



# The SQL Server Crypto Detour

Banging your head when you don't know what to Google!



Adam Chester

SpecterOps



# About Me

**Name:** Adam Chester

**Alias:** XPN

**Company:** SpecterOps

**Role:** Service Architect



@\_xpn\_



xpnsec.com



in/xpn

# How it all Started

With a Simple Request



# How It All Started

## A New Job and a Red Team

- New role at SpecterOps as Service Architect in April 2024 (today is my anniversary \o/)
- One of the primary goals of my role is to drop into engagements and provide additional support, wherever needed
- Received a request from a Red Team to look at a database backup recovered for ManageEngine's ADSelfService product
- Database contained a lot of interesting information, however the interesting values were encrypted
- The request was simple.. "can you recover the encrypted data"?



# How It All Started

## What and Who is ManageEngine?

- A Zoho Company
- Provides a suite of tools to manage Active Directory
- Databases typically contain DA credentials 😊
- Can use different DB engines, but in our case, MSSQL Server was in use

### Active Directory management



#### ADManager Plus

Active Directory, Microsoft 365, and Exchange management and reporting

On-premises | MSP



#### ADAudit Plus

Real-time Active Directory, file, and Windows server change auditing

On-premises



#### ADSelfService Plus

Identity security with adaptive MFA, SSPR, and SSO

On-premises



#### Exchange Reporter Plus

Reporting, auditing, and monitoring for hybrid Exchange and Skype

On-premises



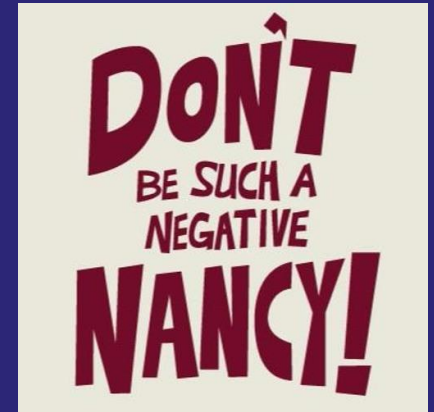
#### RecoveryManager Plus

Active Directory, Microsoft 365, and Exchange backup and recovery

On-premises

# How It All Started

## What Was In The Backup?



- The issue presented was a MSSQL Server .bak file using MSSQL Server Database Encryption
- Recovery was a long shot, and some of the initial ideas I had were:
  - MSSQL crypto is likely linked to the DPAPI master key of the service account running SQL Server, so this backup won't be useful
  - MSSQL crypto may be linked to other keying material on the SQL Server, so backup won't be useful
  - It's unlikely that we will be able to recover any interesting data without access to SQL Server, so let's at least prove this for future reference
- Basically, I was being a Negative Nancy!



# SQL Server Encryption Overview

Sounds Boring.. Because it is!



# SQL Server Encryption Overview

If It Sounds Boring.. That's Because It Is!

- SQL Server encryption is used to protect data within the database
- It does this transparently to the developer
- Once the database is unsealed, queries can be made and the crypto is handled by SQL Server
- Once the database is sealed (or a backup is made), the data is encrypted
- This is “Transparent Data Encryption” (TDE)

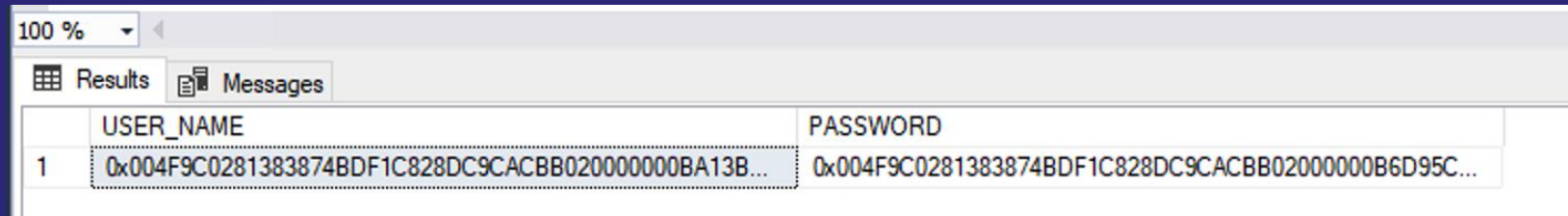




# SQL Server Encryption Overview

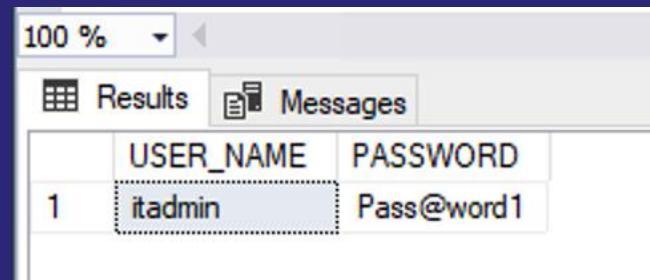
If It Sounds Boring.. That's Because It Is!

- Data encrypted in a table looks like this:



	USER_NAME	PASSWORD
1	0x004F9C0281383874BDF1C828DC9CACBB02000000BA13B...	0x004F9C0281383874BDF1C828DC9CACBB02000000B6D95C...

- Once unsealed, we see the decrypted values like this:

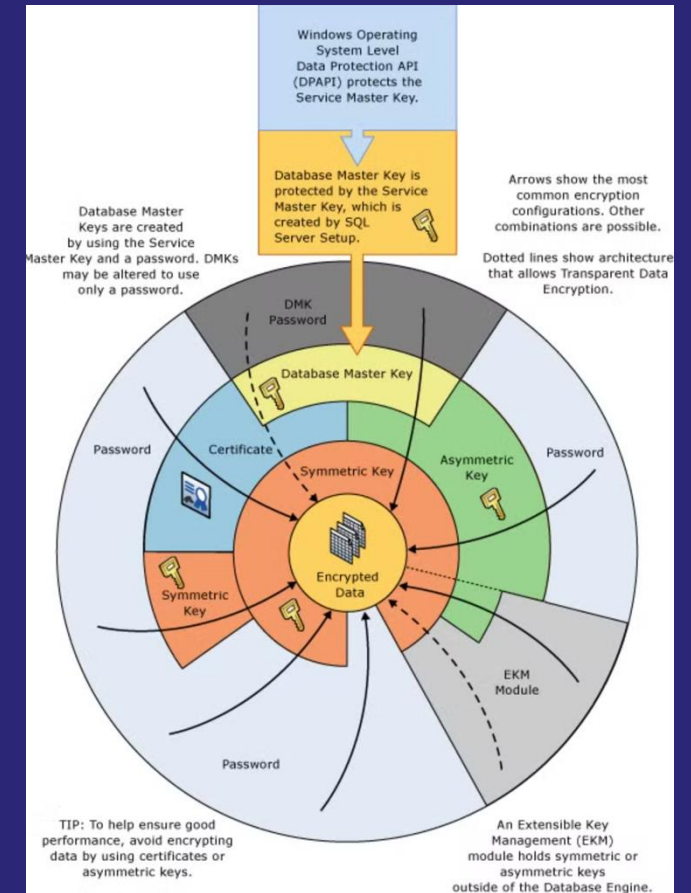


	USER_NAME	PASSWORD
1	itadmin	Pass@word1

# SQL Server Encryption Overview

## Visualisation

- All starts with a Service Master Key (SMK) which is encrypted using DPAPI. The SMK is created during install



# Visualisation

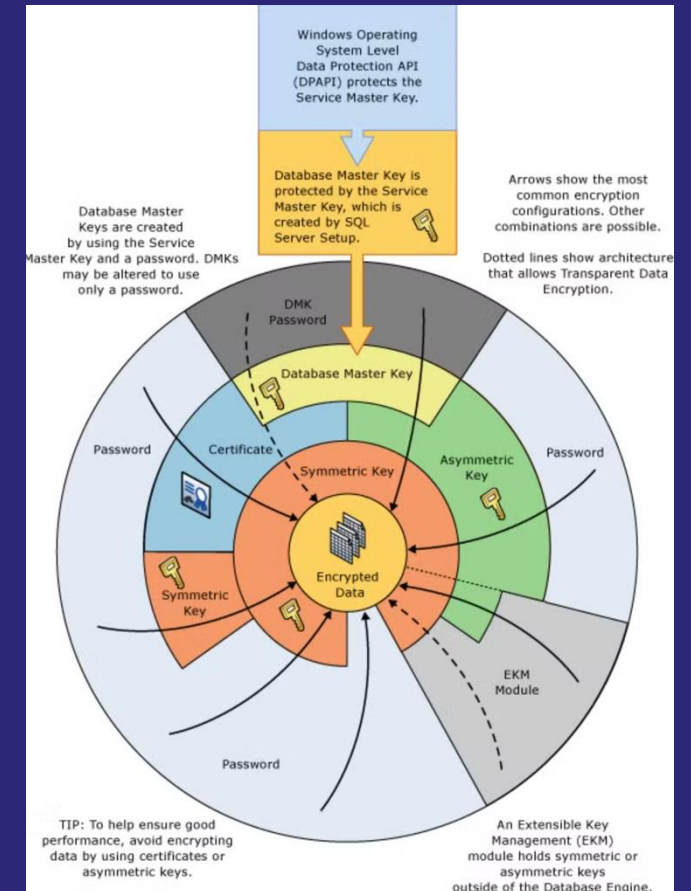
- 



# SQL Server Encryption Overview

## Visualisation

- All starts with a Service Master Key (SMK) which is encrypted using DPAPI. The SMK is created during install
- A Database Master Key (DMK) is encrypted by the SMK
- A Certificate or Async Key is encrypted by the DMK



# Visualisation

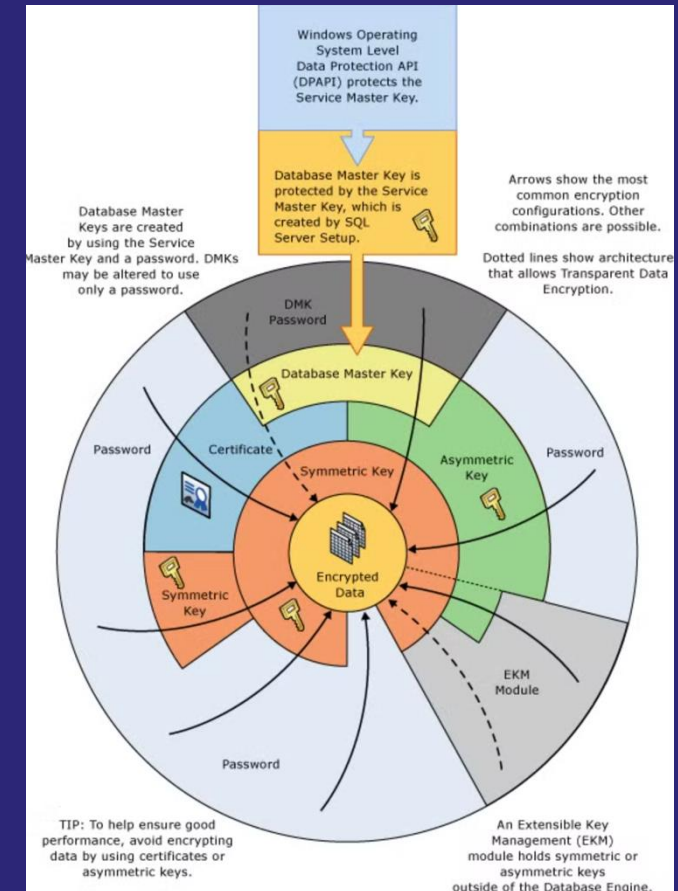
- 
- Windows Operating System Level Data Protection API (DPAPI) protects the Service Master Key.
- Database Master Key is protected by the Service Master Key, which is created by SQL Server Setup.
- Database Master Keys are created by using the Service Master Key and a password. DMKs may be altered to use only a password.
- Arrows show the most common encryption configurations. Other combinations are possible.
- Dotted lines show architecture that allows Transparent Data Encryption.
- TIP: To help ensure good performance, avoid encrypting data by using certificates or asymmetric keys.
- An Extensible Key Management (EKM) module holds symmetric or asymmetric keys outside of the Database Engine.



# SQL Server Encryption Overview

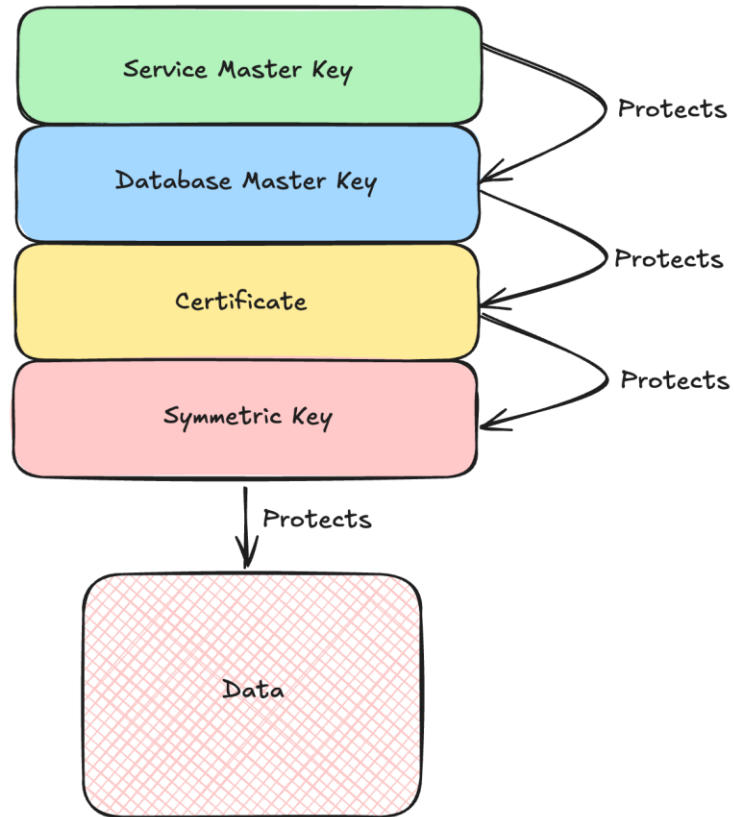
## Visualisation

- All starts with a Service Master Key (SMK) which is encrypted using DPAPI. The SMK is created during install
- A Database Master Key (DMK) is encrypted by the SMK
- A Certificate or Async Key is encrypted by the DMK
- The symmetric key is encrypted by the Certificate or Async Key
- Symmetric key used to encrypt data



# SQL Server Encryption Overview

## Breaking The Crypto Stack



If we can compromise any layer of the SQL Crypto stack, each layer below will fall.

# SQL Server Encryption Overview

## TSQL for TDE

- To start using SQL Server Encryption, we first create the Database Master Key, which is encrypted with the SMK:

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

- The generated Database Master Key row can be viewed using:

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



# SQL Server Encryption Overview

## TDE Tables

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

- Unfortunately, you can't find the generated key value in the table:

Results Messages										
	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

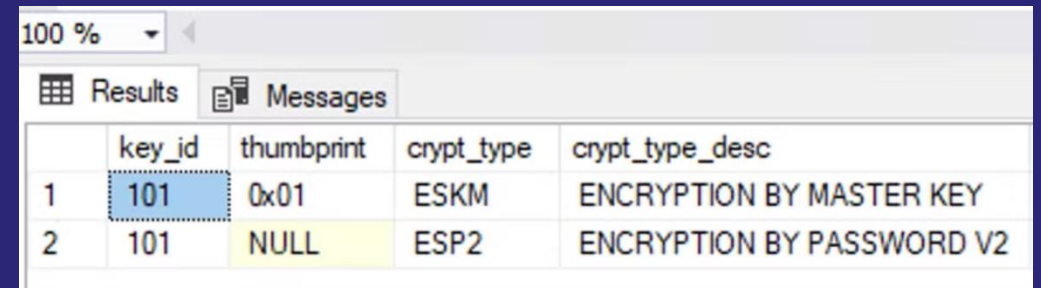
- Instead, it is found in `sys.key_encryptions` within `crypt_property` (in some blob form)

100 % Results Messages					
	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...

# SQL Server Encryption Overview

## TDE Tables

- `crypt_type` and `crypt_type_desc` fields in `sys.key_encryptions` look interesting
- Microsoft provide a table describing each `crypt_type`, but it's vague



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

crypt_type	char(4)	Type of encryption:
		ESKS = Encrypted by symmetric key
		ESKP, ESP2, or ESP3 = Encrypted by password
		EPUC = Encrypted by certificate
		EPUA = Encrypted by asymmetric key
		ESKM = Encrypted by master key



Fire Up The Debugger H4xx0rz... We're  
Going Low Level



# Low Level Analysis

- As we know, the best method of analysis is to set up a lab
- Lab setup consisted of:
  - SQL Server 2008
  - SQL Server 2019
- API Monitor
- x64Dbg (WinDBG was being a pain to install from the store)



# Low Level Analysis

## APIMonitor

BCryptHashData Used

Password Parameter  
Shown

Call stack shows sqlang.dll  
is the caller of Bcrypt API

The screenshot displays the APIMonitor tool interface. The main window shows a list of API calls with columns for the function name, parameters, status, and a numeric value. A green arrow points from the 'BCryptHashData Used' text to the 'BCryptHashData' entry in the list. Another green arrow points from the 'Password Parameter Shown' text to the 'Hex Buffer: 22 bytes (Pre-Call)' section, which displays the password 'P.a.s.s.w.o.r.d.1.2.3.' in hexadecimal. A third green arrow points from the 'Call stack shows sqlang.dll is the caller of Bcrypt API' text to the 'Call Stack: BCryptHashData (Bcrypt.dll)' window, which shows a list of modules and their addresses.

Function	Parameters	Status	Value
BCryptCreateHash	( 0x000002b33384aa40, 0x000000d718bfd568, 0x000002b37082ae50, 294, NULL, ... )	STATUS_SUCCESS	0.0000005
BCryptHashData	( 0x000002b37082ae50, 0x000002b372851490, 22, 0 )	STATUS_SUCCESS	0.0000006
BCryptHashData	( 0x000002b37082ae50, 0x000000d718bfd5a0, 4, 0 )	STATUS_SUCCESS	0.0000001
BCryptGetProperty	( 0x000002b33384aa40, "HashDigestLength", 0x000000d718bfd454, 4, 0x0000... )	STATUS_SUCCESS	0.0000001
BCryptGetProperty	( 0x000002b33384aa40, "HashDigestLength", 0x000000d718bfd3a4, 4, 0x0000... )	STATUS_SUCCESS	0.0000001
BCryptFinishHash	( 0x000002b37082ae50, 0x000000d718bfd434, 16, 0 )	STATUS_SUCCESS	0.0000009
BCryptDestroyHash	( 0x000002b37082ae50 )	STATUS_SUCCESS	0.0000003
BCryptImportKey	( 0x000002b3e3548ba0, NULL, "OpaqueKeyBlob", 0x000002b3e34f7e20, 0x0000... )	STATUS_SUCCESS	0.0000015
BCryptImportKey	( 0x000002b3e3548ba0, NULL, "OpaqueKeyBlob", 0x000002b3e34f8b60, 0x0000... )	STATUS_SUCCESS	0.0000003
BCryptDecrypt	( 0x000002b3e34f7e70, 0x000002b37283800d, 455, 0x000000d718bfd3f0, NULL, 0, 0... )	STATUS_SUCCESS	0.0000007
BCryptDestroyKey	( 0x000002b3e34f7e70 )	STATUS_SUCCESS	0.0000001
BCryptDestroyKey	( 0x000002b3e34f8bb0 )	STATUS_SUCCESS	0.0000001
BCryptCreateHash	( 0x0000000000000021, 0x000000d7177fed80, NULL, 0, NULL, 0, 0 )	STATUS_SUCCESS	0.0000012
BCryptHashData	( 0x000002b3e3542aa0, 0x000002b3e35503c4, 16, 0 )	STATUS_SUCCESS	0.0000001
BCryptHashData	( 0x000002b3e3542aa0, 0x00007ffa3c3bce00, 59, 0 )	STATUS_SUCCESS	0.0000002
BCryptFinishHash	( 0x000002b3e3542aa0, 0x000000d7177fed88, 16, 0 )	STATUS_SUCCESS	0.0000002
BCryptDestroyHash	( 0x000002b3e3542aa0 )	STATUS_SUCCESS	0.0000001
BCryptCreateHash	( 0x0000000000000021, 0x000000d7177fed80, NULL, 0, NULL, 0, 0 )	STATUS_SUCCESS	0.0000002
BCryptHashData	( 0x000002b3e3542aa0, 0x000002b3e35503c4, 16, 0 )	STATUS_SUCCESS	0.0000001
BCryptHashData	( 0x000002b3e3542aa0, 0x00007ffa3c3bce00, 59, 0 )	STATUS_SUCCESS	0.0000001

Post-Call Value: 0x000002b37082ae50

Hex Buffer: 22 bytes (Pre-Call)

0000 50 00 61 00 73 00 73 00 77 00 6f 00 72 00 64 00 31 00 32 00 33 00 P.a.s.s.w.o.r.d.1.2.3.

#	Module	Address	Offset	Location
1	sqlang.dll	0x00007ffa2fc1...	0xd9c112	RaiseCryptoError + 0x2f02
2	sqlang.dll	0x00007ffa2fc2...	0xdad37e	ComparePartialThumbPrint + 0x34e
3	sqlmin.dll	0x00007ffa01ea...	0x515b94	SMD::UnregisterResumableOibSourceRowsets + 0xd374
4	sqlang.dll	0x00007ffa2fc7...	0xdffcfc	GetCredentialSecretFromLogicalMaster + 0x19c6f

# Low Level Analysis

## sqlang.dll

- Call stack shows **sqlang.dll** uses Bcrypt APIs when calling encryption

Call Stack: BCryptHashData (Bcrypt.dll)				
#	Module	Address	Offset	Location
1	sqlang.dll	0x00007ffa2fc1...	0xd9c112	RaiseCryptoError + 0x2f02
2	sqlang.dll	0x00007ffa2fc2...	0xdad37e	ComparePartialThumbPrint + 0x34e
3	sqlmin.dll	0x00007ffa01ea...	0x515b94	SMD::UnregisterResumableOibSourceRowsets + 0xd374
4	sqlang.dll	0x00007ffa2fc7...	0xdffcf	GetCredentialSecretFromLogicalMaster + 0x19c6f

- And symbols are available from Microsoft

```
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> |
```

# Low Level Analysis

## Follow The Data

- Our plan to dig further becomes:
  1. Add a bunch of breakpoints to Crypt API's
  2. Create a new database master key
  3. Hopefully break on a crypto API call
  4. Review the encrypted data





# Low Level Analysis

## Follow The Data

- We hit a breakpoint on `CryptUnprotectData`

```
crypt32.CryptUnprotectData
sqlang.private: class CSECCryptoError __cdecl CSECDPAPIEncryption::InternalUnprotectDataWindows(struct _CRYPTOAPI_BLOB &, unsigned short **, unsigned long, st
sqlang.protected: class CSECCryptoError __cdecl CSECDPAPIEncryption::InternalUnprotectData(struct _CRYPTOAPI_BLOB &, unsigned short **, unsigned long, struct
sqlang.public: virtual class CSECCryptoError __cdecl CSECMachineAccountEncryption::UnprotectData(struct _CRYPTOAPI_BLOB &, unsigned short **, struct _CRYPTOAP
sqlang.private: class CSECCryptoError __cdecl CSECDPAPIEncryptionMechanism::DecryptUsingDPAPI(struct _CRYPTOAPI_BLOB &, unsigned short **, enum CSECEncryption
sqlang.private: class CSECCryptoError __cdecl CSECDPAPIEncryptionMechanism::DecryptUsingMachineAccount(struct _CRYPTOAPI_BLOB &, unsigned short **, enum CSECEncryption
sqlang.public: virtual class CSECCryptoError __cdecl CSECDPAPIEncryptionMechanism::GetDecryptedSMK(struct _CRYPTOAPI_BLOB &, int)+A4
sqlang.public: class CSECCryptoError __cdecl CSECSERVICEMasterKey::Initialize(class IMemObj *, class IMetadataAccess *, int, int, int)+369
sqlang.public: static class CSECCryptoError __cdecl CSECDBMasterKey::Decrypt(class IMemObj *, class IMetadataAccess *, class IMEDObfusKey *, class CSECCryptoc
```

- The stack trace tells a story:
  - `CSECDBMasterKey::Decrypt`
  - `CSECSERVICEMasterKey::Initialize`
- We know that the Service Master Key (SMK) protects the DB Master Key (DMK), and the stack trace reflects this too

# Low Level Analysis

## Follow The Data

- Analysis of the data passed to `CryptUnprotectData` matches the newly created key value in the `master.sys.key_encryptions` table
- The call also includes optional entropy, which we find in the registry:
  - `HKLM\SOFTWARE\Microsoft\Microsoft SQL Server\MSSQL14.MSSQLSERVER\Security`

Name	Type	Data
 (Default)	REG_SZ	(value not set)
 Entropy	REG_BINARY	bd 39 c4 a6 38 5b aa 37 94 bb e1 6f aa cd 3e 33 06 91 14 73 54 eb fb c5 ab db d8 fb b1 48 3b 37 2d

# Low Level Analysis

## Recovering the SMK

- This means that if we have execution rights on a machine running SQL Server, we can recover the Service Master Key using something like:

```
// Read registry key Entropy Value
var rk = Registry.LocalMachine.OpenSubKey(@"SOFTWARE\Microsoft\Microsoft SQL Server\MSSQL14.MSSQLSERVER\Security");
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, ...
};

// Decrypt key
byte[] data = ProtectedData.Unprotect(encryptedData, (byte[])entropy, DataProtectionScope.LocalMachine);
```

# Low Level Analysis

## But That Doesn't Help!

- Unfortunately, this doesn't help with our original quest of recovering encrypted data from a SQL Server backup.
- For this we need the clear-text database master key



# Low Level Analysis

## But That Doesn't Help!

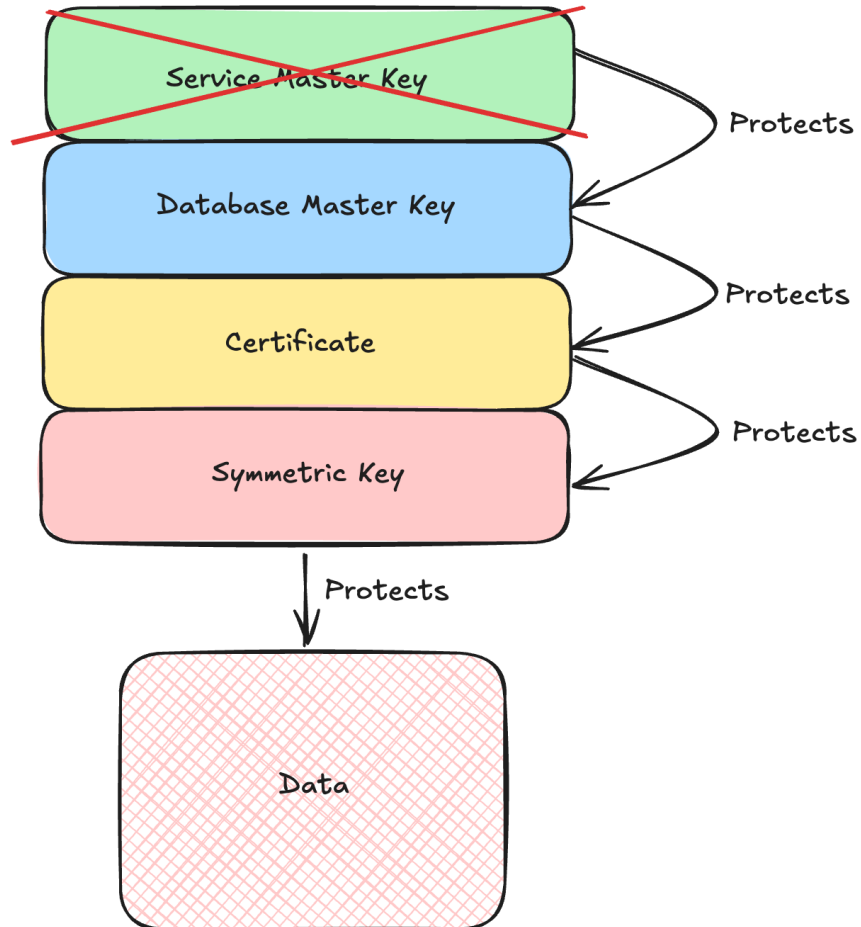
- Unfortunately, this doesn't help with our original quest of recovering encrypted data from a SQL Server backup.
- For this we need the clear-text database master key
- Which we don't have 😞





# Low Level Analysis

## Recovering the SMK



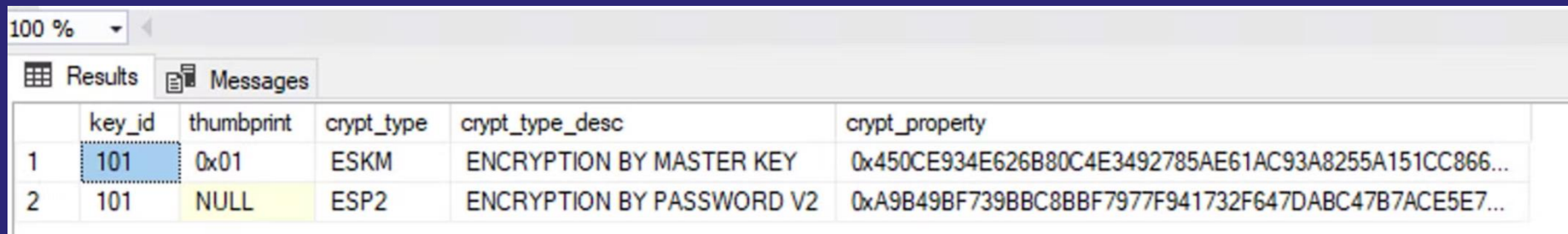
We can rule out the SMK from our quest as this only lives on the MSSQL server instance, not in the database backup

# Low Level Analysis

But Wait...

- But wait...
- Remember when we initialized the database master key...
- What was that password for?

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



A screenshot of the SQL Server Enterprise Manager interface. The 'Results' tab is active, displaying a table with the following data:

	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...

# Low Level Analysis

## Some More Questions

- Some more questions:
  - How is this password tied to the Service Master Key?
  - Is the password ever stored in the database?
  - Is all the keying material for this password stored in a database backup?
  - Can we somehow bruteforce this key?
- Let's find out



# Low Level Analysis

## Finding Out

- So, we add another breakpoint to **BCryptHashData** and recreate the APIMonitor hook we observed before
- This breaks along with the entered password:

Address	Disassembly	Comment
00007FFA3CD03CB0	48:895C24 08	mov qword ptr ss:[rsp+8],rbx
00007FFA3CD03CB5	48:897424 10	mov qword ptr ss:[rsp+10],rsi
00007FFA3CD03CBA	57	push rdi
00007FFA3CD03CB8	48:83EC 40	sub rsp,40
00007FFA3CD03CBF	41:8BF8	mov edi,r8d
00007FFA3CD03CC2	48:8BF2	mov rsi,rdx
00007FFA3CD03CC5	48:8BD9	mov rbx,rcx
00007FFA3CD03CC8	48:8B0D 31C30100	mov rcx,qword ptr ds:[7FFA3CD20000]
00007FFA3CD03CCF	48:8D05 2AC30100	lea rax,qword ptr ds:[7FFA3CD20000]
00007FFA3CD03CD6	48:3BC8	cmp rcx,rax
00007FFA3CD03CD9	74 0A	je bcrypt.7FFA3CD03CE5
00007FFA3CD03CD8	F641 1C 04	test byte ptr ds:[rcx+1C],4
00007FFA3CD03CDF	0F85 83780000	jne bcrypt.7FFA3CD0B568
00007FFA3CD03CE5	48:85DB	test rbx,rbx
00007FFA3CD03CE8	0F84 9F780000	je bcrypt.7FFA3CD0B58D
00007FFA3CD03CEE	833B 28	cmp dword ptr ds:[rbx],28
00007FFA3CD03CF1	0F82 96780000	jb bcrypt.7FFA3CD0B58D

Register	Value	Comment
RAX	00000000FFFFFFFF	
RBX	000000C8531FD530	
RCX	000002BCDF7C8E50	
RDY	000002BCDF7E1490	L"ABCDE"
RBP	000000C8531FD5E0	
RSP	000000C8531FD488	
RSI	000002BCDF7E1490	L"ABCDE"
RDI	0000000000000000	
R8	000000000000000A	
R9	0000000000000000	
R10	000000C8531FD5C8	&L"ABCDE"
R11	0000000000000000	

# Low Level Analysis

## Finding Out

- The stack trace reveals an interesting method name:
  - CMEDProxyObfusKey::SearchEncryptionByUserData

```
bcrypt.BCryptHashData
sqlang.public: class CSECCryptoError __cdecl CSECHash::HashData(struct SECBytes &)+42
sqlang.int __cdecl ComparePartialThumbPrint(unsigned char *, unsigned long, unsigned char *, unsigned long, void *, enum EMDCryptoPropertyType)+34E
sqlmin.public: virtual bool __cdecl CMEDProxyObfusKey::SearchEncryptionByUserData(unsigned char *, unsigned long, enum EMDCryptoPropertyType &, unsigned char *, unsigned long &,
sqlang.public: virtual enum EXRetType __cdecl CStmtOpenDBMasterKey::XretExecute(class CCompExecCtxtStmt const &, class CExecuteStatement *, class CMsglExecContext *) const+4EF
sqlang.private: int __cdecl CMsglExecContext::ExecuteStmts<1, 1>(class CO_Statement &, class CCompExecCtxt const &, unsigned long, bool, bool *)+350
sqlang.public: bool __cdecl CMsglExecContext::FExecute(class CCompExecCtxt const &, bool, class CParamExchange *, unsigned long, bool)+733
sqlang.public: virtual void __cdecl CSQLSource::Execute(class CCompExecCtxtBasic const &, class CParamExchange *, unsigned long)+474
sqlang.enum ECommandResult __cdecl process_request(class IBatch *, class SNI_Conn *, enum RequestType)+815
sqlang.enum ECommandResult __cdecl process_commands_internal(class IBatch *, class CNetConnection *, class CPhysicalConnection *, class ILogonSession *, int &, int &)+122
```

- “Obfus” within a class name is always a good path to follow!

# Low Level Analysis

## Disassembling SearchEncryptionByUserData

- We start the disassembly of `SearchEncryptionByUserData`
- Within this method we find a comparison of a “thumbprint”:

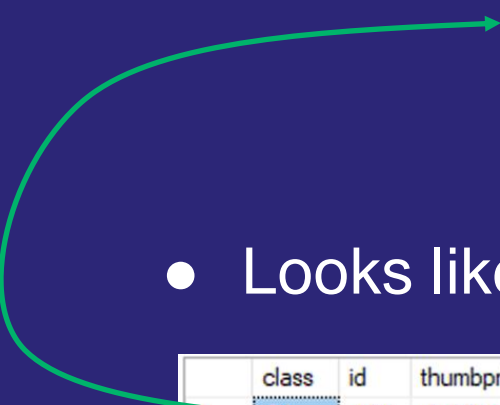
```
if (((EVar2 == EVar3) ||  
    (((EVar2 == 1 || (EVar2 + 0xffffffff5 < 2)) && ((EVar3 == 1 || (EVar3 + 0xffffffff5 < 2))))  
    ) && (iVar5 = ComparePartialThumbPrint  
        (param_1,param_2,*ppuVar8,* (ulong *) (ppuVar8 + 1),param_8,EVar3),  
        iVar5 != 0)) {  
    *param_3 = *(EMDCryptoPropertyType *) ((longlong)ppuVar8 + 0x1c);  
    if (uVar1 <= uVar9) {
```

- Another breakpoint later reveals that the third value passed to this function is a blob of data we haven't seen so far...

# Low Level Analysis

## What is this thumbprint?

- This is a hex encoded value we find in the database `sys.sysobjkeycrypts` (require DAC connection to access this table)



```
00 58 5F 29 4E EA 4B 9D 78 75 E6 EE 39 FD 33 DA 5X_)NèK.xuæ19y3U
35 7D 20 5C 78 27 00 2D 87 68 97 49 F0 D4 3A E0 5} \x'-.h.Iðð:à
05 00 00 00 20 00 00 00 30 00 00 00 00 00 00 00 .....0.....
66 E9 A2 F5 47 AA 28 82 7F A5 2C 28 74 82 3B 4B fécôG^(..¥,(t.;K
05 05 94 9D 15 A5 2D C4 3E 34 B4 2E 22 BC 5A A4 .....¥-Ä>4."¼Z¤
46 CC 01 78 58 57 49 D5 DF D7 F7 61 BB 87 2F 2A Fİ.xXWIOßx÷a»./÷
05 00 00 00 30 00 00 00 20 00 00 00 00 00 00 00 .....0.....
```

- Looks like the `sys.key_encryptions` table but with thumbprint populated:

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



# Low Level Analysis

## ComparePartialThumbPrint

1. The decryption password is hashed

```
Decompile: ComparePartialThumbPrint - (sqlang.dll)
67     local_60 = password;
68     local_58 = passwordLength;
69     CSECHash::HashData((CSECHash *) &local_c8, &local_f8);
70     local_48 = local_e0;
```

# Low Level Analysis

## ComparePartialThumbPrint

1. The decryption password is hashed

```
Decompile: ComparePartialThumbPrint - (sqlang.dll)
67     local_60 = password;
68     local_58 = passwordLength;
69     CSECHash::HashData((CSECHash *) &local_c8, &local_f8);
70     local_48 = local_e0;
```

2. The password hash is salted

```
Decompile: ComparePartialThumbPrint - (sqlang.dll)
82     (**(code **)) (**(longlong **) (uVar1 + 8) + 0x28) ();
83     }
84     }
85     CSECHash::Salt((CSECHash *) &local_c8, (ulong) &local_f8);
86     local_48 = local_e0;
87     if ((int) local_f8.Password == 0) {
```

# Low Level Analysis

## ComparePartialThumbPrint

1. The decryption password is hashed

```
Decompile: ComparePartialThumbPrint - (sqlang.dll)
67     local_60 = password;
68     local_58 = passwordLength;
69     CSECHash::HashData((CSECHash *) &local_c8, &local_f8);
70     local_48 = local_e0;
```

2. The password hash is salted

```
Decompile: ComparePartialThumbPrint - (sqlang.dll)
82     (**(code **)) (**(longlong **) (uVar1 + 8) + 0x28) ();
83     }
84     }
85     CSECHash::Salt((CSECHash *) &local_c8, (ulong) &local_f8);
86     local_48 = local_e0;
87     if ((int) local_f8.Password == 0) {
```

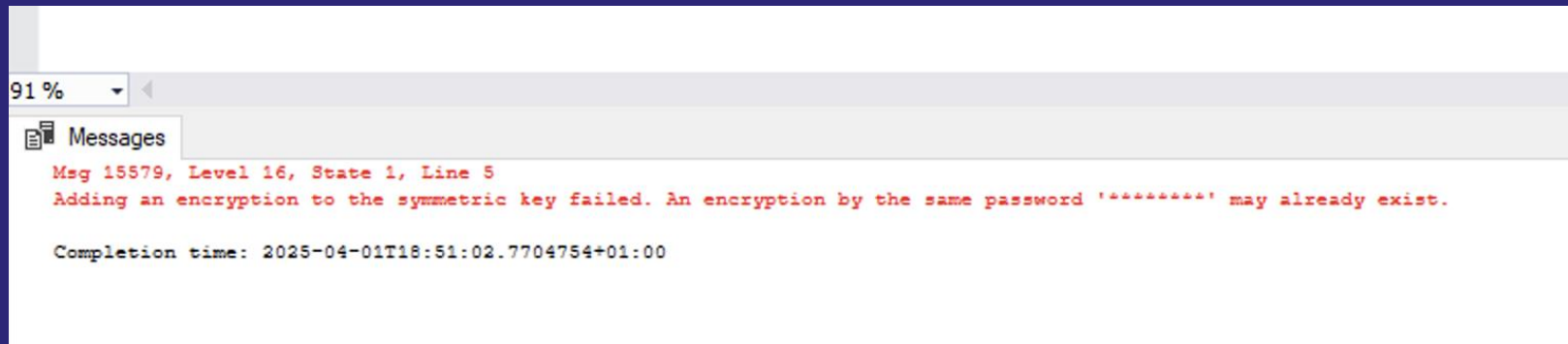
3. The hashed and salted password is compared against the DB thumbprint

```
local_60 = thumbprint;
local_58 = thumbprintLength;
CSECHash::VerifyHash((CSECHash *) &local_c8, &local_f8);
if (local_e0 != 0) {
    uVar1 = local_e0 & 0xffffffffffffffe000;
```

# Low Level Analysis

## Why does this exist?

- Validates if same password is in use
- Adding new symmetric key with same password fails:



A screenshot of a terminal window with a white background and a grey title bar. The title bar shows a zoom level of '91 %'. Below the title bar is a tab labeled 'Messages'. The terminal content is as follows:

```
Msg 15579, Level 16, State 1, Line 5
Adding an encryption to the symmetric key failed. An encryption by the same password '*****' may already exist.

Completion time: 2025-04-01T18:51:02.7704754+01:00
```

# Low Level Analysis

## ComparePartialThumbPrint

- If our hypothesis is correct, this means:
  - All the keying material is in the database (and therefore the backup)
  - Nothing in the thumbprint ties the DMK to the SMK, therefore DPAPI isn't a factor
- But what is the algo used to hash the password?
  - Depends on the version of SQL Server ORIGINALLY used to create the DMK



# Low Level Analysis

## Crypto Versions

- ESP2 – Observed with SQL Server 2012
- ESKP – Observed with SQL Server 2008
- With breakpoints added to `BCryptHashData`, we find that:
  - ESP2:
    - SHA-512 Thumbprint
    - Salted with 8 bytes
    - Result truncated to 24 bytes



# Low Level Analysis

## Bruteforce

- ESP2 needs a custom format due to truncation, but we can do it easily in JtR:

```
[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$28E3C098$
```

26939B2ACE091AC197AC7616A7275C9C46D6793F0DE8F1C77E6B5D473B526E51

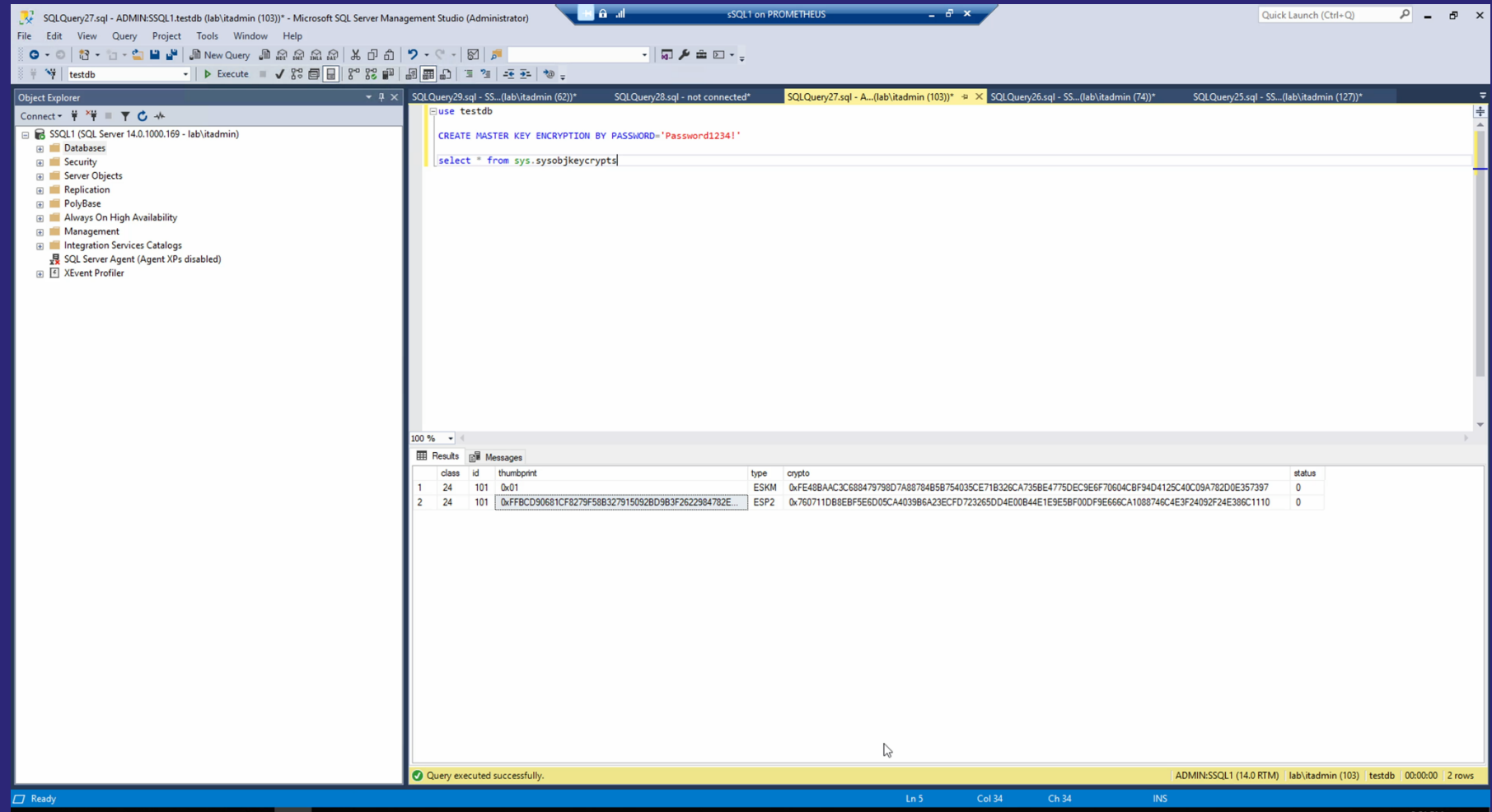
└──────────────────┬──┘

Salt Hash



# Low Level Analysis

## Demo



The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The left pane shows the 'Object Explorer' with the 'testdb' database selected. The right pane shows the 'Query Editor' with the following SQL script:

```
use testdb
CREATE MASTER KEY ENCRYPTION BY PASSWORD='Password1234!'
select * from sys.sysobjkeycrypts
```

The bottom pane shows the 'Results' tab with the following data:

class	id	thumbprint	type	crypto	status
1	24	101	ESKM	0xFE48BAAC3C688479798D7A88784B5B754035CE71B326CA7358E4775DEC9E6F70604CBF94D4125C40C09A782D0E357397	0
2	24	101	ESP2	0xFFBCD90681CF8279F58B327915092BD9B3F2622984782E...	0

The status bar at the bottom indicates 'Query executed successfully.' and 'ADMIN-SSQL1 (14.0 RTM) | lab\itadmin (103) | testdb | 00:00:00 | 2 rows'.

# Low Level Analysis

## Crypto Versions

- ESP2 – Observed with SQL Server 2012
- ESKP – Observed with SQL Server 2008
- By breakpoints added to `BCryptHashData`, we find that:
  - ESP2:
    - SHA-512 Thumbprint
    - Salted with 8 bytes
    - Result truncated to 24 bytes



# Low Level Analysis

## Crypto Versions

- ESP2 – Observed with SQL Server 2012
- ESKP – Observed with SQL Server 2008
- By breakpoints added to **BCryptHashData**, we find that:
  - ESP2:
    - SHA-512 Thumbprint
    - Salted with 8 bytes
    - Result truncated to 24 bytes
  - **ESKP:**
    - **MD5 Thumbprint**
    - **Salted with 4 bytes**



# Low Level Analysis

## Bruteforce

- We can bruteforce ESKP thumbprints with Hashcat:
  - `md5(utf16le($pass).$salt)`

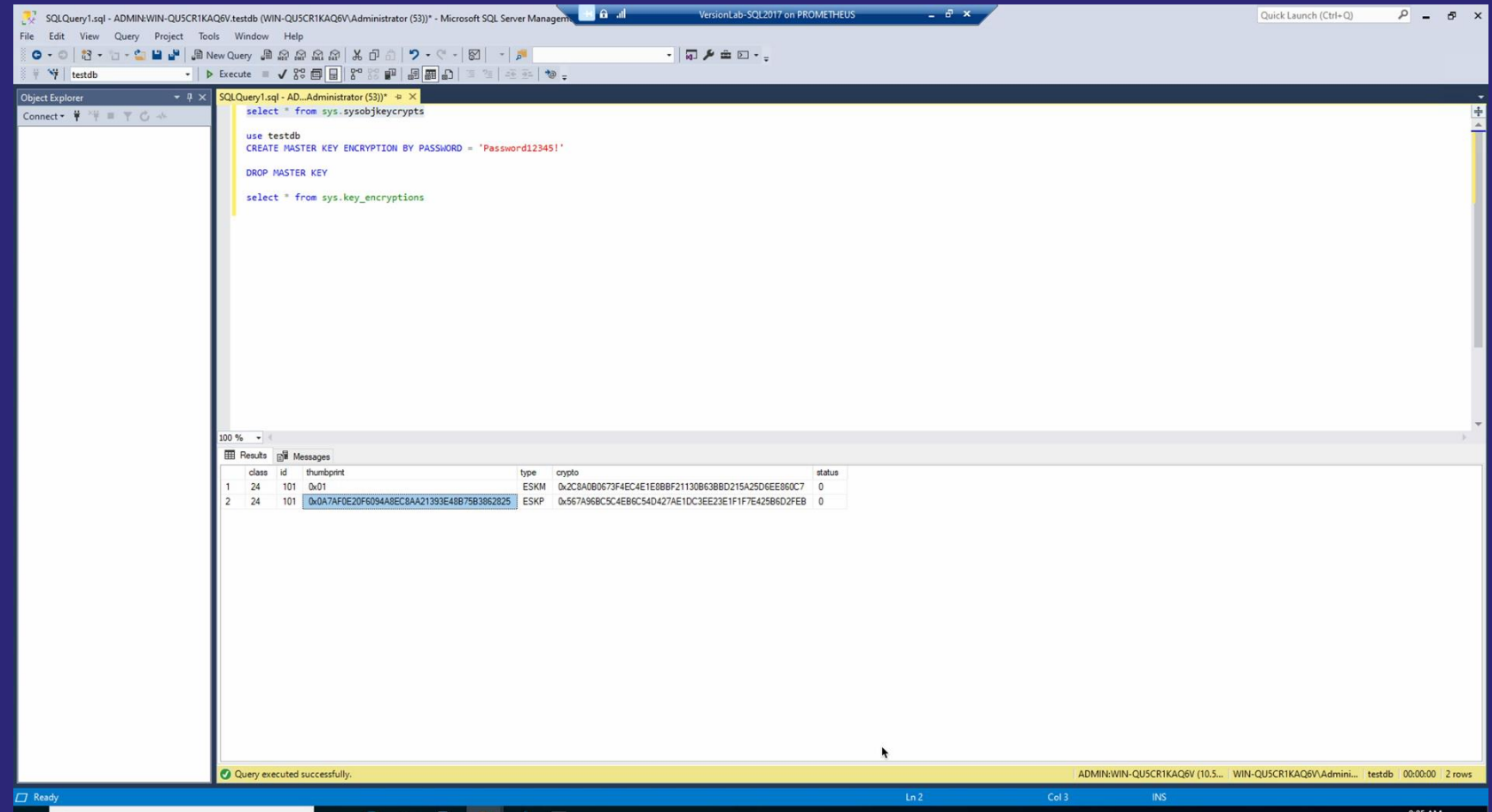
1B0F82784EF7E28054ECB0AE166EAC5D73A608E8

└──────────┴────────────────────────────────┘

Salt Hash

# Low Level Analysis

## Demo



The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The main window shows a query executed in the 'testdb' database. The query is as follows:

```
select * from sys.sysobjkeycrypts

use testdb
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Password12345!'

DROP MASTER KEY

select * from sys.key_encryptions
```

The results pane at the bottom shows the output of the final query, displaying two rows of encryption data:

class	id	thumbprint	type	crypto	status	
1	24	101	0x01	ESKM	0x2C8A0B0673F4EC4E1E8B8F21130B638BD215A25D6EE86C7	0
2	24	101	0x0A7AF0E20F6094A8EC8AA21393E48B75B3862825	ESKP	0x567A968C5C4E86C54D427AE10C3EE23E1F1F7E425B6D2FEB	0

The status bar at the bottom indicates that the query was executed successfully.

# Low Level Analysis

## Crypto Versions

- So what? Nobody uses SQL Server 2008 anymore?!
- Well:
  - Of course they do!
  - But even if they all upgraded:
    - Keys amended are appended (hey, that rhymes), leaving old ESKP (MD5) keys in the database
    - Upgrading SQL Server doesn't rotate the encryption key



# Low Level Analysis

## Crypto Versions

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Password123456!'  
ALTER MASTER KEY ADD ENCRYPTION BY PASSWORD='Wibble11111!'
```

100 %

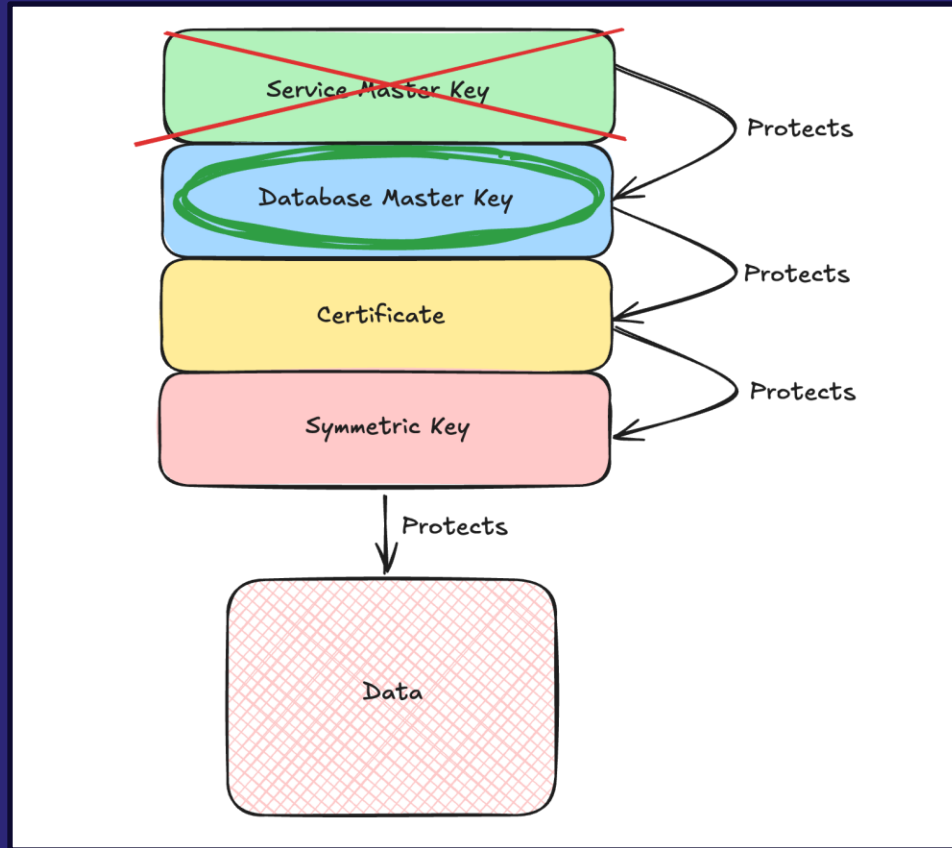
Results Messages

	class	id	thumbprint	type	crypto	status
1	24	101	0x01	ESKM	0x51316AB020D68A8C8EBF675E638C2EAC5138C0146C0E5F9A	0
2	24	101	0x757A5C21D9CE6DC2B84F48EF40D450961E094CB5	ESKP	0xCD61A83664617185819AD8AE5B50FCB36B5FC250A84FD54B	0
3	24	101	0xECD270F950E0C92F9DC1986144D48A455513AD0D	ESKP	0x165837289ED77D1D6B9C97C77AFF2DE013699707D17B2100	0



# Low Level Analysis

## Broken Chain



Database Master Key compromised, so all the other keys fall

# Detour Over

Back To ManageEngine



# ManageEngine Database

## Back to it

- Now that we know how MSSQL crypto works, we can try and bruteforce the key in our original ManageEngine database
- We hoped that it's ESKP format.. And it was! MD5 cracking is a GO!
- The key hadn't been rotated in a looooong time, so we were in with a good shot of cracking it!



# ManageEngine Database

## What's the Craic?

- And it fails to crack 😞
- Experience tells us something is wrong, the password should have cracked!
- So, we have to reverse engineer ManageEngine
- It's Java, so it doesn't take long

```
ot$: elpscrk -ip 222.12.154.1
ot$: scanning complete
ot$: Time elapsed: 14.09987
ot$: Password: No match found
ot$: HOW? HE'S TOO OLD
TO HAVE A COMPLICATED
PASSWORD.
```

# ManageEngine Database

## What the...?!

- In the config file `product-config.xml` we see this in our lab deployment:

```
<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#KL95234n10zBe"/>
</configuration>
```

- Looks random, but we throw that into our password list and... the .bak key CRACKS!

```
Approaching final keyspace - workload adjusted.
4ef7e28054ecb0ae166eac5d73a608e8:1b0f8278:23987hxJ#KL95234n10zBe
Session.....: hashcat
Status.....: Cracked
```

- How? Is this a hardcoded database master key?

# ManageEngine Database

I've seen that password before...

- Googling with the password, we find a few hits, including, Microsoft's documentation!

## To create a database master key

1. Choose a password for encrypting the copy of the master key that will be stored in the database.
2. In Object Explorer, connect to an instance of Database Engine.
3. Expand **System Databases**, right-click **master** and then click **New Query**.
4. Copy and paste the following example into the query window and click **Execute**.

SQL

Copy

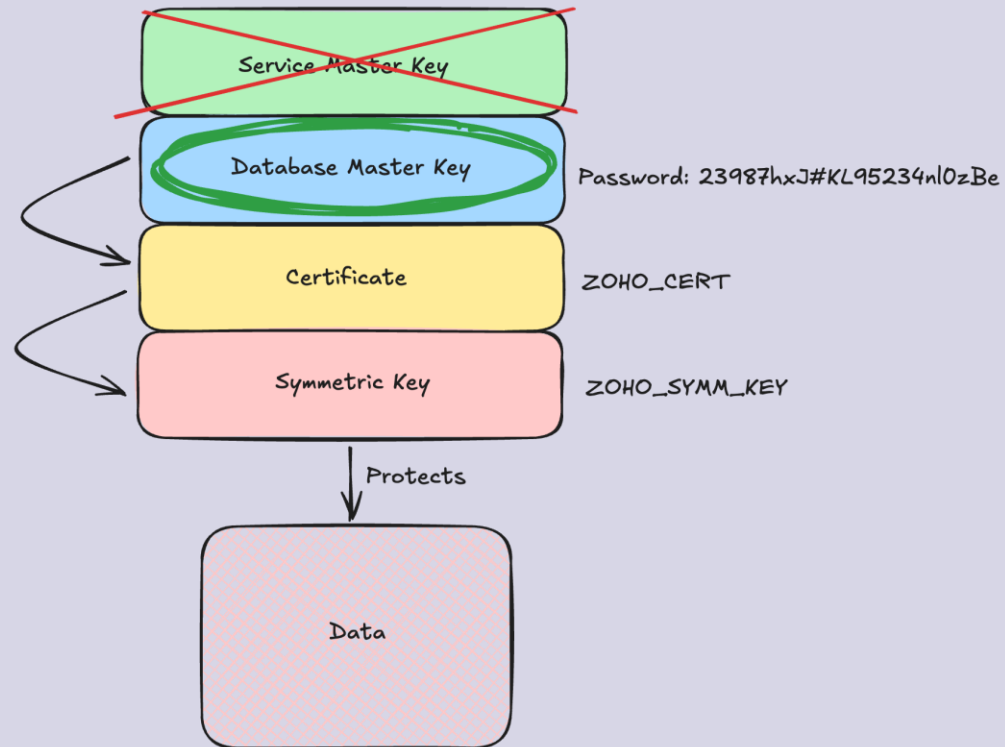
```
-- Creates the master key.  
-- The key is encrypted using the password "23987hxJ#KL95234n!0zBe".  
CREATE MASTER KEY ENCRYPTION BY PASSWORD = '23987hxJ#KL95234n!0zBe';
```

For more information, see [CREATE MASTER KEY \(Transact-SQL\)](#).



# ManageEngine Database

The chain falls...





# ManageEngine Database

- So, the key to ManageEngine ADSelfService product is.. The stock documentation key from the Microsoft website!
- Hopefully, nothing too important is stored...

- 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](#).

# ManageEngine Database

- So, the key to ManageEngine ADSelfService products is.. The stock documentation key from the Microsoft website!
- Hopefully, nothing too important is stored...

```
OPEN MASTER KEY DECRYPTION BY PASSWORD='23987hxJ#KL95234n10zBe'  
OPEN SYMMETRIC KEY ZOHO_SYMM_KEY DECRYPTION BY CERTIFICATE ZOHO_CERT  
SELECT CONVERT(NVARCHAR, DECRYPTBYKEY(USER_NAME)), CONVERT(NVARCHAR, DECRYPTBYKEY(PASSWORD)), * FROM ADSMDomainConfiguration
```

	(No column name)	(No column name)	DOMAIN_NAME	DOMAIN_DNS_NAME	DOMAIN_FLAT_NAME	DOMAIN_CONTROLLER_ID	USER_NAME
1	itadmin	Pass@word1	lab.local	lab.local	lab	2	0x004F9C0281383874BDF1C828D0

# Takeaways

- ManageEngine ADSelfService hardcoded password for the DMK is `23987hxJ#KL95234nl0zBe`
- If you find a MSSQL .bak which uses encryption, check out the `sys.sysobjkeycrypts` table and pray for a type of ESKP which is MD5
- Google is your friend.. If you know what to look for ;)



# Fixes

- Good password hygiene for database crypto (hopefully this talk shows the “why” which is half the battle sometimes)
- Rotate any ADSelfService encryption keys and product-config.xml
- Closely guard any backups from any RedTeamers who come ‘a snoopin!







# Thank you

Any Questions?

Adam Chester | [achester@specterops.com](mailto:achester@specterops.com)

