**Add a Salt with the SQL Server HASHBYTES Function**

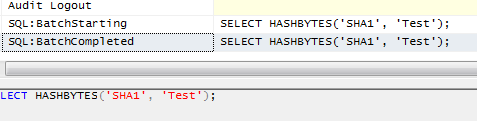
|  |  |
| --- | --- |
|  | By: [K. Brian Kelley](http://www.mssqltips.com/sqlserverauthor/25/k.-brian-kelley/)  FROM: <http://www.mssqltips.com/sqlservertip/3293/add-a-salt-with-the-sql-server-hashbytes-function/?utm_source=dailynewsletter&utm_medium=email&utm_content=text&utm_campaign=20140725> |

**Problem**

I am trying to store password hashes in SQL Server. I know I can generate those hashes using the [HASHBYTES() function](http://www.mssqltips.com/sqlservertip/2988/understanding-the-sql-server-hashbytes-hashing-algorithms/), but I don't see where it takes a salt. I've been told it's good to have a salt. Is there an easy way to do this?

**Solution**

Indeed there is. However, first, a caveat. If you can, you want to generate the hash in the application. If you don't, there is the potential for a DBA to be able to see the password using [SQL Profiler](http://www.mssqltips.com/sql-server-tip-category/83/profiler-and-trace/), a [server side trace](http://www.mssqltips.com/sql-server-tip-category/83/profiler-and-trace/), or through [Extended Events](http://www.mssqltips.com/sql-server-tip-category/99/extended-events/). [HASHBYTES()](http://www.mssqltips.com/sqlservertip/2988/understanding-the-sql-server-hashbytes-hashing-algorithms/) doesn't cause these mechanisms to hide the T-SQL that was passed, as can be seen here:



If you can't do this at the application layer, here's how to do it entirely within SQL Server.

**What to Use as a Salt the SQL Server HASHBYTES() function**

If you're not familiar with what the salt is when it comes to cryptographic functions, it's basically something added to whatever we're trying to encrypt to make it harder to decrypt the data (two way functions, like symmetric and asymmetric key functions) or find a collision (one way functions, AKA hash functions). The salt should be potentially different for every single piece of encrypted data. The salt should be randomly generated.

Since the salt should be randomly generated, this eliminates basic functions derived from date/time or anything of that sort. SQL Server does have a [RAND() function](http://www.mssqltips.com/sqlservertip/3055/generating-random-numbers-in-sql-server-without-collisions/), which does serve as random number generator. In addition, it can be seeded. However, it's a pseudo-random number generator. If you give it the same seed, it'll produce the same results. Therefore, we'll want our potential seed value range to be large.

We can use the time, specifically the hour, minute, second, and millisecond values to generate a reasonable large seed value pool. It is not perfectly random, but nothing ever is when it comes to these functions. Most of the random number generator functions work off of the computer clock and we're basically using that in order to generate the values for our salt. That leads to something like:

DECLARE @Salt VARCHAR(25);

-- Generate the salt

DECLARE @Seed int;

DECLARE @LCV tinyint;

DECLARE @CTime DATETIME;

SET @CTime = GETDATE();

SET @Seed = (DATEPART(hh, @Ctime) \* 10000000) + (DATEPART(n, @CTime) \* 100000)

+ (DATEPART(s, @CTime) \* 1000) + DATEPART(ms, @CTime);

SET @LCV = 1;

SET @Salt = CHAR(ROUND((RAND(@Seed) \* 94.0) + 32, 3));

WHILE (@LCV < 25)

BEGIN

SET @Salt = @Salt + CHAR(ROUND((RAND() \* 94.0) + 32, 3));

SET @LCV = @LCV + 1;

END;

Note that I'm generating the seed value by shifting hour, minute, and second values over by powers of ten. Then I'm using the RAND() function to generate a text string of 25 characters. This will be our salt.

**Putting It All Together**

With the salt generated, it's a simple matter of concatenating the salt and the password, then submitting the combined string into [HASHBYTES()](http://www.mssqltips.com/sqlservertip/2988/understanding-the-sql-server-hashbytes-hashing-algorithms/). This results in a solution which will store both the salt and the salt+password hash:

CREATE DATABASE TestDB;

GO

USE TestDB;

GO

CREATE TABLE dbo.SecurityAccounts (

AccountID INT IDENTITY(1,1),

AccountName varchar(50),

Salt CHAR(25),

AccountPwd varbinary(20),

CONSTRAINT PK\_SecurityAccounts PRIMARY KEY (AccountID)

);

GO

CREATE UNIQUE INDEX NDX\_SecurityAccounts\_AccountName

ON dbo.SecurityAccounts (AccountName) INCLUDE (Salt, AccountPwd);

GO

CREATE PROC dbo.CreateAccount

@NewAccountName VARCHAR(50),

@NewAccountPwd VARCHAR(100)

AS

BEGIN

SET NOCOUNT ON;

DECLARE @Salt VARCHAR(25);

DECLARE @PwdWithSalt VARCHAR(125);

-- Generate the salt

DECLARE @Seed int;

DECLARE @LCV tinyint;

DECLARE @CTime DATETIME;

SET @CTime = GETDATE();

SET @Seed = (DATEPART(hh, @Ctime) \* 10000000) + (DATEPART(n, @CTime) \* 100000)

+ (DATEPART(s, @CTime) \* 1000) + DATEPART(ms, @CTime);

SET @LCV = 1;

SET @Salt = CHAR(ROUND((RAND(@Seed) \* 94.0) + 32, 3));

WHILE (@LCV < 25)

BEGIN

SET @Salt = @Salt + CHAR(ROUND((RAND() \* 94.0) + 32, 3));

SET @LCV = @LCV + 1;

END;

SET @PwdWithSalt = @Salt + @NewAccountPwd;

INSERT INTO dbo.SecurityAccounts

(AccountName, Salt, AccountPwd)

VALUES (@NewAccountName, @Salt, HASHBYTES('SHA1', @PwdWithSalt));

END;

GO

As for verification, we'll need to basically repeat the same steps, except we'll retrieve the stored salt from the database.

CREATE PROC dbo.VerifyAccount

@AccountName VARCHAR(50),

@AccountPwd VARCHAR(100)

AS

BEGIN

SET NOCOUNT ON;

DECLARE @Salt CHAR(25);

DECLARE @PwdWithSalt VARCHAR(125);

DECLARE @PwdHash VARBINARY(20);

SELECT @Salt = Salt, @PwdHash = AccountPwd

FROM dbo.SecurityAccounts WHERE AccountName = @AccountName;

SET @PwdWithSalt = @Salt + @AccountPwd;

IF (HASHBYTES('SHA1', @PwdWithSalt) = @PwdHash)

RETURN 0;

ELSE

RETURN 1;

END;

GO

**Testing the Solution**

We can test it both with a relatively normal sized password and with the longest password allowed.

USE TestDB;

GO

EXEC dbo.CreateAccount @NewAccountName = 'Brian', @NewAccountPwd = 'Str0ngP4ssw0rd!';

GO

SELECT \* FROM dbo.SecurityAccounts;

DECLARE @Result TINYINT;

EXEC @Result = dbo.VerifyAccount @AccountName = 'Brian', @AccountPwd = 'Str0ngP4ssw0rd!';

SELECT @Result;

EXEC @Result = dbo.VerifyAccount @AccountName = 'Brian', @AccountPwd = 'WeakP4ssw0rd!';

SELECT @Result;

GO

DECLARE @Password CHAR(100);

SET @Password = REPLICATE('A', 100);

EXEC dbo.CreateAccount @NewAccountName = 'John', @NewAccountPwd = @Password;

GO

SELECT \* FROM dbo.SecurityAccounts;

GO

DECLARE @Result TINYINT;

DECLARE @Password CHAR(100);

SET @Password = REPLICATE('A', 100);

EXEC @Result = dbo.VerifyAccount @AccountName = 'Brian', @AccountPwd = @Password;

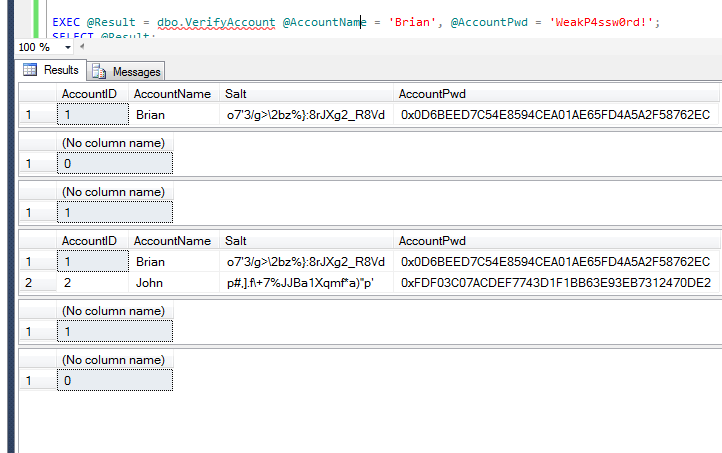
SELECT @Result;

EXEC @Result = dbo.VerifyAccount @AccountName = 'John', @AccountPwd = @Password;

SELECT @Result;

GO

If we get a zero on the return from the stored procedure, we have a match. With a value of 1, we don't. Therefore, if we just run the verification test all at once, we get:



**Next Steps**

* Read up on the [hashing algorithms presented by HASHBYTES() so you can choose the correct one](http://www.mssqltips.com/sqlservertip/2988/understanding-the-sql-server-hashbytes-hashing-algorithms/).
* Learn how to use [authenticators for other forms of encryption within SQL Server](http://www.mssqltips.com/sqlservertip/1760/using-authenticators-with-sql-server-encryption/).
* Know how to restrict what the [DBAs see with respect to data that needs to be encrypted](http://www.mssqltips.com/sqlservertip/2840/sql-server-encryption-to-block-dbas-data-access/).

Last Update: 7/25/2014

[**View all my tips**](http://www.mssqltips.com/sqlserverauthor/25/k.-brian-kelley/)

[Learn More](http://www.mssqltips.com/sql-server-tip-category/19/security/?utm_source=website&utm_medium=rightside&utm_content=learnmore&utm_campaign=20140221)

**Comments and Feedback:**

|  |  |
| --- | --- |
| **Friday, July 25, 2014 - 9:31:28 AM - Jason Carter** | [Read The Tip](http://www.mssqltips.com/sqlservertip/3293/add-a-salt-with-the-sql-server-hashbytes-function/) |
| While I understand that this is just a demo, storing the salt is obviously a terrible idea, unless that salt is another piece of data that would be there anyway, such as the email address.  Using the email address would create a unique salt value for each key, but then that is no good, as someone who has the database could easily figure out the proc and get all the password.  A third idea would be to combine both a user salt (email address)  with a second, system wide salt, which could be stored somewhere local to the system, such as the registry or a file on the disk even, but then again, those present their own issues as well.  Great article. | |