The SQL Server Crypto Detour

specterops.io/blog/2025/04/08/the-sql-server-crypto-detour

adam chester April 8, 2025

Share

As part of my role as Service Architect here at SpecterOps, one of the things I'm tasked with is exploring all kinds of technologies to help those on assessments with advancing their engagement.

Not long after starting this new role, I was approached with an interesting problem. A SQL Server database backup for a ManageEngine's ADSelfService Plus product had been recovered and, while the team had walked through the database recovery, SQL Server database encryption was in use. With a ticking clock, the request was clear… can we do anything to recover sensitive information from the database with only a .bak file available?

One of the things that I love about this job is getting to dig into various technologies and seeing the resulting research being used in real-time. After some research, we had decryption keys, a method of decrypting sensitive data, and DA credentials extracted and ready to go!

This post will explore how this was done, look at how SQL Server encryption works, introduce some new methods of brute-forcing database encryption keys, and show a mistake in ManageEngine's ADSelfService product which allows compromised database backups to reveal privileged credentials.

Manage Engine Protected Data

Let's start with Manage Engine's ADSelfService product. Documentation shows that Domain Admin credentials are likely present:

> - **Information fetched from the domain is stored in the product's database (the in-built PostgreSQL or any other database configured externally). During domain configuration, the credentials provided must have Domain Admin privileges or the individual privileges listed out in this guide.**

If we setup this tool in a lab environment, we find encrypted fields such as the below USER_NAME column:

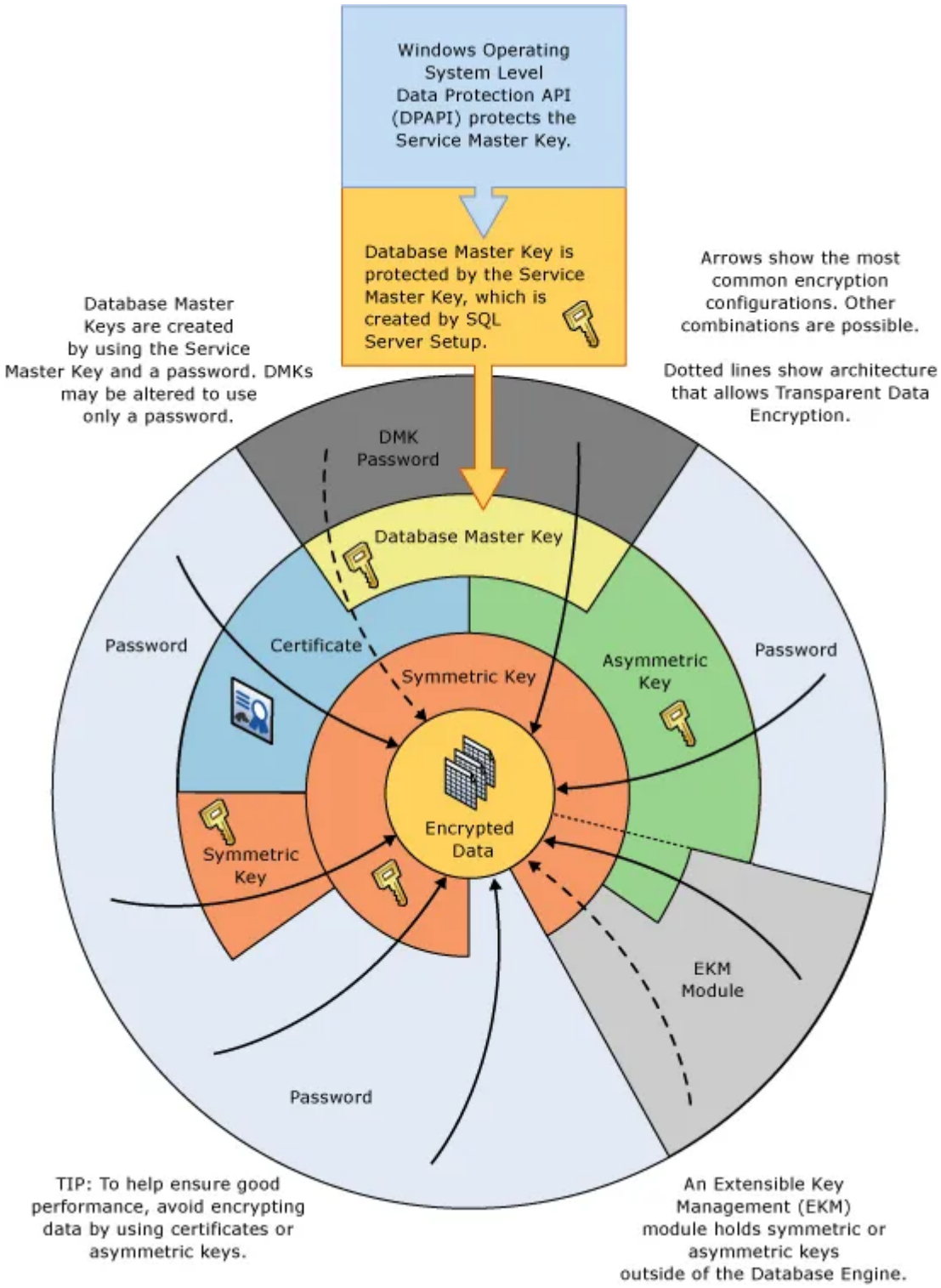
DOMAIN_NAME	DOMAIN_DNS_NAME	DOMAIN_FLAT_NAME	DOMAIN_CONTROLLER_ID	USER_NAME
lab.local	lab.local	lab	2	0x004F9C0281383874BDF1C828D(

Further, if we review the configuration of the database, we see that this is SQL Server's builtin encryption functionality that is being used to protect these fields. So the mission is clear: we need to understand SQL Server Encryption before we can hope to retrieve this data in cleartext.

SQL Server Encryption Overview

The root of the cryptography chain in SQL Server is the Service Master Key (SMK). This key is associated and stored in the master database for the server.

At a database layer, the Database Master Key (DMK) is the start of the encryption chain for each database. This diagram from Microsoft gives a brilliant visualisation of this in action:



https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/encryption-hierarchy?view=sql-server-ver16

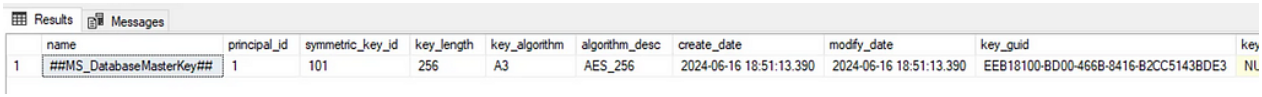
For us to explore this encryption functionality, let's run a few TSQL commands on a lab instance of SQL Server 2019.

First up, we create a new database and master key:

```
USE CryptoDB;
CREATE MASTER KEY ENCRYPTION BY PASSWORD='Password123'
```

We can then view our created master key with:

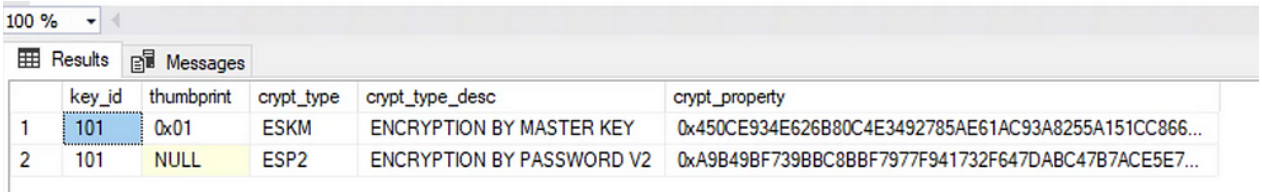
```
SELECT * FROM sys.symmetric_keys
```



	name	principal_id	symmetric_key_id	key_length	key_algorithm	algorithm_desc	create_date	modify_date	key_guid	key
1	##MS_DatabaseMasterKey##	1	101	256	A3	AES_256	2024-06-16 18:51:13.390	2024-06-16 18:51:13.390	EEB18100-BD00-466B-8416-B2CC5143BDE3	NU

Now this doesn't show the actual content of the master key. Instead, to see this, we can use the query to list encryption keys in a database:

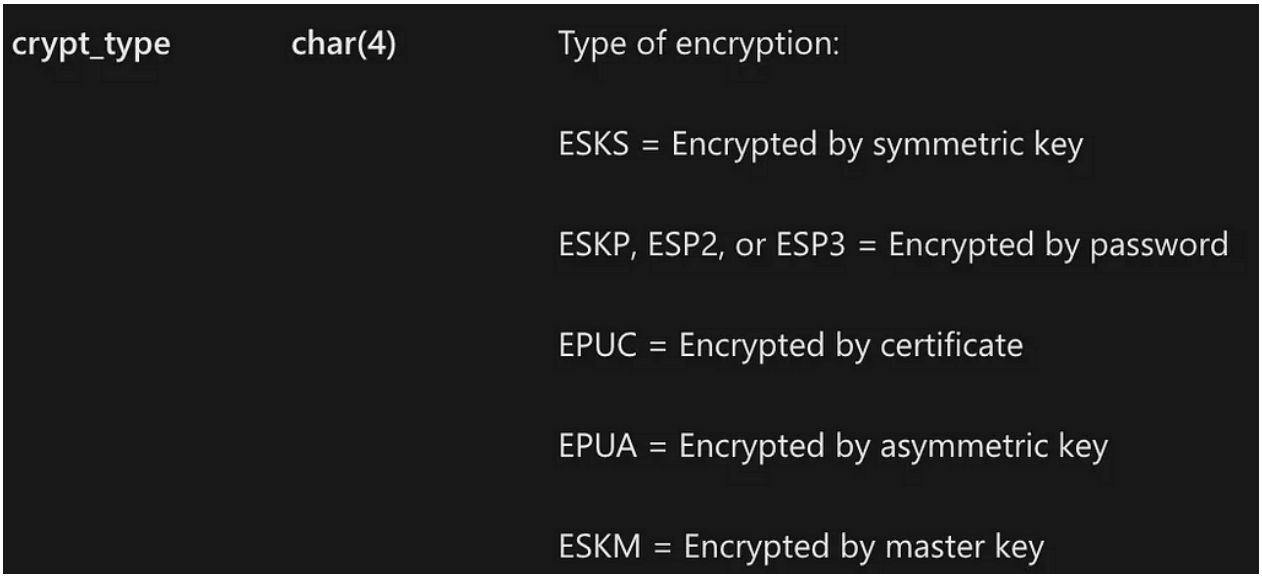
```
SELECT * FROM sys.key_encryptions
```



	key_id	thumbprint	crypt_type	crypt_type_desc	crypt_property
1	101	0x01	ESKM	ENCRYPTION BY MASTER KEY	0x450CE934E626B80C4E3492785AE61AC93A8255A151CC866...
2	101	NULL	ESP2	ENCRYPTION BY PASSWORD V2	0xA9B49BF739BBC8BBF7977F941732F647DABC47B7ACE5E7...

The crypt_property field shows our newly created master key in some form. We can also see that the crypt_type and crypt_type_description fields give a good indication as to each key's type.

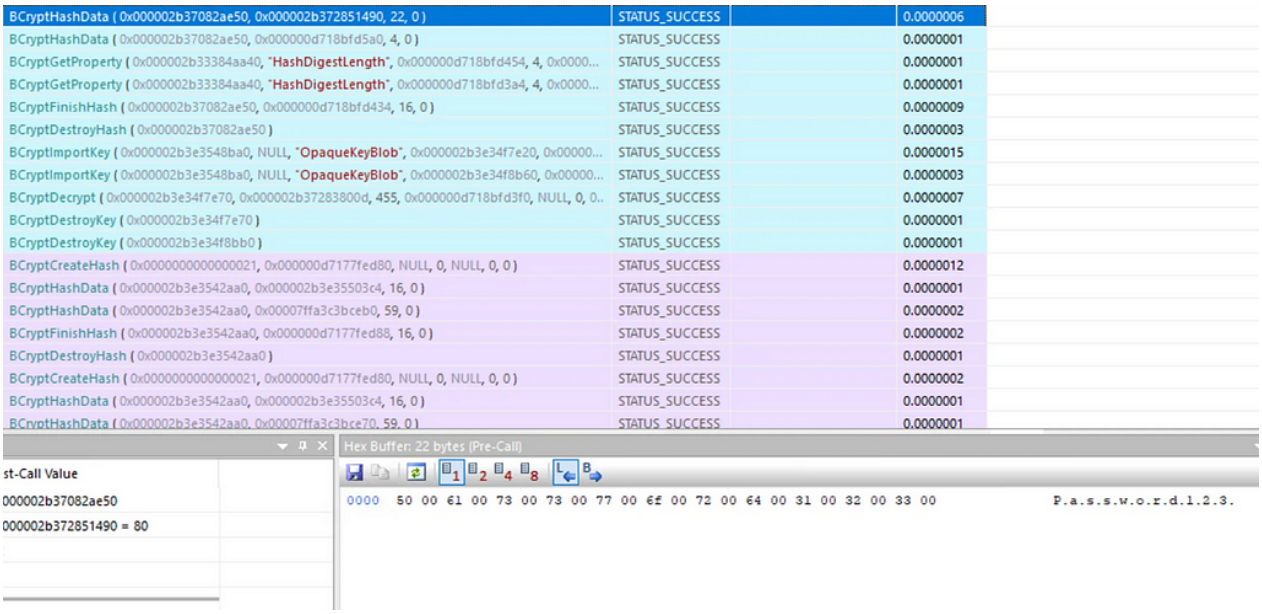
After searching Microsoft's documentation for how these keys are actually stored, or ways that we can extract them, I found a few snippets of information:



Unfortunately none of this is useful for our purpose, so into the disassembler and debugger I needed to go.

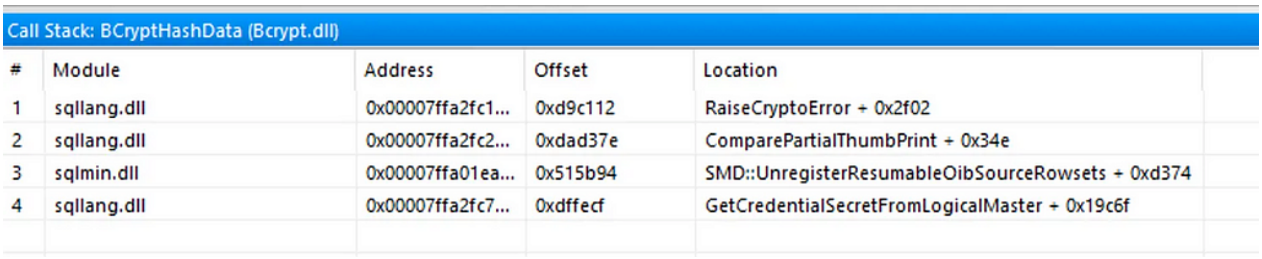
Strap In Peeps.. We're Going Low Level!

For this exercise, it usually makes sense to try and find a good lead as to the APIs that Microsoft SQL Server may use to handle encryption/decryption. My lab ran SQL Server 2017 on Microsoft Windows Server 2019 and installing WinDBG Preview was too much of a pain without access to the Windows Store, so I spun up API Monitor and hooked the Crypto APIs to see if anything indicated their use during cryptographic operations on SQL Server. We execute the TSQL to open the master key and:



As far as indicators go, this was a good one. We see that BCryptHashData was used along with a password provided during the opening of the database master key.

The important part for us is the call stack, which showed sqllang.dll and sqlmin.dll were prime candidates for reversing:



Symbols were available for both of these dynamic-link libraries (DLLs) are grabbed using symchk.exe:

```
PS C:\Program Files (x86)\Windows Kits\10\Debuggers\x64> .\symchk.exe C:\tools\sqlmin.dll /s SRV*c:\symbols\*http://msdl
.microsoft.com/download/symbols

SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
PS C:\Program Files (x86)\Windows Kits\10\Debuggers\x64>
```

Service Master Key Encryption

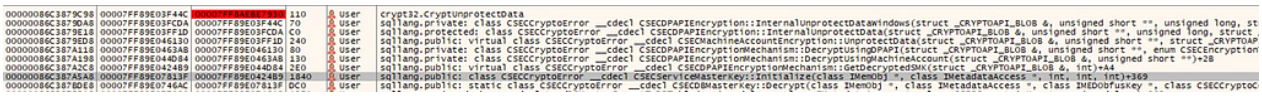
Let's look at how the Service Master Key is generated and stored on SQL Server. This is the root of the encryption chain as shown in Microsoft's diagram, so if we can find a vulnerability here, or some method of cracking this key, everything else will fall!

We know that a Database Master Key is encrypted using the Service Master Key. We also know from Microsoft's documentation that this is likely protected using the data protection APIs (DPAPIs), which means that if we add a breakpoint on CryptUnprotectData / CryptProtectData and create a new DMK, we are in with a shot of seeing where in SQL Server is responsible for using the SMK.

To create the new key we use:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD='Password123'
```

And we hit a breakpoint with a valuable stack trace:



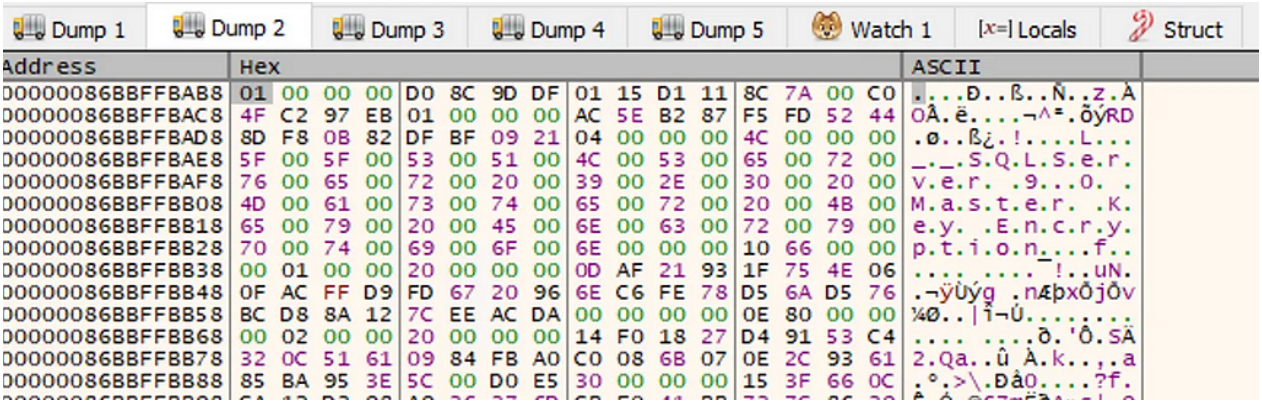
Here we see two method calls which tell us a story:

CSECDBMasterKey::Decrypt

CSECServiceMasterKey::Initialize

This makes sense, because we know that the SMK is used to decrypt the DMK and the DPAPI should protect the SMK.

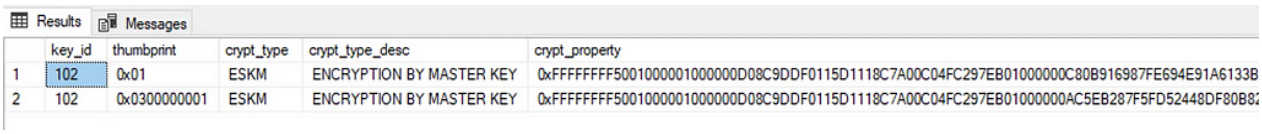
We can pull out the arguments to CryptoUnprotectData and find the following value being decrypted:



And if we use the following TSQL query:

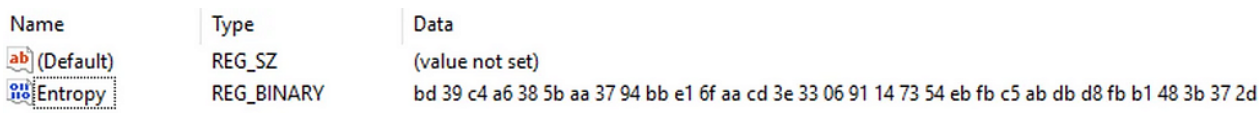
```
SELECT * FROM master.sys.key_encryptions
```

We find that the encrypted SMK matches the encrypted key stored in the master database:



Another caveat is a value passed to CryptUnprotectData as the optional entropy value. After a bit of digging, we find that this value is taken from the registry key:

```
HKLMSOFTWAREMicrosoftMicrosoft SQL ServerMSSQL14.MSSQLSERVERSecurity
```



So what does this mean? Well, if you have execution rights on a machine running SQL Server, we can use the following C# to recover the SMK:

```
using System;
using System.Security.Cryptography;
using Microsoft.Win32;

namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            // Read registry key
            var rk =
Registry.LocalMachine.OpenSubKey(@"SOFTWAREMicrosoftMicrosoft SQL
ServerMSSQL14.MSSQLSERVERSecurity");
            byte[] entropy = (byte[])rk.GetValue("Entropy", new byte[] { 0x41 });
            // SQL Encrypted SMK (minus the first 8 bytes)
            byte[] encryptedData = new byte[]
            {
                0x01, 0x00, 0x00, 0x00, 0xD0, 0x8C, 0x9D, 0xDF, 0x01, 0x15, 0xD1,
0x11, 0x8C, 0x7A, 0x00, 0xC0, 0x4F, 0xC2, 0x97, 0xEB, 0x01, 0x00, 0x00, 0x00,
0xAC, 0x5E, 0xB2, 0x87, 0xF5, ... 0x8E, 0x50, 0x44, 0xFA, 0xDC, 0xBE, 0x47, 0x88,
0x16, 0x57, 0xBF, 0xCB, 0xB3, 0x56, 0x7B, 0x43, 0x86, 0x68, 0x31, 0x7E, 0x30,
0xE3, 0xE4, 0x3A, 0x14, 0xB4
            };
            try
            {
                // Decrypt key
                byte[] data = ProtectedData.Unprotect(encryptedData,
(byte[])entropy, DataProtectionScope.LocalMachine);
               Console.WriteLine("Key Recovered");
            } catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Unfortunately with only the database backup that we hold for ADSelfService, this isn't an option, so we move onto the next crypto layer, the Database Master Key.

Database Master Key Encryption

With DPAPI being used to protect the SMK, next up we tackle the DMK to see what we can unearth here.

We know from our TSQL that when we initialized the DMK, we used a password:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD='Password123'
```

This password is surely a weak link in the chain, but there are a few questions that come up:

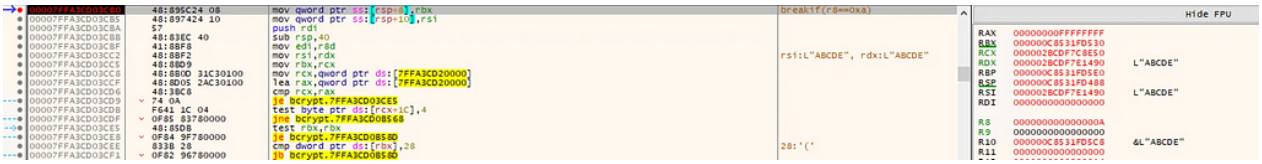
1. How is this password stored in the database?

2. Is all of the keying material for this password stored in a database backup?
3. Can we bruteforce this key?

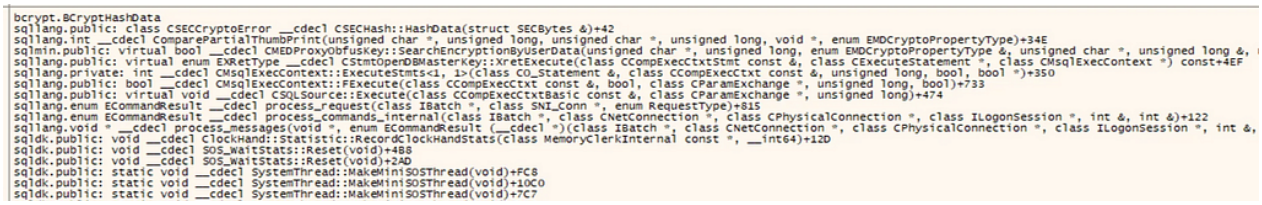
First up, we need to understand how this key is actually stored in the database. We attach a debugger to SQLServr.exeand use a password to attempt to open the DMK:

```
OPEN MASTER KEY DECRYPTION BY PASSWORD='ABCDE'
```

We add breakpoints to the previously observed BCrypt suite of APIs and we find that, after being executed, we land on a method called BCryptHashData:



The call stack shows where this is invoked:



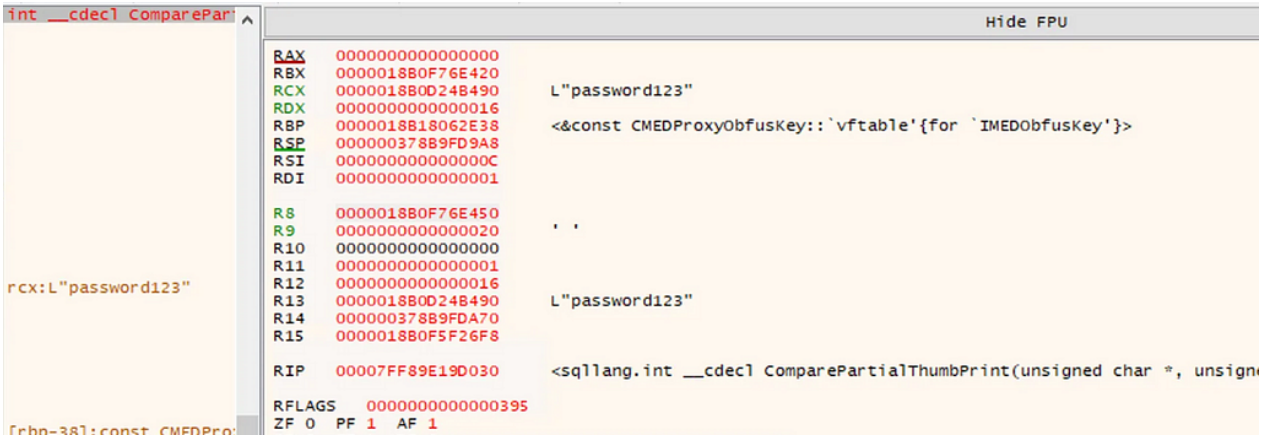
What's interesting is the use of the word Obfus in the method CMEDProxyObfusKey::SearchEncryptionByUserData . Obfuscation usually means something fishy is going on, so we dig into this method a bit more and we find reference to a key thumbprint:

```
EVar2 = *param_3;
EVar3 = *(EMDCryptoPropertyType *)((longlong)ppuVar8 + 0x1c);
if (((EVar2 == EVar3) ||
    (((EVar2 == 1 || (EVar2 + 0xfffffff5 < 2)) && ((EVar3 == 1 || (EVar3 + 0xfffffff5 < 2))))))
    ) && (iVar5 = ComparePartialThumbPrint
                            (param_1,param_2,*ppuVar8,*(ulong *)(ppuVar8 + 1),param_8,EVar3),
          iVar5 != 0)) {
  *param_3 = *(EMDCryptoPropertyType *)((longlong)ppuVar8 + 0x1c);
  if (uVar1 <= uVar9) {
    return false;
```

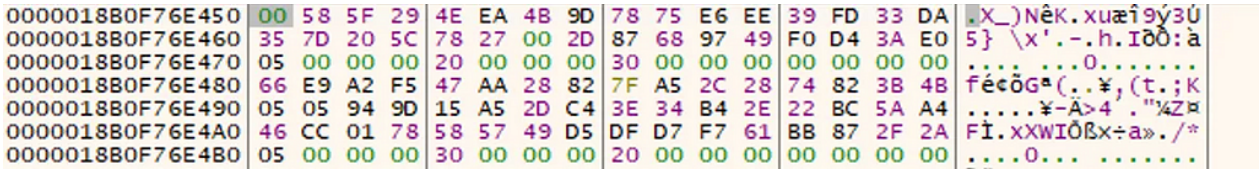
Add a breakpoint to ComparePartialThumbPrint and attempt to open the master key again with an invalid password using:

```
OPEN MASTER KEY DECRYPTION BY PASSWORD='password123'
```

This time we find that our password is passed to this method as an argument, along with the unicode byte length:



But what is this being compared to? Dumping the third argument to this method call shows the following memory content:



This is not something that we've seen so far, but a bit of digging in SQL reveals the following table (requires DAC / diagnostic connection on a live database):

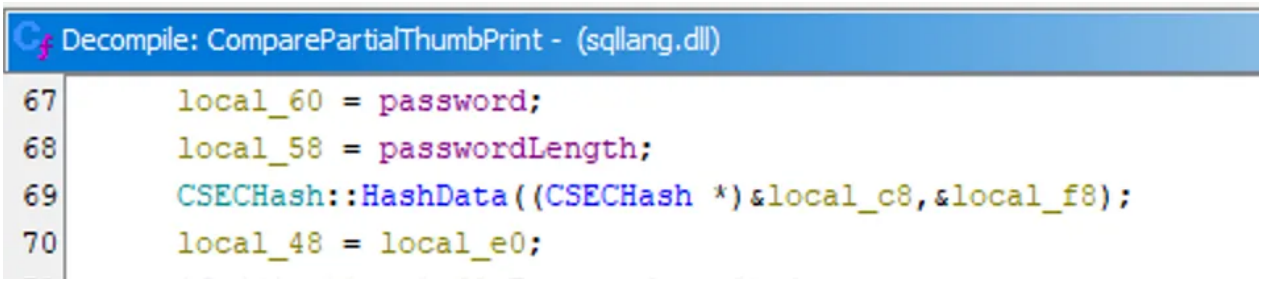
```
SELECT * FROM sys.sysobjkeycrypts
```

	class	id	thumbprint	type	crypto	status
1	24	101	0x00585F294EEA4B9D7875E6EE39FD33DA357D205C7827002D87689749F0D43AE0	ESP2	0x66E9A2F547AA28827FA52C2874823B4B0505949D15A52DC4...	0
2	24	101	0x01	ESKM	0x871127C0F150FB46DAFA94699AEA5AE22F39924AB1E69AE9...	0

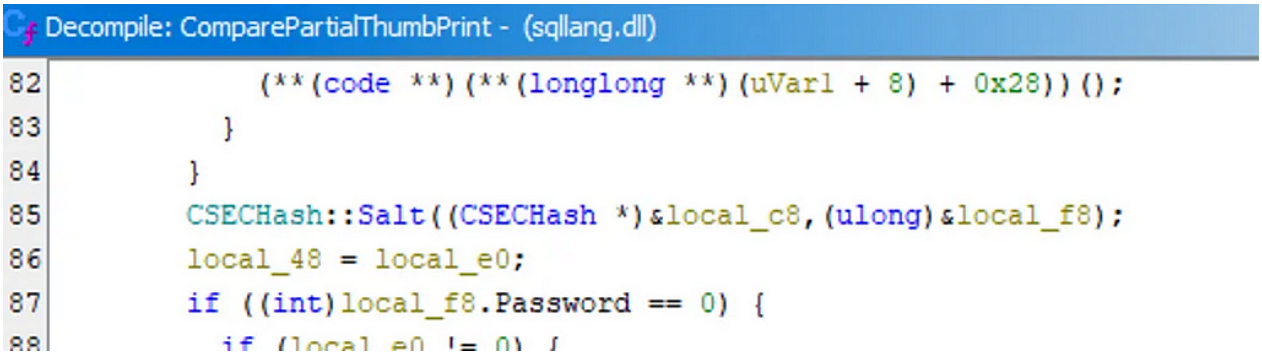
This looks similar to the previous sys.key_encryptions table; however, the thumbprint value is populated this time. What is going on here?

At this point, we know that the thumbprint is being used alongside our plaintext password. Let's look in CompartialThumbPrint to see what the comparison is doing.

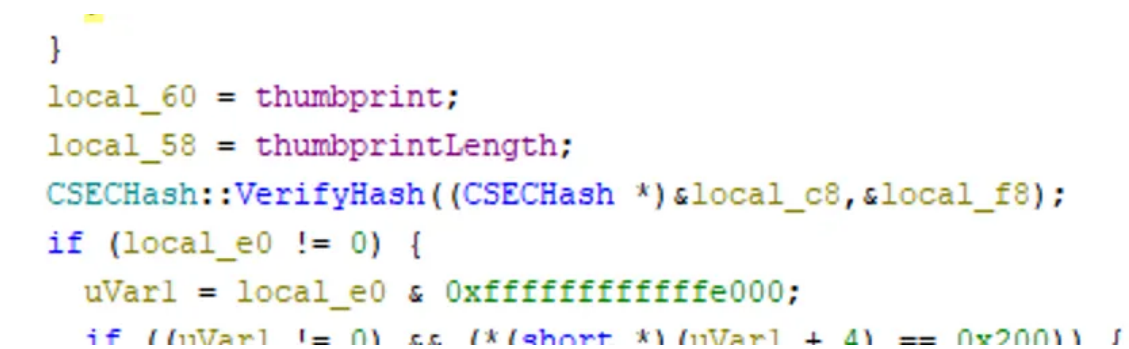
First the provided password is hashed:



Then the hash is salted:



And then the result is compared to the thumbprint:



If this is the case, this gives us a brilliant opportunity to create a brute-force method for our target database. After all:

1. All of the keying material is stored in the database (and therefore the database backup)
2. Nothing relates to the SMK and, therefore, DPAPI

But what are the algorithms used to hash the password? Well, in the type field of sys.key_encryptions we have a number of values:

- ESKP – Observed in databases starting at SQL Server 2008
- ESP2 – Observed in databases starting at SQL Server 2012

Starting with ESP2, if we add a breakpoint to BCryptHashData, we find that this is SHA-512 salted with 8 bytes. The resulting hash is then truncated to 24 bytes and then compared to the thumbprint.

Unfortunately for us, there is an additional step that SQL Server takes when storing the SHA-512 hash of the DMK: the hash truncates to 24 bytes.

This step alone appears to put it out of the reach of stock Hashcat rules; however, if we turn to John The Ripper, we have the option of Dynamic Rules.

A warning in advance: this is going to be slow, but we can add the following dynamic rule which will crack ESP2 keys:

```
[List.Generic:dynamic_2020]
Expression=sha512(utf16le($p).$s) (hash truncated to length 24)
Flag=MGF_SALTED
Flag=MGF_FLAT_BUFFERS
Flag=MGF_INPUT_24_BYTE
SaltLen=8
Func=DynamicFunc__clean_input_kwik
Func=DynamicFunc__setmode_unicode
Func=DynamicFunc__append_keys
Func=DynamicFunc__setmode_normal
Func=DynamicFunc__append_salt
Func=DynamicFunc__SHA512_crypt_input1_to_output1_FINAL
Test=$dynamic_2020$E45AF6FA6601E13A8F2B620FF8A859AE4B459B848D06F5C7$HEX$28E3C09896
```

This dynamic format can then be used with:

```
./run/john --format=dynamic_2020 /tmp/hashes --wordlist=/tmp/wordlist --
encoding=raw
```

A quick demo to show how this works:

The second type is ESKP, which is using MD5 salted with four bytes. The result is then compared to the thumbprint.

Looking at Hashcat, we find a format which suits our cracking format:

```
md5(utf16le($pass).$salt)
```

This means we can crack using:

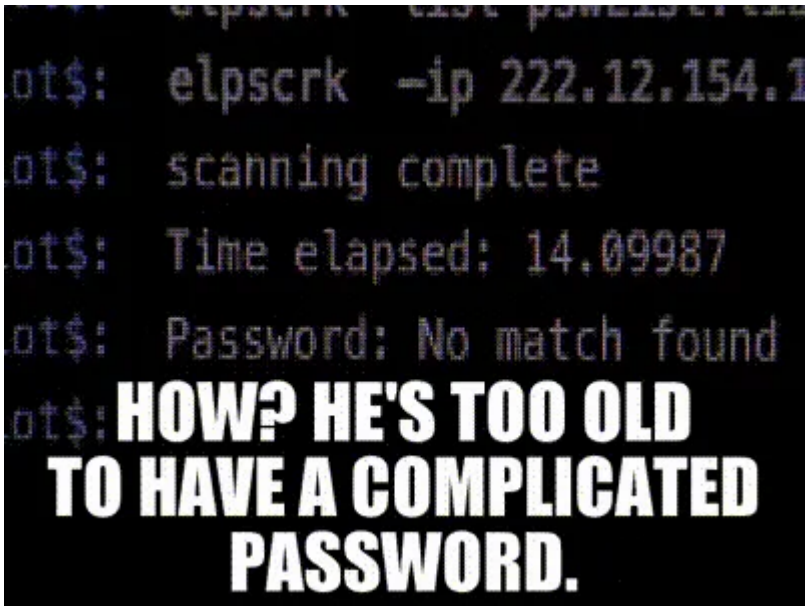
```
hashcat -m 30 --hex-salt /tmp/hashes /tmp/uberwordlist.txt
```

A quick demo to show how this works:

Brute-Forcing the ManageEngine Hashed Database Master Key

So now we have a technique to hopefully recover database encryption keys. We cross our fingers and look in our target database backup and we find ESKP. This means that we have a DMK protected using MD5 and, thankfully, a GPU cracking rig just waiting for us to feed it hashes!

Adding our hash to a file, we fire up our cracking job and…nothing.



The key hasn't been rotated in a long time. Experience tells us that something is wrong here, so I did what I should have done in the first place. I spun up a local instance of ManageEngine to take a look at what was happening.

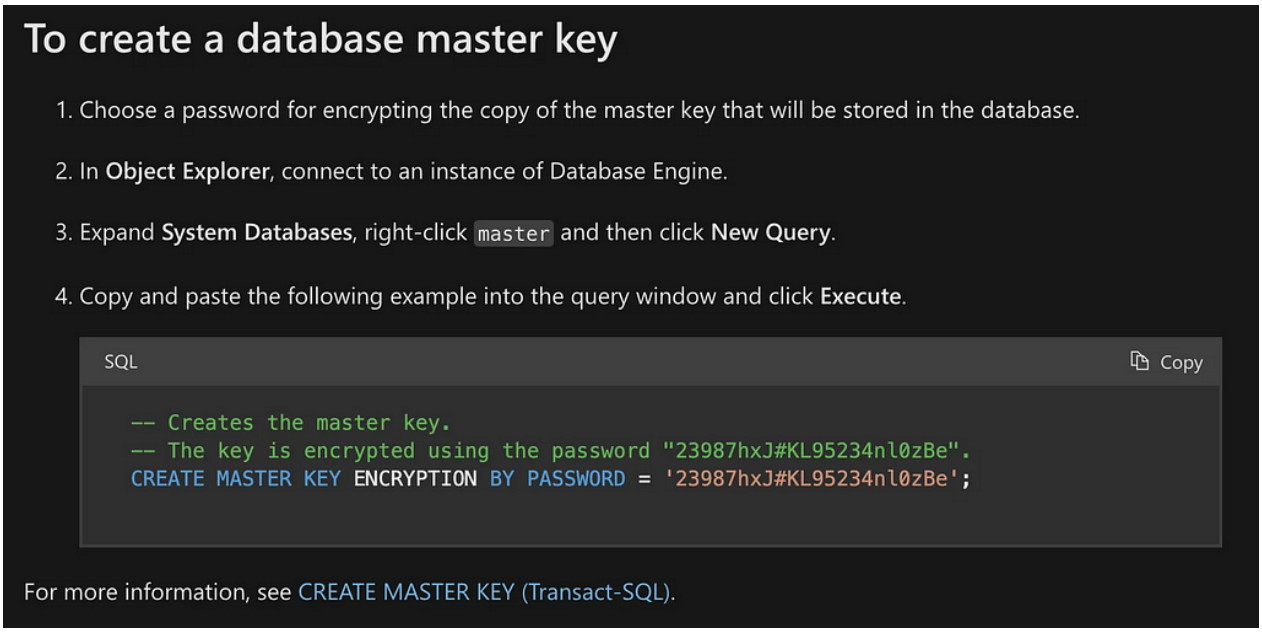
After reviewing, I found a file named *product-config.xml*, which looks like this:

```
<configuration name="mssql" value="">
        <property name="dbadapter" value="com.adventnet.db.adapter.mssql.MssqlDBAdapter"/>
        <property name="sqlgenerator" value="com.adventnet.db.adapter.mssql.MssqlSQLGenerator"/>
        <sql_function_pattern_file value="conf/Persistence/mssql_functionpatterns.txt"/>
        <property name="masterkey.password" value="23987hxJ#KL95234nl0zBe"/>
</configuration>
```

The masterkey.password property has a value of 23987hxJ#KL95234nl0zBe, and if we throw this into our new method of cracking database encryption keys, we find that it cracks.

More concerningly, I then try this against the provided .bak file from the client environment and it cracks!

So what is this key: just a hardcoded value? A quick throw of this password into Google and…



The key is the example key used in Microsoft's documentation for setting up a DMK!

TADA, dopamine hit! Using the database backup, we can now unseal the certificate and symmetric keys ManageEngine uses for decryption and pull out those sensitive credentials:

```
use DATABASE_NAME_HERE -- Update to contain the restored database name
OPEN MASTER KEY DECRYPTION BY PASSWORD = '23987hxJ#KL95234nl0zBe'
OPEN SYMMETRIC KEY ZOHO_SYMM_KEY DECRYPTION BY CERTIFICATE ZOHO_CERT;
```

And we can use this to decrypt any sensitive data contained within:

```
SELECT CONVERT(NVARCHAR(MAX), DecryptByKey((SELECT [Password] FROM
ADSMDomainConfiguration))) as Password, CONVERT(NVARCHAR(MAX),
DecryptByKey((SELECT [USER_NAME] FROM ADSMDomainConfiguration))) as UserName
```

Here's what we've learned during this exercise:

1. ManageEngine ADSelfService backups created use an example key Microsoft provides
2. If you find a database backup that uses a DMK with the ESKP type, you can brute-force the decryption password with the speed of MD5
3. Always lab your target product before spending so much time in a disassembler

This blog post was presented at SOCON 2025. Stay tuned for the video!
