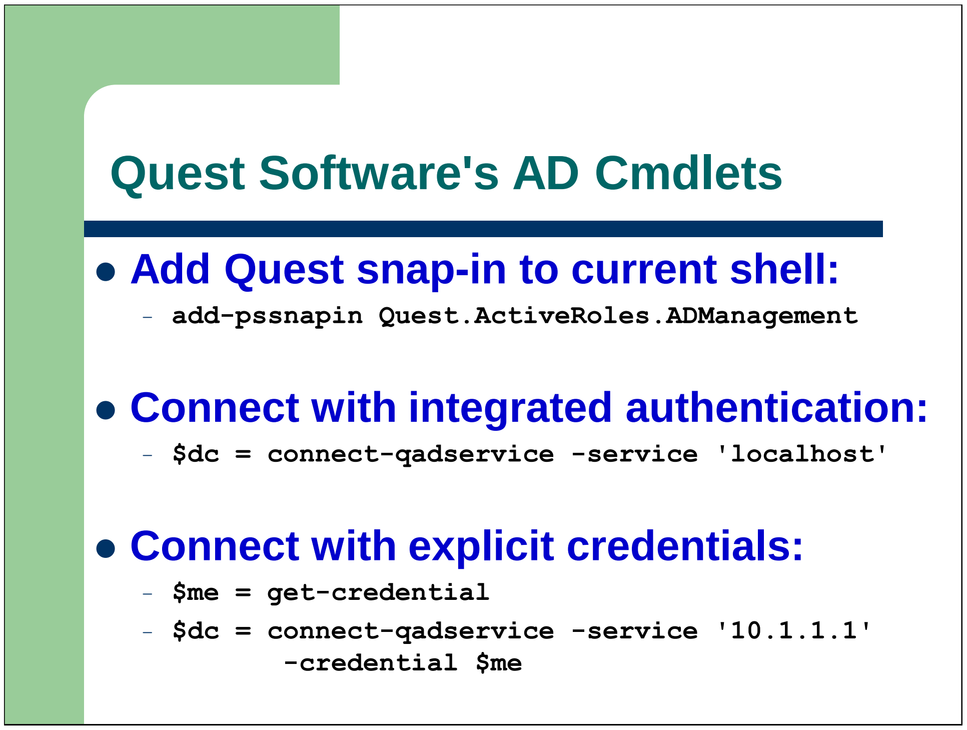


Quest Software's AD Cmdlets

- **Add Quest snap-in to current shell:**
  - `add-pssnapin Quest.ActiveRoles.ADManagement`
- **Connect with integrated authentication:**
  - `$dc = connect-qadservice -service 'localhost'`
- **Connect with explicit credentials:**
  - `$me = get-credential`
  - `$dc = connect-qadservice -service '10.1.1.1' -credential $me`

Quest Software's Active Directory Cmdlets

Until PowerShell 2.0 comes out with built-in cmdlets for Active Directory, there are third-party cmdlets instead. Because cmdlets all have similar *verb-noun* naming conventions and parameter rules, we can easily transition from one set of cmdlets to another.

Quest Software has a great set of cmdlets for Active Directory that are free to download and use (http://www.quest.com/activeroles-server/arms.aspx). These cmdlets are a part of their larger ActiveRoles Server product for doing comprehensive Active Directory management and provisioning (http://www.quest.com/activeroles-server/).

To install Quest's AD cmdlets you must have PowerShell, .NET Framework 2.0, and Windows XP-SP2/Vista/2003-SP1/2008 or later. The cmdlets can be installed on a domain controller, member server or even a stand-alone box (but the cmdlets can't be used for managing local users or groups). Look for the PDF documentation file that comes with the MSI installer package; it's very detailed. Once installed on your computer, the snap-in for Quest's cmdlets can be loaded into your current shell, or, if you modify your profile, loaded into your shell automatically whenever you open it.

To load the snap-in with the Quest cmdlets into your current shell:

```
# Example: Quest_AD.ps1

add-pssnapin Quest.ActiveRoles.ADManagement
```

If prompted during installation to trust Quest as a publisher, select "A" to always trust them. If you want Quest's cmdlets available every time you launch PowerShell, add the above command to your profile script.

To see a list of all Quest-related cmdlets (they all have "QAD" in their names):

```
C:\> get-command *-qad*

      Connect-QADService
      Disconnect-QADService

      Get-QADComputer
      Get-QADGroup
      Get-QADGroupMember
      Get-QADObject
      Get-QADObjectAttributes
      Get-QADUser

      New-QADGroup
      New-QADObject
      New-QADUser

      Add-QADGroupMember

      Remove-QADGroupMember
      Remove-QADObject

      Set-QADObject
      Set-QADUser
```

You can also list cmdlets by the snap-in upon which they depend, for example, "`get-command -pssnapin Quest.ActiveRoles.ADManagement`".

Before any AD cmdlets can be run, you have to establish an authenticated connection to a domain controller. Though it is possible to let the cmdlet find an available domain controller by itself, it works better if you provide the name or IP address of a specific controller. PowerShell will authenticate to the controller automatically as the account under which PowerShell is running (you don't need to provide a username or password). We'll save the connection object to a variable ($dc) just in case we need it later, but this step of saving to a variable actually isn't necessary.

```
$dc = connect-qadservice -service 'localhost'
$dc | format-list *
```

To be prompted for different credentials and connect with those credentials instead:

```
$me = get-credential
$dc = connect-qadservice -service '10.1.1.1' -credential $me
```

Once you have an authenticated connection to a domain controller, you can use that connection (and the variable representing it) with other cmdlets from Quest.

Managing User Properties

To get all user and computer objects from the domain:

```
get-qaduser
get-qadcomputer
```

To view all the properties of the Administrator account from the giac.org domain (notice that you can specify the user by UPN, *domain\username*, or distinguished name (DN)):

```
get-qaduser administrator@giac.org | format-list *
get-qaduser giac\administrator | format-list *
get-qaduser 'cn=Administrator,cn=Users,dc=giac,dc=org'|fl *
```

To reset the passphrase of the Guest account in the giac.org domain:

```
set-qaduser giac\guest -userpassword 'fly tomatoes at nite'
```

To change the pager number of the Justin account in the giac.org domain:

```
set-qaduser justin@giac.org -pager '(214)328-2292'
```

Now, not all properties of all objects are exposed in the cmdlets as simple parameter names like "-pager" or "-userpassword", but you can still manage those other properties. To modify one or more properties you'll enter a hashtable of *propertyname*='*value*' pairs separated by semicolons.

To change the e-mail address for Justin in the giac.org domain:

```
set-qaduser justin@giac.org -objectattributes
@{mail='justin@giac.org'}
```

To change Justin's e-mail address, description and home phone number properties:

```

set-qaduser justin@giac.org -objectattributes
@{mail='justin@giac.org'; description='IT Engineer';
homephone='(222)333-4444'}
```

It can be difficult to figure out the name of the property you want to modify, hence, use the ADSI Edit MMC snap-in to browse the properties of objects to figure out these names. Also, it is indeed possible to manage multi-valued properties using the cmdlets; see Quest's documentation for examples.

Create and Delete Global User Accounts

To create a user account, you must specify the common name (CN) of the user, its backwards-compatible username (samaccountname) and the container in Active Directory where the account will be created (the AD container must be specified using its full distinguished name (DN) path). Though not mandatory, you'll probably also want to assign a password and enable the account too (a generic enabled account has a useraccountcontrol field set to 544). You can set additional optional attributes while creating the account too.

To create a user named "Jessica Parker" in the Users container of the giac.org domain, with a passphrase of "mice ate my leggs again", a username of "JessicaP", and have it be an enabled account too (useraccountcontrol = 544):

```
new-qaduser -name 'Jessica Parker' -parentcontainer
'cn=Users,dc=giac,dc=org' -userpassword 'mice ate my leggs again'
-samaccountname 'JessicaP' -objectattributes
@{useraccountcontrol='544'}
```

To delete a user named "Jessica Parker" in the Users container of the giac.org domain:

```
remove-qadobject 'CN=Jessica Parker,CN=Users,DC=giac,DC=org'
```

> **Note**: Don’t forget that the -WhatIf switch can be used to show what changes would occur if the command were actually run. This is a safe way to test code without accidentally making destructive changes. If you want to be prompted to allow/disallow a change before they're committed, use the -Confirm switch.

To delete an organizational unit named "Test" from the giac.org domain as well as recursively delete (-DeleteTree) everything inside it, including other organizational units, and at the same time suppress the confirmation prompt so it works hands-free (-Force):

```
remove-qadobject 'OU=Test,DC=giac,DC=org' -DeleteTree -Force
```

Managing Groups

To list each group's name, scope (domain local, global, universal) and type (security or distribution group):

```
get-qadgroup | format-table name,groupscope,grouptype -auto
```

To list the members of the Domain Admins group:

```
get-qadgroupmember 'Domain Admins'
```

To add the Administrator account in the giac.org to the Guests group:

```
add-qadgroupmember 'Guests' -member 'administrator@giac.org'
```

To remove the Administrator account from the Guests group in the giac.org domain:

```
remove-qadgroupmember 'Guests' -member 'administrator@giac.org'
```

Find Accounts With Old Passwords

To get a list of user accounts whose passwords have not been changed in *X* days:

```
# Example: Get-UsersWithOldPasswords.ps1

function Get-UsersWithOldPasswords ($DaysAgo, $Controller)
{
    connect-qadservice -service $Controller | out-null

    $TicksAgo = (((get-date).AddDays($DaysAgo * -1)).Ticks -
       504911232000000000)

    get-qaduser -ldapfilter "(&(pwdLastSet<=$TicksAgo)
      (pwdLastSet>=1))" -sizelimit 0

}
```

A lot is going on in the above little function. A "tick" is 100 nanoseconds (100 billionths of a second). The pwdLastSet field is the count of ticks between 12:00PM UTC on 1-Jan-1601 and when the password was last reset or changed (yes, that year is 1601, it's not a typo). Hence, the procedure is to get the ticks between year 0000 and now, minus the ticks for the $DaysAgo interval, minus the ticks from year 0000 to 1-Jan-1601, and then compare this tick count with the ticks in pwdLastSet: if pwdLastSet is smaller, then it is older and should be returned by the function.

If pwdLastSet equals zero, then the password has never been set, and this function doesn't return those accounts. Remove the "(pwdLastSet>=1)" from the -LdapFilter if you do want these to be returned (for more information about LDAP filters, Google on "ldap filter tutorial"). And "-sizelimit 0" tells the cmdlet not to limit the number of matching objects returned (by default, only 1000 objects maximum are returned).

Incidentally, the -LdapFilter is a server-side filter, and one of the properties *not* exposed by Quest's cmdlet is the pwdLastSet property, hence, while we can do a server-side filter on this property, we can't actually see the property locally once the matching user accounts are returned.

What is 504911232000000000? That's the number of ticks between year 0000 and 12:00PM UTC 1-Jan-1601. The following code snippet shows how to get this number.

```
$Jan011601 = new-object System.DateTime -argumentlist
1601,1,1,0,0,0,0,"UTC"

$Jan011601.Ticks       # Returns 504911232000000000
```

Update Description On All Computer Accounts

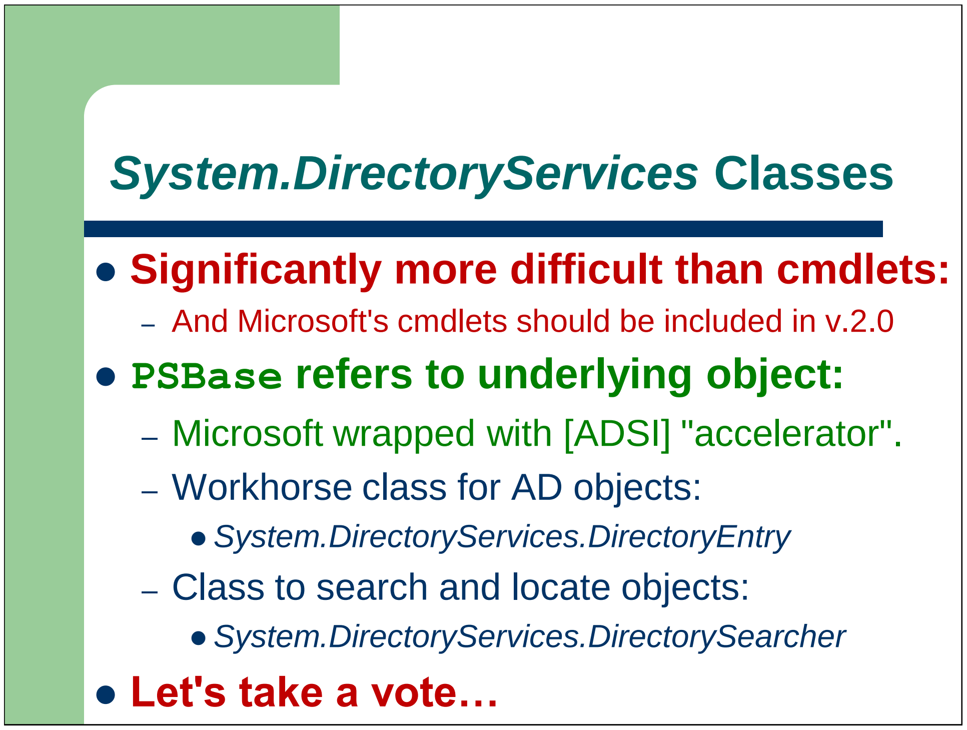
To change the description field on computer objects in particular domain or OU so that the description is the operating system version and Service Pack number:

```
# Example: Update-ComputerDescription.ps1


function Update-ComputerDescription ($DomainController =
'localhost', $Container = $null)
{
    $con = connect-qadservice -service $DomainController

    # Need to get the correct full DN path to the container
    # to be searched, but there are various ways to specify a
    # container other than a full DN.

    switch ($Container)
    {
        {$Container -eq $null}
            {
                $path = $con.DN      # Entire domain.
                break
            }
        {$Container -match $con.DN}  # Full DN path given.
            {
                $path = $(get-qadobject -identity
"$Container").DN
                if ($path -eq $null) {throw "DN path resolution `
                                             failed, please
check`
                                             the path!"}
            }
        default   # Assume simple name entered, not a DN.
            {
                # The following returns an array even if it
                # only contains one element.

                $path = @(get-qadobject -SearchRoot $con.DN -
Name`
                          "$Container" | select-object DN)

                if ( $path.length -eq 1 )
                     { $path = $path[0].DN }
```

```
                 else
                  {
                      "`nThe container name is ambiguous, there `
                        are $($path.length) matches:"

                      $path | select-object DN

                      "`nPlease enter full DN path to the `
                       container instead!`n"

                      throw
                  }
             }
    }


    # Now use correct $path to find and update computer `
    # accounts, but sometimes there are computer accounts `
    # which have never been used by any computer:

    get-qadcomputer -SearchRoot $path -SizeLimit 0 |
    foreach-object {
        if ( $($_.OSName) -eq $null )
         { set-qadobject -identity "$_" -objectattributes `
              @{description="Never Used"} }
        else
         { set-qadobject -identity "$_" -objectattributes `
              @{description="$($_.OSName) $($_.OSServicePack)"}
}
    }

    disconnect-qadservice
}
```

Finally, when you're done with your Active Directory connection, don't forget to disconnect it:

```
disconnect-qadservice
```



System.DirectoryServices Classes

- **Significantly more difficult than cmdlets:**
  – And Microsoft's cmdlets should be included in v.2.0
- **`PSBase` refers to underlying object:**
  – Microsoft wrapped with [ADSI] "accelerator".
  – Workhorse class for AD objects:
    - *System.DirectoryServices.DirectoryEntry*
  – Class to search and locate objects:
    - *System.DirectoryServices.DirectorySearcher*
- **Let's take a vote…**

System.DirectoryServices Classes in .NET

The PowerShell developers have included the [ADSI] solution accelerator, but we're just going to ignore it. In this author's opinion, it was a mistake, no matter how good the intentions were, because it only partially supports discovery through get-member, is a weird hand-waving exercise for the benefit of VBScript/JScript, and requires the somewhat annoying use of the "psbase" keyword whenever the .NET classes are directly used instead. The long-term future is .NET, so that's what we're going to focus on. Hopefully future versions of PowerShell will get rid of this awkward crutch.

Directory Services Classes

In the .NET class library, the workhorse class for managing Active Directory is the *System.DirectoryServices.DirectoryEntry* class. This class represents objects in AD like domains, OUs, groups, computers and users. Another class you might use frequently is *System.DirectoryServices.DirectorySearcher*, which provides methods to search and locate objects in AD when you only know the name of the object and not the full distinguished name path to it.

You will very often require the full distinguished name path to an object in AD in order to script management of that object. This is true of PowerShell, VBScript and command-line tools for AD. An example of a full distinguished name (DN) path to a user is "CN=Administrator,CN=Users,DC=SANS,DC=ORG". The best graphical tool to see these DN paths is the ADSI Edit MMC snap-in from the Support Tools.

Search For An Object

To search for an object by its name, class and domain, then return the full distinguished name path of the object if found:

```
# Example: Get-DistinguishedName.ps1

function Get-DistinguishedName ( $ObjectName,
                                 $Class = "user",
                                 $Domain = "" )
{
 if ($Class -like 'ou') { $Class = 'organizationalUnit' }

 $DirectoryEntry = new-object
         System.DirectoryServices.DirectoryEntry -arg $Domain

 $DirectorySearcher = new-object
   System.DirectoryServices.DirectorySearcher -arg
$DirectoryEntry

 $DirectorySearcher.Filter = "(&(objectClass=$Class)
                              (|(sAMAccountName=$ObjectName)
                              (cn=$ObjectName)(ou=$ObjectName)))"

 $SearchResultCollection = $DirectorySearcher.FindAll()

 if ($SearchResultCollection.Count -eq 1)
    {
        "'" + $SearchResultCollection[0].path + "'"
    }
 elseif ($SearchResultCollection.Count -gt 1)
    {
        $i = 0
        ForEach($Item in $SearchResultCollection)
              { "$i : " + $Item.Path ; $i++ }
        $choice = Read-Host "`nPlease choose which match:"
        "'" + $SearchResultCollection[$choice].Path + "'"
    }
 else
    {
        if ($ObjectName -notmatch '.+\*$')
        {
            $ObjectName += '*'
            Get-DistinguishedName -objectname $ObjectName
                    -class $Class -domain $Domain
        }
        else
        {
            "NOT_FOUND"
        }
    }
}
```

Notice in the above code that if multiple matches are found, the matches are listed and the user can choose which one to use. Also note that if a match is not found and the object name did not end with the "*" wildcard, the search will automatically be performed again with that wildcard appended to the end. If the $Domain parameter is specified, it too must be in DN format, e.g., "DC=SANS,DC=ORG", but, if left blank, it will default to whatever domain the computer running the script is a member of.

Create Global User Account

The keyword "psbase" is used in PowerShell to provide direct access to an object underlying an object-wrapper layer. Direct access to Active Directory-related objects is sometimes desired because the PowerShell developers have "wrapped" some of these objects, hiding some of their members and changing others, in an attempt to make working with the object easier or more VBScript-like. While the intention may be good, in this author's opinion it results in more long-term frustration than it's worth. Hence, whenever you see the "psbase" keyword in script code, just remember that this is not a regular property or method, it's more like a command to PowerShell to bypass the outermost wrapper and instead work directly with the raw underlying .NET object.

Note that in the code below, if you leave the $Domain argument blank, it will default to the Active Directory domain to which the present working computer belongs.

To create and enable a global user account in Active Directory:

```
# Example: Create-GlobalUser.ps1

function Create-GlobalUser ($UserName,
                            $Container = "CN=Users",
                            $Domain = "")
{
   # Get domain object specified, or the local domain if blank.
   $DirectoryEntry = new-object
              System.DirectoryServices.DirectoryEntry -arg
$Domain

   # Construct path to the container which will hold the user.
   $PathToContainer = "LDAP://$Container," +
              "$($DirectoryEntry.DistinguishedName)"

   # Get the collection of objects in the container.
   $Container = new-object
System.DirectoryServices.DirectoryEntry
              -arg $PathToContainer
   $DirectoryEntries = $Container.PSbase.Children

   # Add a user account to the container, set its username, save.
   $User = $DirectoryEntries.Add('CN=' + $UserName, 'User')
```

```
    $User.PSbase.InvokeSet('sAMAccountName', $UserName)
    $User.PSbase.CommitChanges()

    # Enable the new user account, save changes.
    $User.PSbase.InvokeSet('AccountDisabled', 'False')
    $User.PSbase.CommitChanges()

    # Tell the Garbage Collector that we're done with the object.
    $User.PSbase.Dispose()
}
```

The CommitChanges() method is similar to the SetInfo() method in VBScript: it causes the changes to be sent across the network to the domain controllers and asks the controller to write/commit the changes to the AD database. The reason it is called twice is that you must specify the mandatory properties first, call CommitChanges() to create the object, then the optional properties may be set, which then require their own CommitChanges() to actually be saved. The Dispose() call at the end isn't necessary for the function to work, but it's a bit more efficient to tell the .NET Common Language Runtime (CLR) and its garbage collector that we're done with the object, hence, its memory and any other resources can be cleaned up and reclaimed for other purposes.