Windows Management Framework 5.0

Preview February 2015  
Release Notes

# Overview

Windows Management Framework (WMF) 5.0 Preview February 2015 contains functionality that has been updated from WMF 4.0. WMF 5.0 Preview February 2015 is available for installation on Windows 8.1,Windows Server 2012 R2 and Windows Server 2012, and contains updated versions of the following features:

* Windows PowerShell
* Windows PowerShell Integrated Scripting Environment (ISE)
* Windows PowerShell Web Services (Management OData IIS Extension)
* Windows PowerShell Desired State Configuration (DSC)
* Windows Remote Management (WinRM)
* Windows Management Instrumentation (WMI)

The functionality within WMF 5.0 Preview February 2015 has been categorized into those features with an “*experimental design”* and those with a “*stable design”*.

The goal of features with “experimental designs” is to get feedback about their usage. Your feedback is used to improve both design and implementation of experimental design features. This means that the feature design might change in future releases, which can break existing scripts. Nothing that is still in the experimental design phase should be used in production.

WMF 5.0 Preview elements designated as having “stable designs” are supported through a fix-forward approach, and might be used for both testing and production purposes. This means we do not expect to release patches, but rather that we will provide fixes in the next release of WMF 5.0 Preview. We encourage feedback on the stable design elements as well as the experimental.

Users should be sure to back up their environments when they run WMF 5.0 Preview February 2015 in production.

# Requirements

WMF 5.0 Preview February 2015 can be installed only on the following operating systems:

|  |  |  |
| --- | --- | --- |
| Operating System | Service Pack Level | Editions |
| Windows 8.1 |  | All |
| Windows Server 2012 R2 |  | All except IA64 |
| Windows Server 2012 |  | All except IA64 |

Earlier releases of WMF 5.0 Preview (such as the September 2014 or November 2014 releases) are not required to install WMF 5.0 Preview February 2015.

It is not necessary to uninstall any software before installing WMF 5.0 Preview February 2015.

Install the latest Windows updates before installing WMF 5.0 Preview February 2015.

# How to install and uninstall WMF 5.0 Preview February 2015

## Installation instructions

### Windows 8.1, Windows Server 2012 R2, and Windows Server 2012

Ensure all of the following prerequisites have been met:

* **Windows 8.1**, **Windows Server 2012 R2**,or **Windows Server 2012**
* **Windows PowerShell 2.0 (if applicable)**
* Windows PowerShell 2.0 is disabled by default on Windows Server 2012 R2 Server Core. It can be enabled by running the following two commands:

dism /online /enable-feature:MicrosoftWindowsPowerShellV2

dism /online /enable-feature:MicrosoftWindowsPowerShellV2-WOW64

Installation:

* **Double-click** the MSU file to start installation, or run the MSU file directly from **Command Prompt**.

## Uninstallation instructions

### By using Control Panel

1. Open **Control Panel**.
2. Open **Programs**, then open **Uninstall a program**.
3. Click **View installed updates**.
4. Select **Windows Management Framework 5.0 Preview February 2015** from the list of installed updates. This corresponds to KB3037315 or KB3037667. Click **Uninstall.**

### By using Command Prompt

1. Open **Command Prompt**.
2. Run the following command:

wusa /uninstall /kb:3037315

or

wusa /uninstall /kb:3037667

# New scenarios enabled by WMF 5.0 Preview February 2015

The following table provides an overview of the new and updated scenarios in WMF 5.0 Preview February 2015.

|  |  |
| --- | --- |
| **Scenario** | **Design Status** |
| Develop DSC resources with classes in Windows PowerShell | Stable |
| Remove DSC documents delivered to a system | Stable |
| Support for inheritance with classes in Windows PowerShell | Experimental |
| DSC resource script debugging | Experimental |
| Support for new RefreshMode | Experimental |
| Partial configurations support mixed RefreshModes | Experimental |
| PSScriptAnalyzer: static code analysis of Windows PowerShell artifacts | Experimental |

The following table identifies scenarios that were available in older WMF 5.0 preview releases, but that are not currently available.

|  |  |
| --- | --- |
| **Scenario** | **Design Status** |
| Obtaining results from a previously run configuration (Connect-DscConfiguration) | Not Available in February WMF |

The following table identifies scenarios from an older release that are now considered to have stable designs.

|  |  |
| --- | --- |
| **Scenario** | **Design Status** |
| 32-bit support for the configuration keyword in DSC | Stable |
| Generate Windows PowerShell cmdlets based on an OData endpoint with ODataUtils | Stable |
| Manage .ZIP archives through new cmdlets | Stable |
| Audit Windows PowerShell usage by transcription and logging | Stable |
| Interact with symbolic links using improved Item cmdlets | Stable |
| Network Switch management with Windows PowerShell | Stable |
| DSC authoring improvements in Windows PowerShell ISE | Stable |

To see the complete list of scenarios enabled in older releases (and their design statuses), see the section, [Scenarios enabled by a previous release of WMF 5.0 Preview](#_Scenarios_enabled_by).

## Develop DSC resources with classes in Windows PowerShell

We’ve made improvements to the Windows PowerShell language for defining classes and other user-defined types. The goal is to enable developers and IT professionals to embrace Windows PowerShell for a wider range of use cases, simplify development of Windows PowerShell artifacts (such as DSC resources), and accelerate coverage of management surfaces.

### Supported scenarios in this release

* Define DSC resources and their associated types by using the Windows PowerShell language.
* Define custom types in Windows PowerShell by using familiar object-oriented programming constructs, such as classes, properties, methods, etc.
* Debug types by using the Windows PowerShell language.
* Generate and handle exceptions by using formal mechanisms, and at the right level.

### Define DSC resources with classes

Based on feedback, we’ve made some changes to make authoring class-defined DSC resources simpler and easier to understand. The major differences between a class-defined DSC resource and a cmdlet DSC resource provider are:

* A MOF file for the schema is not required.
* A **DSCResource** subfolder in the module folder is not required.
* A Windows PowerShell module file can contain multiple DSC resource classes.

The following is an example of a class-defined DSC resource provider; this is saved as a module, **MyDSCResource.psm1**. Note that you must always include at least one key property in a class-defined DSC resource provider.

enum Ensure

{

Absent

Present

}

<#

This resource manages the file in a specific path.

[DscResource()] indicates the class is a DSC resource

#>

[DscResource()]

class FileResource

{

<#

This property is the fully qualified path to the file that is

expected to be present or absent.

The [DscProperty(Key)] attribute indicates the property is a

key and its value uniquely identifies a resource instance.

Defining this attribute also means the property is required

and DSC will ensure a value is set before calling the resource.

A DSC resource must define at least one key property.

#>

[DscProperty(Key)]

[string]$Path

<#

This property indicates if the settings should be present or absent

on the system. For present, the resource ensures the file pointed

to by $Path exists. For absent, it ensures the file pointed to by

$Path does not exist.

The [DscProperty(Mandatory)] attribute indicates the property is

required and DSC will guarantee it is set.

If Mandatory is not specified or if it is defined as

Mandatory=$false, the value is not guaranteed to be set when DSC

calls the resource. This is appropriate for optional properties.

#>

[DscProperty(Mandatory)]

[Ensure] $Ensure

<#

This property defines the fully qualified path to a file that will

be placed on the system if $Ensure = Present and $Path does not

exist.

NOTE: This property is required because [DscProperty(Mandatory)] is

set.

#>

[DscProperty(Mandatory)]

[string] $SourcePath

<#

This property reports the file's create timestamp.

[DscProperty(NotConfigurable)] attribute indicates the property is

not configurable in a DSC configuration. Properties marked this way

are populated by the Get() method to report additional details

about the resource when it is present.

#>

[DscProperty(NotConfigurable)]

[Nullable[datetime]] $CreationTime

<#

This method is equivalent to the Set-TargetResource script function.

It sets the resource to the desired state.

#>

[void] Set()

{

$fileExists = $this.TestFilePath($this.Path)

if($this.ensure -eq [Ensure]::Present)

{

if(-not $fileExists)

{

$this.CopyFile()

}

}

else

{

if($fileExists)

{

Write-Verbose -Message "Deleting the file $($this.Path)"

Remove-Item -LiteralPath $this.Path -Force

}

}

}

<#

This method is equivalent to the Test-TargetResource script function.

It should return True or False, showing whether the resource

is in a desired state.

#>

[bool] Test()

{

$present = $this.TestFilePath($this.Path)

if($this.Ensure -eq [Ensure]::Present)

{

return $present

}

else

{

return -not $present

}

}

<#

This method is equivalent to the Get-TargetResource script function.

The implementation should use the keys to find appropriate resources.

This method returns an instance of this class with the updated key

properties.

#>

[FileResource] Get()

{

$present = $this.TestFilePath($this.Path)

if ($present)

{

$file = Get-ChildItem -LiteralPath $this.Path

$this.CreationTime = $file.CreationTime

$this.Ensure = [Ensure]::Present

}

else

{

$this.CreationTime = $null

$this.Ensure = [Ensure]::Absent

}

return $this

}

<#

Helper method to check if the file exists and is the right file

#>

[bool] TestFilePath([string] $location)

{

$present = $true

$item = Get-ChildItem -LiteralPath $location -ea Ignore

if ($item -eq $null)

{

$present = $false

}

elseif( $item.PSProvider.Name -ne "FileSystem")

{

throw "Path $($location) is not a file path."

}

elseif($item.PSIsContainer)

{

throw "Path $($location) is a directory path."

}

return $present

}

<#

Helper method to copy file from source to path

#>

[void] CopyFile()

{

if(-not $this.TestFilePath($this.SourcePath))

{

throw "SourcePath $($this.SourcePath) is not found."

}

[System.IO.FileInfo] $destFileInfo = new-object System.IO.FileInfo($this.Path)

if (-not $destFileInfo.Directory.Exists)

{

Write-Verbose -Message "Creating directory $($destFileInfo.Directory.FullName)"

# use CreateDirectory instead of New-Item to avoid code

# to handle the non-terminating error

[System.IO.Directory]::CreateDirectory($destFileInfo.Directory.FullName)

}

if(Test-Path -LiteralPath $this.Path -PathType Container)

{

throw "Path $($this.Path) is a directory path"

}

Write-Verbose -Message "Copying $($this.SourcePath) to $($this.Path)"

#DSC engine catches and reports any error that occurs

Copy-Item -LiteralPath $this.SourcePath -Destination $this.Path -Force

}

}

After creating the class-defined DSC resource provider, and saving it as a module, create a module manifest for the module. In this example, the following module manifest is saved as **MyDscResource.psd1**.

@{

# Script module or binary module file associated with this manifest.

RootModule = 'MyDscResource.psm1'

# Version number of this module.

ModuleVersion = '1.0'

# ID used to uniquely identify this module

GUID = '81624038-5e71-40f8-8905-b1a87afe22d7'

# Author of this module

Author = 'User01'

# Company or vendor of this module

CompanyName = 'Unknown'

# Copyright statement for this module

Copyright = '(c) 2015 User01. All rights reserved.'

# Description of the functionality provided by this module

Description = 'DSC resource provider for FileResource.'

# Minimum version of the Windows PowerShell engine required by this module

PowerShellVersion = '5.0'

# Name of the Windows PowerShell host required by this module

# PowerShellHostName = ''

# Required for DSC to detect PS class-based resources.

DscResourcesToExport = 'FileResource'

}

Deploy the new DSC resource provider by creating a **MyDscResource** folder for it in $pshome\Modules or $env:SystemDrive\Program Files\WindowsPowerShell\Modules. You do not need to create a DSCResource subfolder. Copy the module and module manifest files (**MyDscResource.psm1** and **MyDscResource.psd1**) to the **MyDscResource** folder.

From this point, you create and run a configuration script as you would with any DSC resource. The following is a configuration that references the MyDSCResource module. Save this as a script, **MyResource.ps1**.

Configuration MyConfig

{

Import-Dscresource -ModuleName MyDscResource

FileResource file

{

Path = "C:\thisisatest\te.txt"

SourcePath = "c:\\test.txt"

Ensure = "Present"

}

}

MyConfig

Run this as you would any DSC configuration script. To start the configuration, in an elevated Windows PowerShell console, run the following.

PS C:\test> .\MyResource.ps1

PS C:\test> Start-DscConfiguration c:\test\MyConfig –wait –verbose

### Known issues

In this release, the following are known issues with class-defined DSC resource providers.

* The DSC engine caches resources that are implemented as Windows PowerShell modules for efficiency purposes. When you are authoring and testing a resource simultaneously, set the value of **DebugMode** on the DSC Local Configuration Manager to **All** to force the engine to reload the DSC resource.

## Support for Inheritance with classes in Windows PowerShell

WMF 5.0 Preview February 2015 introduces support for inheritance in classes.

**Note:** In the following examples, keep in mind that class definition scopes are local. Although this should not be a problem if you are working with a script file, if you are working in REPL (i.e. powershell\_ise command line), you cannot declare a class in powershell\_ise REPL in one line, and then use the class in another line. This results in an “Unable to find type” error. It will work fine, however, if you copy the whole example in powershell\_ise.

### Declare base classes for Windows PowerShell classes

You can declare a Windows PowerShell class as a base type for another Windows PowerShell class.

class bar

{

[int]foo() {return 100500}

}

class baz : bar {}

[baz]::new().foo() # return 100500

You can also use existing .NET Framework types as base classes.

class MyIntList : system.collections.generic.list[int]

{

}

$list = [MyIntList]::new()

$list.Add(100)

$list[0] # return 100

### Declare implemented interfaces for Windows PowerShell classes

You can declare implemented interfaces after base types, or immediately after a colon (:), if there is no base type specified. Separate all type names by using commas. It’s very similar to C# syntax.

class MyComparable : system.IComparable

{

[int] CompareTo([object] $obj)

{

return 0;

}

}

class MyComparableBar : bar, system.IComparable

{

[int] CompareTo([object] $obj)

{

return 0;

}

}

### Call base class constructor

To call a base class constructor from a subclass, use the keyword **base**.

class A {

[int]$a

A([int]$a)

{

$this.a = $a

}

}

class B : A

{

B() : base(103) {}

}

[B]::new().a # return 103

If a base class has a default (parameterless) constructor, you can omit an explicit constructor call.

class C : B

{

C([int]$c) {}

}

### Call base class method

You can override existing methods in subclasses. To do this, declare methods with same name and signature.

class baseClass

{

[int]foo() {return 100500}

}

class childClass1 : baseClass

{

[int]foo() {return 200600}

}

[childClass1]::new().foo() # return 200600

To call base class methods from overridden implementations, cast to the base class ([baseclass]$this) on invocation.

class childClass2 : baseClass

{

[int]foo()

{

return 3 \* ([baseClass]$this).foo()

}

}

[childClass2]::new().foo() # return 301500

All Windows PowerShell methods are virtual. You can hide non-virtual .NET methods in a subclass by using the same syntax as you do for an override: just declare methods with same name and signature.

class MyIntList : system.collections.generic.list[int]

{

# Add is final in system.collections.generic.list

[void] Add([int]$arg)

{

([system.collections.generic.list[int]]$this).Add($arg \* 2)

}

}

$list = [MyIntList]::new()

$list.Add(100)

$list[0] # return 200

### Known issues

In this release, the following are known issues of the inheritance feature of classes:

* You cannot use the syntax ([baseclass]$this).protectedFoo() to call a protected method that is defined in the .NET class baseclass. This problem doesn’t occur when the baseclass is a Windows PowerShell class, because all Windows PowerShell class methods are public.
* Currently there is no syntax to declare interfaces in Windows PowerShell.
* Class-based DSC resouces don’t support inheritance in this release.

## DSC resource script debugging

WMF 5.0 Preview February 2015 includes new support for debugging DSC resource scripts while they run on target machines. In older WMF5 releases, we added advanced script debugging features with the ability to attach to local processes (Get-PSHostProcessInfo, Enter-PSHostProcessInfo, Exit-PSHostProcessInfo), enumerate all runspaces in a process, and debug an arbitrary runspace in a process (Get-Runspace, Debug-Runspace).

DSC resource script debugging builds on this work by extending the DSC LCM configuration metadata’s **DebugMode** setting to include a new **ResourceScriptBreakAll** value. The **ResourceScriptBreakAll** debug mode configures the DSC LCM to run Get-TargetResource, Set-TargetResource, and Test-TargetResource scripts in Break All mode, meaning that the script stops at the first script statement and waits for a debugger to be attached. You can then connect to the target computer by using Windows PowerShell remoting, and attach the Windows PowerShell debugger to the LCM process and runspace to debug the script. After you do this, you can set breakpoints and debug the script in the normal Windows PowerShell way.

A typical DSC resource script debugging session goes as follows:

[DscLocalConfigurationManager()]

Configuration ConfigureLCMForDebug

{

Node "MGMT-10001-827"

{

Settings

{

DebugMode = "ResourceScriptBreakAll"

}

}

}

ConfigureLCMForDebug

PS C:\test> Set-DscLocalConfigurationManager .\ConfigureLCMForDebug

PS C:\test> Get-DscLocalConfigurationManager

ActionAfterReboot : ContinueConfiguration

AllowModuleOverWrite : False

CertificateID :

ConfigurationDownloadManagers : {}

ConfigurationID :

ConfigurationMode : ApplyAndMonitor

ConfigurationModeFrequencyMins : 15

Credential :

DebugMode : {ResourceScriptBreakAll}

...

PS C:\test> Start-DscConfiguration .\TestConfig2 -wait -verbose –force

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, ''methodName' =

SendConfigurationApply,'className' = MSFT\_DSCLocalConfigurationManager,'namespaceName' =

root/Microsoft/Windows/DesiredStateConfiguration'.

VERBOSE: An LCM method call arrived from computer MGMT-10005-1146 with user sid

S-1-5-21-2127521184-1604012920-1887927527-6287655.

VERBOSE: [MGMT-10001-827]: LCM: [ Start Set ]

WARNING: [MGMT-10001-827]: [DSCEngine] Warning LCM is in Debug 'ResourceScriptBreakAll'

mode. Resource script processing will be stopped to wait for PowerShell script debugger to attach.

VERBOSE: [MGMT-10001-827]: LCM: [ Start Resource ] [[INIMgmt]MyIniMgmt]

VERBOSE: [MGMT-10001-827]: LCM: [ Start Test ] [[INIMgmt]MyIniMgmt]

WARNING: [MGMT-10001-827]: [[INIMgmt]MyIniMgmt] Resource is waiting for PowerShell script

debugger to attach. Use the following commands to begin debugging this resource script:

Enter-PSSession -ComputerName MGMT-10001-827 -Credential <credentials>

Enter-PSHostProcess -Id 2640 -AppDomainName DscPsPluginWkr\_AppDomain

Debug-Runspace -Id 3

By running the commands shown in the warning message after running Start-DscConfiguration, you attach your client Windows PowerShell script debugger to the appropriate computer, process, application domain and runspace.

# Run this command to optionally use Windows PowerShell remoting to connect to the target computer. Skip this if you already have a remote desktop connection.

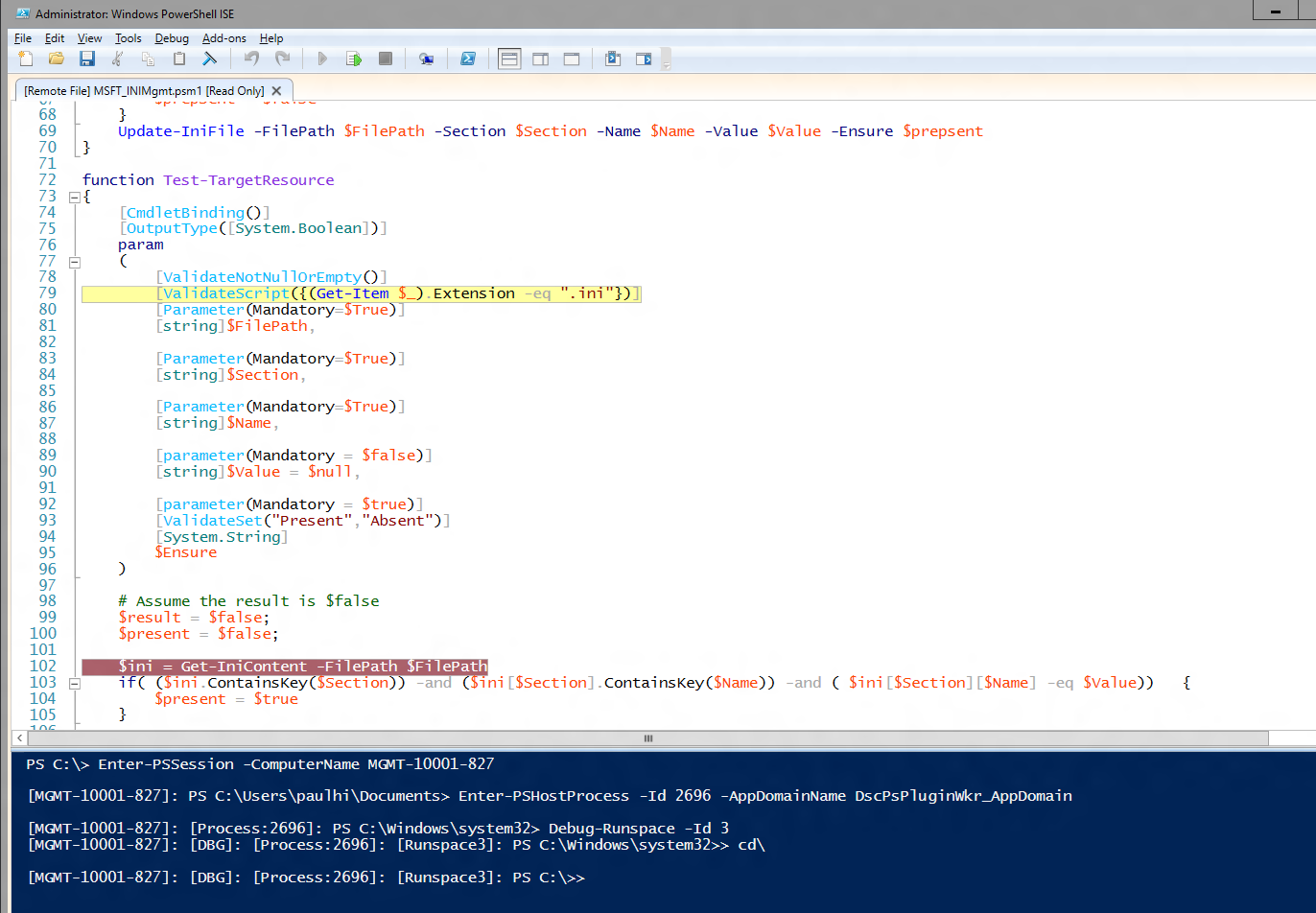
Enter-PSSession -ComputerName MGMT-10001-827

# This command attaches to the DSC LCM host process and app domain where the resource script is running.

Enter-PSHostProcess -Id 2640 -AppDomainName DscPsPluginWkr\_AppDomain

# This last command lets you debug the DSC runspace running the script.

Debug-Runspace -Id 3



After you have the debugger attached, set line breakpoints where you want to investigate script execution and run the **continue** debugger command to let the script run in the debugger. When you are done debugging, you can stop the script from running by typing the **quit** debugger command, or let the script continue to run without the debugger by running the **detach** debugger command.

Notice that with the LCM in **ResourceScriptBreakAll** DebugMode, ***all*** resource scripts are stopped in the debugger. This means that Test-TargetResoruce, Set-TargetResource, and Get-TargetResource scripts will each in turn stop in the debugger. If you don't want to debug a resource script, you can exit debug mode in the runspace by running this command.

Disable-RunspaceDebug -RunspaceId 3

Or, you can attach the debugger by running Debug-Runspace, and then immediately run the **detach** command.

After you are done debugging your resource script, you should stop the DSC configuration by running the following command.

Stop-DscConfiguration –force

Finally, you must reconfigure the target computer LCM debug mode to change the **DebugMode == ResourceScriptBreakAll** setting.

PS C:\test> Set-DscLocalConfigurationManager .\ResetLCMDebug

## PSScriptAnalyzer: static code analysis of Windows PowerShell artifacts

WMF 5.0 Preview February 2015 includes the addition of a new Windows PowerShell Module called PSScriptAnalyzer. This is a static checker for Windows PowerShell artifacts (modules and scripts) and is installed in $env:ProgramFiles/WindowsPowerShell/Modules.

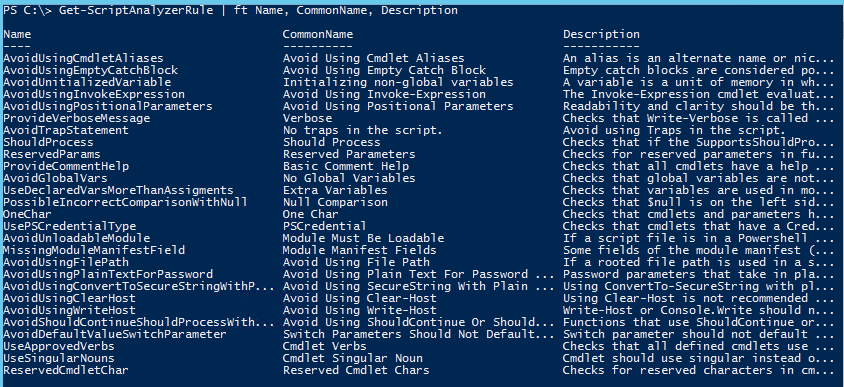


When you use PSScriptAnalyzer, you are applying a list of rules to specified Windows PowerShell artifacts to check for rule violations. PSScriptAnalyzer generates DiagnosticResults (errors and warnings) to inform users about potential code defects in their Windows PowerShell artifacts, and suggests possible solutions for improvements.

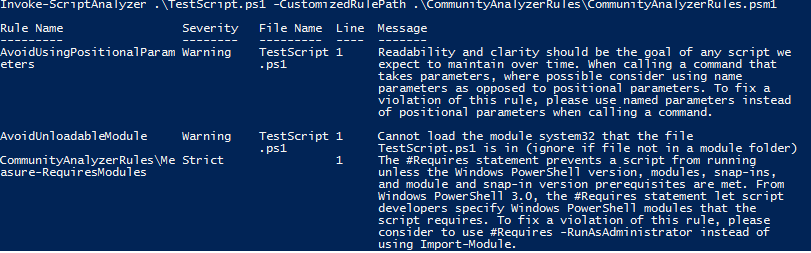
You can choose which rules to include or exclude. You can also write your own rules and import them into PSScriptAnalyzer’s collection of rules.

### Scenarios

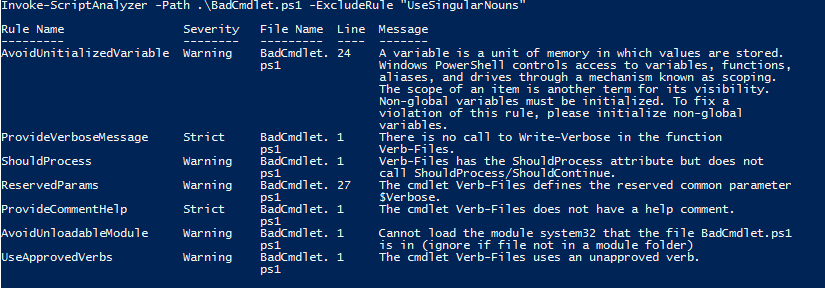
* The following screenshot shows the rules that are included by default, and their descriptions.



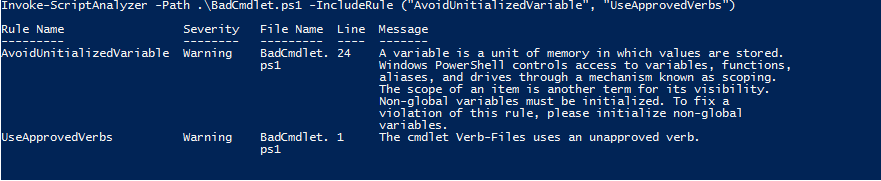
* PSScriptAnalyzer can consume user-provided rules (authored as scripts or compiled assembly – C#)



* The following shows how you exclude rules, by adding the ExcludeRule parameter:



* You can also opt to run only a specific set of rules, by adding the IncludeRule parameter:



## Support for new RefreshMode value

This release introduces a new RefreshMode value, **Disabled**. When this mode is set, LCM does not do document management, and any third party can invoke DSC resources directly (by running the Invoke-DscResource cmdlet).

Configuration LCMSettings

{

    Node localhost

    {

        LocalConfigurationManager

        {

           RefreshMode = 'Disabled'

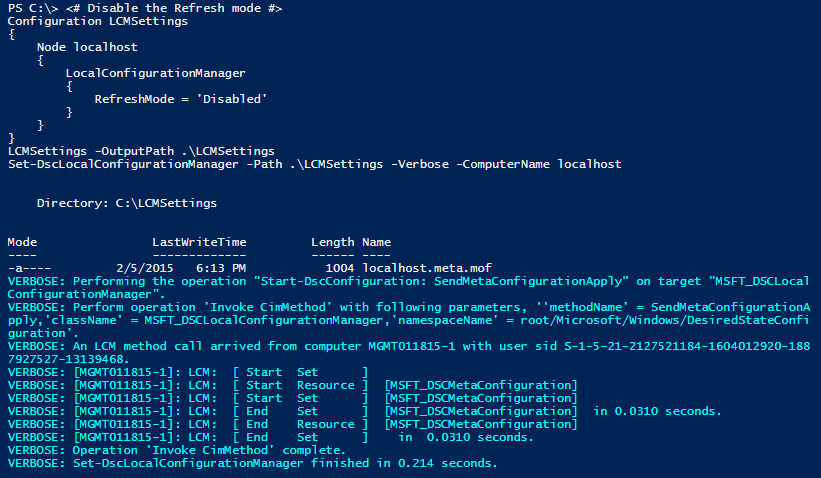
        }

    }

}

LCMSettings -OutputPath .\LCMSettings

Set-DscLocalConfigurationManager -Path .\LCMSettings -Verbose -ComputerName localhost



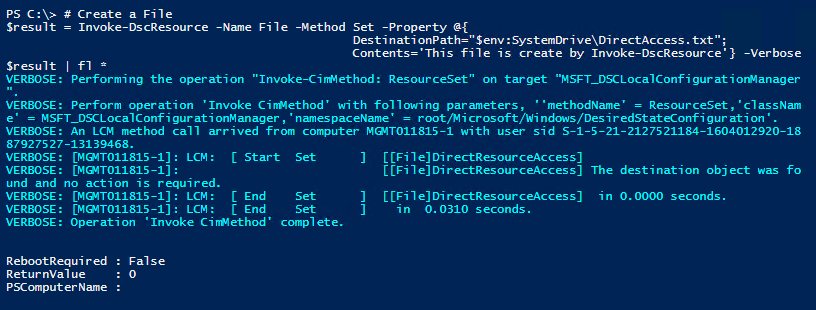
# Create a File

$result = Invoke-DscResource -Name File -Method Set -Property @{

                                DestinationPath="$env:SystemDrive\DirectAccess.txt";

                                Contents='This file is create by Invoke-DscResource'} -Verbose

$result | fl \*



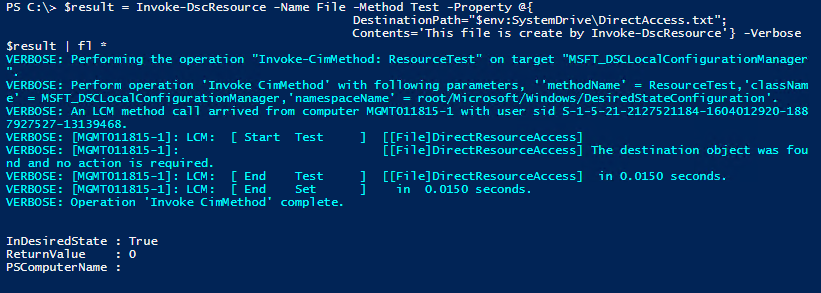
# Test the resource state

$result = Invoke-DscResource -Name File -Method Test -Property @{

                                 DestinationPath="$env:SystemDrive\DirectAccess.txt";

                                 Contents='This file is create by Invoke-DscResource'} -Verbose

$result | fl \*



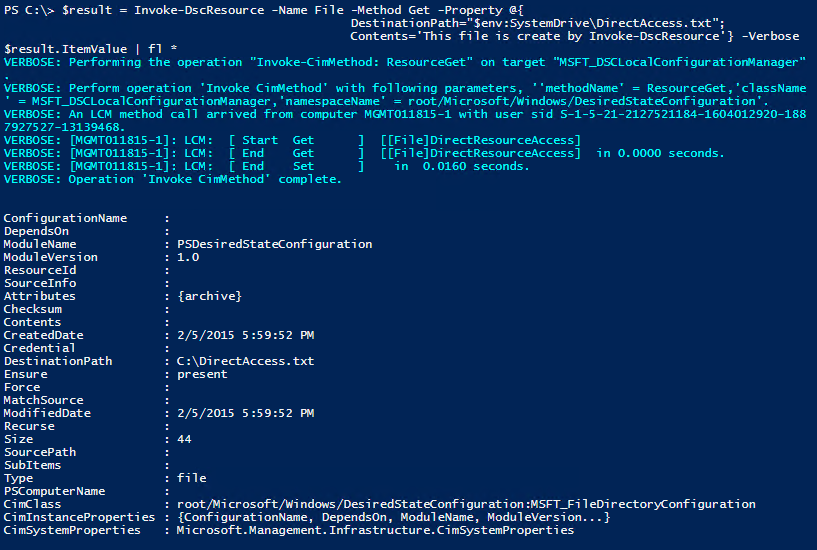
# Get the contents of File

$result = Invoke-DscResource -Name File -Method Get -Property @{

                                DestinationPath="$env:SystemDrive\DirectAccess.txt";

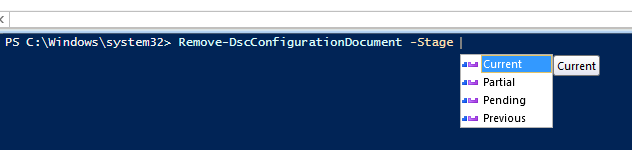
                                Contents='This file is create by Invoke-DscResource'} -Verbose

$result.ItemValue | fl \*



## Remove DSC documents delivered to a system

When a configuration document is delivered to DSC, it goes through various stages (pending, current, previous). When partial configurations are delivered, they are also stored in the system. We added a new cmdlet, Remove-DscConfigurationDocument, as part of KB3000850. This cmdlet is now available in WMF 5.0 Preview February 2015.



## Partial configurations support mixed RefreshModes

When you use partial configurations, you can now define each individual partial configuration with a RefreshMode. Valid values are **PUSH** or **PULL**.

[DscLocalConfigurationManager()]

Configuration partialMeta

{

Settings

{

RefreshMode = "PULL"

RefreshFrequencyMins = 30

ConfigurationModeFrequencyMins = 15

ConfigurationMode = "ApplyAndAutoCorrect"

RebootNodeIfNeeded = $false

ConfigurationId = "922A3987-6647-4665-B770-0FD008436CE3"

}

PartialConfiguration Partial1

{

RefreshMode = "PUSH"

Description = "Partial1"

}

PartialConfiguration Partial2

{

RefreshMode = "PULL"

Description = "Partial1"

ConfigurationSource = "[ConfigurationRepositoryShare]FileShare"

DependsOn = "[PartialConfiguration]Partial1"

}

PartialConfiguration Partial3

{

RefreshMode = "PUSH"

Description = "Partial3"

ConfigurationSource = "[ConfigurationRepositoryWeb]PullServerWeb1"

DependsOn = "[PartialConfiguration]Partial2"

}

ConfigurationRepositoryShare FileShare

{

SourcePath = "\\testserver\pullserver"

}

}

# Scenarios enabled by a previous release of WMF 5.0 Preview

The following table provides an overview of existing scenarios last modified in previous releases of WMF 5.0 Preview.

|  |  |
| --- | --- |
| **Scenario** | **Design Status** |
| Generate Windows PowerShell cmdlets based on an OData endpoint with ODataUtils | Stable |
| Manage .ZIP archives through new cmdlets | Stable |
| Interact with symbolic links using improved Item cmdlets | Stable |
| Network Switch management with Windows PowerShell | Stable |
| DSC authoring improvements in Windows PowerShell ISE | Stable |
| 32-bit support for the configuration keyword in DSC | Stable |
| Audit Windows PowerShell usage by transcription and logging | Stable |
| Extract and parse structured object out of string content | Experimental |
| Configure DSC’s Local Configuration Manager with the meta-configuration attribute | Experimental |
| Configure piece by piece with partial configurations in DSC | Experimental |
| Manage with cross machine dependencies in DSC | Experimental |
| More control over configurations in DSC | Experimental |
| Find more detail about configuration status in DSC | Experimental |
| Discover and install software with OneGet | Experimental |
| Discover modules and DSC resources with PowerShellGet | Experimental |
| Develop with classes in Windows PowerShell | Experimental |
| More control and remoting in Windows PowerShell Debugging | Experimental |
| Report configuration status from DSC to a central location | Experimental |

## Generate Windows PowerShell cmdlets based on an OData endpoint

Export-ODataEndpointProxy is a cmdlet that generates a set of Windows PowerShell cmdlets based on the functionality exposed by a given OData endpoint.

The following example shows how to use this new cmdlet:

# Basic use case of Export-ODataEndpoint Proxy

Export-ODataEndpointProxy -Uri 'http://services.odata.org/v3/(S(snyobsk1hhutkb2yulwldgf1))/odata/odata.svc' -OutputModule C:\Users\user\Generated.psd1

ipmo 'C:\Users\user\Generated.psd1'

# Cmdlets are created based on the following heuristics

# New-<EntityType> -<Key> [-<Other Attributes>…]

#

# Get-<EntityType> [-<Key> -Top –Skip –Filter -OrderBy]

# # If there is a complex key, the keys will actually be -<Key1> -<Key2>…

# # Note this rule applies to any other instances of the key

#

# Set-<EntityType> -<Key> [-<Other Attributes>]

#

# Remove-<EntityType> -<Key>

#

# Invoke-<EntityType><Action> [-<Key> -<Other Parameters>]

#

#

# Cmdlets from associations (Note: Get and Remove get additional parameter sets)

# Get-<EntityType> -<AssociatedEntity>

# New-<EntityType> -<AssociatedEntity> -<Key>

# Remove-<EntityType> -<AssociatedEntity> -<Key>

#

#

# Note: Every cmdlet has a –ConnectionURI parameter for explicitly setting the URI of the endpoint. This normally uses the same address that you gave the Export-ODataEndpointProxy cmdlet, but can be overridden in this fashion for the sake of similar endpoints.

#

There are still parts of key use cases in development for this functionality, including, but not limited to:

* Associations
* Passing streams

## Manage .ZIP archives with new cmdlets

Two new cmdlets, Compress-Archive and Expand-Archive, let you compress and expand ZIP files.

### Compress-Archive

The Compress-Archive cmdlet creates a new archive file from specified files. An archive file allows multiple files to be packaged and optionally compressed into a single file for easier handling and storage. An archive file can be compressed by using a compression algorithm specified in the -CompressionLevel parameter.

|  |
| --- |
| Compress-Archive -LiteralPath <String[]> [-DestinationPath] <String> [-Update] [-CompressionLevel <Microsoft.PowerShell.Commands.CompressionLevel>] |
| Compress-Archive [-Path] <String[]> [-DestinationPath] <String> [-Update] [-CompressionLevel <Microsoft.PowerShell.Commands.CompressionLevel>] |

### Expand-Archive

The Expand-Archive cmdlet extracts files from a specified archive file. An archive file allows multiple files to be packaged and optionally compressed into a single file for easier handling and storage.

|  |
| --- |
| Expand-Archive -LiteralPath <String> [-DestinationPath] <String> |
| Expand-Archive [-Path] <String> [-DestinationPath] <String> |

## DSC authoring improvements in Windows PowerShell ISE

Authoring DSC configurations in Windows PowerShell ISE is much easier, thanks to the following improvements:

* List all DSC resources within a **configuration** block or **node** block by entering **Ctrl+Space** on a blank line within it.
* Automatic completion on resource properties that are of the **enumeration** type.
* Automatic completion on the **DependsOn** property of DSC resources, based on other resource instances in the configuration.
* Better tab completion of resource property values.

**Note:** You must have an empty string for resource property values before you can use Ctrl+Space to list the options. Pressing **Tab** cycles through options.

## Configure DSC’s Local Configuration Manager with the meta-configuration attribute

The **DscLocalConfigurationManager** attribute designates a configuration block as a meta-configuration, which is used to configure the DSC Local Configuration Manager. This attribute restricts a configuration to containing only items which configure the DSC Local Configuration Manager. During processing, this configuration generates a \*.meta.mof file that is then sent to the appropriate target nodes by using the Set-DscLocalConfigurationManager cmdlet.

[DscLocalConfigurationManager()]

configuration meta

{

Node localhost

{

Settings

{

ConfigurationMode = "ApplyAndAutocorrect"

ConfigurationID = "5603f952-d6c6-4971-88c4-948636bf5c3f"

RefreshMode = "Pull"

}

ConfigurationRepositoryWeb PullServer

{

ServerURL = "https://corp.contoso.com/PSDSCPullServer/PSDSCPullServer.svc"

}

}

}

The above example configures the refresh mode for LCM to be in pull mode, changes the configuration mode to ApplyAndAutocorrect, and defines the type and location of the pull server.

This new configuration replaces and extends the functionality of the LocalConfigurationManager resource from DSC v1. LocalConfigurationManager is still supported in configurations without this attribute, for backwards compatibility.

Meta-resources:

|  |  |
| --- | --- |
| Resource Name | Description |
| LocalConfigurationManager | Various settings for DSC engine execution |
| PartialConfiguration | Partial configuration settings |
| ConfigurationRepositoryWeb | Web-based configuration repository |
| ConfigurationRepositoryShare | File share-based configuration repository |
| MSFT\_WebResourceManager | Web-based resource repository |
| MSFT\_FileResourceManager | File-based resource repository |
| MSFT\_WebReportManager | Web-based reporting endpoint for pull scenario |

## Configure piece by piece with Partial configurations in DSC

WMF 5.0 preview helps you deliver configuration documents to a node in fragments. For a node to receive multiple fragments of a configuration document, its Local Configuration Manager must be set first to specify the expected fragments, as shown in this example.

[DSCLocalConfigurationManager()]

configuration SQLServerDSCSettings

{

Node localhost

{

LocalConfigurationManager

{

ConfigurationModeFrequencyMins = 30

}

ConfigurationRepositoryWeb OSConfigServer

{

ServerURL = "https://corp.contoso.com/OSConfigServer/PSDSCPullServer.svc"

}

ConfigurationRepositoryWeb SQLConfigServer

{

ServerURL = "https://corp.contoso.com/SQLConfigServer/PSDSCPullServer.svc"

}

PartialConfiguration OSConfig

{

Description = 'Configuration for the Base OS'

ExclusiveResources = 'PSDesiredStateConfiguration\\*'

ConfigurationSource = '[ConfigurationRepositoryWeb]OSConfigServer'

}

PartialConfiguration SQLConfig

{

Description = 'Configuration for the SQL Server'

ConfigurationSource = '[ConfigurationRepositoryWeb]SQLConfigServer'

DependsOn = '[PartialConfiguration]OSConfig'

}

}

}

While authoring the partial configuration, the configuration name must match what is defined in the Local Configuration Manager. In the above example, the configurations should be named **OSConfig** and **SQLConfig**.

Setting Local Configuration Manager for partial configurations enables configuration coordination, but does NOT provide security functionality.

## Manage with cross-computer dependencies in DSC

By using the built-in WaitFor\* resources (WaitForAll, WaitForAny, and WaitForSome), you can now specify dependencies across computers during configuration runs, without external orchestrations. These resources provide node-to-node synchronization by using CIM connections over the WS-Man protocol. By using these resources, a configuration can wait for another computer’s specific resource state to change.

For example, in the following configuration, the target node is waiting for the **xADDomain** resource to finish on the **MyDC** node with a few retries, before the target node can join the domain.

configuration JoinDomain

{

Import-DscResource -Module xComputerManagement

WaitForAll DC

{

ResourceName = '[xADDomain]NewDomain'

NodeName = 'MyDC'

RetryIntervalSec = 15

RetryCount = 30

}

xComputer JoinDomain

{

Name = 'MyPC'

DomainName = 'Contoso.com'

Credential = (get-credential)

DependsOn ='[WaitForAll]DC'

}

}

## Find more detail about configuration status in DSC

### Get-DscConfigurationStatus

The Get-DscConfigurationStatus cmdlet gets high level information about configuration status from a target node. You can obtain the status of the last or all configurations.

Get-DscConfigurationStatus -All [-CimSession <CimSession[]>] [-ThrottleLimit <int>] [-AsJob] [<CommonParameters>]

Get-DscConfigurationStatus [-CimSession <CimSession[]>] [-ThrottleLimit <int>] [-AsJob] [<CommonParameters>]

### Compare-DscConfiguration

The Compare-DscConfiguration cmdlet compares a specified configuration with the actual state of one or more target nodes.

Compare-DscConfiguration [[-ReferenceConfiguration] <string>] [[-ComputerName] <string[]>] [-AsJob] [-Credential <pscredential>] [-ThrottleLimit <int>] [<CommonParameters>]

Compare-DscConfiguration [[-ReferenceConfiguration] <string>] -CimSession <CimSession[]> [-AsJob] [-throttleLimit <int>] [<CommonParameters>]

## More control over configurations in DSC

### Publish-DscConfiguration

The Publish-DscConfiguration cmdlet copies a configuration MOF file to a target node, but does not apply the configuration. This configuration is applied during the next consistency pass, or when you run the Update-DscConfiguration cmdlet.

Publish-DscConfiguration [-Path] <string> [[-ComputerName] <string[]>] [-Force] [-Credential <pscredential>]

[-ThrottleLimit <int>] [-WhatIf] [-Confirm] [<CommonParameters>]

Publish-DscConfiguration [-Path] <string> -CimSession <CimSession[]> [-Force] [-ThrottleLimit <int>] [-WhatIf]

[-Confirm] [<CommonParameters>]

### Update-DscConfiguration

The Update-DscConfiguration cmdlet forces the configuration to be processed. If the LCM is in pull mode, it gets the configuration from the pull server before applying it.

Update-DscConfiguration [[-ComputerName] <string[]>] [-Wait] [-Force] [-JobName <string>] [-Credential

<pscredential>] [-ThrottleLimit <int>] [-WhatIf] [-Confirm] [<CommonParameters>]

Update-DscConfiguration -CimSession <CimSession[]> [-Wait] [-Force] [-JobName <string>] [-ThrottleLimit <int>]

[-WhatIf] [-Confirm] [<CommonParameters>]

## Configuring the Windows PowerShell Desired State Configuration service

For reference, to set up a server as a Windows PowerShell Desired State Configuration pull server, additional steps are needed:

**On Windows Server 2012 and Windows Server 2012 R2:**

* Install Windows PowerShell Desired State Configuration Service (DSC-Service) by running the **Install-WindowsFeature -Name DSC-Service** command in an elevated Windows PowerShell session, or you can run the **Add Roles and Features Wizard** in Server Manager.
* It installs all dependent features, such as Web Server (IIS) (Web-Server), Management OData IIS Extension (ManagementOdata), etc.

On either Windows Server 2012 or Windows Server 2012 R2, after installing DSC-Service, you must create a pull server IIS endpoint. Follow the steps in section “**4. Create the pull server” in the** pull server MSDN documentation (<http://technet.microsoft.com/library/dn249913.aspx> ) to get the most current xDSCService resource for installing a pull server endpoint on your computer from the [TechNet Gallery](http://gallery.technet.microsoft.com/xPSDesiredStateConfiguratio-417dc71d). You can read more about this pull server resource in the blog post, [PowerShell DSC Resource for Configuring Pull Server Environment](http://blogs.msdn.com/b/powershell/archive/2013/11/21/powershell-dsc-resource-for-configuring-pull-server-environment.aspx).

## Audit Windows PowerShell usage by using transcription and logging

### Enhanced Windows PowerShell transcription options

Windows PowerShell transcription has been improved to apply to all hosting applications (such as Windows PowerShell ISE) rather than just the console host (powershell.exe).

In addition to extending for transcripting, the transcripting functionality itself has been updated to support arbitrary nesting of transcripts, additional metadata in the resulting transcript header, and the ability to set a transcription output directory (to support centralized log collection).

Transcription options (including enabling a system-wide transcript) can be configured by configuring the **Turn on PowerShell Transcription** Group Policy setting (in Administrative Templates -> Windows Components -> Windows PowerShell).

### Script tracing and logging

While Windows PowerShell already has the **LogPipelineExecutionDetails** Group Policy setting to log the invocation of cmdlets, Windows PowerShell’s scripting language has plenty of features that you might want to log and/or audit. The new Detailed Script Tracing feature lets you enable detailed tracking and analysis of Windows PowerShell scripting use on a system. After you enable detailed script tracing, Windows PowerShell logs all script blocks to the ETW event log, **Microsoft-Windows-PowerShell/Operational**. If a script block creates another script block (for example, a script that calls the Invoke-Expression cmdlet on a string), that resulting script block is logged as well.

Logging of these events can be enabled through the **Turn on PowerShell Script Block Logging** Group Policy setting (in Administrative Templates -> Windows Components -> Windows PowerShell).

The events are:

|  |  |
| --- | --- |
| Channel | Operational |
| Level | Verbose |
| Opcode | Create |
| Task | CommandStart |
| Keyword | Runspace |
| EventId | Engine\_ScriptBlockCompiled (0x1008 = 4104) |
| Message | Creating Scriptblock text (%1 of %2):  %3  ScriptBlock ID: %4 |

The text embedded in the message is the extent of the script block compiled. The ID is a GUID that is retained for the life of the script block.

When you enable verbose logging, the feature writes begin and end markers:

|  |  |
| --- | --- |
| Channel | Operational |
| Level | Verbose |
| Opcode | Open (/ Close) |
| Task | CommandStart (/ CommandStop) |
| Keyword | Runspace |
| EventId | ScriptBlock\_Invoke\_Start\_Detail (0x1009 = 4105) /  ScriptBlock\_Invoke\_Complete\_Detail (0x100A = 4106) |
| Message | Started (/ Completed) invocation of ScriptBlock ID: %1  Runspace ID: %2 |

The ID is the GUID representing the script block (that can be correlated with event ID 0x1008), and the Runspace ID represents the runspace in which this script block was run.

Percent signs in the invocation message represent structured ETW properties. While they are replaced with the actual values in the message text, a more robust way to access them is to retrieve the message with the Get-WinEvent cmdlet, and then use the **Properties** array of the message.

Here's an example of how this functionality can help unwrap a malicious attempt to encrypt and obfuscate a script:

## Malware

function SuperDecrypt

{

    param($script)

    $bytes = [Convert]::FromBase64String($script)

    ## XOR “encryption”

    $xorKey = 0x42

    for($counter = 0; $counter -lt $bytes.Length; $counter++)

    {

        $bytes[$counter] = $bytes[$counter] -bxor $xorKey

    }

    [System.Text.Encoding]::Unicode.GetString($bytes)

}

$decrypted = SuperDecrypt "FUIwQitCNkInQm9CCkItQjFCNkJiQmVCEkI1QixCJkJlQg=="

Invoke-Expression $decrypted

Running this generates the following log entries:

Compiling Scriptblock text (1 of 1):

function SuperDecrypt

{

param($script)

$bytes = [Convert]::FromBase64String($script)

## XOR "encryption"

$xorKey = 0x42

for($counter = 0; $counter -lt $bytes.Length; $counter++)

{

$bytes[$counter] = $bytes[$counter] -bxor $xorKey

}

[System.Text.Encoding]::Unicode.GetString($bytes)

}

ScriptBlock ID: ad8ae740-1f33-42aa-8dfc-1314411877e3

Compiling Scriptblock text (1 of 1):

$decrypted = SuperDecrypt "FUIwQitCNkInQm9CCkItQjFCNkJiQmVCEkI1QixCJkJlQg=="

ScriptBlock ID: ba11c155-d34c-4004-88e3-6502ecb50f52

Compiling Scriptblock text (1 of 1):

Invoke-Expression $decrypted

ScriptBlock ID: 856c01ca-85d7-4989-b47f-e6a09ee4eeb3

Compiling Scriptblock text (1 of 1):

Write-Host 'Pwnd'

ScriptBlock ID: 5e618414-4e77-48e3-8f65-9a863f54b4c8

If the script block length exceeds what ETW is capable of holding in a single event, Windows PowerShell breaks the script into multiple parts. Here is sample code to recombine a script from its log messages:

    $created = Get-WinEvent -FilterHashtable @{ ProviderName="Microsoft-Windows-PowerShell"; Id = 4104 } |

        Where-Object { $\_.<...> }

    $sortedScripts = $created | sort { $\_.Properties[0].Value }

    $mergedScript = -join ($sortedScripts | % { $\_.Properties[2].Value })

As with all logging systems that have a limited retention buffer (i.e., ETW logs), one attack against this infrastructure is to flood the log with spurious events to hide earlier evidence. To protect yourself from this attack, ensure that you have some form of event log collection set up (i.e., Windows Event Forwarding, <http://www.nsa.gov/ia/_files/app/Spotting_the_Adversary_with_Windows_Event_Log_Monitoring.pdf>) to move event logs off of the computer as soon as possible.

### CMS (Cryptographic Message Syntax) cmdlets

The Cryptographic Message Syntax cmdlets support encryption and decryption of content using the IETF standard format for cryptographically protecting messages as documented by [RFC5652](http://tools.ietf.org/html/rfc5652).

  Get-CmsMessage [-Content] <string>

  Get-CmsMessage [-Path] <string>

  Get-CmsMessage [-LiteralPath] <string>

  Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-Content] <string> [[-OutFile] <string>]

  Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-Path] <string> [[-OutFile] <string>]

  Protect-CmsMessage [-To] <CmsMessageRecipient[]> [-LiteralPath] <string> [[-OutFile] <string>]

  Unprotect-CmsMessage [-EventLogRecord] <EventLogRecord> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

  Unprotect-CmsMessage [-Content] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

  Unprotect-CmsMessage [-Path] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

  Unprotect-CmsMessage [-LiteralPath] <string> [[-To] <CmsMessageRecipient[]>] [-IncludeContext]

The CMS encryption standard implements public key cryptography, where the keys used to encrypt content (the *public key*) and the keys used to decrypt content (the *private key*) are separate.

Your public key can be shared widely, and is not sensitive data. If any content is encrypted with this public key, only your private key can decrypt it. For more information about Public Key Cryptography, see: <http://en.wikipedia.org/wiki/Public-key_cryptography>.

To be recognized in Windows PowerShell, encryption certificates require a unique key usage identifier (EKU) to identify them as data encryption certificates (like the identifiers for 'Code Signing', 'Encrypted Mail').

Here is an example of creating a certificate that is good for Document Encryption:

(Change the text in **Subject** to your name, email, or other identifier), and put in a file (i.e.: DocumentEncryption.inf):

  [Version]

  Signature = "$Windows NT$"

  [Strings]

  szOID\_ENHANCED\_KEY\_USAGE = "2.5.29.37"

  szOID\_DOCUMENT\_ENCRYPTION = "1.3.6.1.4.1.311.80.1"

  [NewRequest]

  Subject = "[cn=me@somewhere.com](mailto:cn=me@somewhere.com)"

  MachineKeySet = false

  KeyLength = 2048

  KeySpec = AT\_KEYEXCHANGE

  HashAlgorithm = Sha1

  Exportable = true

  RequestType = Cert

  KeyUsage = "CERT\_KEY\_ENCIPHERMENT\_KEY\_USAGE | CERT\_DATA\_ENCIPHERMENT\_KEY\_USAGE"

  ValidityPeriod = "Years"

  ValidityPeriodUnits = "1000"

  [Extensions]

  %szOID\_ENHANCED\_KEY\_USAGE% = "{text}%szOID\_DOCUMENT\_ENCRYPTION%"

Then run:

  certreq -new DocumentEncryption.inf DocumentEncryption.cer

And you can now encrypt and decrypt content:

106 [C:\temp]  
>> $protected = "Hello World" | Protect-CmsMessage -To "\*me@somewhere.com\*"

107 [C:\temp]  
>> $protected  
  
-----BEGIN CMS-----  
MIIBqAYJKoZIhvcNAQcDoIIBmTCCAZUCAQAxggFQMIIBTAIBADA0MCAxHjAcBgNVBAMMFWxlZWhv  
bG1AbWljcm9zb2Z0LmNvbQIQQYHsbcXnjIJCtH+OhGmc1DANBgkqhkiG9w0BAQcwAASCAQAnkFHM  
proJnFy4geFGfyNmxH3yeoPvwEYzdnsoVqqDPAd8D3wao77z7OhJEXwz9GeFLnxD6djKV/tF4PxR  
E27aduKSLbnxfpf/sepZ4fUkuGibnwWFrxGE3B1G26MCenHWjYQiqv+Nq32Gc97qEAERrhLv6S4R  
G+2dJEnesW8A+z9QPo+DwYU5FzD0Td0ExrkswVckpLNR6j17Yaags3ltNVmbdEXekhi6Psf2MLMP  
TSO79lv2L0KeXFGuPOrdzPAwCkV0vNEqTEBeDnZGrjv/5766bM3GW34FXApod9u+VSFpBnqVOCBA  
DVDraA6k+xwBt66cV84OHLkh0kT02SIHMDwGCSqGSIb3DQEHATAdBglghkgBZQMEASoEEJbJaiRl  
KMnBoD1dkb/FzSWAEBaL8xkFwCu0e1ZtDj7nSJc=  
-----END CMS-----

108 [C:\temp]  
>> $protected | Unprotect-CmsMessage  
Hello World

Any parameter of type **CMSMessageRecipient** supports identifiers in the following formats:

* An actual certificate (as retrieved from the certificate provider)
* Path to the a file containing the certificate
* Path to a directory containing the certificate
* Thumbprint of the certificate (used to look in the certificate store)
* Subject name of the certificate (used to look in the certificate store)

To view document encryption certificates in the certificate provider, you can use the -**DocumentEncryptionCert** dynamic parameter:

58 [Cert:\currentuser\my]

>> dir -DocumentEncryptionCert

## Extract and parse structured objects out of string content

In collaboration with Microsoft Research, a new **ConvertFrom-String** cmdlet has been added.

This cmdlet supports two modes: basic delimited parsing, and auto generated example-driven parsing.

Delimited parsing, by default, splits the input at white space, and assigns property names to the resulting groups. You can customize the delimiter:

1 [C:\temp]  
>> "Hello World" | ConvertFrom-String | Format-Table -Auto

P1    P2  
--    --

The cmdlet also supports auto-generated example-driven parsing based on the [FlashExtract](http://research.microsoft.com/en-us/um/people/sumitg/flashextract.html) research work in [Microsoft Research](http://research.microsoft.com).

To get started, consider a text-based address book:

    Ana Trujillo

    Redmond, WA

    Antonio Moreno

    Renton, WA

    Thomas Hardy

    Seattle, WA

    Christina Berglund

    Redmond, WA

    Hanna Moos

    Puyallup, WA

Copy a few examples into a file, which you will use as your template:

    Ana Trujillo

    Redmond, WA

    Antonio Moreno

    Renton, WA

Put curly braces around data that you want to extract, giving it a name as you do so. Because the **Name** property (and its associated other properties) can appear multiple times, use an asterisk (\*) to indicate that this results in multiple records (rather than extracting a bunch of properties into one record):

    {Name\*:Ana Trujillo}

    {City:Redmond}, {State:WA}

    {Name\*:Antonio Moreno}

    {City:Renton}, {State:WA}

From this set of examples, **ConvertFrom-String** can now automatically extract object-based output from input files with similar structure.

2 [C:\temp]

>> Get-Content .\addresses.output.txt | ConvertFrom-String -TemplateFile .\addresses.template.txt |  
>>> Format-Table -Auto

ExtentText                     Name               City     State  
----------                     ----               ----     -----  
Ana Trujillo...                Ana Trujillo       Redmond  WA  
Antonio Moreno...              Antonio Moreno     Renton   WA  
Thomas Hardy...                Thomas Hardy       Seattle  WA  
Christina Berglund...          Christina Berglund Redmond  WA  
Hanna Moos...                  Hanna Moos         Puyallup WA

To do additional data manipulation on extracted text, the **ExtentText** property captures the raw text from which the record was extracted. To provide feedback on this feature, or to share content that you are having difficulty writing examples for, please email [psdmfb@microsoft.com](mailto:psdmfb@microsoft.com).

## Interact with Symbolic Links using improved Item cmdlets

To support symbolic links, \*-Item and a few related cmdlets have been extended. Now you can create symbolic links in a single, simple line with New-Item.

You’ll notice that the Item related cmdlets (Remove-Item, Get-ChildItem) behave very similarly to before.

The following shows some use cases of the new capabilities:

### SYMBOLIC LINK FILES

# Create a new symbolic link file named MySymLinkFile.txt in C:\Temp which links to $pshome\profile.ps1

cd C:\Temp

New-Item -ItemType SymbolicLink -Name MySymLinkFile.txt -Target $pshome\profile.ps1 # File

# Target is an alias to the Value parameter

# Equivalent to above

New-Item -ItemType SymbolicLink -Path C:\Temp -Name MySymLinkFile.txt -Value $pshome\profile.ps1

# Equivalent to above

New-Item -ItemType SymbolicLink -Path C:\Temp\MySymLinkFile.txt -Value $pshome\profile.ps1

# Equivalent to above

New-Item -ItemType SymbolicLink -Name C:\Temp\MySymLinkFile.txt -Value $pshome\profile.ps1

### SYMBOLIC LINK DIRECTORIES

# Create a new symbolic link directory named MySymLinkDir in C:\Temp which links to the $pshome folder

# ItemType is the same for files and directories - autodetect based on specified target

cd C:\Temp

New-Item -ItemType SymbolicLink -Name MySymLinkDir -Target $pshome # Directory

# Target is an alias to the Value parameter

# Similar to above, any combination of Path and Name also works

New-Item -ItemType SymbolicLink -Path C:\Temp -Name MySymLinkDir -Value $pshome

New-Item -ItemType SymbolicLink -Path C:\Temp\MySymLinkDir -Value $pshome

New-Item -ItemType SymbolicLink -Name C:\Temp\MySymLinkDir -Value $pshome

### HARD LINKS

New-Item -ItemType HardLink -Path C:\Temp -Name MyHardLinkFile.txt -Value $pshome\profile.ps1

# Same combinations of Path and Name allowed as described above

### DIRECTORY JUNCTIONS

New-Item -ItemType Junction -Path C:\Temp\MyJunctionDir -Value $pshome

# Same combinations of Path and Name allowed as described above

# GET-CHILDITEM

# Append link type column to Mode property and display with Get-ChildItem

# Use 'l' for all link types

# Increase the width of the Length column by 4 (from 10 to 14)

Get-ChildItem C:\Temp | sort LastWriteTime -Descending

Directory: C:\Temp

Mode LastWriteTime Length Name

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

-a---- 6/13/2014 3:00 PM 16 File.txt

-a---- 6/13/2014 3:00 PM 98956046499840 My90TB.vhd

d----- 6/13/2014 3:00 PM Directory

-a---l 6/13/2014 3:21 PM 0 MySymLinkFile.txt

d----l 6/13/2014 3:22 PM MySymLinkDir

-a---l 6/13/2014 3:23 PM 23304 MyHardLinkFile.txt

d----l 6/13/2014 3:24 PM MyJunctionDir

# New Target property

# Works with any link type

# Not displayed in the default table view

# Displayed in the default list view

# New LinkType property with values: SymbolicLink

Get-ChildItem C:\Temp\MySymLinkFile.txt | Format-List

Directory: C:\Temp

Name : MySymLinkFile.txt

Length : 0

Mode : -a---l

LinkType : SymbolicLink

Target : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1

CreationTime : 6/16/2014 3:21:01 PM

LastWriteTime : 6/16/2014 3:21:01 PM

LastAccessTime : 6/16/2014 3:21:01 PM

VersionInfo : File: C:\Temp\MySymLinkFile.txt

InternalName:

OriginalFilename:

FileVersion:

FileDescription:

Product:

ProductVersion:

Debug: False

Patched: False

PreRelease: False

PrivateBuild: False

SpecialBuild: False

Language:

# REMOVE-ITEM

# Works like any other item type

# Removes MySymLinkFile.txt

Remove-Item C:\Temp\MySymLinkFile.txt

# Returns an error as this is a reparse point.

Remove-Item C:\Temp\MySymLinkDir

# Removes the files under the target directory and MySymLinkDir

Remove-Item C:\Temp\MySymLinkDir -Force

## Network Switch management with Windows PowerShell (improvements)

The Get-NetworkSwitchEthernetPort cmdlet now returns the following additional information with instances:

* IPAddress – the IP address associated with the port
* PortMode – the port mode: access, route or trunk
* AccessVLAN – the ID of the VLAN associated with this port in access mode
* TrunkedVLANList – a list of IDs of VLANs associated with this port in trunk mode

### Fundamental network switch management with Windows PowerShell

The Network Switch cmdlets, introduced in the first WMF 5.0 Preview, enable you to apply switch, virtual LAN (VLAN), and basic Layer 2 network switch port configuration to Windows Server 2012 R2 logo-certified network switches. Microsoft remains committed to supporting the [Datacenter Abstraction](http://technet.microsoft.com/en-us/cloud/dal.aspx) Layer (DAL) vision, and to show value for our customers and partners in this space. Using these cmdlets you can perform:

* + - Global switch configuration, such as:
      * Sett host name
      * Set switch banner
      * Persist configuration
      * Enable or disable feature
    - VLAN configuration:
      * Create or remove VLAN
      * Enable or disable VLAN
      * Enumerate VLAN
      * Set friendly name to a VLAN
    - Layer 2 port configuration:
      * Enumerate ports
      * Enable or disable ports
      * Set port modes and properties
      * Add or associate VLAN to Trunk or Access on the port

Start exploring by looking for all of the NetworkSwitch cmdlets!

PS> Get-Command \*-NetworkSwitch\*

|  |  |  |
| --- | --- | --- |
| CommandType | Name | Source |
| ----------- | ---- | ------ |
| Function | Disable-NetworkSwitchEthernetPort | NetworkSwitch |
| Function | Disable-NetworkSwitchFeature | NetworkSwitch |
| Function | Disable-NetworkSwitchVlan | NetworkSwitch |
| Function | Enable-NetworkSwitchEthernetPort | NetworkSwitch |
| Function | Enable-NetworkSwitchFeature | NetworkSwitch |
| Function | Enable-NetworkSwitchVlan | NetworkSwitch |
| Function | Get-NetworkSwitchEthernetPort | NetworkSwitch |
| Function | Get-NetworkSwitchFeature | NetworkSwitch |
| Function | Get-NetworkSwitchGlobalData | NetworkSwitch |
| Function | Get-NetworkSwitchVlan | NetworkSwitch |
| Function | New-NetworkSwitchVlan | NetworkSwitch |
| Function | Remove-NetworkSwitchEthernetPortIPAddress | NetworkSwitch |
| Function | Remove-NetworkSwitchVlan | NetworkSwitch |
| Function | Restore-NetworkSwitchConfiguration | NetworkSwitch |
| Function | Save-NetworkSwitchConfiguration | NetworkSwitch |
| Function | Set-NetworkSwitchEthernetPortIPAddress | NetworkSwitch |
| Function | Set-NetworkSwitchPortMode | NetworkSwitch |
| Function | Set-NetworkSwitchPortProperty | NetworkSwitch |
| Function | Set-NetworkSwitchVlanProperty | NetworkSwitch |

More information is available in Jeffrey Snover’s WMF 5.0 Preview announcement blog post: <http://blogs.technet.com/b/windowsserver/archive/2014/04/03/windows-management-framework-v5-preview.aspx>

## More control and remoting in Windows PowerShell debugging

### Break All

The Windows PowerShell console and Windows PowerShell ISE now allow you to break into the debugger for running scripts. This works in both local and remote sessions.

In the console, press **Ctrl+Break**.

In ISE, press **Ctrl+B**, or use the **Debug->Break All** menu command.

### Remote debugging and remote file editing in Windows PowerShell ISE

Windows PowerShell ISE now lets you open and edit files in a remote session by running the PSEdit command.

For example, you can open a file for editing from the command line in a remote session as follows:

[RemoteComputer1]: PS C:\> PSEdit C:\DebugDemoScripts\Test-GetMutex.ps1

In addition, you can now edit and save changes in a remote file that is automatically opened in Windows PowerShell ISE when you hit a breakpoint.

Now, you can debug a script file that is running on a remote computer, edit the file to fix an error, and then rerun the modified script.

### Advanced script debugging

There are new, advanced debugging features that let you attach to any local computer process that has loaded Windows PowerShell, and debug arbitrary runspaces in that process.

#### Runspace debugging

New cmdlets have been added that let you list current runspaces in a process, and attach the Windows PowerShell console or ISE debugger to that runspace for script debugging.

* Get-Runspace
* Debug-Runspace
* Enable-RunspaceDebug
* Disable-RunspaceDebug
* Get-RunspaceDebug

#### Attach to process

You can now attach to any computer process that has Windows PowerShell loaded. You do this by entering into an interactive session with the process, similarly to how you enter into an interactive remote session by running the Enter-PSSession cmdlet.

* Enter-PSHostProcess
* Exit-PSHostProcess

## Discover and install software with OneGet

OneGet is a new way to discover and install software packages from around the web. It is a manager or multiplexor of existing package managers (also called package providers) that unifies Windows package management with a single Windows PowerShell interface. With OneGet, you can do the following.

* Manage a list of software repositories in which packages can be searched, acquired, and installed
* Search and filter your repositories to find the packages you need
* Seamlessly install and uninstall packages from one or more repositories with a single Windows PowerShell command

### Install package providers and bootstrap OneGet

A Windows PowerShell module provider is included by default with OneGet.

OneGet includes a bootstrapper provider that enables on-demand installation of other package providers. Available package providers can be discovered by using the command:

PS C:\> Find-Package -Provider Bootstrap

You can install new providers by running the **Install-Package** command. Alternatively, they can be installed on-demand by running any OneGet command that has a -Provider parameter, and specifying the package provider name as the value of the -Provider parameter. If the package provider isn't currently installed, the bootstrap provider tries to download and install it. Add -Force or -ForceBootstrap to skip the prompts.

# Downloads and installs the NuGet provider on-demand, if it's not already installed.

PS C:\>Get-PackageProvider -Provider nuget -ForceBootstrap

In the WMF 5.0 September Preview, the only optional, external provider that was available is the NuGet provider. A new Chocolatey provider prototype is available in the WMF 5.0 November Preview.

PS> Import-Module -Name OneGet

More information is available in Jeffrey Snover’s WMF 5.0 Preview announcement blog: <http://blogs.technet.com/b/windowsserver/archive/2014/04/03/windows-management-framework-v5-preview.aspx>

## Discover and install software with OneGet

OneGet, introduced in early releases of WMF 5.0 Preview, has more improvements:

* The Chocolatey provider is back! You can bootstrap and install it with OneGet by running **Get-PackageProvider –Name NuGet –ForceBootstrap**
* Huge performance improvements
* Tagging is supported by the NuGet provider. This can be leveraged by any providers that use NuGet, including PowerShellGet.
* Added new package providers, including Add Remove Program (ARP), and MSI.

### Use additional package providers from the Internet

With this release, OneGet can now bootstrap additional package providers from the Internet.

This implementation downloads managed providers to one of two locations:

* If the user is running Windows PowerShell with elevated rights: **$env:PROGRAMFILES\OneGet\ProviderAssemblies**
* If the user is not running with elevated rights: **$env:LOCALAPPDATA\OneGet\ProviderAssemblies**

If users are not running Windows PowerShell with elevated rights when they install the package provider, the provider is also available for use in an elevated session, and can be subject to privilege escalation. Be careful when you install providers from the Internet.

### Additional information

The source for the OneGet component is the OneGet project website, <https://oneget.org>.

More information about the design and implementation is available on the OneGet project webpage.

## Discover modules and DSC resources with PowerShellGet

PowerShellGet has been further improved in November 2014 release of WMF:

* Find-Module can now filter on module metadata with the -Tag parameter
* Find-Module can filter on repository-specific search language with the -Filter parameter
* Find-Module can filter based on module contents with the -Command, -DscResource, and -Includes parameters
* Find-DscResource allows discovery of individual DSC resources in repositories
* Huge performance improvements
* Support for installing from and publishing to file shares with NuGet

### Example commands

# Find all modules with tags Tag1 or Tag3

Find-Module -Tag Tag1,Tag3

# Find modules with a specific DscResource

Find-Module -DscResource DscTestResource

#Find modules with specific commands

Find-Module -Command Test-PSGetTestCmdlet, Get-ContosoServer, Get-ContosoClient

# Find all modules with Dsc resources

Find-Module -Includes DscResource

# Find all modules with cmdlets

Find-Module -Includes Cmdlet

# Find all modules with functions

Find-Module -Includes Function

# Find all DSC resources

Find-DscResource

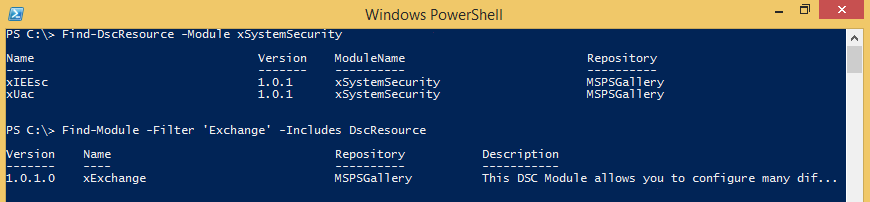
# Find all DSC resources contained within a specific module

Find-DscResource -ModuleName DscTestModule

# Find all DSC resources in modules with Tag1 or Tag3

Find-DscResource -Tag Tag1,Tag3

### Example output



### Register a PSRepository with PowerShellGet

In this release, it is much easier to configure PowerShellGet to operate against internal repositories. This is done by the following additions:

* Register-PSRepository: Registers a repository for the current user.
* Unregister-PSRepository: Removes a registered repository for the current user.
* Set-PSRepository: Set values for a registered repository.
* Get-PSRepository: Get all registered repositories for the current user.

After a repository is registered, you can use Find-Module and Install-Module to work with it.

#Register a default repository

Register-PSRepository –Name DemoRepo –SourceLocation “https://www.myget.org/F/powershellgetdemo/api/v2” –PublishLocation “<https://www.myget.org/F/powershellgetdemo/api/v2>/package” –InstallationPolicy –Trusted

#Get all of the registered repositories

Get-PSRepository

Name SourceLocation

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

PSGallery https://msconfiggallery.cloudapp...

DemoRepo https://www.myget.org/F/powershe...

#Search only the new repository for modules

Find-Module -Repository DemoRepo

Repository Version Name

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

DemoRepo 1.0.1 xActiveDirectory

DemoRepo 1.1.1 SomeModule

#By default, PowerShellGet operates against all registered repositories when none is specified. In this example, the “SomeModule” module is installed from the DemoRepo.

Install-Module SomeModule

#Removing a repository

Unregister-PSRepository DemoRepo

## 32-bit support for the Configuration keyword in DSC

The Configuration keyword is now supported in WOW64 on a 64-bit computer. This means that a DSC configuration can be defined and compiled within a 32-bit process such as Windows PowerShell ISE (x86) running on a 64-bit computer.

## Report configuration status from DSC to central location

High-level information about DSC configuration status (Success or Failure) can be sent to a server running the DSC service. By defining the report server in a nodes meta-configuration, this status is sent to the server when an error occurs or the configuration completes successfully. This information is sent when a node is configured in either pull or push mode. Status information is stored in the DSC service database.

A new OData endpoint is created with the DSC service which exposes this status information. By passing a configuration ID to the endpoint, all of the status for a node can be collected and parsed.

#### Sample meta-configuration for reporting status

[DscLocalConfigurationManager()]

Configuration ReportingClientMetaConfig  
{  
    Settings  
        {  
            ConfigurationID = "edf2d027-69cf-4374-8685-a0b18e6941f6"  
            RefreshFrequencyMins = 30  
            RefreshMode = "PUSH"  
            ConfigurationModeFrequencyMins = 15  
            AllowModuleOverwrite = $true  
        }

        MSFT\_WebReportManager ReportManager  
        {  
            Name = "WebDownloadManager"  
            ServerUrL = "<http://localhost:8080/PSDSCPullServer/PSDSCPullserver.svc>"  
            AllowUnsecureConnection  = $true  
        }             
}

#### Sample web request to gather configuration status

$guid='edf2d027-69cf-4374-8685-a0b18e6941f6'

$d = Invoke-WebRequest -Uri "<http://localhost:8080/psdscpullserver/psdscpullserver.svc/ErrorReport(ConfigurationId='$guid')/GetErrorReport>" -UseBasicParsing -UseDefaultCredentials -ContentType "application/json;odata=minimalmetadata;streaming=true;charset=utf-8" -Method Post -Headers @{Accept = "application/json"}

## Develop with classes in Windows PowerShell

WMF 5.0 Preview September 2014 adds support to the Windows PowerShell language for defining classes and other user-defined types, by using formal syntax and semantics that are similar to other object-oriented programming languages. The goal is to enable developers and IT professionals to embrace Windows PowerShell for a wider range of use cases, simplify development of Windows PowerShell artifacts (such as DSC resources), and accelerate coverage of management surfaces.

### Define custom types in Windows PowerShell

WMF 5.0 Preview September 2014 introduces the following new language elements in Windows PowerShell.

| Element | Description |
| --- | --- |
| **Class** keyword | Defines a new class. This is a true .NET Framework type.  Class members are public, but only public within the module scope. You can't refer to the type name as a string (for example, New-Object doesn't work), and in this release, you can't use a type literal (for example, [MyClass]) outside the script/module file in which the class is defined.  class MyClass  {  } |
| **Enum** keyword and enumerations | Support for the **enum** keyword has been added, which uses newline as the delimiter.  Current limitations: you cannot define an enumerator in terms of itself, but you can initialize enum in terms of another enum, as shown in the following example. The base type cannot currently be specified; it is always [int].  enum Color2  {  Yellow = [Color]::Blue  }  An enumerator value must be a parse time constant; you cannot set it to the result of an invoked command.  enum MyEnum  {  Enum1  Enum2  Enum3 = 42  Enum4 = [int]::MaxValue  }  Enums support arithmetic operations, as shown in the following example.  enum SomeEnum { Max = 42 }  enum OtherEnum { Max = [SomeEnum]::Max + 1 } |
| Import-DscResource | Import-DscResource is now a true dynamic keyword. Windows PowerShell parses the specified module’s root module, searching for classes that contain the **DscResource** attribute. |
| ImplementingAssembly | A new field, **ImplementingAssembly**, has been added to ModuleInfo. It is set to the dynamic assembly created for a script module if the script defines classes, or the loaded assembly for binary modules. It is not set when ModuleType = Manifest.  Reflection on the **ImplementingAssembly** field discovers resources in a module. This means you can discover resources written in either PowerShell or other managed languages.  Fields with initializers:  [int] $i = 5  Static is supported; it works like an attribute, as do the type constraints, so it can be specified in any order.  static [int] $count = 0  A type is optional.  $s = "hello"  All members are public. |
| Constructors and instantiation | Windows PowerShell classes can have constructors; they have the same name as their class. Constructors can be overloaded. Static constructors are supported. Properties with initialization expressions are initialized before running any code in a constructor. Static properties are initialized before the body of a static constructor, and instance properties are initialized before the body of the non-static constructor. Currently, there is no syntax for calling a constructor from another constructor (like the C# syntax ": this()"). The workaround is to define a common Init method.  The following are ways of instantiating classes in this release.  # Instantiating by using the default constructor. Note that New-Object is not supported in this release.  $a = [MyClass]::new()  # Calling a constructor with a parameter  $b = [MyClass]::new(42)  # Passing an array to a constructor with multiple parameters  $c = [MyClass]::new(@(42,43,44), "Hello")  In this release, New-Object does not work with classes defined in Windows PowerShell. Also for this release, the type name is only visible lexically, meaning it is not visible outside of the module or script that defines the class. Functions can return instances of a class defined in Windows PowerShell, and instances work well outside of the module or script.  Get-Member -Static lists constructors, so you can view overloads like any other method. The performance of this syntax is also considerably faster than New-Object.  The pseudo-static method named **new** works with .Net types, as shown in the following example.  [hashtable]::new()  You can now see constructor overloads with Get-Member, or as shown in this example:  PS> [hashtable]::new  OverloadDefinitions  -------------------  hashtable new()  hashtable new(int capacity)  hashtable new(int capacity, float loadFactor) |
| Methods | A Windows PowerShell class method is implemented as a ScriptBlock that has only an end block. All methods are public. The following shows an example of defining a method named **DoSomething**.  class MyClass  {  DoSomething($x)  {  $this.\_doSomething($x) # method syntax  }  private \_doSomething($a) {}  }  Method invocation:  $b = [MyClass]::new()  $b.DoSomething(42)  Overloaded methods—that is, those that are named the same as an existing method, but differentiated by their specified values—are also supported. |
| Properties | All properties are public. Properties require either a newline or semicolon. If no object type is specified, the property type is object.  Properties that use validation attributes or argument transformation attributes (e.g. [ValidateSet("aaa")]) work as expected. |
| Hidden | A new keyword, **hidden**, has been added. **Hidden** can be applied to properties and methods (including constructors).  Hidden members are public, but do not appear in the output of Get-Member unless the -Force parameter is added.  Hidden members are not included when tab completing or using Intellisense unless the completion occurs in the class defining the hidden member.  A new attribute, System.Management.Automation.HiddenAttribute has been added so that C# code can have the same semantics within Windows PowerShell. |
| Return types | Return type is a contract; the return value is converted to the expected type. If no return type is specified, the return type is void. There is no streaming of objects; objects cannot be written to the pipeline either intentionally or by accident. |
| Attributes | Four new attributes, **DscResource**, **DscResourceKey**, **DscResourceMandatory**, and **DscResourceOut** have been added. |
| Lexical scoping of variables | The following shows an example of how lexical scoping works in this release.  $d = 42 # Script scope  function bar  {  $d = 0 # Function scope  [MyClass]::DoSomething()  }  class MyClass  {  static [object] DoSomething()  {  return $d # error, not found dynamically  return $script:d # no error  $d = $script:d  return $d # no error, found lexically  }  }  $v = bar  $v -eq $d # true |

The following example creates several new, custom classes to implement an HTML dynamic style sheet language (DSL). Then, the example adds helper functions to create specific element types as part of the element class, such as heading styles and tables, because types cannot be used outside the scope of a module.

# Classes that define the structure of the document

#

class Html

{

[string] $docType

[HtmlHead] $Head

[Element[]] $Body

[string] Render()

{

$text = "<html>`n<head>`n"

$text += $this.Head

$text += "`n</head>`n<body>`n"

$text += $this.Body -join "`n" # Render all of the body elements

$text += "</body>`n</html>"

return $text

}

[string] ToString() { return $this.Render() }

}

class HtmlHead

{

$Title

$Base

$Link

$Style

$Meta

$Script

[string] Render() { return "<title>$($this.Title)</title>" }

[string] ToString() { return $this.Render() }

}

class Element

{

[string] $Tag

[string] $Text

[hashtable] $Attributes

[string] Render() {

$attributesText= ""

if ($this.Attributes)

{

foreach ($attr in $this.Attributes.Keys)

{

#BUGBUG - need to validate keys against attribute

$attributesText += " $attr=`"$($this.Attributes[$attr])`""

}

}

return "<$($this.tag)${attributesText}>$($this.text)</$($this.tag)>`n"

}

[string] ToString() { return $this.Render() }

}

#

# Helper functions for creating specific element types on top of the classes.

# These are required because types aren’t visible outside of the module.

#

function H1 { [Element] @{ Tag = "H1" ; Text = $args.foreach{$\_} -join " " }}

function H2 { [Element] @{ Tag = "H2" ; Text = $args.foreach{$\_} -join " " }}

function H3 { [Element] @{ Tag = "H3" ; Text = $args.foreach{$\_} -join " " }}

function P { [Element] @{ Tag = "P" ; Text = $args.foreach{$\_} -join " " }}

function B { [Element] @{ Tag = "B" ; Text = $args.foreach{$\_} -join " " }}

function I { [Element] @{ Tag = "I" ; Text = $args.foreach{$\_} -join " " }}

function HREF

{

param (

$Name,

$Link

)

return [Element] @{

Tag = "A"

Attributes = @{ HREF = $link }

Text = $name

}

}

function Table

{

param (

[Parameter(Mandatory)]

[object[]]

$Data,

[Parameter()]

[string[]]

$Properties = "\*",

[Parameter()]

[hashtable]

$Attributes = @{ border=2; cellpadding=2; cellspacing=2 }

)

$bodyText = ""

# Add the header tags

$bodyText += $Properties.foreach{TH $\_}

# Add the rows

$bodyText += foreach ($row in $Data)

{

TR (-join $Properties.Foreach{ TD ($row.$\_) } )

}

$table = [Element] @{

Tag = "Table"

Attributes = $Attributes

Text = $bodyText

}

$table

}

function TH { ([Element] @{ Tag = "TH" ; Text = $args.foreach{$\_} -join " " }) }

function TR { ([Element] @{ Tag = "TR" ; Text = $args.foreach{$\_} -join " " }) }

function TD { ([Element] @{ Tag = "TD" ; Text = $args.foreach{$\_} -join " " }) }

function Style

{

return [Element] @{

Tag = "style"

Text = "$args"

}

}

# Takes a hash table, casts it to and HTML document

# and then returns the resulting type.

#

function Html ([HTML] $doc) { return $doc }

### Known issues

These are the known issues with creating new types.

* New-Object does not work with Windows PowerShell classes in this release. As a workaround, use the static new method; for example, [Point]::new().

## More control and remoting in Windows PowerShell debugging

There have been some improvements to Windows PowerShell debugging beyond what was released in WMF 5.0 September Preview.

* The Enter-PSHostProcess cmdlet has a new, optional **AppDomainName** parameter. If you specify this parameter, Enter-PSHostProcess tries to connect to the specified app domain in the target process. If you do not specify this parameter, Enter-PSHostProcess tries to connect to the default domain in the process.
* Get-PSHostProcessInfo now returns AppDomain names along with processes to which you can connect by running the Enter-PSHostProcess cmdlet.

# Functionality that is still in early stages of development

There is some functionality that can be found that is not yet in a usable state. You may be able to find some small pieces of functionality, but there should be no expectation for it to complete a scenario successfully.

## Centralized DSC error reporting

Rich error information is not only logged in the event log, but it can be sent to a central location for later analysis. Administrators would be able to use this central location to view all errors that have occurred for any server in their environment. After the report server is defined in the meta-configuration, all errors are sent to the report server, and then stored in a database. You can set up this functionality regardless of whether or not the target node is configured to pull its configuration from a pull server.

Some artifacts from this scenario are visible, but are not expected to be in a supported state.

## Select scenarios in Export-ODataEndpointProxy

There are still parts of key use cases in development for this functionality, including, but not limited to:

* Associations
* Passing streams

# Known issues

## Reverting to a Windows 8.1 (2012 R2) build from WMF 5.0 Preview can break DSC cmdlets because of updates in metaconfig.mof

Resolution: Delete metaconfig.mof.

1. Open **powershell.exe** with elevated user rights (run as administrator).
2. Run the following command in the console:
   1. Remove-Item -Path $env:SystemRoot\system32\Configuration\metaconfig.mof

## WMF 5.0 Preview installation can appear to succeed, but the installation rolls back after the system is restarted (or just fails)

Resolution: Delete the **\\root\microsoft\windows\desiredstateconfiguration** namespace in WMI.

1. Open **powershell.exe** with elevated user rights (**Run as Administrator**).
2. Run the following commands:
   1. $dscNamespace = Get-CimInstance -Namespace root\microsoft\windows -Query "select \* from \_\_namespace where name = 'desiredstateconfiguration'"
   2. $dscNamespace | Remove-CimInstance
   3. mofcomp.exe $env:windir\system32\wbem\DSCCoreConfProv.mof
3. Install the WMF 5.0 Preview package.

# Known incompatibilities

**Systems that are running the following server applications should not run Windows Management Framework 5.0 at this time.**

* + - System Center 2012 Configuration Manager (not including SP1)
    - Windows Small Business Server 2011 Standard

# Feedback and contact

For issues or feedback you would like to report to us, use our Connect website:

<https://connect.microsoft.com/PowerShell/Feedback>

# Additional links

Windows PowerShell Team Blog: http://blogs.msdn.com/b/powershell/