Differences between Windows PowerShell 5.1 and PowerShell 7.x - PowerShell

sdwheeler ⋮ 42-53 minutes

Differences between Windows PowerShell 5.1 and PowerShell 7.x

- Article
- 04/02/2024
-

In this article

Windows PowerShell 5.1 is built on top of the .NET Framework v4.5. With the release of PowerShell 6.0, PowerShell became an open source project built on .NET Core 2.0. Moving from the .NET Framework to .NET Core allowed PowerShell to become a cross-platform solution. PowerShell runs on Windows, macOS, and Linux.

There are few differences in the PowerShell language between Windows PowerShell and PowerShell. The most notable differences are in the availability and behavior of PowerShell cmdlets between Windows and non-Windows platforms and the changes that stem from the differences between the .NET Framework and .NET Core.

This article summarizes the significant differences and breaking changes between Windows PowerShell and the current version of PowerShell. This summary does not include new features or cmdlets that have been added. Nor does this article discuss what changed between versions. The goal of this article is to present the current state of PowerShell and how that is different from Windows

PowerShell. For a detailed discussion of changes between versions and the addition of new features, see the **What's New** articles for each version.

- What's new in PowerShell 7.5
- What's new in PowerShell 7.4
- What's new in PowerShell 7.3
- What's new in PowerShell 7.2
- What's new in PowerShell 7.1
- What's new in PowerShell 7.0
- What's new in PowerShell 6.x

.NET Framework vs .NET Core

PowerShell on Linux and macOS uses .NET core, which is a subset of the full .NET Framework on Microsoft Windows. This is significant because PowerShell provides direct access to the underlying framework types and methods. As a result, scripts that run on Windows may not run on non-Windows platforms because of the differences in the frameworks. For more information about changes in .NET Core, see Breaking changes for migration from .NET Framework to .NET Core.

Each new release of PowerShell is built on a newer version of .NET. There can be breaking changes in .NET that affect PowerShell.

- PowerShell 7.5 - Built on .NET 9.0
- PowerShell 7.4 - Built on .NET 8.0
- PowerShell 7.3 - Built on .NET 7.0
- PowerShell 7.2 (LTS-current) - Built on .NET 6.0 (LTS-current)
- PowerShell 7.1 - Built on .NET 5.0
- PowerShell 7.0 (LTS) - Built on .NET Core 3.1 (LTS)
- PowerShell 6.2 - Built on .NET Core 2.1
- PowerShell 6.1 - Built on .NET Core 2.1
- PowerShell 6.0 - Built on .NET Core 2.0

With the advent of .NET Standard 2.0, PowerShell can load many traditional Windows PowerShell modules without modification. Additionally, PowerShell 7 includes a Windows PowerShell Compatibility feature that allows you to use Windows PowerShell modules that still require the full framework.

For more information see:

- about_Windows_PowerShell_Compatibility
- PowerShell 7 module compatibility

Be aware of .NET method changes

While .NET method changes are not specific to PowerShell, they can affect your scripts, especially if you are calling .NET methods directly. Also, there might be new overloads for constructors. This can have an impact on how you create objects using `New-Object` or the `[type]::new()` method.

For example, .NET added overloads to the `[System.String]::Split()` method that aren't available in .NET Framework 4.5. The following list shows the overloads for the `Split()` method available in Windows PowerShell 5.1:

```
PS> "".Split

OverloadDefinitions
-------------------
string[] Split(Params char[] separator)
```

```
string[] Split(char[] separator, int count)
string[] Split(char[] separator, System.StringSplitOptions options)
string[] Split(char[] separator, int count, System.StringSplitOptions
options)
string[] Split(string[] separator, System.StringSplitOptions options)
string[] Split(string[] separator, int count, System.StringSplitOptions
options)
```

The following list shows the overloads for the `Split()` method available in PowerShell 7:

```
"".Split

OverloadDefinitions
-------------------
string[] Split(char separator, System.StringSplitOptions options)
string[] Split(char separator, int count, System.StringSplitOptions
options)
string[] Split(Params char[] separator)
string[] Split(char[] separator, int count)
string[] Split(char[] separator, System.StringSplitOptions options)
string[] Split(char[] separator, int count, System.StringSplitOptions
options)
string[] Split(string separator, System.StringSplitOptions options)
string[] Split(string separator, int count, System.StringSplitOptions
options)
string[] Split(string[] separator, System.StringSplitOptions options)
string[] Split(string[] separator, int count, System.StringSplitOptions
options)
```

In Windows PowerShell 5.1, you could pass a character array (`char[]`) to the `Split()` method as a `string`. The method splits the target string at any occurrence of a character in the array. The following command splits the target string in Windows PowerShell 5.1, but not in PowerShell 7:

```
# PowerShell 7 example
"1111p2222q3333".Split('pq')
```

```
1111p2222q3333
```

To bind to the correct overload, you must typecast the string to a character array:

```
# PowerShell 7 example
"1111p2222q3333".Split([char[]]'pq')
```

```
1111
2222
3333
```

Modules no longer shipped with PowerShell

For various compatibility reasons, the following modules are no longer included in PowerShell.

- ISE
- Microsoft.PowerShell.LocalAccounts
- Microsoft.PowerShell.ODataUtils
- Microsoft.PowerShell.Operation.Validation
- PSScheduledJob
- PSWorkflow
- PSWorkflowUtility

PowerShell Workflow

PowerShell Workflow is a feature in Windows PowerShell that builds on top of Windows Workflow Foundation (WF) that enables the creation of robust runbooks for long-running or parallelized tasks.

Due to the lack of support for Windows Workflow Foundation in .NET Core, we removed PowerShell Workflow from PowerShell.

In the future, we would like to enable native parallelism/concurrency in the PowerShell language without the need for PowerShell Workflow.

If there is a need to use checkpoints to resume a script after the OS restarts, we recommend using Task Scheduler to run a script on OS startup, but the script would need to maintain its own state (like persisting it to a file).

Cmdlets removed from PowerShell

For the modules that are included in PowerShell, the following cmdlets were removed from PowerShell for various compatibility reasons or the use of unsupported APIs.

CimCmdlets

- `Export-BinaryMiLog`

Microsoft.PowerShell.Core

- `Add-PSSnapin`
- `Export-Console`
- `Get-PSSnapin`
- `Remove-PSSnapin`
- `Resume-Job`
- `Suspend-Job`

Microsoft.PowerShell.Diagnostics

- `Export-Counter`
- `Import-Counter`

Microsoft.PowerShell.Management

- `Add-Computer`
- `Checkpoint-Computer`
- `Clear-EventLog`
- `Complete-Transaction`
- `Disable-ComputerRestore`
- `Enable-ComputerRestore`
- `Get-ComputerRestorePoint`
- `Get-ControlPanelItem`
- `Get-EventLog`
- `Get-Transaction`
- `Get-WmiObject`
- `Invoke-WmiMethod`
- `Limit-EventLog`
- `New-EventLog`
- `New-WebServiceProxy`
- `Register-WmiEvent`
- `Remove-Computer`
- `Remove-EventLog`
- `Remove-WmiObject`
- `Reset-ComputerMachinePassword`
- `Restore-Computer`
- `Set-WmiInstance`
- `Show-ControlPanelItem`
- `Show-EventLog`
- `Start-Transaction`
- `Test-ComputerSecureChannel`
- `Undo-Transaction`
- `Use-Transaction`
- `Write-EventLog`

Microsoft.PowerShell.Utility

- `Convert-String`
- `ConvertFrom-String`

PSDesiredStateConfiguration

- `Disable-DscDebug`
- `Enable-DscDebug`
- `Get-DscConfiguration`
- `Get-DscConfigurationStatus`
- `Get-DscLocalConfigurationManager`
- `Publish-DscConfiguration`
- `Remove-DscConfigurationDocument`
- `Restore-DscConfiguration`
- `Set-DscLocalConfigurationManager`
- `Start-DscConfiguration`
- `Stop-DscConfiguration`

- `Test-DscConfiguration`
- `Update-DscConfiguration`

WMI v1 cmdlets

The following WMI v1 cmdlets were removed from PowerShell:

- `Register-WmiEvent`
- `Set-WmiInstance`
- `Invoke-WmiMethod`
- `Get-WmiObject`
- `Remove-WmiObject`

The CimCmdlets module (aka WMI v2) cmdlets perform the same function and provide new functionality and a redesigned syntax.

`New-WebServiceProxy` cmdlet removed

.NET Core does not support the Windows Communication Framework, which provide services for using the SOAP protocol. This cmdlet was removed because it requires SOAP.

`*-Transaction` cmdlets removed

These cmdlets had very limited usage. The decision was made to discontinue support for them.

- `Complete-Transaction`
- `Get-Transaction`
- `Start-Transaction`
- `Undo-Transaction`
- `Use-Transaction`

`*-EventLog` cmdlets

Due to the use of unsupported APIs, the `*-EventLog` cmdlets have been removed from PowerShell. `Get-WinEvent` and `New-WinEvent` are available to get and create events on Windows.

Cmdlets that use the Windows Presentation Framework (WPF)

.NET Core 3.1 added support for WPF, so the release of PowerShell 7.0 restored the following Windows-specific features:

- The `Show-Command` cmdlet
- The `Out-GridView` cmdlet
- The **ShowWindow** parameter of `Get-Help`

PowerShell Desired State Configuration (DSC) changes

`Invoke-DscResource` was restored as an experimental feature in PowerShell 7.0.

Beginning with PowerShell 7.2, the PSDesiredStateConfiguration module has been removed from PowerShell and has been published to the PowerShell Gallery. For more information, see the announcement in the PowerShell Team blog.

PowerShell executable changes

Renamed `powershell.exe` to `pwsh.exe`

The binary name for PowerShell has been changed from `powershell(.exe)` to `pwsh(.exe)`. This change provides a deterministic way for users to run PowerShell on machines and support side-by-side installations of Windows PowerShell and PowerShell.

Additional changes to `pwsh(.exe)` from `powershell.exe`:

- Changed the first positional parameter from `-Command` to `-File`. This change fixes the usage of `#!` (aka as a shebang) in PowerShell scripts that are being executed from non-PowerShell shells on non-Windows platforms. It also means that you can run commands like `pwsh foo.ps1` or `pwsh fooScript` without specifying `-File`. However, this change requires that you explicitly specify `-c` or `-Command` when trying to run commands like `pwsh.exe -Command Get-Command`.
- `pwsh` accepts the `-i` (or `-Interactive`) switch to indicate an interactive shell. This allows PowerShell to be used as a default shell on Unix platforms.
- Removed parameters `-ImportSystemModules` and `-PSConsoleFile` from `pwsh.exe`.
- Changed `pwsh -version` and built-in help for `pwsh.exe` to align with other native tools.
- Invalid argument error messages for `-File` and `-Command` and exit codes consistent with Unix standards
- Added `-WindowStyle` parameter on Windows. Similarly, package-based installations updates on non-Windows platforms are in-place updates.

The shortened name is also consistent with naming of shells on non-Windows platforms.

Support running a PowerShell script with bool parameter

Previously, using `pwsh.exe` to execute a PowerShell script using `-File` provided no way to pass `$true`/`$false` as parameter values. Support for `$true`/`$false` as parsed values to parameters was added. Switch values are also supported.

Improved backwards compatibility with Windows PowerShell

For Windows, a new switch parameter **UseWindowsPowerShell** is added to `Import-Module`. This switch creates a proxy module in PowerShell 7 that uses a local Windows PowerShell process to implicitly run any cmdlets contained in that module. For more information, see Import-Module.

For more information on which Microsoft modules work with PowerShell 7.0, see the Module Compatibility Table.

Microsoft Update support for Windows

PowerShell 7.2 added support for Microsoft Update. When you enable this feature, you'll get the latest PowerShell 7 updates in your traditional Windows Update (WU) management flow, whether that's with Windows Update for Business, WSUS, SCCM, or the interactive WU dialog in Settings.

The PowerShell 7.2 MSI package includes following command-line options:

- `USE_MU` - This property has two possible values:
  - `1` (default) - Opts into updating through Microsoft Update or WSUS
  - `0` - Do not opt into updating through Microsoft Update or WSUS

- `ENABLE_MU`
  - `1` (default) - Opts into using Microsoft Update the Automatic Updates or Windows Update
  - `0` - Do not opt into using Microsoft Update the Automatic Updates or Windows Update

Engine changes

Support PowerShell as a default Unix shell

On Unix, it is a convention for shells to accept `-i` for an interactive shell and many tools expect this behavior (`script` for example, and when setting PowerShell as the default shell) and calls the shell with the `-i` switch. This change is breaking in that `-i` previously could be used as short hand to match `-inputformat`, which now needs to be `-in`.

Custom snap-ins

PowerShell snap-ins are a predecessor to PowerShell modules that do not have widespread adoption in the PowerShell community.

Due to the complexity of supporting snap-ins and their lack of usage in the community, we no longer support custom snap-ins in PowerShell.

Experimental feature flags

PowerShell 6.2 enabled support for Experimental Features. This allows PowerShell developers to deliver new features and get feedback before the design is complete. This way we avoid making breaking changes as the design evolves.

Use `Get-ExperimentalFeature` to get a list of available experimental features. You can enable or disable these features with `Enable-ExperimentalFeature` and `Disable-ExperimentalFeature`.

Load assembly from module base path before trying to load from the GAC

Previously, when a binary module has the module assembly in GAC, we loaded the assembly from GAC before trying to load it from module base path.

Skip null-element check for collections with a value-type element type

For the `Mandatory` parameter and `ValidateNotNull` and `ValidateNotNullOrEmpty` attributes, skip the null-element check if the collection's element type is value type.

Preserve `$?` for ParenExpression, SubExpression and ArrayExpression

This PR alters the way we compile subpipelines `(...)`, subexpressions `$(...)` and array expressions `@()` so that `$?` is not automatically **true**. Instead the value of `$?` depends on the result of the pipeline or statements executed.

Fix `$?` to not be `$false` when native command writes to `stderr`

`$?` is not set to `$false` when native command writes to `stderr`. It is common for native commands to write to `stderr` without intending to indicate a failure. `$?` is set to `$false` only when the native command has a non-zero exit code.

Make `$ErrorActionPreference` not affect `stderr` output of native commands

It is common for native commands to write to `stderr` without intending to indicate a failure. With this change, `stderr` output is still captured in **ErrorRecord** objects, but the runtime no longer applies `$ErrorActionPreference` if the **ErrorRecord** comes from a native command.

Change `$OutputEncoding` to use `UTF-8 NoBOM` encoding rather than ASCII

The previous encoding, ASCII (7-bit), would result in incorrect alteration of the output in some cases. Making `UTF-8 NoBOM` the default preserves Unicode output with an encoding supported by most tools and operating systems.

Unify cmdlets with parameter `-Encoding` to be of type `System.Text.Encoding`

The `-Encoding` value `Byte` has been removed from the filesystem provider cmdlets. A new parameter, `-AsByteStream`, is now used to specify that a byte stream is required as input or that the output is a stream of bytes.

Change `New-ModuleManifest` encoding to `UTF8NoBOM` on non-Windows platforms

Previously, `New-ModuleManifest` creates `psd1` manifests in UTF-16 with BOM, creating a problem for Linux tools. This breaking change changes the encoding of `New-ModuleManifest` to be UTF (no BOM) in non-Windows platforms.

Remove `AllScope` from most default aliases

To speed up scope creation, `AllScope` was removed from most default aliases. `AllScope` was left for a few frequently used aliases where the lookup was faster.

`-Verbose` and `-Debug` no longer overrides `$ErrorActionPreference`

Previously, if `-Verbose` or `-Debug` were specified, it overrode the behavior of `$ErrorActionPreference`. With this change, `-Verbose` and `-Debug` no longer affect the behavior of `$ErrorActionPreference`.

Also, the `-Debug` parameter sets `$DebugPreference` to **Continue** instead of **Inquire**.

Make `$PSCulture` consistently reflect in-session culture changes

In Windows PowerShell, the current culture value is cached, which can allow the value to get out of sync with the culture is change after session-startup. This caching behavior is fixed in PowerShell core.

Allow explicitly specified named parameter to supersede the same one from hashtable splatting

With this change, the named parameters from splatting are moved to the end of the parameter list so that they are bound after all explicitly specified named parameters are bound. Parameter binding for simple functions doesn't throw an error when a specified named parameter cannot be found. Unknown named parameters are bound to the `$args` parameter of the simple function. Moving splatting to the end of the argument list changes the order the parameters appears in `$args`.

For example:

```
function SimpleTest {
    param(
        $Name,
        $Path
    )
    "Name: $Name; Path: $Path; Args: $args"
}
```

In the previous behavior, **MyPath** is not bound to `-Path` because it's the third argument in the argument list. ## So it ends up being stuffed into '$args' along with `Blah = "World"`

```
PS> $hash = @{ Name = "Hello"; Blah = "World" }
PS> SimpleTest @hash "MyPath"
Name: Hello; Path: ; Args: -Blah: World MyPath
```

With this change, the arguments from `@hash` are moved to the end of the argument list. **MyPath** becomes the first argument in the list, so it is bound to `-Path`.

```
PS> SimpleTest @hash "MyPath"
Name: Hello; Path: MyPath; Args: -Blah: World
```

Language changes

Null-coalescing operator ??

The null-coalescing operator `??` returns the value of its left-hand operand if it isn't null. Otherwise, it evaluates the right-hand operand and returns its result. The `??` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

```
$x = $null
$x ?? 100
```

```
100
```

In the following example, the right-hand operand won't be evaluated.

```
[string] $todaysDate = '1/10/2020'
$todaysDate ?? (Get-Date).ToShortDateString()
```

```
1/10/2020
```

Null-coalescing assignment operator ??=

The null-coalescing assignment operator `??=` assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to null. The `??=` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

```
$x = $null
$x ??= 100
$x
```

```
100
```

In the following example, the right-hand operand won't be evaluated.

```
[string] $todaysDate = '1/10/2020'
$todaysDate ??= (Get-Date).ToShortDateString()
```

```
1/10/2020
```

Null-conditional operators

Note

This feature was moved from experimental to mainstream in PowerShell 7.1.

A null-conditional operator applies a member access, `?.`, or element access, `?[]`, operation to its operand only if that operand evaluates to non-null; otherwise, it returns null.

Since PowerShell allows ? to be part of the variable name, formal specification of the variable name is required for using these operators. So it is required to use `{}` around the variable names like `${a}` or when ? is part of the variable name `${a?}`.

In the following example, the value of **PropName** is returned.

```
$a = @{ PropName = 100 }
${a}?.PropName
```

```
100
```

The following example will return null, without trying to access the member name **PropName**.

```
$a = $null
${a}?.PropName
```

Similarly, the value of the element will be returned.

```
$a = 1..10
${a}?[0]
```

```
1
```

And when the operand is null, the element isn't accessed and null is returned.

```
$a = $null
${a}?[0]
```

Note

The variable name syntax of `${<name>}` should not be confused with the `$()` subexpression operator. For more information, see Variable name section of about_Variables.

Added & operator for job control

Putting & at the end of a pipeline causes the pipeline to be run as a PowerShell job. When a pipeline is backgrounded, a job object is returned. Once the pipeline is running as a job, all of the standard `*-Job` cmdlets can be used to manage the job. Variables (ignoring process-specific variables) used in the pipeline are automatically copied to the job so `Copy-Item $foo $bar &` just works. The job is also run in the current directory instead of the user's home directory.

New methods/properties on `PSCustomObject`

We've added new methods and properties to `PSCustomObject`. `PSCustomObject` now includes a `Count`/`Length` property like other objects.

```
$PSCustomObject = [pscustomobject]@{foo = 1}

$PSCustomObject.Length
```

```
1
```

```
$PSCustomObject.Count
```

```
1
```

This work also includes `ForEach` and `Where` methods that allow you to operate and filter on `PSCustomObject` items:

```
$PSCustomObject.ForEach({$_.foo + 1})
```

```
2
```

```
$PSCustomObject.Where({$_.foo -gt 0})
```

```
foo
---
  1
```

Conversions from PSMethod to Delegate

You can convert a `PSMethod` into a delegate. This allows you to do things like passing `PSMethod` `[M]::DoubleStrLen` as a delegate value into `[M]::AggregateString`:

```
class M {
    static [int] DoubleStrLen([string] $value) { return 2 *
$value.Length }

    static [long] AggregateString([string[]] $values, [func[string,
int]] $selector) {
        [long] $res = 0
        foreach($s in $values){
            $res += $selector.Invoke($s)
        }
        return $res
    }
}

[M]::AggregateString((gci).Name, [M]::DoubleStrLen)
```

String comparison behavior changed in PowerShell 7.1

PowerShell 7.1 is built on .NET 5.0, which introduced the following breaking change:

- Behavior changes when comparing strings on .NET 5+

As of .NET 5.0, culture invariant string comparisons ignore non-printing control characters.

For example, the following two strings are considered to be identical:

```
# Escape sequence "`a" is Ctrl-G or [char]7
'Food' -eq "Foo`ad"
```

```
True
```

New cmdlets

New Get-Uptime cmdlet

The Get-Uptime cmdlet returns the time elapsed since the last boot of the operating system. The cmdlet was introduced in PowerShell 6.0.

New Remove-Alias cmdlet

The Remove-Alias cmdlet removes an alias from the current PowerShell session. The cmdlet was introduced in PowerShell 6.0.

New cmdlet Remove-Service

The Remove-Service cmdlet removes a Windows service in the registry and in the service database. The `Remove-Service` cmdlet was introduced in PowerShell 6.0.

New Markdown cmdlets

Markdown is a standard for creating readable plaintext documents with basic formatting that can be rendered into HTML.

The following cmdlets were added in PowerShell 6.1:

- ConvertFrom-Markdown - Convert the contents of a string or a file to a **MarkdownInfo** object.
- Get-MarkdownOption - Returns the current colors and styles used for rendering Markdown content in the console.
- Set-MarkdownOption - Sets the colors and styles used for rendering Markdown content in the console.
- Show-Markdown - Displays Markdown content in the console or as HTML

New Test-Json cmdlet

The Test-Json cmdlet tests whether a string is a valid JavaScript Object Notation (JSON) document and can optionally verify that JSON document against a provided schema.

This cmdlet was introduced in PowerShell 6.1

New cmdlets to support Experimental Features

The following cmdlets were added in PowerShell 6.2 to support Experimental Features.

- Disable-ExperimentalFeature
- Enable-ExperimentalFeature
- Get-ExperimentalFeature

New Join-String cmdlet

The Join-String cmdlet combines objects from the pipeline into a single string. This cmdlet was added in PowerShell 6.2.

New view ConciseView and cmdlet Get-Error

PowerShell 7.0 enhances the display of error messages to improve the readability of interactive and script errors with a new default view, **ConciseView**. The views are user-selectable through the preference variable `$ErrorView`.

With **ConciseView**, if an error is not from a script or parser error, then it's a single line error message:

```
Get-Childitem -Path c:\NotReal
```

```
Get-ChildItem: Cannot find path 'C:\NotReal' because it does not exist
```

If the error occurs during script execution or is a parsing error, PowerShell returns a multiline error message that contains the error, a pointer, and an error message showing where the error is in that line. If the terminal doesn't support ANSI color escape sequences (VT100), then colors are not displayed.

The default view in PowerShell 7 is **ConciseView**. The previous default view was **NormalView** and you can select this by setting the preference variable `$ErrorView`.

```
$ErrorView = 'NormalView' # Sets the error view to NormalView
$ErrorView = 'ConciseView' # Sets the error view to ConciseView
```

Note

A new property **ErrorAccentColor** is added to `$Host.PrivateData` to support changing the accent color of the error message.

The new `Get-Error`cmdlet provides a complete detailed view of the fully qualified error when desired. By default the cmdlet displays the full details, including inner exceptions, of the last error that occurred.

The `Get-Error` cmdlet supports input from the pipeline using the built-in variable `$Error`. `Get-Error` displays all piped errors.

```
$Error | Get-Error
```

The `Get-Error` cmdlet supports the **Newest** parameter, allowing you to specify how many errors from the current session you wish displayed.

```
Get-Error -Newest 3 # Displays the lst three errors that occurred in
the session
```

For more information, see Get-Error.

Cmdlet changes

Parallel execution added to ForEach-Object

Beginning in PowerShell 7.0, the `ForEach-Object` cmdlet, which iterates items in a collection, now has built-in parallelism with the new **Parallel** parameter.

By default, parallel script blocks use the current working directory of the caller that started the parallel tasks.

This example retrieves 50,000 log entries from 5 system logs on a local Windows machine:

```
$logNames = 'Security','Application','System','Windows
PowerShell','Microsoft-Windows-Store/Operational'

$logEntries = $logNames | ForEach-Object -Parallel {
    Get-WinEvent -LogName $_ -MaxEvents 10000
} -ThrottleLimit 5

$logEntries.Count

50000
```

The **Parallel** parameter specifies the script block that is run in parallel for each input log name.

The new **ThrottleLimit** parameter limits the number of script blocks running in parallel at a given time. The default is 5.

Use the `$_` variable to represent the current input object in the script block. Use the `$using:` scope to pass variable references to the running script block.

For more information, see ForEach-Object.

Check `system32` for compatible built-in modules on Windows

In the Windows 10 1809 update and Windows Server 2019, we updated a number of built-in PowerShell modules to mark them as compatible with PowerShell.

When PowerShell starts up, it automatically includes `$windir\System32` as part of the `PSModulePath` environment variable. However, it only exposes modules to `Get-Module` and `Import-Module` if its `CompatiblePSEdition` is marked as compatible with `Core`.

You can override this behavior to show all modules using the `-SkipEditionCheck` switch parameter. We've also added a `PSEdition` property to the table output.

`-lp` alias for all `-LiteralPath` parameters

We created a standard parameter alias `-lp` for all the built-in PowerShell cmdlets that have a `-LiteralPath` parameter.

Fix `Get-Item -LiteralPath a*b` if `a*b` doesn't actually exist to return error

Previously, `-LiteralPath` given a wildcard would treat it the same as `-Path` and if the wildcard found no files, it would silently exit. Correct behavior should be that `-LiteralPath` is literal so if the file doesn't exist, it should error. Change is to treat wildcards used with `-Literal` as literal.

Set working directory to current directory in `Start-Job`

The `Start-Job` cmdlet now uses the current directory as the working directory for the new job.

Remove `-Protocol` from `*-Computer` cmdlets

Due to issues with RPC remoting in CoreFX (particularly on non-Windows platforms) and ensuring a consistent remoting experience in PowerShell, the `-Protocol` parameter was removed from the `\*-Computer` cmdlets. DCOM is no longer supported for remoting. The following cmdlets only support WSMAN remoting:

- `Rename-Computer`
- `Restart-Computer`
- `Stop-Computer`

Remove `-ComputerName` from `*-Service` cmdlets

In order to encourage the consistent use of PSRP, the `-ComputerName` parameter was removed from `*-Service` cmdlets.

Fix `Get-Content -Delimiter` to not include the delimiter in the returned lines

Previously, the output while using `Get-Content -Delimiter` was inconsistent and inconvenient as it required further processing of the data to remove the delimiter. This change removes the delimiter in returned lines.

Changes to `Format-Hex`

The `-Raw` parameter is now a "no-op" (in that it does nothing). Going forward all output is displayed with a true representation of numbers that includes all of the bytes for its type. This is what the `-Raw` parameter was doing prior to this change.

Typo fix in Get-ComputerInfo property name

`BiosSerialNumber` was misspelled as `BiosSeralNumber` and has been changed to the correct spelling.

Add `Get-StringHash` and `Get-FileHash` cmdlets

This change is that some hash algorithms are not supported by CoreFX, therefore they are no longer available:

- `MACTripleDES`
- `RIPEMD160`

Add validation on `Get-*` cmdlets where passing $null returns all objects instead of error

Passing `$null` to any of the following now throws an error:

- `Get-Credential -UserName`
- `Get-Event -SourceIdentifier`
- `Get-EventSubscriber -SourceIdentifier`
- `Get-Help -Name`
- `Get-PSBreakpoint -Script`
- `Get-PSProvider -PSProvider`
- `Get-PSSessionConfiguration -Name`
- `Get-Runspace -Name`
- `Get-RunspaceDebug -RunspaceName`
- `Get-Service -Name`
- `Get-TraceSource -Name`
- `Get-Variable -Name`

Add support for the W3C Extended Log File Format in `Import-Csv`

Previously, the `Import-Csv` cmdlet cannot be used to directly import the log files in W3C extended log format and additional action would be required. With this change, W3C extended log format is supported.

`Import-Csv` applies `PSTypeNames` upon import when type information is present in the CSV

Previously, objects exported using `Export-CSV` with `TypeInformation` imported with `ConvertFrom-Csv` were not retaining the type information. This change adds the type information to `PSTypeNames` member if available from the CSV file.

`-NoTypeInformation` is the default on `Export-Csv`

Previously, the `Export-CSV` cmdlet would output a comment as the first line containing the type name of the object. The change excludes the type information by default because it's not understood by most CSV tools. This change was made to address customer feedback.

Use `-IncludeTypeInformation` to retain the previous behavior.

Allow `*` to be used in registry path for `Remove-Item`

Previously, `-LiteralPath` given a wildcard would treat it the same as `-Path` and if the wildcard found no files, it would silently exit. Correct behavior should be that `-LiteralPath` is literal so if the file doesn't exist, it should error. Change is to treat wildcards used with `-Literal` as literal.

Group-Object now sorts the groups

As part of the performance improvement, `Group-Object` now returns a sorted listing of the groups. Although you should not rely on the order, you could be broken by this change if you wanted the first group. We decided that this performance improvement was worth the change since the impact of being dependent on previous behavior is low.

Standard deviation in `Measure-Object`

The output from `Measure-Object` now includes a `StandardDeviation` property.

```
Get-Process | Measure-Object -Property CPU -AllStats
```

```
Count             : 308
Average           : 31.3720576298701
Sum               : 9662.59375
Maximum           : 4416.046875
Minimum           :
StandardDeviation : 264.389544720926
Property          : CPU
```

`Get-PfxCertificate -Password`

`Get-PfxCertificate` now has the `Password` parameter, which takes a `SecureString`. This allows you to use it non-interactively:

```
$certFile = '\\server\share\pwd-protected.pfx'
$certPass = Read-Host -AsSecureString -Prompt 'Enter the password for
certificate: '

$certThumbPrint = (Get-PfxCertificate -FilePath $certFile -Password
$certPass ).ThumbPrint
```

Removal of the `more` function

In the past, PowerShell shipped a function on Windows called `more` that wrapped `more.com`. That function has now been removed.

Also, the `help` function changed to use `more.com` on Windows, or the system's default pager specified by `$env:PAGER` on non-Windows platforms.

`cd DriveName:` now returns users to the current working directory in that drive

Previously, using `Set-Location` or `cd` to return to a PSDrive sent users to the default location for that drive. Users are now sent to the last known current working directory for that session.

`cd -` returns to previous directory

```
C:\Windows\System32> cd C:\
C:\> cd -
C:\Windows\System32>
```

Or on Linux:

```
PS /etc> cd /usr/bin
PS /usr/bin> cd -
PS /etc>
```

Also, `cd` and `cd --` change to `$HOME`.

`Update-Help` as non-admin

By popular demand, `Update-Help` no longer needs to be run as an administrator. `Update-Help` now defaults to saving help to a user-scoped folder.

`Where-Object -Not`

With the addition of `-Not` parameter to `Where-Object`, can filter an object at the pipeline for the non-existence of a property, or a null/empty property value.

For example, this command returns all services that don't have any dependent services defined:

```
Get-Service | Where-Object -Not DependentServices
```

Changes to Web Cmdlets

The underlying .NET API of the Web Cmdlets has been changed to `System.Net.Http.HttpClient`. This change provides many benefits. However, this change along with a lack of interoperability with Internet Explorer have resulted in several breaking changes within `Invoke-WebRequest` and `Invoke-RestMethod`.

- `Invoke-WebRequest` now supports basic HTML Parsing only. `Invoke-WebRequest` always returns a `BasicHtmlWebResponseObject` object. The `ParsedHtml` and `Forms` properties have been removed.
- `BasicHtmlWebResponseObject.Headers` values are now `String[]` instead of `String`.
- `BasicHtmlWebResponseObject.BaseResponse` is now a `System.Net.Http.HttpResponseMessage` object.
- The `Response` property on Web Cmdlet exceptions is now a `System.Net.Http.HttpResponseMessage` object.
- Strict RFC header parsing is now default for the `-Headers` and `-UserAgent` parameter. This can be bypassed with `-SkipHeaderValidation`.
- `file://` and `ftp://` URI schemes are no longer supported.
- `System.Net.ServicePointManager` settings are no longer honored.
- There is currently no certificate based authentication available on macOS.
- Use of `-Credential` over an `http://` URI will result in an error. Use an `https://` URI or supply the `-AllowUnencryptedAuthentication` parameter to suppress the error.
- `-MaximumRedirection` now produces a terminating error when redirection attempts exceed the provided limit instead of returning the results of the last redirection.
- In PowerShell 6.2, a change was made to default to UTF-8 encoding for JSON responses. When a charset is not supplied for a JSON response, the default encoding should be UTF-8 per RFC 8259.
- Default encoding set to UTF-8 for `application-json` responses

- Added `-SkipHeaderValidation` parameter to allow `Content-Type` headers that aren't standards-compliant
- Added `-Form` parameter to support simplified `multipart/form-data` support
- Compliant, case-insensitive handling of relation keys
- Added `-Resume` parameter for web cmdlets

Invoke-RestMethod returns useful info when no data is returned

When an API returns just `null`, `Invoke-RestMethod` was serializing this as the string `"null"` instead of `$null`. This change fixes the logic in `Invoke-RestMethod` to properly serialize a valid single value JSON `null` literal as `$null`.

Web Cmdlets warn when `-Credential` is sent over unencrypted connections

When using HTTP, content including passwords are sent as clear-text. This change is to not allow this by default and return an error if credentials are being passed insecurely. Users can bypass this by using the `-AllowUnencryptedAuthentication` switch.

Make `-OutFile` parameter in web cmdlets to work like `-LiteralPath`

Beginning in PowerShell 7.1, the **OutFile** parameter of the web cmdlets works like **LiteralPath** and does not process wildcards.

API changes

Remove `AddTypeCommandBase` class

The `AddTypeCommandBase` class was removed from `Add-Type` to improve performance. This class is only used by the `Add-Type` cmdlet and should not impact users.

Removed `VisualBasic` as a supported language in Add-Type

In the past, you could compile Visual Basic code using the `Add-Type` cmdlet. Visual Basic was rarely used with `Add-Type`. We removed this feature to reduce the size of PowerShell.

Removed `RunspaceConfiguration` support

Previously, when creating a PowerShell runspace programmatically using the API, you could use the legacy `RunspaceConfiguration` or the newer `InitialSessionState` classes. This change removed support for `RunspaceConfiguration` and only supports `InitialSessionState`.

`CommandInvocationIntrinsics.InvokeScript` bind arguments to `$input` instead of `$args`

An incorrect position of a parameter resulted in the args passed as input instead of as args.

Remove `ClrVersion` and `BuildVersion` properties from `$PSVersionTable`

The `ClrVersion` property of `$PSVersionTable` is not useful with CoreCLR. End users should not be using that value to determine compatibility.

The `BuildVersion` property was tied to the Windows build version, which is not available on non-Windows platforms. Use the `GitCommitId` property to retrieve the exact build version of PowerShell.

Implement Unicode escape parsing

`` `u#### `` or `` `u{####} `` is converted to the corresponding Unicode character. To output a literal `` `u ``, escape the backtick: ``` ``u ```.

Parameter binding problem with `ValueFromRemainingArguments` in PS functions

`ValueFromRemainingArguments` now returns the values as an array instead of a single value which itself is an array.

Cleaned up uses of `CommandTypes.Workflow` and `WorkflowInfoCleaned`

Clean up code related to the uses of `CommandTypes.Workflow` and `WorkflowInfo` in **System.Management.Automation**.

These minor breaking changes mainly affect help provider code.

- Change the public constructors of `WorkflowInfo` to internal. We don't support workflow anymore, so it makes sense to not allow people to create `Workflow` instances.
- Remove the type **System.Management.Automation.DebugSource** since it's only used for workflow debugging.
- Remove the overload of `SetParent` from the abstract class **Debugger** that is only used for workflow debugging.
- Remove the same overload of `SetParent` from the derived class **RemotingJobDebugger**.

Do not wrap return result in `PSObject` when converting a `ScriptBlock` to a delegate

When a `ScriptBlock` is converted to a delegate type to be used in C# context, wrapping the result in a `PSObject` brings unneeded troubles:

- When the value is converted to the delegate return type, the `PSObject` essentially gets unwrapped. So the `PSObject` is unneeded.
- When the delegate return type is `object`, it gets wrapped in a `PSObject` making it hard to work with in C# code.

After this change, the returned object is the underlying object.

Remoting Support

PowerShell Remoting (PSRP) using WinRM on Unix platforms requires NTLM/Negotiate or Basic Auth over HTTPS. PSRP on macOS only supports Basic Auth over HTTPS. Kerberos-based authentication is not supported for non-Windows platforms.

PowerShell also supports PowerShell Remoting (PSRP) over SSH on all platforms (Windows, macOS, and Linux). For more information, see SSH remoting in PowerShell.

PowerShell Direct for Containers tries to use pwsh first

PowerShell Direct is a feature of PowerShell and Hyper-V that allows you to connect to a Hyper-V VM or Container without network connectivity or other remote management services.

In the past, PowerShell Direct connected using the built-in Windows PowerShell instance on the Container. Now, PowerShell Direct first attempts to connect using any available `pwsh.exe` on the `PATH` environment variable. If `pwsh.exe` isn't available, PowerShell Direct falls back to use `powershell.exe`.

`Enable-PSRemoting` now creates separate remoting endpoints for preview versions

`Enable-PSRemoting` now creates two remoting session configurations:

- One for the major version of PowerShell. For example, `PowerShell.6`. This endpoint that can be relied upon across minor version updates as the "system-wide" PowerShell 6 session configuration
- One version-specific session configuration, for example: `PowerShell.6.1.0`

This behavior is useful if you want to have multiple PowerShell 6 versions installed and accessible on the same machine.

Additionally, preview versions of PowerShell now get their own remoting session configurations after running the `Enable-PSRemoting` cmdlet:

```
C:\WINDOWS\system32> Enable-PSRemoting
```

Your output may be different if you haven't set up WinRM before.

```
WinRM is already set up to receive requests on this computer.
WinRM is already set up for remote management on this computer.
```

Then you can see separate PowerShell session configurations for the preview and stable builds of PowerShell 6, and for each specific version.

```
Get-PSSessionConfiguration
```

```
Name          : PowerShell.6.2-preview.1
PSVersion     : 6.2
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\INTERACTIVE AccessAllowed,
BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users
AccessAllowed

Name          : PowerShell.6-preview
PSVersion     : 6.2
StartupScript :
RunAsUser     :
```

```
Permission    : NT AUTHORITY\INTERACTIVE AccessAllowed,
BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users
AccessAllowed

Name          : powershell.6
PSVersion     : 6.1
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\INTERACTIVE AccessAllowed,
BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users
AccessAllowed

Name          : powershell.6.1.0
PSVersion     : 6.1
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\INTERACTIVE AccessAllowed,
BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users
AccessAllowed
```

`user@host:port` syntax supported for SSH

SSH clients typically support a connection string in the format `user@host:port`. With the addition of SSH as a protocol for PowerShell Remoting, we've added support for this format of connection string:

```
Enter-PSSession -HostName fooUser@ssh.contoso.com:2222
```

Telemetry can only be disabled with an environment variable

PowerShell sends basic telemetry data to Microsoft when it is launched. The data includes the OS name, OS version, and PowerShell version. This data allows us to better understand the environments where PowerShell is used and enables us to prioritize new features and fixes.

To opt-out of this telemetry, set the environment variable `POWERSHELL_TELEMETRY_OPTOUT` to `true`, `yes`, or `1`. We no longer support deletion of the file `DELETE_ME_TO_DISABLE_CONSOLEHOST_TELEMETRY` to disable telemetry.
Collaborate with us on GitHub
The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

In this article

Previous Chapter Next Chapter