# PowerShell

**RUNAS**
Starting with PowerShell 4.0, we can specify that a script requires administrative privileges by including a #Requires statement with the -RunAsAdministrator switch parameter.#Requires -RunAsAdministrator

**Run a script on a remote computer**
-- invoke-command -computername machine1, machine2 -filepath c:\Script\script.ps1

**Remotely shut down another machine after one minute**
-- Start-Sleep 60; Restart-Computer –Force –ComputerName TARGETMACHINE

**Install an MSI package on a remote computer**
-- (Get-WMIObject -ComputerName TARGETMACHINE -List | Where-Object -FilterScript {$_.Name -eq "Win32_Product"}).Install(\\MACHINEWHEREMSIRESIDES\path\package.msi)

**Upgrade an installed application with an MSI-based application upgrade package**
-- (Get-WmiObject -Class Win32_Product -ComputerName . -Filter "Name='name_of_app_to_be_upgraded'").Upgrade(\\MACHINEWHEREMSIRESIDES\path\upgrade_package.msi)

**Remove an MSI package from the current computer**
-- (Get-WmiObject -Class Win32_Product -Filter "Name='product_to_remove'" -ComputerName . ).Uninstall()

## Collecting information

**Get information about the make and model of a computer**
-- Get-WmiObject -Class Win32_ComputerSystem

**Get information about the BIOS of the current computer**
-- Get-WmiObject -Class Win32_BIOS -ComputerName .

**List installed hotfixes (QFEs, or Windows Update files)**
-- Get-WmiObject -Class Win32_QuickFixEngineering -ComputerName .

**Get the username of the person currently logged on to a computer**
-- Get-WmiObject -Class Win32_ComputerSystem -Property UserName -ComputerName .

**Find just the names of installed applications on the current computer**
-- Get-WmiObject -Class Win32_Product -ComputerName . | Format-Wide -Column 1

**Get IP addresses assigned to the current computer**
-- Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE -ComputerName . | Format-Table -Property IPAddress

**Get a more detailed IP configuration report for the current machine**
-- Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE -ComputerName . | Select-Object -Property [a-z]* -ExcludeProperty IPX*,WINS*

**To find network cards with DHCP enabled on the current computer**
-- Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "DHCPEnabled=true" -ComputerName .

**Enable DHCP on all network adapters on the current computer**
-- Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=true -ComputerName . | ForEach-Object -Process {$_.EnableDHCP()}

**Navigate the Windows Registry like the file system** -- cd hkcu:

**Search recursively for a certain string within files** -- dir –r | select string "searchforthis"

**Find the five processes using the most memory** -- ps | sort –p ws | select –last 5

**Cycle a service (stop, and then restart it) like DHCP** -- Restart-Service DHCP

**List all items within a folder** -- Get-ChildItem – Force

**Recurse over a series of directories or folders** -- Get-ChildItem –Force c:\directory –Recurse

**Remove all files within a directory without being prompted for each -- Remove-Item C:\tobedeleted –Recurse**

**Restart the current computer --** (Get-WmiObject -Class Win32_OperatingSystem -ComputerName .).Win32Shutdown(2)

## Set-ExecutionPolicy

Although you can create and execute PowerShell scripts, Microsoft has disabled scripting by default in an effort to prevent malicious code from executing in a PowerShell environment. You can use the Set-ExecutionPolicy command to control the level of security surrounding PowerShell scripts. Four levels of security are available to you:

- **Restricted** -- Restricted is the default execution policy and locks PowerShell down so that commands can be entered only interactively. PowerShell scripts are not allowed to run.
- **All Signed** -- If the execution policy is set to All Signed then scripts will be allowed to run, but only if they are signed by a trusted publisher.
- **Remote Signed** -- If the execution policy is set to Remote Signed, any PowerShell scripts that have been locally created will be allowed to run. Scripts created remotely are allowed to run only if they are signed by a trusted publisher.
- **Unrestricted** -- As the name implies, Unrestricted removes all restrictions from the execution policy.

You can set an execution policy by entering the Set-ExecutionPolicy command followed by the name of the policy. For example, if you wanted to allow scripts to run in an unrestricted manner you could type:

**Set-ExecutionPolicy Unrestricted**

## Get-ExecutionPolicy

If you're working on an unfamiliar server, you'll need to know what execution policy is in use before you attempt to run a script. You can find out by using the **Get-ExecutionPolicy** command.

## Get-Service

The **Get-Service** command provides a list of all of the services that are installed on the system. If you are interested in a specific service you can append the -Name switch and the name of the service (wildcards are permitted) When you do, Windows will show you the service's state.

## Export-CSV

Just as you can create an HTML report based on PowerShell data, you can also export data from PowerShell into a CSV file that you can open using Microsoft Excel. The syntax is similar to that of converting a command's output to HTML. At a minimum, you must provide an output filename. For example, to export the list of system services to a CSV file, you could use the following command:

**Get-Service | Export-CSV c:\service.csv**

## Select-Object

If you tried using the command above, you know that there were numerous properties included in the CSV file. It's often helpful to narrow things down by including only the properties you are really interested in. This is where the Select-Object command comes into play. The Select-Object command allows you to specify specific properties for inclusion. For example, to create a CSV file containing the name of each system service and its status, you could use the following command:

**Get-Service | Select-Object Name, Status | Export-CSV c:\service.csv**

## Get-Process

Just as you can use the Get-Service command to display a list of all of the system services, you can use the **Get-Process** command to display a list of all of the processes that are currently running on the system.

## Stop-Process

Sometimes, a process will freeze up. When this happens, you can use the Get-Process command to get the name or the process ID for the process that has stopped responding. You can then terminate the process by using the Stop-Process command. You can terminate a process based on its name or on its process ID. For example, you could terminate Notepad by using:

**Stop-Process -Name notepad**

**Stop-Process -ID 2668**

# PowerShell Active Directory

## Reset a User Password

Let's start with a typical IT pro task: resetting a user's password. We can easily accomplish this by using the Set-ADAccountPassword cmdlet. The tricky part is that the new password must be specified as a secure string: a piece of text that's encrypted and stored in memory for the duration of your PowerShell session. So first, we'll create a variable with the new password:

```
PS C:\> $new=Read-Host "Enter the new password" -AsSecureString
```

Next, we'll enter the new password:

```
PS C:\>
```

Now we can retrieve the account (using the samAccountname is best) and provide the new password. Here's the change for user Jack Frost:

```
PS C:\> Set-ADAccountPassword jfrost -NewPassword $new
```

Unfortunately, there's a bug with this cmdlet: -Passthru, -Whatif, and -Confirm don't work. If you prefer a one-line approach, try this:

```
PS C:\> Set-ADAccountPassword jfrost -NewPassword

(ConvertTo-SecureString -AsPlainText -String

"P@ssw0rd1z3" -force)
```

Finally, I need Jack to change his password at his next logon, so I'll modify the account by using Set-ADUser:

```
PS C:\> Set-ADUser jfrost -ChangePasswordAtLogon $True
```

The command doesn't write to the pipeline or console unless you use -True. But I can verify success by retrieving the username via the Get-ADUser cmdlet and specifying the PasswordExpired property, shown in Figure 2.

## Disable and Enable a User Account

Next, let's disable an account. We'll continue to pick on Jack Frost. This code takes advantage of the -Whatif parameter, which you can find on many cmdlets that change things, to verify my command without running it:

```
PS C:\> Disable-ADAccount jfrost -whatif

What if: Performing operation "Set" on Target "CN=Jack
Frost,

OU=staff,OU=Testing,DC=GLOBOMANTICS,DC=local".
```

Now to do the deed for real:

```
PS C:\> Disable-ADAccount jfrost
```

When the time comes to enable the account, can you guess the cmdlet name?

```
PS C:\> Enable-ADAccount jfrost
```

These cmdlets can be used in a pipelined expression to enable or disable as many accounts as you need. For example, this code disables all user accounts in the Sales department:

```
PS C:\> get-aduser -filter "department -eq 'sales'" |

disable-adaccount
```

## Unlock a User Account

Now, Jack has locked himself out after trying to use his new password. Rather than dig through the GUI to find his account, I can unlock it by using this simple command:

```
PS C:\> Unlock-ADAccount jfrost
```

## Delete a User Account

Deleting 1 or 100 user accounts is easy with the Remove-ADUser cmdlet. I don't want to delete Jack Frost, but if I did, I could use this code:

```
PS C:\> Remove-ADUser jfrost -whatif

What if: Performing operation "Remove" on Target

"CN=Jack
Frost,OU=staff,OU=Testing,DC=GLOBOMANTICS,DC=local".
```

Or I could pipe in a bunch of users and delete them with one simple command:

```
PS C:\> get-aduser -filter "enabled -eq 'false'"

-property WhenChanged -SearchBase "OU=Employees,

DC=Globomantics,DC=Local" | where {$_.WhenChanged

-le (Get-Date).AddDays(-180)} | Remove-ADuser -whatif
```

This one-line command would find and delete all disabled accounts in the Employees organizational unit (OU) that haven't been changed in at least 180 days.

## Add Members to a Group

Let's add Jack Frost to the Chicago IT group:

```
PS C:\> add-adgroupmember "chicago IT" -Members jfrost
```

It's that simple. You can just as easily add hundreds of users to a group, although doing so is a bit more awkward than I would like:

```
PS C:\> Add-ADGroupMember "Chicago Employees" -
member

(get-aduser -filter "city -eq 'Chicago'")
```

I used a parenthetical pipelined expression to find all users with a City property of Chicago. The code in the parentheses is executed and the resulting objects are piped to the -Member parameter. Each user object is then added to the Chicago Employees group. It doesn't matter whether there are 5 or 500 users; updating group membership takes only a few seconds This expression could also be written using ForEach-Object, which might be easier to follow.

```
PS C:\> Get-ADUser -filter "city -eq 'Chicago'" | foreach

{Add-ADGroupMember "Chicago Employees" -Member $_}
```

## Enumerate Members of a Group

You might want to see who belongs to a given group. For example, you should periodically find out who belongs to the Domain Admins group:

```
PS C:\> Get-ADGroupMember "Domain Admins"
```

The cmdlet writes an AD object for each member to the pipeline. But what about nested groups? My Chicago All Users group is a collection of nested groups. To get a list of all user accounts, all I need to do is use the -Recursive parameter:

```
PS C:\> Get-ADGroupMember "Chicago All Users"

-Recursive | Select DistinguishedName
```

## Disable a Computer Account

Perhaps when you find those inactive or obsolete accounts, you'd like to disable them. Easy enough. We'll use the same cmdlet that we use with user accounts. You can specify it by using the account's samAccountname:

```
PS C:\> Disable-ADAccount -Identity "chi-srv01$" -whatif
```

What if: Performing operation "Set" on Target "CN=CHI-SRV01,
CN=Computers,DC=GLOBOMANTICS,DC=local".

Or you can use a pipelined expression:

```
PS C:\> get-adcomputer "chi-srv01" | Disable-ADAccount
```

I can also take my code to find obsolete accounts and disable all those accounts:

```
PS C:\> get-adcomputer -filter "Passwordlastset

-lt '1/1/2012'" -properties *| Disable-ADAccount
```

## Find Computers by Type

The last task that I'm often asked about is finding computer accounts by type, such as servers or laptops. This requires a little creative thinking on your part. There's nothing in AD that distinguishes a server from a client, other than the OS. If you have a laptop or desktop running Windows Server 2008, you'll need to get extra creative.
You need to filter computer accounts based on the OS. It might be helpful to get a list of those OSs first:

```
PS C:\> Get-ADComputer -Filter * -Properties
OperatingSystem |

Select OperatingSystem -unique | Sort OperatingSystem
```

I want to find all the computers that have a server OS:

```
PS C:\> Get-ADComputer -Filter "OperatingSystem -like

'*Server*'" -properties OperatingSystem,OperatingSystem

ServicePack | Select Name,Op* | format-list
```

As with the other AD Get cmdlets, you can fine-tune your search parameters and limit your query to a specific OU if necessary. All the expressions that I've shown you can be integrated into larger PowerShell expressions. For example, you can sort, group, filter, export to a comma-separated value (CSV), or build and email an HTML report, all from PowerShell and all without writing a single PowerShell script! In fact, here's a bonus: a user password-age report, saved as an HTML file:

```
PS C:\> Get-ADUser -Filter "Enabled -eq 'True' -AND

PasswordNeverExpires -eq 'False'" -Properties

PasswordLastSet,PasswordNeverExpires,PasswordExpired
|

Select
DistinguishedName,Name,pass*,@{Name="PasswordAge"
;

Expression={(Get-Date)-$_.PasswordLastSet}} |sort

PasswordAge -Descending | ConvertTo-Html -Title

"Password Age Report" | Out-File c:\Work\pwage.htm
```