Where I work, we have a number of applications that use Active Directory and a Single Sign On methodology, and each application then has a table with the users login and mappings to various internal roles to determine what parts of the application the user can access. Also, except for a handful of applications that need SQL logins, SQL server access is granted via AD Groups or Ad Logins.

As a result, when a user reports an issue, we have to start at active directory to determine if the user is locked out, disabled, or password expired before we can even start looking to see if the problem is in an application.

Having a list of Active Directory Users in a SQL Table can really open up your ability to analyze your systems, and produce smart relevant reports about your user base.

In my case, I’m using PowerShell to query Active Directory for several different items, and insert the results into a suite of tables. I’m not going to go deep into PowerShell ins and outs, but hope to give you an incredible tool to put in your toolbox.

### The Rabbit Hole

When you put your SQL Hat on, since you grant access to both groups and users, you realize you need tables filled with AD Users, AD groups and a AD Group Members table featuring who belongs in those groups. As you start pulling the AD Groups, you end up getting groups that contain other groups, objects like servers and computers and so much more. Add in multiple domains or forests, and it starts getting scary! The whole process goes pretty deep down that rabbit hole, so you get to benefit from the investigation and solution I’ve built so far.

So for my project, I’m gathering quite a bit of information; I’m getting a list of all Active Directory Users, all the groups, and all the group members. Groups can contain groups, so you might need to create a recursive CTE to enumerate just which GroupMembers is in a given group.

### The Core Commands

Here’s a script example that is getting the Top 10 of users and Top 100 groups as an example. The last is enumerating the members in the “users” group, which is often another group name.

Each example is using the **Out-GridView** command to throw the data into a grid, which keeps us SQL nerds in our comfort zone.

Import-Module ActiveDirectory

Get-ADUser -Filter \* -ResultSetSize 100 -Properties \* | Out-GridView

Get-ADGroup -Filter \* -ResultSetSize 100 -Properties \* | Out-GridView

Get-ADGroupMember -Identity "Users" | Out-GridView



### Review the Results

Let’s look at the results for Get-ADUser, since that’s the most important item; everything else is just more information about our core investigation: the Users in AD.

A side note that is probably relevant: if you ask for specific properties, and you misspell it or it does not exist, you get a blank answer, which can be confusing. In my example below, I included a column “Email”, but AD does not have a property for it; it does have EmailAddress though. In my world, we end up populating that [email] address outside of this process for other reasons.

Get-ADUser -Filter {samaccountname -eq " Leonard\_Hofstadter"} -ResultSetSize 10 -Properties \* | select -property sAMAccountName,ou,GivenName,SurName,DisplayName,email,emailaddress | Out-GridView



For the most part, what I’m grabbing is very straightforward, since the values are reasonably self describing; GivenName and Surname(FirstName/LastName) are a bit of unusual vernacular, but all other values are pretty understandable.

I’ve added a pair of inline calculations to calculate when an account or password expires, which can be helpful. The values stored in AD are not a datetime value, but number of ticks, which is something that is not easily consumable, so I’m converting them up front.

The final command I actually end up looks like this for reference, and has the properties you see here:

Get-ADUser -Filter \* -Properties \* | select -property

CanonicalName,

sAMAccountName,

ou,

GivenName,

SurName,

DisplayName,

email,

emailaddress,

StreetAddress,

City,

State,

PostalCode,

HomePhone,

MobilePhone,

OfficePhone,

Fax,

Company,

Organization,

Department,

Title,

Description,

Office,

extensionAttribute1,

extensionAttribute2,

extensionAttribute3,

extensionAttribute4,

extensionAttribute5,

@{Name='AccountExpires';Expression={[DATETIME]::fromFileTime($\_.accountExpires)}},

Enabled,

PasswordLastSet,

@{n="PasswordExpirationDate";e={$\_.PasswordLastSet.AddDays($maxPasswordAge)}},

PasswordNeverExpires,

PasswordExpired,

LastLogonDate,

whenCreated

### Caveats and Gotchas for new to Powershell folks

An interesting difference between TSQL and Powershell: asterisk (\*) does not mean “all” in PowerShell! As a SQL developer, you expect asterisk (\*) to mean everything, but instead, there is a default set of Properties that is returned as part of the asterisk results. You’ll see why in a minute.

The command below returns 30 columns in our grid; that might be an artifact of the Out-Grid command, because you don’t get all values in the grid!

Get-ADUser -Filter \* -ResultSetSize 10 -Properties \* | Out-GridView

The same command sent to the text viewer, in my environment, returns 142 values!

Get-ADUser -Filter \* -ResultSetSize 10 -Properties \*

But wait, there’s more! In my shop, I **know** we use some of the twelve optional properties related to the ADUser to store information; **extensionAttribute1** through **extensionAttribute12** exist, but are empty unless you populate them. I *know* they exist, but they are not in that 142 items I just grabbed. So instinctively, I already know there are more properties that I’m currently seeing. It depends on your AD environment, but there are actually more than 700 properties related to an ADUser!

If you need to go down that tangent, I’ve added a text file (Get-ADUser\_All\_Properties.txt) that enumerates all the Properties I saw in my domain. Also, two resources for you to make it easy:

Download Dell (Formerly Quest) ActiveRoles Management Shell for Active Directory 1.7

<https://support.software.dell.com/download-install-detail/5024645>

Dmitry Sotniko’s blog has an example using the above script to get a list of ALL user properties.

<https://dmitrysotnikov.wordpress.com/2007/06/28/get-a-list-of-all-user-properties/>

My own script adaptation of those two resources is below:

###################################################################################################

#load snapin if not already loaded

# https://support.software.dell.com/download-install-detail/5024645

###################################################################################################

if ( (Get-PSSnapin -Name Quest.ActiveRoles.ADManagement -ErrorAction SilentlyContinue) -eq $null )

{

Add-PsSnapin Quest.ActiveRoles.ADManagement

}

Get-QADUser -ReturnPropertyNamesOnly -IncludeAllProperties

## What if we have multiple forests or child domains?

My company was recently absorbed by another company. Get-ADUser has an optional **–Server** parameter that you can use to identify the other domains.

In my example, My Home domain is the BigBang domain was absorbed by the Disney domain.

I know that fully qualified SamAccountName are actually something like **BigBang.com\** **Leonard\_Hofstadter**, but the other domain is actually users like **Disney.local\Aladdin\_StreetRat**

So I could add the parameter to the GetADUser call to pull the other domain:

Get-ADUser -Filter \* -ResultSetSize 10 -Server bigbang.com -Properties \* | Out-GridView

Get-ADUser -Filter \* -ResultSetSize 10 -Server disney.local -Properties \* | Out-GridView

I don’t have child domains set up, but it would be the same logic; identify the child domain, and pass that to the server;

Get-ADUser -Filter \* -ResultSetSize 10 -Server Avengers.MarvelUniverse.com -Properties \* | Out-GridView

Get-ADUser -Filter \* -ResultSetSize 10 -Server XMen.MarvelUniverse.com -Properties \* | Out-GridView

### PowerShell to SQL Time!

OK so I rambled on about how there is a LOT of info out there, but what is relevant? I recommend you start with what I found relevant to me, and then add or subtract other properties that might be useful. I went through a lot of iterations to determine what I wanted to actually capture, and the SQL table that captures it is in the shape below: but I built a suite of views to make the information even more relevant.

The key things for most people are the classics; FirstName/LastNname/EmailAddress and whether they are locked out, disabled or password expired. I’ve included a lot of values that we actually use regularly;

I’ve got a companion process that updates information in Active directory, so that if we needed to say, update everyone’s address or contact phone numbers, or Department, I can use Set-ADUser and update those values from my SQL Table. If that seems relevant, I’ll add a companion article to this one. It can be really scary updating Active Directory in monster batches though, a lot of people don’t have that level of comfort.

Since this information is not related to one specific application, I put the tables and views related to this in a separate database, DBA\_Utilities; when you install the two scripts, you will need to create the objects in the right database, and change just two parameters the PowerShell script

###################################################################################################

#SQL Connection Setup

###################################################################################################

$SQLServer = "localhost";

$SQLDBName = "DBA\_Utilities";

{Placeholder for FullScript ActiveDirectoryTablesAndViews.sql}

### ActiveDirectoryTablesAndViews Details

The tables here are what I felt we needed at the time; you can strip out or add columns, but you’ll also need to strip out or add parameters and columns in the matching PowerShell script. All in all, it’s a robust collection of items to start your baseline with.

There’s a few views I created that help gather the data together for convenience, and they are not mandatory or anything, just some suggestions on how to visualize the data a little better.

### FootPrint?

The PowerShell below gets All AD Users every time, I haven’t bothered to look into trying to get incremental changes; my local domain environment only has ~3000 ADUsers, many of them past employees, so I didn’t see a need to streamline the amount; if you worked for a big company, you might want to look at tuning it to get only new records. Here’s the core of the Get-ADUsers script; get the users in memory, and loop through each, calling an update-insert SQL command to update/insert the data into the table.

Timewise, it takes less than three minutes to run the script for users, but the script for Groups and Group members often takes more than ten minutes; it’s grabbing a lot more records and making a lot more calls.

In a real life scenario, for just a single forest, here’s some rowcounts for my tables, which contains two domain forests, Each with roughly 3K users ; you can see the Group Members is roughly 10X the User Count, because users exist in multiple groups.

|  |  |  |  |
| --- | --- | --- | --- |
| SchemaName | ObjectName | TheCount | cmd |
| dbo | GetActiveDirectoryGroupMembers | 62054 | SELECT \* FROM [dbo].[GetActiveDirectoryGroupMembers] |
| dbo | GetActiveDirectoryUsers | 6602 | SELECT \* FROM [dbo].[GetActiveDirectoryUsers] |
| dbo | GetActiveDirectoryGroups | 2464 | SELECT \* FROM [dbo].[GetActiveDirectoryGroups] |

### GetActiveDirectoryUsers Details

The logic in the script below is to basically call a parameterized upsert command for each row returned from the Get-ADUser Call. A Get-String function simply returns empty strings for Nothing/NULL values.

{Placeholder for FullScript Get-ActiveDirectoryUsers}

### GetActiveDirectoryGroups Details

{Placeholder for FullScript Get-ActiveDirectoryGroups}

The logic in the script below is similar to the first example : call a parameterized upsert command for each row returned from the Get-ADGroup Call., but for each group, we also Call Get-ADGroupMembers and upsert those into a third table.

So this one proc is populating two tables for us. The Get-ADGroupMembers call gives us four different ObjectClass values; user/group/computer or blank values.

It’s pretty handy to now have a list of all known computers in active directory, as it might be the basis for scanning your computers ofr services, putting them in workstation vs server logical groups, etc.

select \* from [GetActiveDirectoryGroupMembers] where objectclass = 'computer'

### Run This Regularly or not?

I’ve included a scripted Job as part of the article as well. However, running things that Get AD info requires a domain user, so you need all that that would entail: a Credential, an Operator, and operator permissions.

I’ve created a scheduled job that scans for users M-W-F, but I end up refreshing Groups and Group members rarely, if ever. I’ve disabled the job step in my case, since I rarely need to compare group information. I might run it manually once a month or so, but my own needs tend towards reviewing end users.

### So What Can I do with it?

So you’ve got some nifty tables full of this information, but if you don’t actually use it, it has no real value. Here’s just some of the things I do with it.

1. Remove SQL users that are disabled. I create a script that does an IF EXISTS() and drops users and logins. Pasting that into a Central Management Server multi server query, and I cleanup tons of servers at a time.
2. Generate and verify the list of users who should have access to various resources based on the groups they belong to.
3. Adding AD users quickly to our application via insert…from syntax.
4. Finding users who exist in recursive groups.
5. Inventorying Servers and workstations.
6. Providing spreadsheets to be used for Updating Active Directory; we have had several initiatives to update phone number and addresses and custom string values; I provide a spreadsheet, and then generate Set-ADUser commands to update AD, which in turn gets pulled back into my tables.

I hope you find this suite of tools for your toolbox helpful; It’s one of those Secret Sauce implementations a lot of folks might have developed independantly, but are afraid to share, since they put a lot of time into it. I’ve been in countless meetings where someone says “how many people in active directory? How many active in the last 30 days? how many are just active period?”; being able to pull that info up in 30 secodns makes you look extremely competent.

Please join the discussion and give some feedback! Everything we add makes the SQL community a little better.

### TLDR:

1. run the ActiveDirectoryTablesAndViews.sql in a database named DBA\_Utilities(or the db of your choice) to create the Tables And Views.
2. Modify the two scripts GetActiveDirectoryUsers.ps1 And GetActiveDirectoryGroups.ps1 to point to the right **server** and **database**, if it’s not localhost and DBA\_Utilities.
3. Run each Of the two PowerShell Scripts manually from either a PowerShell prompt, a PowerShellIDE, or a commandline call .

Optional SQL Job is going to require a Credential and Operator, that is an AD User:

1. Find the CredentialHelper.sql script, so that you can modify it to create the Credential, Operator, and permissions to the **PowerShell** and **CmdExec** subsystem.
2. Copy both PowerShell files to a location on the server, ie L:\PowerShell\
3. Modify the Script SQLJob\_ETL\_GetActiveDirectoryData.sql in two places to have the **filepath** you decided on in Step 3. Modify the Operator if you didn’t stick with “TaskRunner”, and run it to create the Job.
4. Run the Job manually, and review the contents of the new tables and views.

Link To Attachments