

# DON'T LET YOUR PERMISSIONS BE HIJACKED!

<https://sqlb.it/?9560>

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?



# AGENDA

- The evil DDL trigger and the Principle of Least Privilege.
- The db\_ddladmin role and DDL triggers.
- The benefit of sandbox users and logins.
- Certificate signing and the permissions of application logins.
- Running application tasks in SQL Server Agent.
- Attacks through deployment.

# 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.

# THE EXECUTE AS COMMANDS

- **EXECUTE AS USER:** run commands as a database user.
  - Stripped of any permissions on server level.
  - Cannot access other user databases.
  - Completely locked out from linked servers.
  - Need IMPERSONATE permission on the user.
- **EXECUTE AS LOGIN:** run commands as a server login.
  - No sandboxing inside SQL Server.
  - No access to linked servers with self-mapping.
  - Linked servers with login-mapping works.
  - Need IMPERSONATE on the login (server permission).



# 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.
  - If you are running a script you cannot trust.

# 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

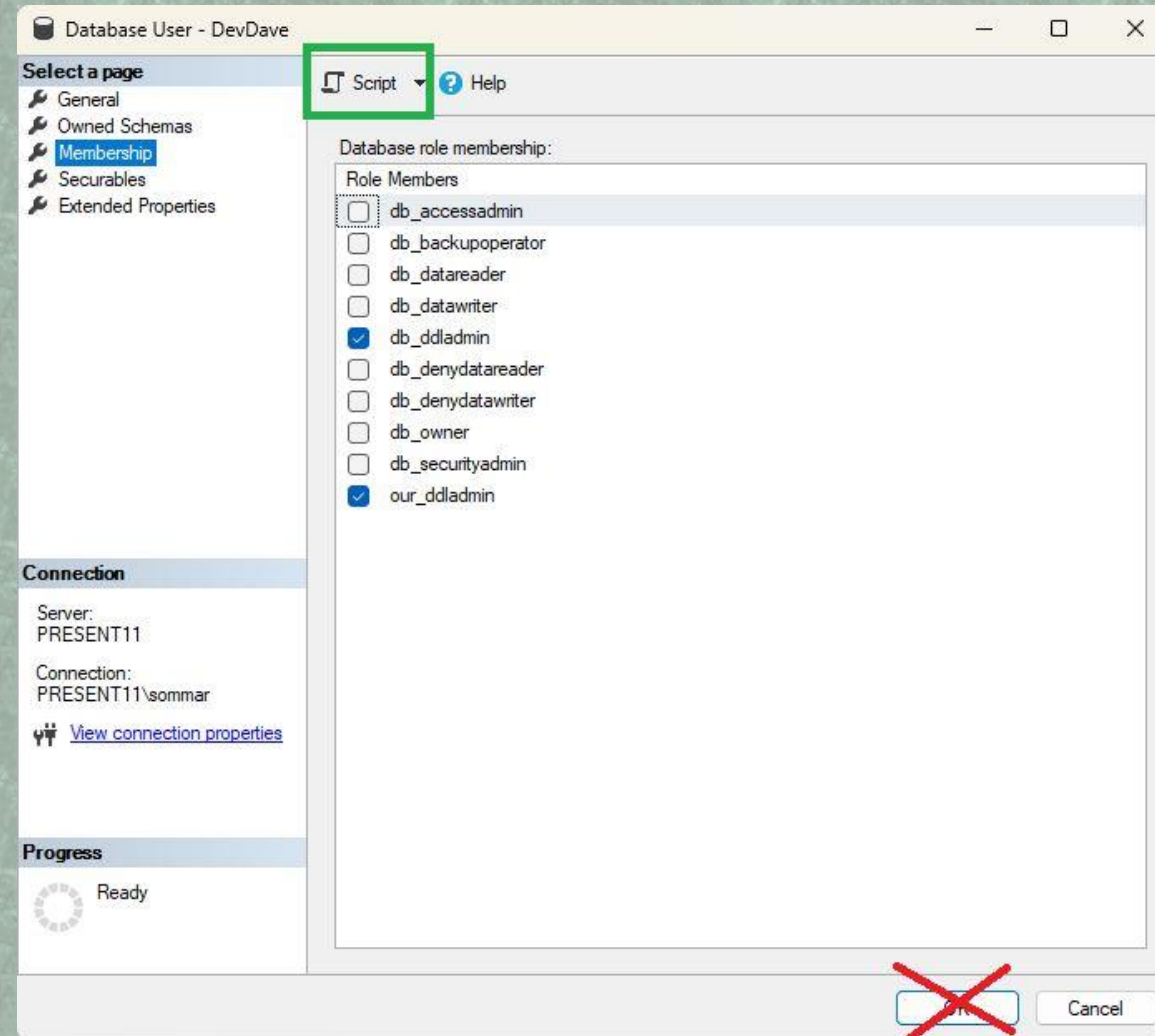
- [Ola Hallengren's Maintenance Solution](#).  
IndexOptimize accepts the parameter `@ExecuteAsUser`, which you set to `'dbo'` (or another user of your choice).
- The maintenance plans in SSMS – no provision for **EXECUTE AS USER**.
  - DBCC and backup is safe, it's index and stats maintenance that can be dangerous.

# DDL IN GENERAL

- For the server DBA any DDL on database level is dangerous: creating indexes, users, granting permissions etc.
- Corollary: cannot use UI in SSMS directly. Must use SQL commands.
- Or use the script button, see next slide.



# SCRIPT BUTTON IN SSMS UI



# DEVELOPERS AND DDL TRIGGERS

- Debbie gets an assistant, DevDave who is going to help with writing stored procedures, create tables etc.
- Dave is not trusted to create users etc, so she adds Dave to the **db\_ddladmin** role.
- But db\_ddladmin gives Dave permission to create DDL triggers.
- So Dave can write a DDL trigger that hijacks Debbie's permission and, for instance, adds him to db\_owner.



# HOW TO KEEP DEVS OUT FROM DDL TRIGGERS

- DDL triggers are not application code.
  - ➔ Developers have no business there.
- Create a role, e.g. **our\_ddladmin** (or request an AD group.)
- Make this role a member of db\_ddladmin.
- **DENY ALTER ANY DATABASE DDL TRIGGER TO our\_ddladmin**
- Add developers to this role, not to db\_ddladmin.
- DENY db\_ddladmin directly? SQL Server does not permit that.

# 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'
```

# PLAYING THE SANDBOX USER

As sysadmin, you need to run things in an unknown database.

1. Get any data needed from outside sources into temp tables.
2. Impersonate dbo with EXECUTE AS USER.
3. Create sandbox user WITHOUT LOGIN and grant permissions.
4. Impersonate sandbox user.
5. Do your task.
6. REVERT back to dbo.
7. Drop sandbox user.
8. REVERT back to your true self.



# 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 outside sources?  
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.

# 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 a 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](#).
- Upcoming: [SQL Friday #110](#), April 14<sup>th</sup>, 12:00 CET, online.



# 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 Agent jobs 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.

The screenshot shows the 'Job Step Properties - Step1' dialog box. The 'Advanced' tab is selected. The 'Run as user' field is circled in black, and the 'AppLogin' button is also circled in black.

Job Step Properties - Step1

Select a page

- General
- Advanced

Script Help

On success action:  
Quit the job reporting success

Retry attempts:  
0

On failure action:  
Quit the job reporting failure

Transact-SQL script (T-SQL)

Output file:

☐ Append output to existing file

☐ Log to table

☐ Append output to existing entry in table

☐ Include step output in history

Connection

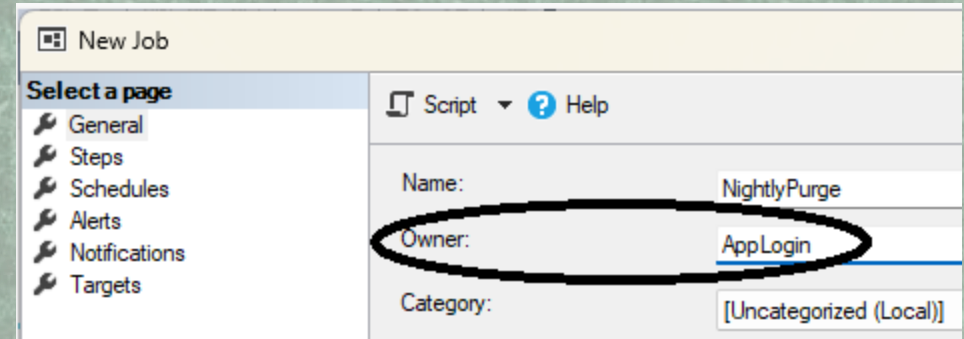
Server:  
PRESENT11

Connection:  
PRESENT11\sommar

Run as user AppLogin

# 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.
- Thus, no restriction with cross-database or server access.
- But linked servers with self-mapping will still not work.
- Side effect: job owner gets added as a user to msdb and as a member of SQLAgentUserRole on next install of a CU.





# 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

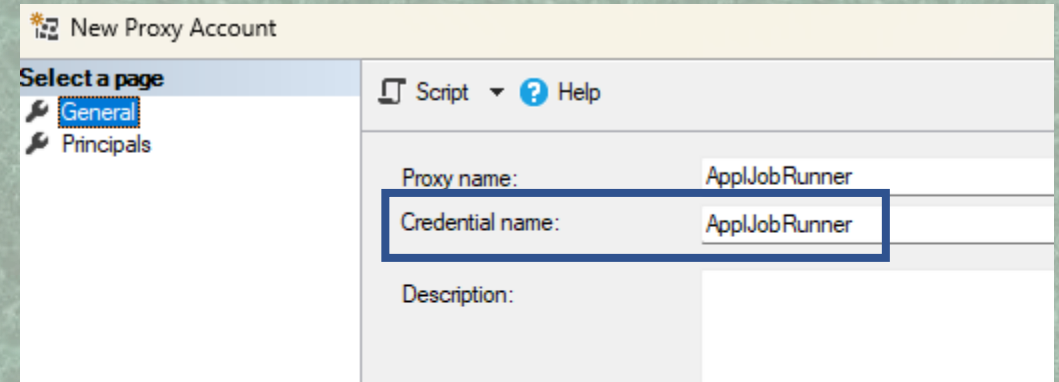
- First request an AD account.
- Create login in SQL Server.
  - For linked server access, this may include more than one instance.
- Create database user and grant application permissions.
  - For instance, add it to the ApplicationUsers role.
- Create a credential for this account in SQL Server:

```
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

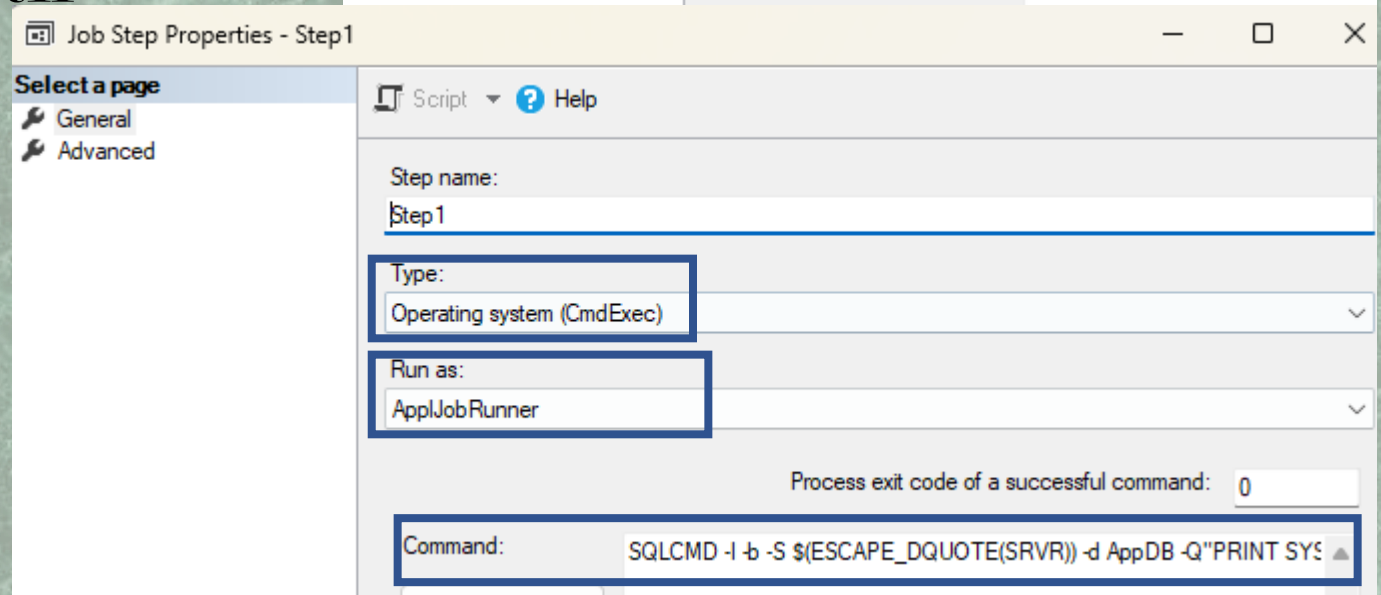
Select a page: General, Principals

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...



# SUMMARY

- A user with some elevated permission can lure you to unknowingly run code that abuses your permissions to run malicious actions.
- Protection is based on 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.

# SUMMARY II

- Application users should have limited permissions, at most SELECT, INSERT, UPDATE and DELETE on tables.
- 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 be in practice be equal to db\_owner – even if they don't have access to the server.

# THAT'S ALL FOLKS!

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

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

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

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

Feedback link!



<https://sqlb.it/?9560>



Link for submitting  
feedback!

<https://sqlb.it/?9560>

