Active Directory Friday:

All Articles

Jaap Brasser

Content

This ebook is created as a result of a blog post series on my blog www.jaapbrasser.com. The Active Directory Friday series in which common and some uncommon scenarios are discussed. I decided to bundle these tips and to make them available to the wider technical community. Feel free to reach out if you find any errors in this document or if you have suggestions or feedback in regards to this resource.

- Jaap Brasser

Creating Active Directory groups using PowerShell

Creating a group in Active Directory using PowerShell is relatively simple when using the Active Directory module. To create a Global Distribution Group the following code can be executed:

```
New-ADGroup -Name NewGlobalDG_1 -GroupScope Global -GroupCategory
Distribution
```

When creating a Domain Local Security Group the GroupScope can be changed to DomainLocal and GroupCategory can be omitted since the default is a Security Group:

```
New-ADGroup -Name NewDLSG_1 -GroupScope DomainLocal
```

Creating groups using the [adsi] provider is a three step process. First we bind to the OU in which the group should be created. Secondly we enter the name and the properties of the group that should be created. And by finally calling the SetInfo() method to create the group. The following code will create a group:

```
$TargetOU = [adsi]'LDAP://OU=Groups,DC=jaapbrasser,DC=com'
$Group = $TargetOU.Create('group','cn=System_Operators')
$Group.put('grouptype',0x80000004)
$Group.put('samaccountname','System_Operators')
$Group.SetInfo()
```

To specify the Group Type a hexadecimal value is required as specified in the following MSDN article: 2.2.12 Group Type Flags. The following table lists all the possible values:

Symbolic name	Value
GROUP_TYPE_BUILTIN_LOCAL_GROUP	0x00000001
GROUP_TYPE_ACCOUNT_GROUP	0x00000002
GROUP_TYPE_RESOURCE_GROUP	0x00000004
GROUP_TYPE_UNIVERSAL_GROUP	0x00000008

GROUP_TYPE_APP_BASIC_GROUP	0x00000010
GROUP_TYPE_APP_QUERY_GROUP	0x00000020
GROUP_TYPE_SECURITY_ENABLED	0x80000000

It is important to note that only four values are relevant to us when creating Active Directory accounts:

- GROUP_TYPE_ACCOUNT_GROUP – 0x00000002
- GROUP_TYPE_RESOURCE_GROUP – 0x00000004
- GROUP_TYPE_UNIVERSAL_GROUP – 0x00000008
- GROUP_TYPE_SECURITY_ENABLED – 0x80000000

To simplify the creation of groups these values can be place in a hashtable:

```
$GroupType = @{
    DomainLocal = 0x00000002
    Global = 0x00000004
    Universal = 0x00000008
    Security =  0x80000000
}
```

Using the values stored in the hash table it is now possible to create any of the three group scopes as either a distribution group or security group. The following example uses the -bor operator to combine the values to create a Universal Security Group:

```
$TargetOU = [adsi]'LDAP://OU=Groups,DC=jaapbrasser,DC=com'
$Group = $TargetOU.Create('group','cn=Universal_Operators')
$Group.put('grouptype',($GroupType.Universal -bor $GroupType.Security))
$Group.put('samaccountname','Universal_Operators')
$Group.SetInfo()
```

That is all there is to it, using this methodology it is possible to create any type of Active Directory group using either the Active Directory module or the [adsi] type accelerator. Below I have included some links in regards to this topic.

Creating Active Directory Groups

New-ADGroup

Understanding Groups

2.2.12 Group Type Flags

Determine the forest functional level

Knowing the Forest Functional Level can be important when implementing new products or when considering to upgrade your functional level. This information can be view in the 'Active Directory Domains and Trusts' console, but for the purpose of this article we will take a look how this information can be retrieved programmatically, or to be more specific: How to retrieve this using PowerShell.

In the following example we use the Get-ADForest cmdlet to Retrieve information about the current forest. In particular the property we are interested in is the ForestMode property:

```
Get-ADForest | Select-Object ForestMode
```

Alternatively the [adsi] type accelerator can be used, this has the advantage that it works on any computer that has PowerShell installed as it does not rely on having the Active Directory module installed, the following code will retrieve the Forest Functional level:

```
([adsi]"LDAP://CN=Partitions,$(([adsi](“LDAP://RootDSE”)).`
configurationNamingContext)").'msDS-Behavior-Version'
```

The problem with this is that the value of the Forest Functional Level is stored as an integer. Luckily for us this integer can be found on MSDN. So by combining the previous command with a switch statement we can get the expected output:

```
switch (([adsi]"LDAP://CN=Partitions,$(([adsi](“LDAP://RootDSE”)).`
configurationNamingContext)").'msDS-Behavior-Version') {
    0 {'DS_BEHAVIOR_WIN2000'}
    1 {'DS_BEHAVIOR_WIN2003_WITH_MIXED_DOMAINS'}
    2 {'DS_BEHAVIOR_WIN2003'}
    3 {'DS_BEHAVIOR_WIN2008'}
    4 {'DS_BEHAVIOR_WIN2008R2'}
    5 {'DS_BEHAVIOR_WIN2012'}
    6 {'DS_BEHAVIOR_WIN2012R2'}
```

```
}
```

For more information about the Forest Functional Level I have included a TechNet article that goes in depth about the implications of the various forest and domain functional levels. For more information about the msDS-Behavior-Version attribute I have included the link to the MSDN entry.

Forest Functional Level

Understanding Active Directory Domain Services (AD DS) Functional Levels

msDS-Behavior-Version: Forest Functional Level

Find empty Organizational Unit

As an Active Directory Administrator there are some moments, few and far in between where you might have a moment to yourself. In this article I will give you a short line of code so you can use this moment to find out if you have any empty Organizational Units in your domain. The definition of empty is an OU that does not contain any child objects. By this definition an OU containing another OU would not be considered empty. Because there is no LDAP filter for this we will take a look at how to do this using the Cmdlets and the [adsisearcher] type accelerator.

In the following example I will use Get-ADOrganizationalUnit in combination with an if-statement and Get-ADObject to gather empty OUs:

```
Get-ADOrganizationalUnit -Filter * | ForEach-Object {
   if (-not (Get-ADObject -SearchBase $_ -SearchScope OneLevel -Filter *)){
       $_
   }
}
```

So lets have a look at what this code does, the first portion is straight forward, gather all OUs using the Get-ADOrganizationalUnit cmdlet and pipe it into the ForEach-Object cmdlet. The if-statement is the interesting part here, I am using the Get-ADObject cmdlet to establish if this OU contains any child object, by setting the SearchBase to that OU and setting the SearchScope to OneLevel. Setting the SearchScope to OneLevel will only return direct child objects of the parent, the OU, without returning the OU itself. Because of this Get-ADObject will not return any objects if the OU is empty.

For more information about the SearchScope parameter and the possible arguments have a look at the following link: Specifying the Search Scope

Because you might not have the ActiveDirectory module loaded in your current PowerShell session it can be useful to know the [adsisearcher] alternative:

```
([adsisearcher]'(objectcategory=organizationalunit)').FindAll() |
Where-Object {
    -not (-join $_.GetDirectoryEntry().psbase.children)
}
```

This is a slightly different approach to illustrate a different method of gathering empty OUs, here we check the Children property part of the base object that is retrieved. The -join operator is used to ensure the -not does not evaluate the empty System.DirectoryServices.DirectoryEntries object as true.

Using the logic in this post it is also possible to filter for other specific objects contained in the OUs. For example display OUs that only have user objects, display OUs with both user and computer objects and so on.

For more information on this subject please refer to the following links:

Additional resources

Specifying the Search Scope

Get-ADObject

Get-ADOrganizationalUnit

Use the ANR filter for LDAP Queries

ANR or Ambiguous Name Resolution is used to query for objects in Active Directory if the exact identity of an object is not known. A query containing Ambigious Name Resolution will query for all the attributes for example, Given Name, Sur Name, Display Name and samaccountname. For Windows Server 2008 and later versions this is the full list of ANR Attributes included in the search results:

For a full list of all the attributes that are queried please refer to the following TechNet article: ANR Attributes.

- Display-Name
- Given-Name
- Physical-Delivery-Office-Name
- Proxy-Addresses
- RDN
- SAM-Account-Name
- Surname
- Legacy-Exchange-DN
- ms-DS-Additional-Sam-Account-Name
- ms-DS-Phonetic-Company-Name
- ms-DS-Phonetic-Department
- ms-DS-Phonetic-Display-Name
- ms-DS-Phonetic-First-Name
- ms-DS-Phonetic-Last-Name

For a full list of all the attributes that are queried please refer to the following TechNet article:ANR Attributes.

An ANR query is useful in a number of scenarios, for example when relying on user input in your script. In this case querying against a samaccountname might fail if the spelling does not match the samaccountname. Similarly an export from a different department or database might be close to what is stored in Active Directory but not an exact match, again this is somewhere where an ANR query might be useful. Something that should be kept in mind is that this is a relatively expensive query and therefore should be avoided when it is not required. In this article we will discuss how to create an ANR filter and what happens exactly in such a query.

In the next example we will be using Get-ADUser cmdlet, which is part of the ActiveDirectory module, in combination with the LDAPFilter parameter in order to execute our query:

```
Get-ADUser -LDAPFilter '(anr=Jaap Brasser)'
```

This will query against all the attributes in the list as 'Jaap Brasser*' and two additionally queries: 'GivenName=Jaap*' and 'SurName=Brasser*' as well as 'GivenName=Brasser*' and 'SurName=Jaap*'. As a result more than one result might be returned, as different attributes of a user account might overlap or are not unique to a single user account. This is the downside of this method of querying.

In the following example I will use the [adsisearcher] type accelerator to execute the same query:

```
([adsisearcher]'(anr=Jaap Brasser)').FindAll()
```

Alternatively the DirectorySearcher object can be manually created to execute a query:

```
$ADSearcher = New-Object DirectoryServices.DirectorySearcher -Property @{
 Filter = '(anr=Jaap Brasser)'
 PageSize = 100
}
$ADSearcher.FindAll()
```

For more information on this Ambiguous Name Resolution (ANR) have a look at the following resources:

Ambiguous Name Resolution
MSDN Ambiguous Name Resolution
ANR Attributes
KB Ambiguous Name Resolution for LDAP in Windows 2000

Find users with password never expires

Having password set to never expires might be something that is not allowed by your IT policy, or perhaps you would like to get some insight about how widespread this setting is in your domain. In order to find accounts the Search-ADAccount cmdlet can be used. In order to find all user accounts that do have the *password never expires* option enabled the following code can be used:

```
Search-ADAccount -UsersOnly -PasswordNeverExpires
```

Alternatively the Get-ADObject cmdlet can also be used in combination with an LDAP filter to filter out the user accounts and the *password never expires* option. To filter out user accounts we should filter the following: '**(objectCategory=person)(objectClass=user)**'. To search for*password never expires* the following filter is used: '**(userAccountControl:1.2.840.113556.1.4.803:=65536)**'. Combined that gives us the following code:

```
Get-ADObject -LDAPFilter "(&(objectCategory=person)(objectClass=user)`
(userAccountControl:1.2.840.113556.1.4.803:=65536))"
```

It is of course also possible to do this using the DirectoryServices.DirectorySearcher. This time we use a slightly different LDAP filter, instead of filtering on '**(objectCategory=person)(objectClass=user)**' we filter on '**(sAMAccountType=805306368)**' which gives the same output but is a more efficient query. Also we set pagesize to 100 so we ensure that all results are displayed:

```
$ADSearcher = New-Object DirectoryServices.DirectorySearcher -Property @{
  Filter = `
'(&(sAMAccountType=805306368)(userAccountControl:1.2.840.113556.1.4.803:=65536))'
  PageSize = 100
}
$ADSearcher.FindAll()
```

And that is all that is required in order to find AD users with the *password never expires* option set, with or without the ActiveDirectory module.

Change a user's password

It is one of the most common tasks Active Directory administrators face, changing a user's password or unlocking their account. Today we will discuss how this can be done in Powershell using either the Active Directory module or [adsi] type accelerator for this purpose.

Setting or resetting a password is rather straight forward using the Active Directory cmdlets, simply use Get-ADUser to get the AD user object and pipe it into Set-ADAccountPassword:

```
Get-ADUser jaapbrasser | Set-ADAccountPassword -Reset -NewPassword`
(ConvertTo-SecureString -AsPlainText "secretpassword01" -Force)
```

To unlock an account the Unlock-ADAccount cmdlet can be used:

```
Get-ADUser jaapbrasser | Unlock-ADAccount
```

To both unlock and change the password of a user using the ADSI type accelerator the following code can be used:

```
$User = [adsi]([adsisearcher]'samaccountname=jaapbrasser').findone().path
$User.SetPassword("secretpassword01")
$User.lockoutTime = 0
$User.SetInfo()
```

Create new OU

PowerShell can be used to create any number of objects in Active Directory. Today I will demonstrate how to create an organizational unit using both the ActiveDirectory module as well as the [adsi] alternative.

Creating an OU using the New-ADOrganizationalUnit is quite straight forward:

```
New-ADOrganizationalUnit -Name Departments -Path`
"ou=Resources,DC=jaapbrasser,DC=com'
```

Using the [adsi] accelerator to create an OU requires some additional steps. First the parent object has to be selected, in this example the Resources OU in the jaapbrasser.com domain. The next step

is create a new object, an organizationalunit in this case, finally the changes are committed to Active Directory by using the SetInfo() method.

```
$TargetOU = [adsi]'LDAP://ou=Resources,DC=jaapbrasser,DC=com'
$NewOU = $TargetOU.Create('organizationalUnit','ou=Departments')
$NewOU.SetInfo()
```

That is all there is to it, creating an Organizational Unit in Active Directory is quite easy, with or without the ActiveDirectory module.

Determine tombstone lifetime

In Active Directory objects are tomb stoned after a deletion occurs. This is allow replication to occur between domain controllers before an object is deleted from the Active Directory data store. The default value depends on the server when the forest was initially created, Microsoft recommends that this is set at 180 days.

The tombstone lifetime is set at the forest level and can be viewed by running the following code:

```
([adsi]"LDAP://CN=Directory Service,CN=Windows NT,CN=Services,`
$(([adsi]("LDAP://RootDSE")).configurationNamingContext)").tombstoneLifetime
```

Alternatively this can also be retrieved by using the Get-ADObject cmdlet:

```
$HashSplat = @{
    Identity = 'CN=Directory Service,CN=Windows
NT,CN=Services,CN=Configuration,DC=jaapbrasser,DC=com'
    Partition = 'CN=Configuration,DC=jaapbrasser,DC=com'
    Properties = 'tombstoneLifetime'
}
Get-ADObject @HashSplat | Select-Object -Property tombstoneLifetime
```

Search for computers accounts

I have decided to reintroduce Active Directory Friday on my blog, so today is the start of the new series of articles on Friday. The format remains the same as the previous posts. Usually the examples will be written by using .Net objects or the [adsi] and [adsisearcher] accelerators, although occasionally examples using the Active Directory cmdlets will be posted. My preference for avoiding the cmdlets is mostly compatibility, usually there is only a select number of systems that has access to the Active Directory module, so it pays off to know the native method as well.

Today we will take a look at how to find computer objects in Active Directory using the DirectoryServices.DirectorySearcher object. In order to search for computer objects the following properties of this object will be set:

- Filter – This contains the LDAP filter used to select only the computer objects by specifying the objectcategory
- PageSize – This allows for paging to occur, by specifying the pagesize more than 1000 results can be returned

```
$Searcher = New-Object DirectoryServices.DirectorySearcher -Property @{
    Filter = '(objectCategory=computer)'
    PageSize = 500
}
$Searcher.FindAll()
```

To search in a specific organizational unit the SearchRoot property can be used, only computer objects in the Servers OU will be returned by this search:

```
$Searcher = New-Object DirectoryServices.DirectorySearcher -Property @{
    Filter = '(objectCategory=computer)'
    PageSize = 500
    SearchRoot = 'LDAP://OU=Servers,DC=jaapbrasser,DC=com'
    SearchScope = 'Subtree'
}
$Searcher.FindAll()
```

The SearchScope property has been set to Subtree, which means that the OU will be recursively searched through and all child-ous will be included in the search. There are a total of three options available for the SearchRoot property:

- Base – Only returns a single objects
- OneLevel – Only searches the current container, will not recursively search
- Subtree – Searches recursively through all child containers

List password information for Domain Administrators

In today's Active Directory Friday we touch the subject of security of Domain Administrator accounts. Although this should not be overlooked it is not uncommon for passwords to be unchanged for a long period of time.

To find the members of the Domain Admins group we can use following LDAP Filter:

```
"(memberof=CN=Domain Admins,CN=Users,DC=jaapbrasser,DC=com)"
```

Then for each account found a PowerShell Custom Object is created with the following three properties:

- Samaccountname
- PasswordAge
- Account Enabled

So combing all these statements the complete code is as follows:

```
$Searcher = New-Object DirectoryServices.DirectorySearcher -Property @{
    Filter = "(memberof=CN=Domain Admins,CN=Users,DC=jaapbrasser,DC=com)"
    PageSize = 500
}
$Searcher.FindAll() | ForEach-Object {
    New-Object -TypeName PSCustomObject -Property @{
        samaccountname = $_.Properties.samaccountname -join ''
        pwdlastset =
[datetime]::FromFileTime([int64]($_.Properties.pwdlastset -join ''))
        enabled = -not [boolean]([int64]($_.properties.useraccountcontrol -
join '') -band 2)
    }
}
```

Get DistinguishedName of current domain

To determine the DistinguishedName of the current domain the [adsi] accelerator can be utilized.

The following piece of code can be used to retrieve the DN of the current domain:

```
New-Object -TypeName System.DirectoryServices.DirectoryEntry |
Select -ExpandProperty distinguishedName
```

Alternatively the [adsi] accelerator can be utilized for this purpose, as this requires less code and it is easier to remember:

```
([adsi]'').distinguishedName
```

The value returned by this line of code a System.DirectoryServices.PropertyValueCollection instead of a string object. To unwrap this code can be used:

```
([adsi]'').distinguishedName[0]
```

Now the object returned is a string and the methods and properties of a string object are available, so it is possible to manipulate the output for example by doing a text replace:

```
([adsi]'').distinguishedName[0].replace('com','jaap')
```

Note that in PowerShell v3 and up it is not required to unwrap the array, as the Member Enumeration feature of PowerShell will ensure that the methods and properties of underlying objects in an array are available. As demonstrated in the following line of code:

```
([adsi]'').distinguishedName.replace('com','jaap')
```

Query Group Policy Objects in Active Directory

For the second Active Directory Friday we have Group Policies on our radar. To query for Group Policy objects the following LDAP filter can be used:

```
'(objectClass=groupPolicyContainer)'
```

To get the full list of Group Policy objects the adsisearcher accelerator should be used in combination with the LDAP filter. This will return all group policy objects:

```
([adsisearcher]'(objectClass=groupPolicyContainer)').FindAll()
```

To generate a short report with relevant information about the following code can be used:

```
$GPOSearcher = New-Object DirectoryServices.DirectorySearcher -Property @{
    Filter = '(objectClass=groupPolicyContainer)'
    PageSize = 100
}
$GPOSearcher.FindAll() | ForEach-Object {
    New-Object -TypeName PSCustomObject -Property @{
        'DisplayName' = $_.properties.displayname -join ''
        'CommonName' = $_.properties.cn -join ''
        'FilePath' = $_.properties.gpcfilesyspath -join ''
        'DistinguishedName' = $_.properties.distinguishedname -join ''
    } | Select-Object -Property
DisplayName,CommonName,FilePath,DistinguishedName
}
```

This will display a list of all Group Policy Objects and display the following properties:

- DisplayName
- CommonName
- FilePath
- DistinguishedName

The full script is also available in the TechNet Script Gallery:http://gallery.technet.microsoft.com/Get-GroupPolicyObject-05aaef2d

Find user accounts that have not changed password in 90 days

Today I am starting a new section my blog. Each friday I will post an example of a task I have performed in Active Directory using PowerShell. For this I will usually not use any of the Active Directory Cmdlets, so there is no dependancy on any modules to be present on a system in order to execute these queries. If you have any suggestions for a task or query that could be discussed, please drop me a line in the comments and I will consider it for next week. Today I will start with a query that gathers the samaccountname, pwdlastset and if an account is currently enabled or disabled. Note that the commands in this article only query Active Directory so no changes to objects will be made. First we will create a variable, $PwdDate, that contains the filetime of a date ninety days ago:

```
$PwdDate = (Get-Date).AddDays(-90).ToFileTime()
```

Then an DirectoryServices.DirectorySearcher object will be created with the LDAP Query to locate only user accounts that have their passwords last set on a date 90 or more days ago:

```
$Searcher = New-Object DirectoryServices.DirectorySearcher -Property @{
    Filter =
"(&(objectclass=user)(objectcategory=person)(pwdlastset&lt;=$PwdDate))"
    PageSize = 500
}
```

ForEach user account found we output its samaccountname, pwdlastset and enabled or disabled state of the account:

```
$Searcher.FindAll() | ForEach-Object {
    New-Object -TypeName PSCustomObject -Property @{
        samaccountname = $_.Properties.samaccountname -join ''
        pwdlastset =
[datetime]::FromFileTime([int64]($_.Properties.pwdlastset -join ''))
        enabled = -not [boolean]([int64]($_.properties.useraccountcontrol -
join '') -band 2)
    }
}
```

Note that the -join ” operator this is used to unwrap the properties, which are by default provided as System.DirectoryServices.ResultPropertyValueCollection objects. Alternatively the array indexing notation [0] could be used, this has the downside that when a property is empty it will cause the script to display errors. The full code used in this example is available here in the TechNet Script Repository: http://gallery.technet.microsoft.com/scriptcenter/Query-for-AD-Users-that-b87acf2f