The Internals of WITH ENCRYPTION

May 20, 2016 by Paul White in SQL Performance | 16 Comments

It is pretty easy for a SQL Server administrator to recover the text of stored procedures, views, functions, and triggers protected using `WITH ENCRYPTION`. Many articles have been written about this, and several commercial tools are available. The basic outline of the common method is to:

1. Obtain the encrypted form (A) using the Dedicated Administrator Connection.
2. Start a transaction.
3. Replace the object definition with known text (B) of at least the same length as the original.
4. Obtain the encrypted form for the known text (C).
5. Roll back the transaction to leave the target object in its initial state.
6. Obtain the unencrypted original by applying an exclusive-or to each character: `A XOR (B XOR C)`

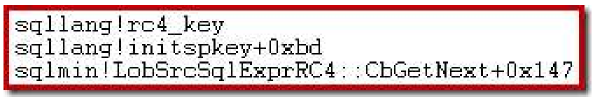
That is all pretty straightforward, but seems a bit like magic: It does not explain much about *how and why it works*. This article covers that aspect for those of you that find these sorts of details interesting, and provides an alternative method for decryption that is more illustrative of the process.

The Stream Cipher

The underlying encryption algorithm SQL Server uses for *module encryption* is the RC4™ stream cipher. An outline of the encryption process is:

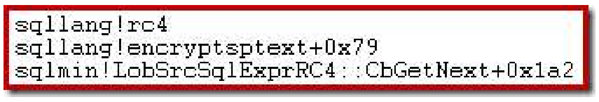
1. Initialize the RC4 cipher with a cryptographic key.
2. Generate a pseudorandom stream of bytes.
3. Combine the module plain text with the byte stream using exclusive-or.

We can see this process occurring using a debugger and public symbols. For example, the stack trace below shows SQL Server initializing the RC4 key while preparing to encrypt the module text:



```
sqllang!rc4_key
sqllang!initspkey+0xbd
sqlmin!LobSrcSqlExprRC4::CbGetNext+0x147
```

This next one shows SQL Server encrypting the text using the RC4 pseudorandom byte stream:



```
sqllang!rc4
sqllang!encryptsptext+0x79
sqlmin!LobSrcSqlExprRC4::CbGetNext+0x1a2
```

Like most stream ciphers, the process of decryption is the same as encryption, making use of the fact that exclusive-or is reversible (`A XOR B XOR B = A`).

The use of a stream cipher is the reason *exclusive-or* is used in the method described at the start of the article. There is nothing inherently unsafe about using exclusive-or, provided that a secure encryption method is used, the initialization key is kept secret, and the key is not reused.

RC4 is not particularly strong, but that is not the main issue here. That said, it is worth noting that encryption using RC4 is being gradually removed from SQL Server, and is deprecated (or disabled, depending on version and database compatibility level) for user operations like creating a symmetric key.

The RC4 Initialization Key

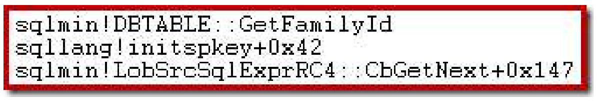
SQL Server uses three pieces of information to generate the key used to initialize the RC4 stream cipher:

1. The database family GUID.
   This can be obtained most easily by querying *sys.database_recovery_status*. It is also visible in undocumented commands like `DBCC DBINFO` and `DBCC DBTABLE`.

2. The target module's object ID.
   This is just the familiar object ID. Note that not all modules that allow encryption are schema-scoped. You will need to use metadata views (*sys.triggers* or *sys.server_triggers*) to get the object ID for DDL and server-scoped triggers, rather than *sys.objects* or `OBJECT_ID`, since these only work with schema-scoped objects.

3. The target module's sub-object ID.
   This is the procedure number for numbered stored procedures. It is 1 for an unnumbered stored procedure, and zero in all other cases.

Using the debugger again, we can see the family GUID being retrieved during key initialization:



```
sqlmin!DBTABLE::GetFamilyId
sqllang!initspkey+0x42
sqlmin!LobSrcSqlExprRC4::CbGetNext+0x147
```

The database family GUID is typed *uniqueidentifier*, object ID is *integer*, and sub-object ID is *smallint*.

Each part of the key **must** be converted to a specific binary format. For the database family GUID, converting the *uniqueidentifier* type to *binary(16)* produces the correct binary representation. The two IDs must be converted to binary in little-endian representation (least significant byte first).

Note: Be very careful not to accidentally provide the GUID as a string! It must be typed *uniqueidentifier*.

The code snippet below shows correct conversion operations for some sample values:

```
DECLARE
    @family_guid binary(16) = CONVERT(binary(16), {guid 'B1FC892E-5824-4FD3-AC48-FBCD91D57763'}),
    @objid binary(4) = CONVERT(binary(4), REVERSE(CONVERT(binary(4), 800266156))),
    @subobjid binary(2) = CONVERT(binary(2), REVERSE(CONVERT(binary(2), 0)));
```

The final step to generate the RC4 initialization key is to concatenate the three binary values above into a single binary(22), and compute the SHA-1 hash of the result:

```
DECLARE
    @RC4key binary(20) = HASHBYTES('SHA1', @family_guid + @objid + @subobjid);
```

For the sample data given above, the final initialization key is:

```
0x6C914908E041A08DD8766A0CFEDC113585D69AF8
```

The contribution of the target module's object ID and sub-object ID to the SHA-1 hash are hard to see in a single debugger screenshot, but the interested reader can refer to the disassembly of a portion of *initspkey* below:

```
call    sqllang!A_SHAInit
lea     rdx,[rsp+40h]
lea     rcx,[rsp+50h]
mov     r8d,10h
call    sqllang!A_SHAUpdate
lea     rdx,[rsp+24h]
lea     rcx,[rsp+50h]
```

```
mov     r8d,4
call    sqllang!A_SHAUpdate
lea     rdx,[rsp+20h]
lea     rcx,[rsp+50h]
mov     r8d,2
call    sqllang!A_SHAUpdate
lea     rdx,[rsp+0D0h]
lea     rcx,[rsp+50h]
call    sqllang!A_SHAFinal
lea     r8,[rsp+0D0h]
mov     edx,14h
mov     rcx,rbx
call    sqllang!rc4_key (00007fff`89672090)
```

The *SHAInit* and *SHAUpdate* calls add components to the SHA hash, which is eventually computed by a call to *SHAFinal*.

The *SHAInit* call contributes 10h bytes (16 decimal) stored at [rsp+40h], which is the **family GUID**. The first *SHAUpdate* call adds 4 bytes (as indicated in the r8d register), stored at [rsp+24h], which is the **object** ID. The second *SHAUpdate* call adds 2 bytes, stored at [rsp+20h], which is the **subobjid**.

The final instructions pass the computed SHA-1 hash to the RC4 key initialization routine *rc4_key*. The length of the hash is stored in register edx: 14h (20 decimal) bytes, which is the defined hash length for SHA and SHA-1 (160 bits).

The RC4 Implementation

The core RC4 algorithm is well-known, and relatively simple. It would be better implemented in a .Net language for efficiency and performance reasons, but there is a T-SQL implementation below.

These two T-SQL functions implement the RC4 key-scheduling algorithm and pseudorandom number generator, and were originally written by SQL Server MVP Peter Larsson. I have a made some minor modifications to improve performance a little, and allow LOB-length binaries to be encoded and decoded. This part of the process could be replaced by any standard RC4 implementation.

```
/*
** RC4 functions
** Based on http://www.sqlteam.com/forums/topic.asp?TOPIC_ID=76258
** by Peter Larsson (SwePeso)
*/
IF OBJECT_ID(N'dbo.fnEncDecRc4', N'FN') IS NOT NULL
    DROP FUNCTION dbo.fnEncDecRc4;
GO
IF OBJECT_ID(N'dbo.fnInitRc4', N'TF') IS NOT NULL
    DROP FUNCTION dbo.fnInitRc4;
GO
CREATE FUNCTION dbo.fnInitRc4
    (@Pwd varbinary(256))
RETURNS @Box table
    (
        i tinyint PRIMARY KEY,
        v tinyint NOT NULL
    )
WITH SCHEMABINDING
AS
BEGIN
    DECLARE @Key table
    (
        i tinyint PRIMARY KEY,
        v tinyint NOT NULL
    );

    DECLARE
        @Index smallint = 0,
```

```sql
        @PwdLen tinyint = DATALENGTH(@Pwd);

    WHILE @Index <= 255
    BEGIN
        INSERT @Key
            (i, v)
        VALUES
            (@Index, CONVERT(tinyint, SUBSTRING(@Pwd, @Index % @PwdLen + 1, 1)));

        INSERT @Box (i, v)
        VALUES (@Index, @Index);

        SET @Index += 1;
    END;

    DECLARE
        @t tinyint = NULL,
        @b smallint = 0;

    SET @Index = 0;

    WHILE @Index <= 255
    BEGIN
        SELECT @b = (@b + b.v + k.v) % 256
        FROM @Box AS b
        JOIN @Key AS k
            ON k.i = b.i
        WHERE b.i = @Index;

        SELECT @t = b.v
        FROM @Box AS b
        WHERE b.i = @Index;

        UPDATE b1
        SET b1.v = (SELECT b2.v FROM @Box AS b2 WHERE b2.i = @b)
        FROM @Box AS b1
        WHERE b1.i = @Index;

        UPDATE @Box
        SET v = @t
        WHERE i = @b;

        SET @Index += 1;
    END;

    RETURN;
END;
GO
CREATE FUNCTION dbo.fnEncDecRc4
(
    @Pwd varbinary(256),
    @Text varbinary(MAX)
)
RETURNS varbinary(MAX)
WITH
    SCHEMABINDING,
    RETURNS NULL ON NULL INPUT
AS
BEGIN
    DECLARE @Box AS table
    (
        i tinyint PRIMARY KEY,
        v tinyint NOT NULL
    );

    INSERT @Box
```

```
        (i, v)
    SELECT
        FIR.i, FIR.v
    FROM dbo.fnInitRc4(@Pwd) AS FIR;

    DECLARE
        @Index integer = 1,
        @i smallint = 0,
        @j smallint = 0,
        @t tinyint = NULL,
        @k smallint = NULL,
        @CipherBy tinyint = NULL,
        @Cipher varbinary(MAX) = 0x;

    WHILE @Index <= DATALENGTH(@Text)
    BEGIN
        SET @i = (@i + 1) % 256;

        SELECT
            @j = (@j + b.v) % 256,
            @t = b.v
        FROM @Box AS b
        WHERE b.i = @i;

        UPDATE b
        SET b.v = (SELECT w.v FROM @Box AS w WHERE w.i = @j)
        FROM @Box AS b
        WHERE b.i = @i;

        UPDATE @Box
        SET v = @t
        WHERE i = @j;

        SELECT @k = b.v
        FROM @Box AS b
        WHERE b.i = @i;

        SELECT @k = (@k + b.v) % 256
        FROM @Box AS b
        WHERE b.i = @j;

        SELECT @k = b.v
        FROM @Box AS b
        WHERE b.i = @k;

        SELECT
            @CipherBy = CONVERT(tinyint, SUBSTRING(@Text, @Index, 1)) ^ @k,
            @Cipher = @Cipher + CONVERT(binary(1), @CipherBy);

        SET @Index += 1;
    END;

    RETURN @Cipher;
END;
GO
```

The Encrypted Module Text

The easiest way for a SQL Server administrator to get this is to read the *varbinary(max)* value stored in the *imageval* column of *sys.sysobjvalues*, which is only accessible via the Dedicated Administrator Connection (DAC).

This is the same idea as the routine method described in the introduction, though we add a filter on *valclass* = 1. This internal table is also a convenient place to get the *subobjid*. Otherwise, we would need to check *sys.numbered_procedures* when the target object is a procedure, use 1 for an unnumbered procedure, or zero for anything else, as described previously.

It is possible to **avoid using the DAC** by reading the *imageval* from *sys.sysobjvalues* directly, using multiple `DBCC PAGE` calls. This involves a bit more work to locate the pages from metadata, follow the *imageval* LOB chain, and read the target binary data from each page. The latter step is a lot easier to do in a programming language other than T-SQL. Note that `DBCC PAGE` will work, even though the base object is not normally readable from a non-DAC connection. If the page is not in memory, it will be read in from persistent storage as normal.

The extra effort to avoid the DAC requirement pays off by allowing multiple users to use the decrypting process concurrently. I will use the DAC approach in this article for simplicity and space reasons.

Worked Example

The following code creates a test encrypted scalar function:

```
CREATE FUNCTION dbo.FS()
RETURNS varchar(255)
WITH ENCRYPTION, SCHEMABINDING AS
BEGIN
    RETURN
    (
        SELECT 'My code is so awesome is needs to be encrypted!'
    );
END;
```

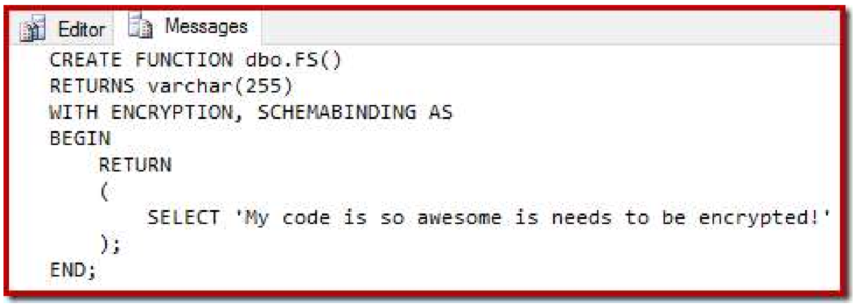
The complete decryption implementation is below. The only parameter that needs changing to work for other objects is the initial value of `@objectid` set in the first `DECLARE` statement.

```
-- *** DAC connection required! ***
-- Make sure the target database is the context
USE Sandpit;

DECLARE
    -- Note: OBJECT_ID only works for schema-scoped objects
    @objectid integer = OBJECT_ID(N'dbo.FS', N'FN'),
    @family_guid binary(16),
    @objid binary(4),
    @subobjid binary(2),
    @imageval varbinary(MAX),
    @RC4key binary(20);

-- Find the database family GUID
SELECT @family_guid = CONVERT(binary(16), DRS.family_guid)
FROM sys.database_recovery_status AS DRS
WHERE DRS.database_id = DB_ID();

-- Convert object ID to little-endian binary(4)
SET @objid = CONVERT(binary(4), REVERSE(CONVERT(binary(4), @objectid)));

SELECT
    -- Read the encrypted value
    @imageval = SOV.imageval,
    -- Get the subobjid and convert to little-endian binary
    @subobjid = CONVERT(binary(2), REVERSE(CONVERT(binary(2), SOV.subobjid)))
FROM sys.sysobjvalues AS SOV
WHERE
    SOV.[objid] = @objectid
    AND SOV.valclass = 1;

-- Compute the RC4 initialization key
```

```
SET @RC4key = HASHBYTES('SHA1', @family_guid + @objid + @subobjid);

-- Apply the standard RC4 algorithm and
-- convert the result back to nvarchar
PRINT CONVERT
    (
        nvarchar(MAX),
        dbo.fnEncDecRc4
        (
            @RC4key,
            @imageval
        )
    );
```

Note the final conversion to *nvarchar* because module text is typed as *nvarchar(max)*.

The output is:



```
CREATE FUNCTION dbo.FS()
RETURNS varchar(255)
WITH ENCRYPTION, SCHEMABINDING AS
BEGIN
    RETURN
    (
        SELECT 'My code is so awesome is needs to be encrypted!'
    );
END;
```

Conclusion

The reasons the method described in the introduction works are:

- SQL Server uses the RC4 stream cipher to reversibly exclusive-or the source text.
- The RC4 key depends only on the database family guid, object id, and subobjid.
- Temporarily replacing the module text means the same (SHA-1 hashed) RC4 key is generated.
- With the same key, the same RC4 stream is generated, allowing exclusive-or decryption.

Users that do not have access to system tables, database files, or other admin-level access, cannot retrieve encrypted module text. Since SQL Server itself needs to be able to decrypt the module, there is no way to prevent suitably privileged users from doing the same.

16 thoughts on “The Internals of WITH ENCRYPTION”

tobi says:
May 20, 2016 at 6:51 PM
What is `{guid 'B1FC892E-5824-4FD3-AC48-FBCD91D57763'}`? I do not recognize that syntax.

Paul White says:
May 20, 2016 at 9:16 PM
It is the ODBC escape sequence for a GUID literal. You can see the same thing in execution plans, for example look at the Compute Scalar's Defined Values for:

```
SELECT TOP (1) CONVERT(uniqueidentifier, '986FF056-A1C7-444B-B8C0-184AFB1311BA');
```

Coyote Ugly says:
May 23, 2016 at 6:46 PM