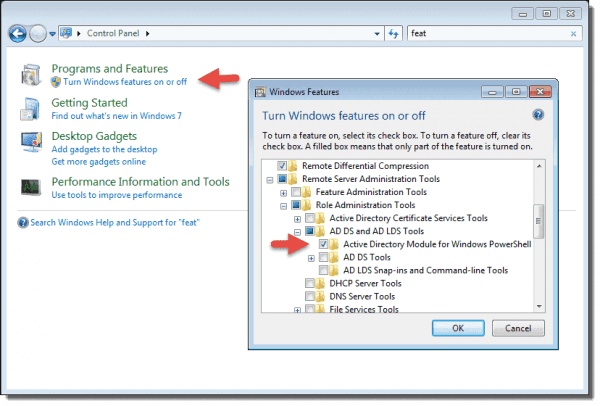
**How to install the PowerShell Active Directory module**

## Windows 7

On a Windows 7 computer, you can follow this procedure to install the Active Directory module:

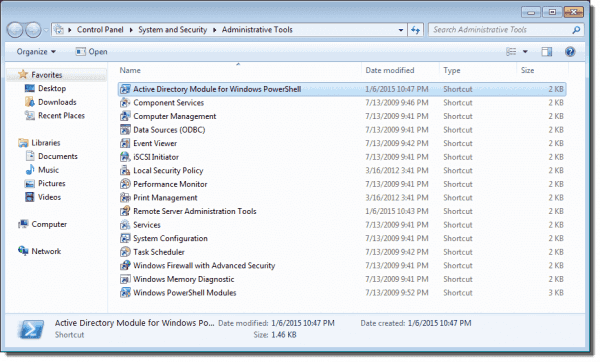
1. Download the Remote Server Administration Tools (RSAT) for Windows 7.
2. Open the **Control Panel**, start typing features, and then click Turn Windows features on or off.
3. Scroll down to Remote Server Administration Tools and enable the **Active Directory Module for Windows PowerShell** in **Remote Server Administration Tools > Role Administration Tools > AD DS and AD LDS Tools**.
4. Run Import-Module ActiveDirectory on a PowerShell console.

*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/Active-Directory-Module-for-Windows-PowerShell-on-Windows-7.png)*

*Active Directory Module for Windows PowerShell on Windows 7*

If the Windows 7 machine only has PowerShell 2.0 installed, you have to add the Import-Module ActiveDirectory command to your profile because PowerShell doesn't load modules automatically. For instance, you can import the module in %UserProfile%\My Documents\WindowsPowerShell\profile.ps1. Makes sure you've set your execution policy to either RemoteSigned or Unrestricted: Set-ExecutionPolicy RemoteSigned.

Another option is to open the module from the Administrative Tools folder in the Control Panel.

*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/Active-Directory-Module-in-Administrative-Tools.png)*

*Active Directory Module in Administrative Tools*

## Windows Server 2008 R2 ^

If your Windows Server 2008 R2 machine is a domain controller, the PowerShell Active Directory Module is already installed. You only have to install the module on member servers. The procedure on Windows Server 2008 R2 is similar to that on Windows 7. (Note that the module is not available for Windows Server 2008.)

One difference is that you don't have to download RSAT because the tools are already available on Windows Server 2008 R2.

1. In **Server Manager**, click **Add features**, and then:
2. Select **Active Directory module for Windows PowerShell** in **Remote Server Administration Tools > Role Administration Tools > AD DS and AD LDS Tools**.

Alternatively, you can install the module from a PowerShell console:

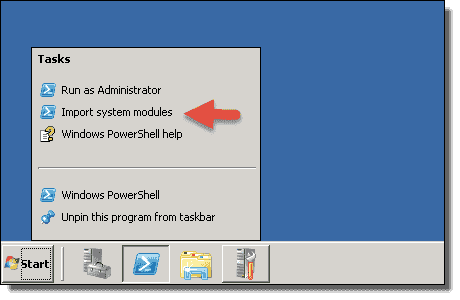
Import-Module ServerManagerAdd-WindowsFeature RSAT-AD-PowerShell

After copying the module to your computer, you have to import it:



|  |  |
| --- | --- |
| 1 | Import-Module ActiveDirectory |

Or you can right-click the PowerShell icon on the taskbar and select **Import system modules**.

*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/Import-system-modules.png)*

*Import system modules*

As on Windows 7, if you want to make the import permanent, you have to add the above import command to your PowerShell profile. Notice this description assumes you haven't updated PowerShell 2 on your Windows Server 2008 R2 machine (see the description about Windows 7).

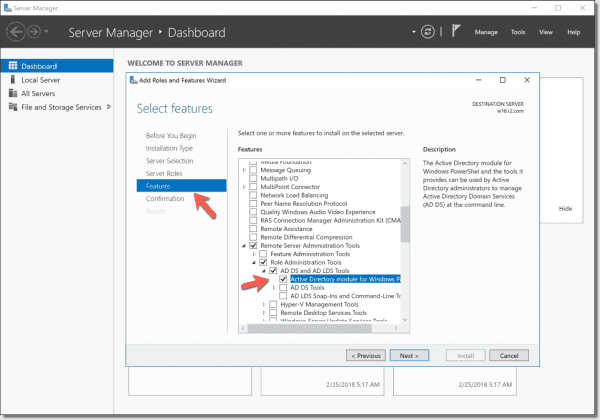
## Windows 8, Windows 8.1, Windows 10 ^

Things are a lot easier in Windows 8, Windows 8.1, and Windows 10. All you have to do is download and install RSAT (Windows 8, Windows 8.1, Windows 10). The installation enables all tools by default, and you also don't have to import the module. You can use the AD module right away after you install RSAT.

## Windows Server 2012, Windows Server 2012 R2, Windows Server 2016 ^

As on Windows Server 2008 R2, the AD module is already installed on domain controllers on Windows Server 2012, Windows Server 2012 R2, and Windows Server 2016. On member servers, you can add the module as a feature in Server Manager.

1. Start **Server Manager**.
2. Click **Manage > Add Roles and Features**.
3. Click **Next** until you reach **Features**.
4. Enable **Active Directory module for Windows PowerShell** in **Remote Server Administration Tools > Role Administration Tools > AD DS and AD LDS Tools**.

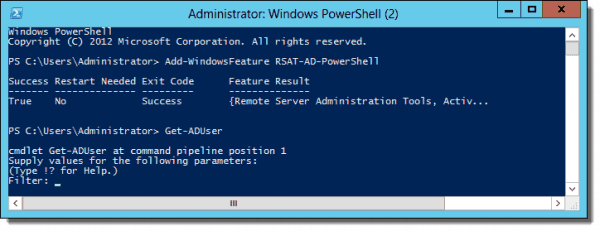
*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/Install-the-AD-module-on-Windows-Server-2016.png)*

*Install the AD module on Windows Server 2016*

Alternatively, you can install the module from a PowerShell console:



|  |  |
| --- | --- |
| 1 | Add-WindowsFeature RSAT-AD-PowerShell |

*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/Installing-the-AD-module-on-Windows-Server-2012-with-PowerShell.png)*

*Installing the AD module on Windows Server 2012 with PowerShell*

There's no need to import the Server Manager module first, as on Windows Server 2008 R2. You also don't have to import the AD module after the installation.

If you want to verify successful installation of the module, you can just run the Get-ADuser cmdlet.

## Install the AD module on PowerShell Core 6.0 ^

Currently, the AD module for Windows PowerShell does not work with PowerShell Core 6.0. If you try to import the module, you'll receive this error message:

import-module : Could not load type 'System.Management.Automation.PSSnapIn' from assembly 'System.Management.Automation, Version=6.0.1.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'.

The reason is that the module requires the .NET Framework, and PowerShell Core uses .NET Core, which is only subset of the .NET Framework.

It is unknown when Microsoft will deliver a version of the AD module that will work with PowerShell Core 6. However, you have three options to work with the AD module on PowerShell Core.

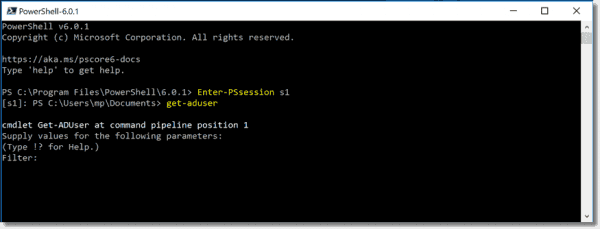
### Create an interactive remote session

The simplest option is to create an interactive remote session to your domain controller with the Enter-PSsession cmdlet:



|  |  |
| --- | --- |
| 1 | Enter-PSsession MyDomainConroller |

You can then work right away with the AD cmdlets. This option is good if you only occasionally manage AD on a PowerShell console and if you don't have to execute local scripts.

*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/Managing-Active-Directory-on-PowerShell-Core-in-an-interactive-remote-session.png)*

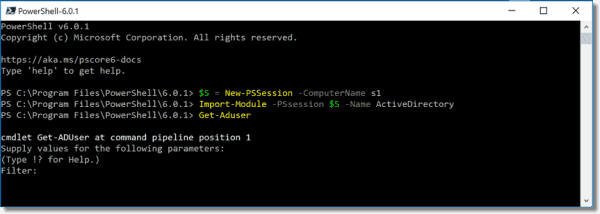
*Managing Active Directory on PowerShell Core in an interactive remote session*

### Import the AD module from a remote session

The second option uses implicit remoting and allows you to run the AD cmdlets from a local session. However, you execute the AD cmdlets remotely on a domain controller. In practice, you won't notice much of difference in locally installed cmdlets. To import the AD module on PowerShell Core 6.0, execute these commands:



|  |  |
| --- | --- |
| 1  2 | $S = New-PSSession -ComputerName MyDomainConroller  Import-Module -PSsession $S -Name ActiveDirectory |

*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/Import-the-AD-module-on-PowerShell-Core-6.0.png)*

*Import the AD module on PowerShell Core 6.0*

The first command creates a PowerShell session (PSsession) on the domain controller (replace MyDomainController with the name of your DC) and establishes a persistent connection. Next, we import the ActiveDirectory module from this remote PSsession into our local session.

You can now use all AD module cmdlets on your local PowerShell Core console. Just keep in mind the commands always execute remotely.

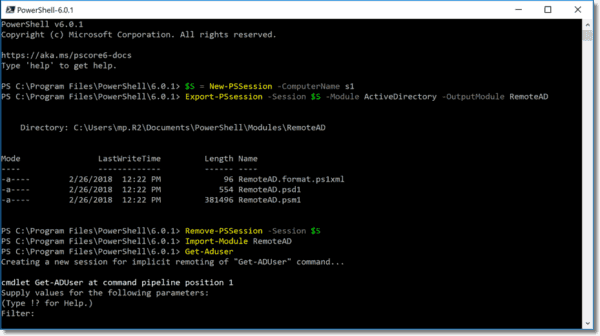
If you often work with AD, you can add the above commands to your profile, for instance in Documents\PowerShell\Profile.ps1.

### Export the remote AD module to a local module

Alternatively, you can export the AD cmdlets from a remote session to a local module:



|  |  |
| --- | --- |
| 1  2  3  4 | $S = New-PSSession -ComputerName MyDomainController  Export-PSsession -Session $S -Module ActiveDirectory -OutputModule RemoteAD  Remove-PSSession -Session $S  Import-Module RemoteAD |

*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/Exporting-the-Active-Directory-module-to-a-local-module.png)*

*Exporting the Active Directory module to a local module*

These commands will create a local module in your Documents folder under PowerShell\Modules\RemoteAD. However, like with the above solution, you will be working with implicit remoting, and all cmdlets will execute remotely. The local RemoteAD module only links to the cmdlets on the domain controller. If you want to use the RemoteAD module on other machines with PowerShell Core, simply copy the RemoteAD folder to the PowerShell Core module folder on the second machine.

The difference with the "import solution" is that in the "export solution," PowerShell only establishes a connection to the domain controller when you use an AD cmdlet the first time. You also don't have to add the above commands to your profile because PowerShell will load the local RemoteAD module automatically. However, the downside to this option is you might have to repeat the procedure after updating the AD module on the domain controller.

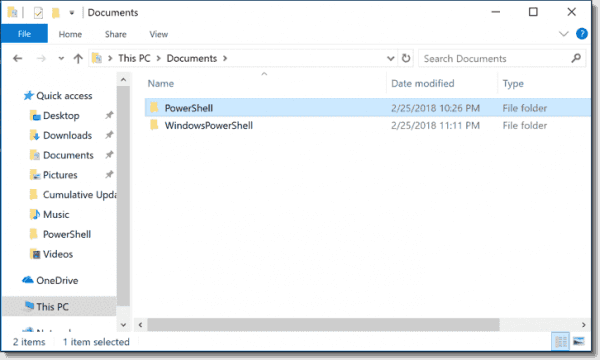
## PowerShell Core and Windows PowerShell modules ^

Note that you can use Windows PowerShell together with PowerShell Core on the same machine and work with the different AD modules in both shells. If you installed RSAT, the AD module for Windows PowerShell will reside in this folder:

$env:windir/System32/WindowsPowerShell/v1.0/Modules/ActiveDirectory

If you used the export solution, the RemoteAD module will be in this folder:

$env:userprofile/Documents/PowerShell/Modules/RemoteAD

*[](https://4sysops.com/wp-content/uploads/bp-attachments/449877/PowerShell-Core-and-Windows-PowerShell-use-different-folders.png)*

*PowerShell Core and Windows PowerShell use different folders*

PowerShell Core does not import modules in WindowsPowerShell folders, and Windows PowerShell does not load PowerShell Core modules, which are always in PowerShell folders. Thus, you don't have to worry about conflicts between the different AD modules in PowerShell Core and Windows PowerShell.

## Examples

**-------------------------- EXAMPLE 1 : Get list of user from AD--------------------------**

PowerShellCopy

C:\PS>Get-ADUser -Filter \* -SearchBase "OU=Finance,OU=UserAccounts,DC=FABRIKAM,DC=COM"

(Get-ADUser -Filter \* -SearchBase

"OU=Finance,OU=UserAccounts,DC=FABRIKAM,DC=COM").Count

Get-ADUser –Filter \* SearchBase “OU:Noida,OU:Branch1,UserType=Desktop”

Description

Get all users under the container 'OU=Finance,OU=UserAccounts,DC=FABRIKAM,DC=COM'.

**-------------------------- EXAMPLE 2 --------------------------**

PowerShellCopy

C:\PS>Get-ADUser -Filter 'Name -like "\*SvcAccount"' | FT Name,SamAccountName -A

Wild card:

\*SvcAccount : end with SvcAccount

SvcAccount \* : start with SvcAccount

\* SvcAccount\* : contains SvcAccount

Name SamAccountName

---- --------------

SQL01 SvcAccount SQL01

SQL02 SvcAccount SQL02

IIS01 SvcAccount IIS01

Description

Get all users that have a name that ends with 'SvcAccount'.

**-------------------------- EXAMPLE 3 --------------------------**

PowerShellCopy

C:\PS>Get-ADUser GlenJohn -Properties \*

Surname : John

Name : Glen John

UserPrincipalName :

GivenName : Glen

Enabled : False

SamAccountName : GlenJohn

ObjectClass : user

SID : S-1-5-21-2889043008-4136710315-2444824263-3544

ObjectGUID : e1418d64-096c-4cb0-b903-ebb66562d99d

DistinguishedName : CN=Glen John,OU=NorthAmerica,OU=Sales,OU=UserAccounts,DC=FABRIKAM,DC=COM

Description

Get all properties of the user with samAccountName 'GlenJohn'.

**-------------------------- EXAMPLE 4 --------------------------**

PowerShellCopy

C:\PS>Get-ADUser -Filter {Name -eq "GlenJohn"} -SearchBase "DC=AppNC" -Properties mail -Server lds.Fabrikam.com:50000

Description

Get the user with name 'GlenJohn' on the AD LDS instance.

## Examples Add the AD Group

**-------------------------- EXAMPLE 1 --------------------------**

PowerShellCopy

C:\PS>New-ADGroup -Name "RODC Admins"

-SamAccountName RODCAdmins -GroupCategory Security -GroupScope Global -DisplayName "RODC Administrators" -Path "CN=Users,DC=Fabrikam,DC=Com" -Description "Members of this group are RODC Administrators"

Description

Create a new group named 'RODC Admins' in the container 'CN=Users,DC=Fabrikam,DC=Com' and set the GroupCategory, DisplayName, GroupScope, and Description properties on the new object.

**-------------------------- EXAMPLE 2 --------------------------**

PowerShellCopy

C:\PS>Get-ADGroup FabrikamBranch1 -Properties Description | New-ADGroup -Name Branch1Employees -SamAccountName Branch1Employees -GroupCategory Distribution -PassThru

GroupScope : Universal

Name : Branch1Employees

GroupCategory : Distribution

SamAccountName : Branch1Employees

ObjectClass : group

ObjectGUID : 8eebce44-5df7-4bed-a98b-b987a702103e

SID : S-1-5-21-41432690-3719764436-1984117282-1117

DistinguishedName : CN=Branch1Employees,CN=Users,DC=Fabrikam,DC=com

Description

Create a new group using the property values from a current group.

**-------------------------- EXAMPLE 3 --------------------------**

PowerShellCopy

C:\PS>New-ADGroup -Server localhost:60000 -Path "OU=AccountDeptOU,DC=AppNC" -Name AccountLeads -GroupScope DomainLocal -GroupCategory Distribution

Description

Create a new group named 'AccountLeads' on an AD LDS instance.

## Examples : Create new user in AD Group

**-------------------------- EXAMPLE 1 --------------------------**

PowerShellCopy

C:\PS>New-ADUser GlenJohn -Certificate (new-object System.Security.Cryptography.X509Certificates.X509Certificate -ArgumentList "export.cer")

Description

Create a new user named 'GlenJohn' with a certicate imported from the file "export.cer".

**-------------------------- EXAMPLE 2 --------------------------**

PowerShellCopy

C:\PS>New-ADUser GlenJohn -OtherAttributes @{title="director";mail="glenjohn@fabrikam.com"}

Description

Create a new user named 'GlenJohn' and set the title and mail properties on the new object.

**-------------------------- EXAMPLE 3 --------------------------**

PowerShellCopy

C:\PS>New-ADUser GlenJohn -Type iNetOrgPerson -Path "DC=AppNC" -server lds.Fabrikam.com:50000

Description

Create a new inetOrgPerson named 'GlenJohn' on an AD LDS instance.

## Examples: Remove Group

**-------------------------- EXAMPLE 1 --------------------------**

PowerShellCopy

C:\PS>remove-adgroup SanjaysReports

Confirm

Are you sure you want to perform this action?

Performing operation "Remove" on Target "CN=SanjayReports,DC=Fabrikam,DC=com".

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):

Description

Remove the group that has samAccountName 'SanjaysReports'.

**-------------------------- EXAMPLE 2 --------------------------**

PowerShellCopy

C:\PS>get-adgroup -filter 'Name -like "Sanjay\*"' | remove-adgroup

Confirm

Are you sure you want to perform this action?

Performing operation "Remove" on Target "CN=SanjaysReports,DC=Fabrikam,DC=com".

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):

Description

Get all groups whose name starts with 'Sanjay' and then remove them.

## Examples: Remove User

**-------------------------- EXAMPLE 1 --------------------------**

PowerShellCopy

C:\PS>Remove-ADUser -Identity GlenJohn

Description

Remove the user with samAccountName 'GlenJohn'.

**-------------------------- EXAMPLE 2 --------------------------**

PowerShellCopy

C:\PS>Search-ADAccount -AccountDisabled | where {$\_.ObjectClass -eq 'user'} | Remove-ADUser

Description

Search for any users that have disabled accounts and remove them.

**-------------------------- EXAMPLE 3 --------------------------**

PowerShellCopy

C:\PS>Remove-ADUser -Identity "CN=Glen John,OU=Finance,OU=UserAccounts,DC=FABRIKAM,DC=COM"

Description

Remove the user with DistinguishedName 'CN=Glen John,OU=Finance,OU=UserAccounts,DC=FABRIKAM,DC=COM'.

**-------------------------- EXAMPLE 4 --------------------------**

PowerShellCopy

C:\PS>Get-ADUser "cn=glenjohn,dc=appnc" -Server Lds.Fabrikam.com:50000 | Remove-ADUser

Description

Get the user with DistinguishedName 'cn=glenjohn,dc=appnc' from the AD LDS instance and remove it.

User Modified

# Powershell script to get the list of users from Active Directory which were modified in last 24 hrs

Get-QADUser -SearchRoot 'ou=,dc=,dc=com'-IncludedProperties \* |

Where{$\_.WhenChanged -gt [datetime]::Now.AddHours(-24)} |

select EmployeeID,SamAccountName,mail,ipPhone,whenChanged,ChangedBy,

Get-QADUser -SearchRoot 'ou=,dc=,dc=com'-IncludedProperties \* |

Where{$\_.PasswordChanged -gt [datetime]::Now.AddHours(-24)} |

select EmployeeID,SamAccountName,mail,ipPhone,whenChanged,ChangedBy,

Get-QADUser -SearchRoot 'ou=,dc=,dc=com'-IncludedProperties \* |

Where{$\_.WhenCreated -gt [datetime]::Now.AddHours(-24)} |

select EmployeeID,SamAccountName,mail,ipPhone,whenChanged,ChangedBy,

Get-ADUser and LastModified date from their home directory

$Results = @()

$users = Get-ADUSer -filter \* -Properties

scriptpath,homedrive,homedirectory | Select Name,displayname,scriptpath,homedrive,homedirectory,distinguishedname

Foreach($user in $users){

$lastmod = (gci $user.homedirectory -recurse | sort

LastWriteTime | select -last 1).LastWriteTime

$Results += New-Object PSObject @{

Username = $user.samaccountname

HomeDirectory = $user.homedirectory

LastModified = $lastMod

}

}

$Results | export-csv C:\File.csv –NoType

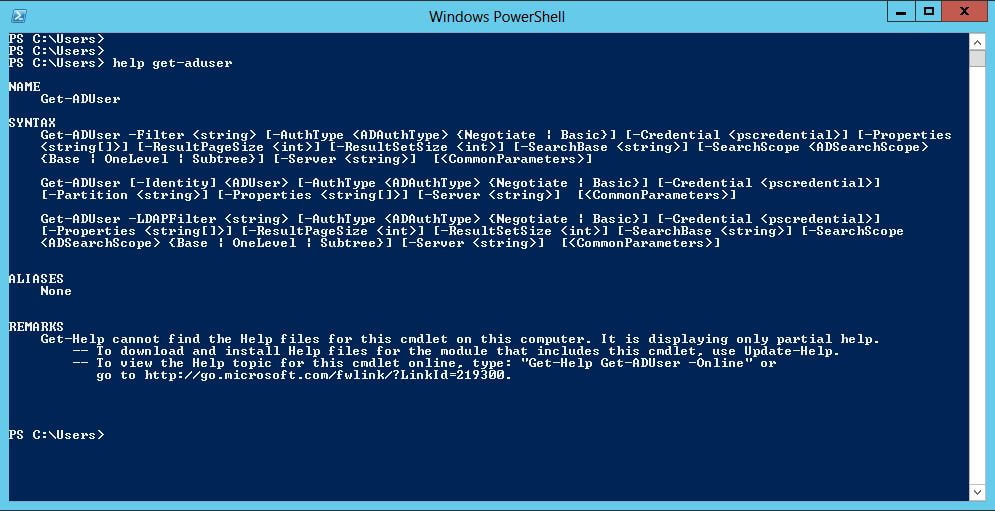
# PowerShell: How to use Get-ADUser to list all recently created accounts (and recently changed accounts)

For the next couple of posts I’ll be looking into AD security and auditing. In this article we’ll look at how to use Get-ADUser to list all recently created accounts.

With the increasing number of cyber attacks, security is at the top of most IT departments agenda. There are many checks you can perform to make sure AD is safe and secure, and that only valid or approved modifications have been made to user accounts. I’ll look at AD auditing in a future post, but this will be a handy snippet of PowerShell to help you identify recently created AD accounts, and a bonus bit of code to identify recently modified accounts!

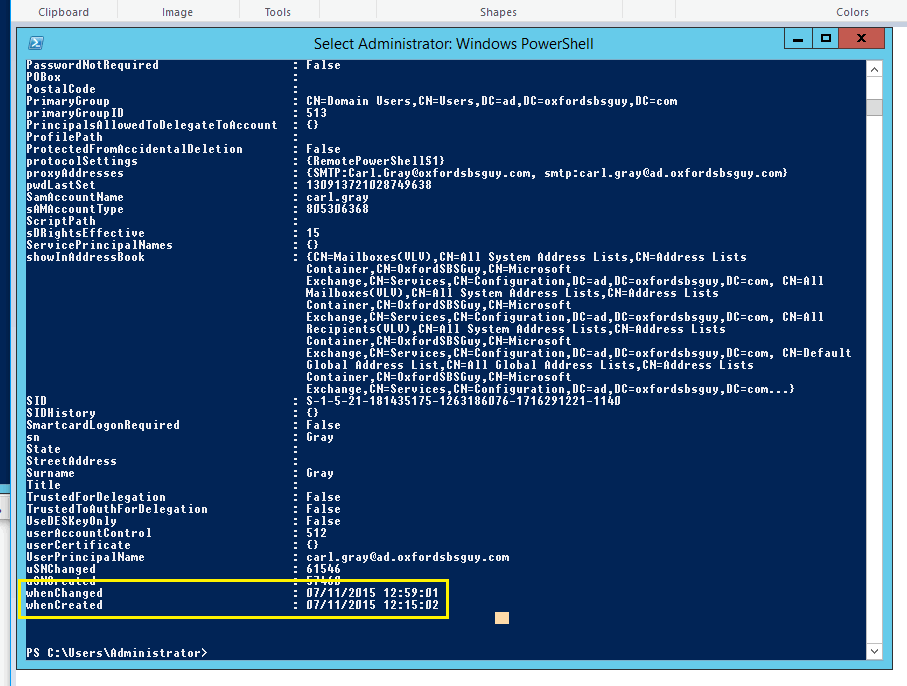
As a quick recap, to view the available options with Get-ADUser type:

**help Get-ADUser**

[](https://www.oxfordsbsguy.com/wp-content/uploads/2013/11/help-get-aduser.jpg)

Next we want to find out the full list of properties Get-ADUser can give us so we can identify the specific property to search for. Pick a user at random and type:

**Get-ADUser -identity *username* -property \***

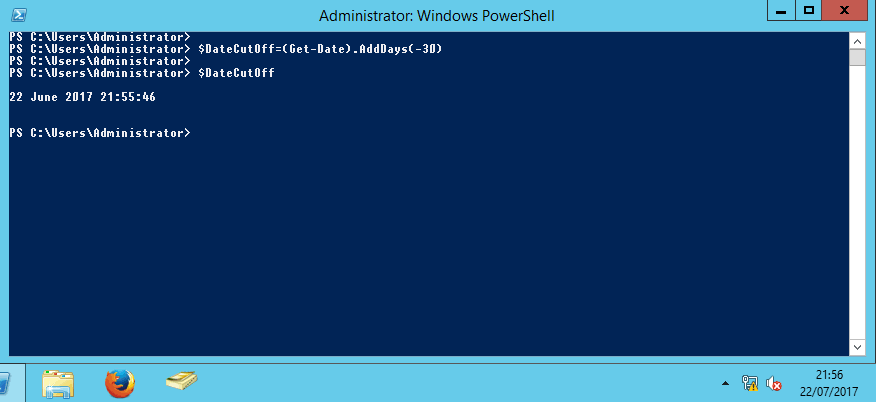
[](https://www.oxfordsbsguy.com/wp-content/uploads/2017/07/get-aduser-identity-username-property-star-whenCreated-whenChanged.png)

Looking through the  properties you can see the very last two, **whenChanged** and **whenCreated**. These are the two properties we’ll be working with

Now we need to do some date manipulation to feed into the Get-ADUser cmdlet so that we can list only accounts created within xx days. To do this we’ll use the variable **$DateCutOff**, along with the **Get-Date**cmdlet and **AddDays**. Type:

**$DateCutOff=(Get-Date).AddDays(-30)**

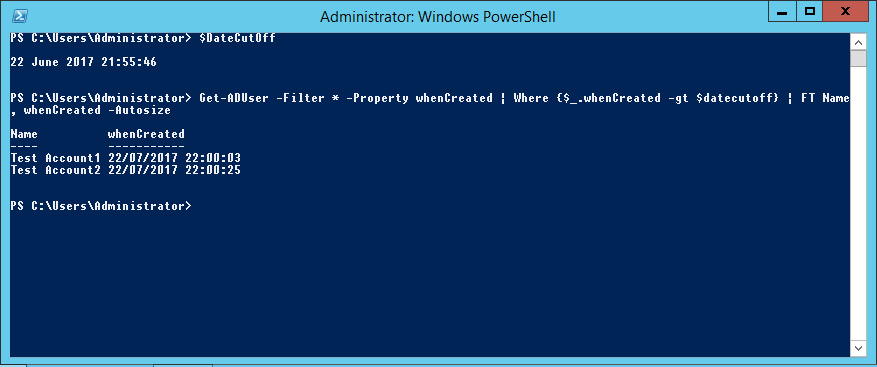
This will take today’s date and effectively add -30 days to it.

[](https://www.oxfordsbsguy.com/wp-content/uploads/2017/07/datecutoff-equals-get-date-adddays-minus30.png)

So you can see today is 22nd July, but the **$DateCutOff** variable is 22nd June. Now we can use this along with the **Where-Object** cmdlet on the **whenCreated** property.

Type:

**Get-ADUser -Filter \* -Property whenCreated | Where {$\_.whenCreated -gt $datecutoff} | FT Name, whenCreated -Autosize**

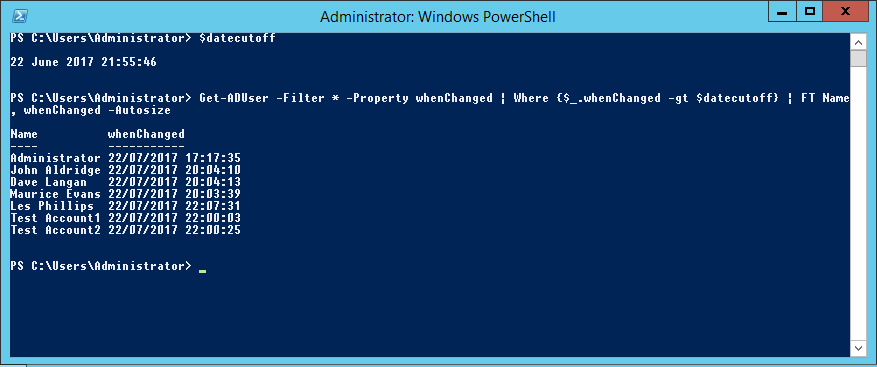
[](https://www.oxfordsbsguy.com/wp-content/uploads/2017/07/get-aduser-filter-star-property-whencreated-pipe-where-whencreated-greaterthan-datecutoff-ft-name-whencreate-autosize.png)

So above you can see two test accounts have been created in the last 30 days.

Now if we change the whenCreated property for the whenChanged property, we can see all accounts that have been modified in the last 30 days.

Type:

**Get-ADUser -Filter \* -Property whenChanged | Where {$\_.whenChanged -gt $datecutoff} | FT Name, whenChanged -Autosize**

[](https://www.oxfordsbsguy.com/wp-content/uploads/2017/07/get-aduser-filter-star-property-whenchanged-pipe-where-whenchanged-greaterthan-datecutoff-ft-name-whenchanged-autosize.png)

Here you can see all the accounts that have been modified in the last 30 days.

Out-File

## Examples

**Example 1: Send output to a file**

PowerShell Copy

PS C:\> Get-Process | Out-File -filepath C:\Test1\process.txt

This command sends a list of processes on the computer to the Process.txt file. If the file does not exist, Out-File creates it. Because the name of the FilePath parameter is optional, you can omit it and submit the equivalent command "Get-Process | Out-File C:\Test1\process.txt".

**Example 2: Send output to a file without overwriting**

PowerShell Copy

PS C:\> Get-Process | Out-File C:\Test1\process.txt -NoClobber

Out-File : File C:\Test1\process.txt already exists and NoClobber was specified.

At line:1 char:23

+ Get-Process | Out-File <<<< process.txt -NoClobber

This command also sends a list of processes to the Process.txt file, but it uses the NoClobber parameter, which prevents an existing file from being overwritten. The output shows the error message that appears when NoClobber is used with an existing file.

**Example 3: Send output to a file in ASCII format**

PowerShell Copy

PS C:\> $A = Get-Process

PS C:\> Out-File -FilePath C:\Test1\process.txt -InputObject $A -Encoding ASCII -Width 50

These commands send a list of processes on the computer to the Process.txt file. The text is encoded in ASCII format so that it can be read by search programs like Findstr and Grep. By default, Out-File uses Unicode format.

The first command gets the list of processes and stores them in the $A variable. The second command uses the Out-File cmdlet to send the list to the Process.txt file.

The command uses the InputObject parameter to specify that the input is in the $A variable. It uses the Encoding parameter to convert the output to ASCII format. It uses the Width parameter to limit each line in the file to 50 characters. Because the lines of output are truncated at 50 characters, the rightmost column in the process table is omitted.

**Example 4: Send output from outside a file system drive**

PowerShell Copy

PS C:\> Set-Location hklm:\software

PS C:\> Get-Acl mycompany\mykey | Out-File -FilePath c:\ps\acl.txt

PS C:\> Get-Acl mycompany\mykey | Out-File -FilePath filesystem::acl.txt

These commands show how to use the Out-File cmdlet when you are not in a FileSystem drive.

The first command sets the current location to the HKLM:\Software registry key.

The second and third commands have the same effect. They use the 1Get-Acl1 cmdlet to get the security descriptor of the MyKey registry subkey (HKLM\Software\MyCompany\MyKey). A pipeline operator passes the result to the Out-File cmdlet, which sends it to the Acl.txt file.

Because Out-File is not supported by the PowerShell Registry provider, you must specify either the file system drive name, such as "c:", or the name of the provider followed by two colons, "FileSystem::", in the value of the FilePath parameter. The second and third commands demonstrate these methods.

## Examples

**Example 1: Output processes to a grid view**

PowerShell Copy

PS C:\> Get-Process | Out-GridView

This command gets the processes running on the local computer and sends them to a grid view window.

**Example 2: Use a variable to output processes to a grid view**

PowerShell Copy

PS C:\> $P = Get-Process

PS C:\> $P | Out-GridView

This command also gets the processes running on the local computer and sends them to a grid view window.

The first command uses the Get-Process cmdlet to get the processes on the computer and then saves the process objects in the $P variable.

The second command uses a pipeline operator to send the $P variable to **Out-GridView**.

**Example 3: Display a formatted table in a grid view**

PowerShell Copy

PS C:\> Get-Process | Select-Object -Property Name, WorkingSet, PeakWorkingSet | Sort-Object -Property WorkingSet -Descending | Out-GridView

This command displays a formatted table in a grid view window.

It uses the Get-Process cmdlet to get the processes on the computer.

Then, it uses a pipeline operator (|) to send the process objects to the Select-Object cmdlet. The command uses the **Property** parameter of **Select-Object** to select the Name, WorkingSet, and PeakWorkingSet properties to be displayed in the table.

Another pipeline operator sends the filtered objects to the Sort-Object cmdlet, which sorts them in descending order by the value of the **WorkingSet** property.

The final part of the command uses a pipeline operator (|) to send the formatted table to **Out-GridView**.

You can now use the features of the grid view to search, sort, and filter the data.

**Example 4: Save output to a variable, and then output a grid view**

PowerShell Copy

PS C:\> ($A = Get-ChildItem -Path $pshome -Recurse) | Out-GridView

This command saves its output in a variable and sends it to **Out-GridView**.

The command uses the Get-ChildItem cmdlet to get the files in the Windows PowerShell installation directory and its subdirectories. The path to the installation directory is saved in the $pshome automatic variable.

The command uses the assignment operator (=) to save the output in the $A variable and the pipeline operator (|) to send the output to **Out-GridView**.

The parentheses in the command establish the order of operations. As a result, the output from the Get-ChildItem command is saved in the $A variable before it is sent to **Out-GridView**.

**Example 5: Output processes for a specified computer to a grid view**

PowerShell Copy

PS C:\> Get-Process -ComputerName "Server01" | ogv -Title "Processes - Server01"

This command displays the processes that are running on the Server01 computer in a grid view window.

The command uses ogv, which is the built-in alias for the **Out-GridView** cmdlet, it uses the Title parameter to specify the window title.

**Example 6: Output data from remote computers to a grid view**

PowerShell Copy

PS C:\> Invoke-Command -ComputerName S1, S2, S3 -ScriptBlock {Get-Culture} | Out-GridView

This example shows the correct format for sending data collected from remote computers to the **Out-GridView** cmdlet.

The command uses the Invoke-Command cmdlet to run a Get-Culture command on three remote computers. It uses a pipeline operator to send the data that is returned to the **Out-GridView** cmdlet.

Notice that the script block that contains the commands that are run remotely does not include the **Out-GridView** command. If it did, the command would fail when it tried to open a grid view window on each of the remote computers.

**Example 7: Pass multiple items through Out-GridView**

PowerShell Copy

PS C:\> Get-Process | Out-GridView -PassThru | Export-Csv -Path .\ProcessLog.csv

This command lets you select multiple processes from the **Out-GridView** window. The processes that you select are passed to the **Export-Csv** command and written to the ProcessLog.csv file.

The command uses the PassThru parameter of **Out-GridView**, which lets you send multiple items down the pipeline. The PassThru parameter is equivalent to using the Multiple value of the OutputMode parameter.

**Example 8: Create a Windows shortcut to Out-GridView**

PowerShell Copy

PS C:\> Powershell.exe -Command "Get-Service | Out-GridView -Wait"

This command shows how to use the Wait parameter of **Out-GridView** to create a Windows shortcut to the **Out-GridView** window. Without the Wait parameter, Windows PowerShell would exit as soon as the **Out-GridView** window opened, which would close the **Out-GridView** window almost immediately.

## Examples

**Example 1: Export process properties to a CSV file**

This example selects **Process** objects with specific properties, exports the objects to a CSV file.

PowerShell Copy

Get-Process -Name WmiPrvSE | Select-Object -Property BasePriority,Id,SessionId,WorkingSet |

Export-Csv -Path .\WmiData.csv -NoTypeInformation

Import-Csv -Path .\WmiData.csv

BasePriority Id SessionId WorkingSet

------------ -- --------- ----------

8 976 0 20267008

8 2292 0 36786176

8 3816 0 30351360

8 8604 0 15011840

8 10008 0 8830976

8 11764 0 14237696

8 54632 0 9502720

The Get-Process cmdlet gets the **Process** objects. The **Name** parameter filters the output to include only the WmiPrvSE process objects. The process objects are sent down the pipeline to the Select-Object cmdlet. Select-Object uses the **Property** parameter to select a subset of process object properties. The process objects are sent down the pipeline to the Export-Csv cmdlet. Export-Csv converts the process objects to a series of CSV strings. The **Path** parameter specifies that the WmiData.csv file is saved in the current directory. The **NoTypeInformation** parameter removes the **#TYPE** information header from the CSV output and is not required in PowerShell 6. The Import-Csv cmdlet uses the **Path** parameter to display the file located in the current directory.

**Example 2: Export processes to a comma-delimited file**

This example gets **Process** objects and exports the objects to a CSV file.

PowerShell Copy

Get-Process | Export-Csv -Path .\Processes.csv -NoTypeInformation

Get-Content -Path .\Processes.csv

"Name","SI","Handles","VM","WS","PM","NPM","Path","Parent","Company","CPU","FileVersion", ...

"ApplicationFrameHost","4","511","2203597099008","35364864","21979136","30048", ...

The Get-Process cmdlet gets **Process** objects. The process objects are sent down the pipeline to the Export-Csv cmdlet. Export-Csv converts the process objects to a series of CSV strings. The **Path** parameter specifies that the Processes.csv file is saved in the current directory. The **NoTypeInformation** parameter removes the **#TYPE** information header from the CSV output and is not required in PowerShell 6. The Get-Content cmdlet uses the **Path** parameter to display the file located in the current directory.

**Example 3: Export processes to a semicolon delimited file**

This example gets **Process** objects and exports the objects to a file with a semicolon delimiter.

PowerShell Copy

Get-Process | Export-Csv -Path .\Processes.csv -Delimiter ';' -NoTypeInformation

Get-Content -Path .\Processes.csv

"Name";"SI";"Handles";"VM";"WS";"PM";"NPM";"Path";"Parent";"Company";"CPU";"FileVersion"; ...

"ApplicationFrameHost";"4";"509";"2203595321344";"34807808";"21770240";"29504"; ...

The Get-Process cmdlet gets **Process** objects. The process objects are sent down the pipeline to the Export-Csv cmdlet. Export-Csv converts the process objects to a series of CSV strings. The **Path** parameter specifies that the Processes.csv file is saved in the current directory. The **Delimiter** parameter specifies a semicolon to separate the string values. The **NoTypeInformation** parameter removes the **#TYPE** information header from the CSV output and is not required in PowerShell 6. The Get-Content cmdlet uses the **Path** parameter to display the file located in the current directory.

**Example 4: Export processes using the current culture's list separator**

This example gets **Process** objects and exports the objects to a file. The delimiter is the current culture's list separator.

PowerShell Copy

(Get-Culture).TextInfo.ListSeparator

Get-Process | Export-Csv -Path .\Processes.csv -UseCulture -NoTypeInformation

Get-Content -Path .\Processes.csv

"Name","SI","Handles","VM","WS","PM","NPM","Path","Parent","Company","CPU","FileVersion", ...

"ApplicationFrameHost","4","511","2203597099008","35364864","21979136","30048", ...

The Get-Culture cmdlet uses the nested properties **TextInfo** and **ListSeparator** and displays the current culture's default list separator. The Get-Process cmdlet gets **Process** objects. The process objects are sent down the pipeline to the Export-Csv cmdlet. Export-Csv converts the process objects to a series of CSV strings. The **Path** parameter specifies that the Processes.csv file is saved in the current directory. The **UseCulture** parameter uses the current culture's default list separator as the delimiter. The **NoTypeInformation** parameter removes the **#TYPE** information header from the CSV output and is not required in PowerShell 6. The Get-Content cmdlet uses the **Path** parameter to display the file located in the current directory.

**Example 5: Export processes with type information**

This example explains how to include the **#TYPE** header information in a CSV file. The **#TYPE** header is the default in versions prior to PowerShell 6.0.

PowerShell Copy

Get-Process | Export-Csv -Path .\Processes.csv -IncludeTypeInformation

Get-Content -Path .\Processes.csv

#TYPE System.Diagnostics.Process

"Name","SI","Handles","VM","WS","PM","NPM","Path","Company","CPU","FileVersion", ...

"ApplicationFrameHost","4","507","2203595001856","35139584","20934656","29504", ...

The Get-Process cmdlet gets **Process** objects. The process objects are sent down the pipeline to the Export-Csv cmdlet. Export-Csv converts the process objects to a series of CSV strings. The **Path** parameter specifies that the Processes.csv file is saved in the current directory. The **IncludeTypeInformation** includes the **#TYPE** information header in the CSV output. The Get-Content cmdlet uses the **Path** parameter to display the file located in the current directory.

**Example 6: Export and append objects to a CSV file**

This example describes how to export objects to a CSV file and use the **Append** parameter to add objects to an existing file.

PowerShell Copy

$AppService = (Get-Service -DisplayName \*Application\* | Select-Object -Property DisplayName, Status)

$AppService | Export-Csv -Path .\Services.Csv -NoTypeInformation

Get-Content -Path .\Services.Csv

$WinService = (Get-Service -DisplayName \*Windows\* | Select-Object -Property DisplayName, Status)

$WinService | Export-Csv -Path ./Services.csv -NoTypeInformation -Append

Get-Content -Path .\Services.Csv

"DisplayName","Status"

"Application Layer Gateway Service","Stopped"

"Application Identity","Running"

"Windows Audio Endpoint Builder","Running"

"Windows Audio","Running"

"Windows Event Log","Running"

The Get-Service cmdlet gets service objects. The **DisplayName** parameter returns services that contain the word Application. The service objects are sent down the pipeline to the Select-Object cmdlet. Select-Object uses the **Property** parameter to specify the **DisplayName** and **Status** properties. The $AppService variable stores the objects.

The $AppService objects are sent down the pipeline to the Export-Csv cmdlet. Export-Csv converts the service objects to a series of CSV strings. The **Path** parameter specifies that the Services.csv file is saved in the current directory. The **NoTypeInformation** parameter removes the **#TYPE** information header from the CSV output and is not required in PowerShell 6. The Get-Content cmdlet uses the **Path** parameter to display the file located in the current directory.

The Get-Service and Select-Object cmdlets are repeated for services that contain the word Windows. The $WinService variable stores the service objects. The Export-Csv cmdlet uses the **Append** parameter to specify that the $WinService objects are added to the existing Services.csv file. The Get-Content cmdlet is repeated to display the updated file that includes the appended data.

**Example 7: Format cmdlet within a pipeline creates unexpected results**

This example shows why it is important not to use a format cmdlet within a pipeline. When unexpected output is received, troubleshoot the pipeline syntax.

PowerShell Copy

Get-Date | Select-Object -Property DateTime, Day, DayOfWeek, DayOfYear |

Export-Csv -Path .\DateTime.csv -NoTypeInformation

Get-Content -Path .\DateTime.csv

"DateTime","Day","DayOfWeek","DayOfYear"

"Wednesday, January 2, 2019 14:59:34","2","Wednesday","2"

Get-Date | Format-Table -Property DateTime, Day, DayOfWeek, DayOfYear |

Export-Csv -Path .\FTDateTime.csv -NoTypeInformation

Get-Content -Path .\FTDateTime.csv

"ClassId2e4f51ef21dd47e99d3c952918aff9cd","pageHeaderEntry","pageFooterEntry","autosizeInfo", ...

"033ecb2bc07a4d43b5ef94ed5a35d280",,,,"Microsoft.PowerShell.Commands.Internal.Format. ...

"9e210fe47d09416682b841769c78b8a3",,,,,

"27c87ef9bbda4f709f6b4002fa4af63c",,,,,

"4ec4f0187cb04f4cb6973460dfe252df",,,,,

"cf522b78d86c486691226b40aa69e95c",,,,,

The Get-Date cmdlet gets the **DateTime** object. The object is sent down the pipeline to the Select-Object cmdlet. Select-Object uses the **Property** parameter to select a subset of object properties. The object is sent down the pipeline to the Export-Csv cmdlet. Export-Csv converts the object to a CSV format. The **Path** parameter specifies that the DateTime.csv file is saved in the current directory. The **NoTypeInformation** parameter removes the **#TYPE** information header from the CSV output and is not required in PowerShell 6. The Get-Content cmdlet uses the **Path** parameter to display the CSV file located in the current directory.

When the Format-Table cmdlet is used within the pipeline to select properties unexpected results are received. Format-Table sends table format objects down the pipeline to the Export-Csv cmdlet rather than the **DateTime** object. Export-Csv converts the table format objects to a series of CSV strings. The Get-Content cmdlet displays the CSV file which contains the table format objects.

**Example 8: Using the Force parameter to overwrite read-only files**

This example creates an empty, read-only file and uses the **Force** parameter to update the file.

PowerShell Copy

New-Item -Path .\ReadOnly.csv -ItemType File

Set-ItemProperty -Path .\ReadOnly.csv -Name IsReadOnly -Value $true

Get-Process | Export-Csv -Path .\ReadOnly.csv -NoTypeInformation

Export-Csv : Access to the path 'C:\ReadOnly.csv' is denied.

At line:1 char:15

+ Get-Process | Export-Csv -Path .\ReadOnly.csv -NoTypeInformation

+ ~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : OpenError: (:) [Export-Csv], UnauthorizedAccessException

+ FullyQualifiedErrorId : FileOpenFailure,Microsoft.PowerShell.Commands.ExportCsvCommand

Get-Process | Export-Csv -Path .\ReadOnly.csv -NoTypeInformation -Force

Get-Content -Path .\ReadOnly.csv

"Name";"SI";"Handles";"VM";"WS";"PM";"NPM";"Path";"Parent";"Company";"CPU";"FileVersion"; ...

"ApplicationFrameHost";"4";"509";"2203595321344";"34807808";"21770240";"29504"; ...

The New-Item cmdlet uses the **Path** and **ItemType** parameters to create the ReadOnly.csv file in the current directory. The Set-ItemProperty cmdlet uses the **Name** and **Value** parameters to change the file's **IsReadOnly** property to true. The Get-Process cmdlet gets **Process** objects. The process objects are sent down the pipeline to the Export-Csv cmdlet. Export-Csv converts the process objects to a series of CSV strings. The **Path** parameter specifies that the ReadOnly.csv file is saved in the current directory. The **NoTypeInformation** parameter removes the **#TYPE** information header from the CSV output and is not required in PowerShell 6. The output shows that the file is not written because access is denied.

The **Force** parameter is added to the Export-Csv cmdlet to force the export to write to the file. The Get-Content cmdlet uses the **Path** parameter to display the file located in the current directory.

**Example 9: Using the Force parameter with Append**

This example shows how to use the **Force** and **Append** parameters. When these parameters are combined, mismatched object properties can be written to a CSV file.

PowerShell Copy

$Content = [PSCustomObject]@{Name = 'PowerShell Core'; Version = '6.0'}

$Content | Export-Csv -Path .\ParmFile.csv -NoTypeInformation

$AdditionalContent = [PSCustomObject]@{Name = 'Windows PowerShell'; Edition = 'Desktop'}

$AdditionalContent | Export-Csv -Path .\ParmFile.csv -NoTypeInformation -Append

Export-Csv : Cannot append CSV content to the following file: ParmFile.csv.

The appended object does not have a property that corresponds to the following column:

Version. To continue with mismatched properties, add the -Force parameter, and then retry

the command.

At line:1 char:22

+ $AdditionalContent | Export-Csv -Path .\ParmFile.csv -NoTypeInformation -Append

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidData: (Version:String) [Export-Csv], InvalidOperationException

+ FullyQualifiedErrorId : CannotAppendCsvWithMismatchedPropertyNames,Microsoft.PowerShell. ...

$AdditionalContent | Export-Csv -Path .\ParmFile.csv -NoTypeInformation -Append -Force

Import-Csv -Path .\ParmFile.csv

Name Version

---- -------

PowerShell Core 6.0

Windows PowerShell

An expression creates the **PSCustomObject** with **Name** and **Version** properties. The values are stored in the $Content variable. The $Content variable is sent down the pipeline to the Export-Csv cmdlet. Export-Csv uses the **Path** parameter and saves the ParmFile.csv file in the current directory. The **NoTypeInformation** parameter removes the **#TYPE** information header from the CSV output and is not required in PowerShell 6.

Another expression creates a **PSCustomObject** with the **Name** and **Edition** properties. The values are stored in the $AdditionalContent variable. The $AdditionalContent variable is sent down the pipeline to the Export-Csv cmdlet. The **Append** parameter is used to add the data to the file. The append fails because there is a property name mismatch between **Version** and **Edition**.

The Export-Csv cmdlet **Force** parameter is used to force the export to write to the file. The **Edition** property is discarded. The Import-Csv cmdlet uses the **Path** parameter to display the file located in the current directory.