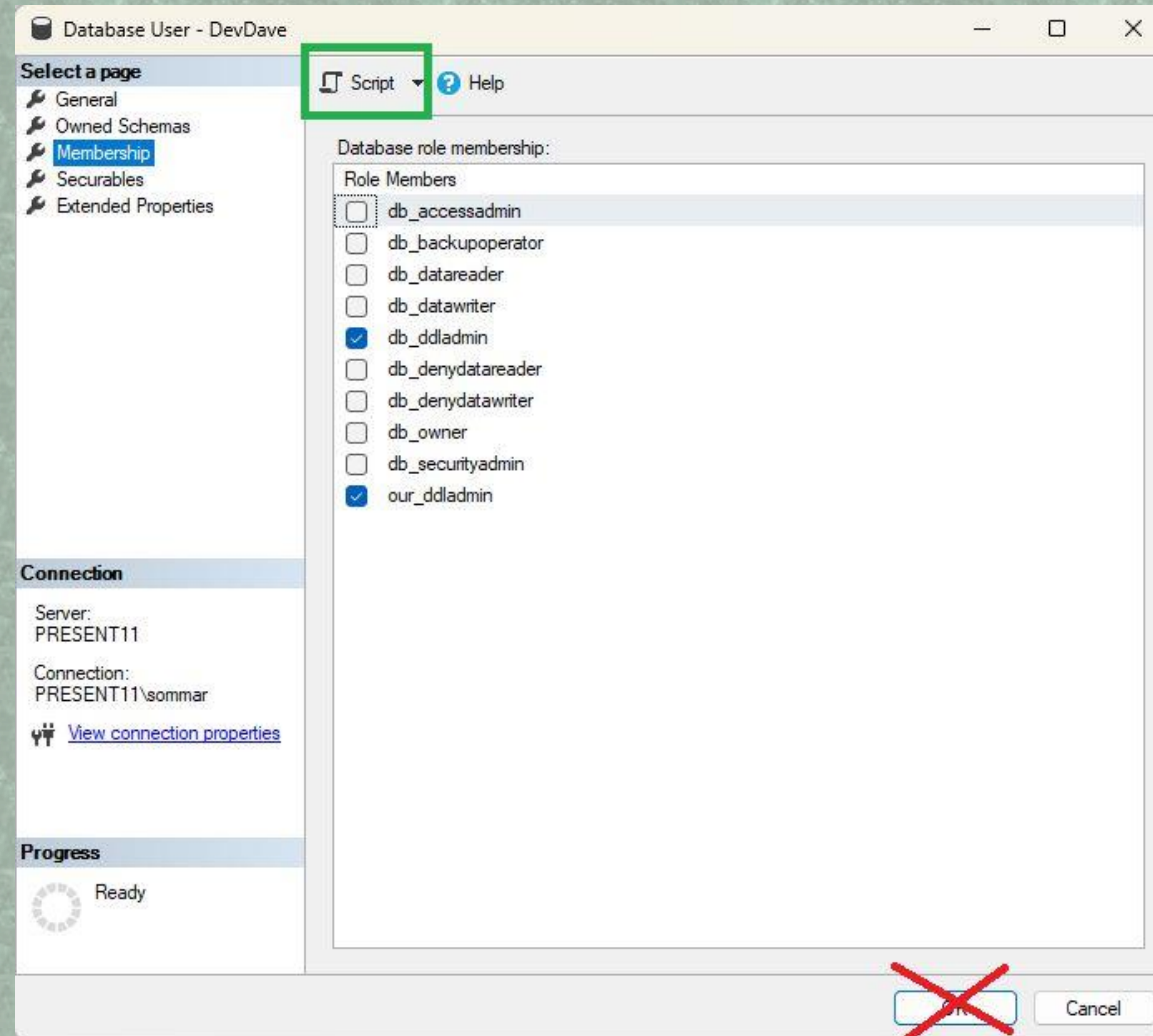
- Only do backup and DBCC. Tell users to run index and stats maintenance themselves.
- Or for index and stats maintenance, have one task per database. Each task has its own login that is `db_owner` solely in that database. (Could be the database owner.)



# DDL IN GENERAL

- For the server DBA any DDL on database level is dangerous: creating indexes, users, granting permissions etc.
- Must bracket all DDL with `EXECUTE AS USER = 'dbo'` (when there is a local db\_owner or similar).
- Corollary: cannot use UI in SSMS directly. Must use SQL commands.
- Or use the script button, see next slide.

# SCRIPT BUTTON IN SSMS UI





# DEVS AND DDL TRIGGERS

[03\\_db\\_ddladmin.sql](#)

- A user in db\_ddladmin could create a DDL trigger to hijack the permissions of a db\_owner user.
- This role should not have rights to create DDL triggers!
- Take out all users from db\_ddladmin.
- Create a new role, e.g. our\_ddladmin (or ask for an AD group) that you add as a member to db\_ddladmin.
- Add devs to this role.
- Deny our\_ddladmin ALTER ANY DATABASE DDL TRIGGER.
- Why not DENY db\_ddladmin directly? SQL Server doesn't let us.

# INTERLUDE: EXECUTE AS CMDS

- **EXECUTE AS USER:** run commands as a database user.
  - Cannot access other databases.
  - Can access tempdb and master.
  - Need IMPERSONATE permission on the user.
- **EXECUTE AS LOGIN:** run commands as a server login.
  - No sandboxing inside SQL Server.
  - Need IMPERSONATE on login (server-level permission).
- Both are great for testing permissions.
- And EXECUTE AS USER is good for removing your server-level permissions.



# INTERLUDE: ABOUT REVERT

- Impersonation is in force until the REVERT command is executed *in the same scope*.
- REVERT must be executed from the same database as where EXECUTE AS was executed.
- Tip: Always put REVERT in its own batch, so that is executed also if there is an error.
- Extraneous REVERT are ignored without error.
- Extra precaution: EXECUTE AS ... WITH NO REVERT.
  - When you are running a script you cannot trust.

# INTERLUDE: LINKED SERVERS

- After `EXECUTE AS USER`, you cannot access linked servers.
  - Because they are server objects.
- There are two choices for authentication for a linked server:
  - Self-mapping: Users log on the data source with their own credentials.
  - Login-mapping: Log in with username/pwd pre-stored in SQL Server. If remote server is SQL Server, this must be an SQL login.
  - Mapping is per user or for everyone.
- With `EXECUTE AS LOGIN`, self-mapping does not work.
  - Impersonation is not valid outside the scope where it was performed.



# DML TRIGGERS AND STORED PROCEDURES

- When you insert/update/delete data in tables, your permissions can be hijacked by a trigger.
- When running stored procedures, a developer may have hidden malicious code somewhere.
- If you only need to protect sysadmin (no one between db\_owner and plain users) same as before: impersonate dbo.
- To get data from outside the database, read into temp tables before starting impersonation.

# HOW TO PROTECT DB\_OWNER?

- If there are users who can change stored procedures, triggers etc, they can hijack db\_owner.
- Protection: PoLP, Principle of Least Privilege.
- Create a sandbox user with limited permissions and impersonate that user. For instance:

```
CREATE USER SandboxUser WITHOUT LOGIN
GRANT EXECUTE TO SandboxUser
GRANT SELECT, INSERT, UPDATE, DELETE TO SandboxUser
EXECUTE AS USER = 'SandboxUser'
```



# PERMISSIONS FOR SANDBOX USER

- In many cases, a sandbox user with permission to run all stored procedures and full rights on all tables is enough.
- Specifically, it is sufficient with someone like DevDave who already has these permissions.
- But what if there is Bettie BusinessAnalyst who has her own schema where she can create tables, and you need to work with her tables?
- You may need a sandbox user with permission only to Bettie's schema.

# ACCESS OUTSIDE THE DATABASE

- With `EXECUTE AS USER` you are sandboxed into the current database.
- What about stored procedures with access to other databases?
- What if you need to join to big tables in other databases? Getting data into temp tables beforehand may be impractical.
- You need to use a sandbox login instead that you impersonate with `EXECUTE AS LOGIN` (or log in as it).
- Easy for sysadmin. A `db_owner` needs to ask the server DBA to create one and get `IMPERSONATE` rights on that login.



# ACCESS TO LINKED SERVER?

- Your operation may include access to linked servers, directly or through stored procedures.
- If linked server set up with login-mapping, impersonating a sandbox login still works.
- If linked server has self-mapping, it gets complicated...
  - Set up login-mapping for the sandbox login only. (Remote server must be enabled for SQL authentication.)
  - Ask the AD admin to create a Windows user that you can use as a sandbox login. Start SSMS with RUNAS to log in as it.

# APPLICATION USERS

- Application may log in users with integrated security or application may use its own login/password.
- Ideally users and application logins should only have permission to run stored procedures.
- If application submits direct SQL statements, they also need SELECT, INSERT, UPDATE and DELETE permissions.
- But they should never have more permissions. **Never!**
  - Minimises the damage of any SQL-injection hole.
- Application logins can serve well as sandbox users.



# APPLICATIONS AND ELEVATED PERMISSIONS

- What if an application needs permissions beyond EXECUTE and read/write on tables?
  - Example: show a count of users connected to the database; requires server-level permission to use DMV.
- Rather than granting the permission to application login, you can package the permission inside a stored procedure with full control over how the permission is used.
- You sign the procedure with a certificate, and from this certificate you create a user/login which you grant the required permission(s).

# CERTIFICATE SIGNING: ARTICLE AND VIDEO

- On my web site, there is an in-depth article about this technique: *Packaging Permissions in Stored Procedures*, <https://www.sommarskog.se/grantperm.html>.
- Article includes stored procedure and script to automate the process.
- There is also a recording on [YouTube](#).



# CERTIFICATE SIGNING KEY FACTS

- When you change a signed procedure, signature and permissions disappear.
- Thus, it must be signed again.
- Example: Debbie Owner wants a procedure that shows user count.
- Server DBA reviews it and approves it by signing and granting permissions.
- If Debbie changes the procedure, she needs new approval.
- Thus, server DBA is in full control.

# CERTIFICATE SIGNING AND PERMISSION HIJACKING

- The packaged permissions apply inside dynamic SQL invoked by the signed procedure.
- They also apply inside system procedures called by the SP.
- But they are removed when entering user-written modules: sub-procedures, triggers and functions.
- That is, the packaged permissions cannot be hijacked!
- But watch out for dynamic SQL!



# AGENT JOBS

- DevDave approaches you and says: we need to run purges at night, can you create an Agent job to run this procedure?
- If you schedule it as sysadmin, Debbie and Dave can now make all sorts of changes to it.
- First line of defence: Hey, why don't you schedule it from Task Scheduler on the application server?
  - After all, Agent is for management tasks, not application jobs, isn't it?
- But you may not win that battle...

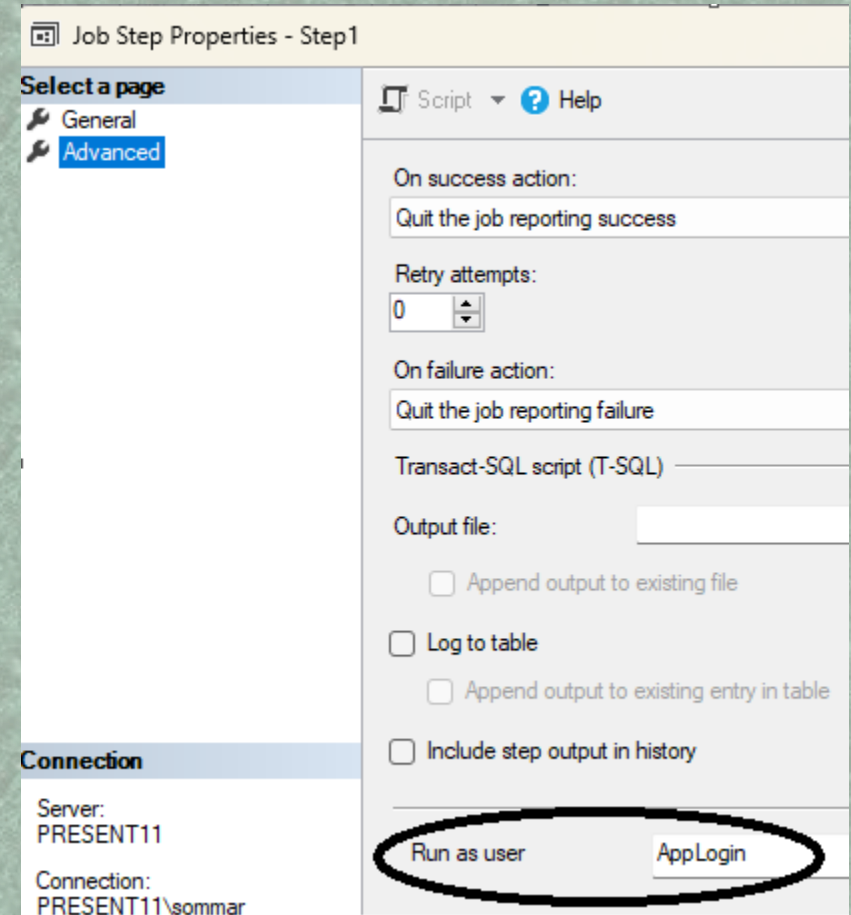
# SOME AGENT PRINCIPLES

- Jobs owners should not be persons who can leave the company.
- Avoid adding users to msdb.
- From this follows that setting up an Agent job is always a sysadmin task.
- Application jobs in Agent should execute with same permissions as application users.
- If job needs to perform actions that require higher permissions, use signed procedures.



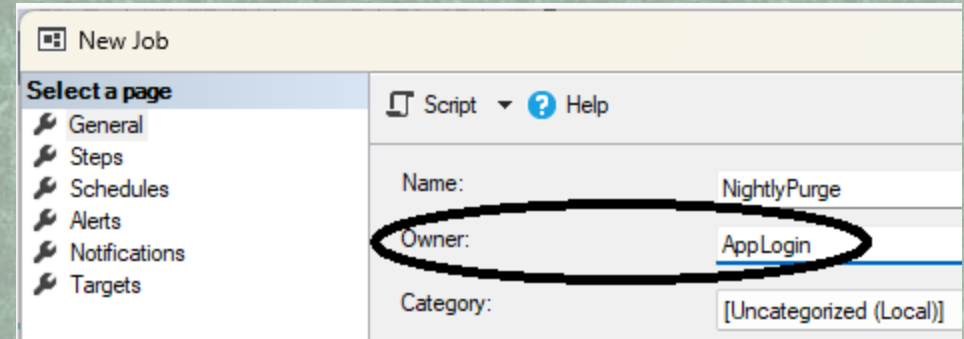
# ALTERNATIVE 2: SET USER ON JOB STEP

- Set user on the Advanced tab on the job step.
- Agent impersonates user with **EXECUTE AS USER**.
- Sandboxed into database, no cross-database access or linked servers.
- Job must be owned by sysadmin.
  - If not, setting is ignored.



# ALTERNATIVE 3: SET JOB OWNER

- Set job owner on first page of job.
- SQL login or Windows login with “application permissions”.
- Agent impersonates the owner with EXECUTE AS LOGIN.
- No restriction with cross-database or server access.
- Linked servers with login-mapping works. Those with self-mapping *do not*.
- Side effect: job owner gets added as a user to msdb and as a member of SQLAgentUserRole on next install of a CU.



The screenshot shows the 'New Job' dialog box with the 'General' tab selected. The 'Name' field is 'NightlyPurge', the 'Owner' field is 'AppLogin' (circled in black), and the 'Category' field is '[Uncategorized (Local)]'. The left sidebar shows 'General' as the selected page, with other options like 'Steps', 'Schedules', 'Alerts', 'Notifications', and 'Targets'.



# ALTERNATIVE 4: CMDEXEC JOB WITH PROXY

- More steps and involves more people.
- No impersonation, since Agent performs a real login in Windows.
- Thus, no restrictions with linked servers.
- While the most complicated, the most general.

# ALTERNATIVE 4: CMDEXEC JOB WITH PROXY

- First request to get a AD account.
- Create login in SQL Server.
  - For linked server access, this may include more than one instance.
- Create database user and grant application permissions.
  - For instance, add it to the ApplicationUsers role.
- Create a credential for this account in SQL Server:

```
CREATE CREDENTIAL App1JobRunner  
  WITH IDENTITY = 'DOMAIN\App1JobRunner',  
  SECRET = 'TheWindowsPassword'
```



# CMDEXEC JOB WITH PROXY

- Create a proxy in Agent with access to CmdExec.
  - Use credential from previous step.
- Create job in Agent with sysadmin as owner.
- Create job step.
  - Type: CmdExec.
  - RunAs: The proxy.
  - Command: SQLCMD.

**New Proxy Account**

Select a page: General, Principals

Script Help

Proxy name: ApplJobRunner

Credential name: ApplJobRunner

Description:

**Job Step Properties - Step1**

Select a page: General, Advanced

Script Help

Step name: Step1

Type: Operating system (CmdExec)

Run as: ApplJobRunner

Process exit code of a successful command: 0

Command: SQLCMD -i b -S \$(ESCAPE\_DQUOTE(SRVR)) -d AppDB -Q "PRINT SYS"

# CMDEXEC JOB WITH PROXY RUNNING SQLCMD

```
SQLCMD -I -b -S $(ESCAPE_DQUOTE(SRVR))  
        -d AppDB -Q"EXEC somesp"
```

- I – For SET QUOTED\_IDENTIFIER ON.
- b – Propagate error code, so job can be reported as failed.
- S – Use Agent token for server name to make it easier to move job to another instance.
- d – The database to connect to.
- Q – Command to run.



# NOTIFICATIONS TO DEVS

- Since developers and application admin are not in msdb, they don't have access to Agent history.
- Set up mail notifications, so they get mail if job fails. (On success as well, if they want.)
- On the Advanced page of the job step, you can define an output file. (Does not work when you set job owner.)
- Direct output file to a share where application team has access.

# ATTACKS THROUGH DEPLOYMENT

- You get a deployment package from the application team.
- It could include all sorts of nasty things:
  - Adding extra logins/users with elevated permissions.
  - Manipulate data (in any database).
- Or what about this:
  - Set up a temporary linked server to system B with sensitive data and where you also are sysadmin.
  - They set up another linked server to a test system C they have access to and dump the sensitive data there.
- How do you defend yourself against hijacking?

# DEFENCE AGAINST DEPLOYMENT ATTACKS

- For a plain script, one means of protection:  
`EXECUTE AS USER = 'dbo' WITH NO REVERT`
- `WITH NO REVERT` counteracts any `REVERT` in the script.
- Rather than `dbo`, use a sandbox user in our `_ddladmin`, or whatever permission you are prepared to grant the app team.
- Use a sandbox login, if actions are expected outside the database.
- First run in QA, this will reveal unexpected but legit actions.
  - But not necessarily malicious actions; they may check the permissions.



# DEPLOYMENT ATTACKS, CONT'D

- Deployment script may have legit actions that requires high permissions.
- Or deployment is through DACPAC or MSI install, with no options to control login.
- Auditing may be your only choice here.
- There are several options, but SQL Server Audit is the only that audits that it has been turned off.
- DDL Triggers? They can be turned off with `DISABLE TRIGGER` – which does not fire DDL triggers...

# ATTACKS THROUGH APPLICATIONS

- You are using the time-reporting system or some other local application.
- The developer of that application has a hook that recognises you and sets up a connection to an instance with sensitive data, abusing your permissions for malicious actions.
- Possible defence: One Windows user for normal company work. and a second login for DBA stuff, maybe through jump servers.
- Rigid network control of which machines can access each other.

# SUMMARY: WHAT THIS HAS BEEN ABOUT

- Malicious users with some elevated permission can inject code that you unknowingly execute with your permissions, performing actions you don't agree to.
- “Elevated permission” here includes access to the deployment pipeline!



# SUMMARY: MEANS OF PROTECTION

- Principle of Least Privilege, PoLP.
- Impersonate dbo to get rid of your server-level permissions.
- To protect db\_owner from being hijacked, use sandbox users or logins.
- Never put people directly in db\_ddladmin, but create a role our\_ddladmin which you deny the rights to DDL triggers.
- Application users should have limited permissions, at most SELECT, INSERT, UPDATE and DELETE on tables.

# SUMMARY: MEANS OF PROTECTION II

- Application tasks in Agent jobs should run with application permissions, never elevated permissions.
- Use procedure signing when application users need to perform actions beyond their permission set.
- The application team may in practice be equal to db\_owner – even if they don't have access to the server.

# FINAL WORDS

- Some of the attacks I have suggested may seem very contrived.
- Remember: if the stakes are high enough, a possible attack will be attempted, sooner or later at some place.
- Don't overlook the benevolent attacker.
  - That is, someone who performs an attack not out of malicious intent, but to circumvent corporate red tape to get their job done.
- You could have made those users sysadmin, but you decided to lock down their permissions. Certainly not a bad idea in itself.
- But if you don't watch out, they may find a way to “work around” that restriction.



# THAT'S ALL FOLKS!

Erland Sommarskog, [esquel@sommarskog.se](mailto:esquel@sommarskog.se).

Slides and scripts at <https://www.sommarskog.se/present>.

Don't Let Your Permissions be Hijacked:  
<https://www.sommarskog.se/perm-hijack.html>.

Clean-up script: [05\\_cleanup.sql](#)