3

Exploring Compatibility with Windows PowerShell

In this chapter, we cover the following recipes:

* Exploring compatibility with Windows PowerShell
* Using the Windows PowerShell compatibility solution
* Exploring compatibility solution limitations
* Exploring the module deny list
* Importing format XML
* Leveraging compatibility

# Introduction

Microsoft built Windows PowerShell to work with Microsoft’s .NET Framework. You can think of PowerShell as a layer on top of .NET. When you use Get-ChildItem to return file or folder details, the cmdlet invokes a .NET class to do much of the heavy lifting involved. In Chapter 5, Exploring .NET, you learn more about .NET.

As Windows PowerShell evolved, each new version took advantage of improvements in newer versions of .NET to provide additional features.

In 2014, Microsoft announced that they would release the .NET Framework as open source, to be known as .NET Core. Microsoft also decided to freeze development of Windows PowerShell, in favor of open sourcing Windows PowerShell. The first two initial versions were known as PowerShell Core. With the release of PowerShell 7.0, the team dropped the name “Core” and announced that future versions are to be known as just PowerShell. This book refers to the latest version as simply PowerShell 7.

As an IT professional, the critical question you should be asking is whether PowerShell 7 can run your specific workload. The answer, as ever, is “it depends”. It depends on the specific Windows PowerShell features and modules/commands on which you rely. And since there are many product teams involved, a fully coordinated solution is not straightforward. As this book shows, there is very little you were able to do in Windows PowerShell that you cannot do in PowerShell 7.

## Module compatibility

In terms of compatibility with Windows PowerShell, there are three sets of modules containing different commands to look at:

* Modules/commands shipped with Windows PowerShell and on which your scripts depend.
* Modules/commands included as part of a Windows feature – these contain commands to manage aspects of Windows service, usually as part of the **Remote Server Administration Tool** (**RSAT**), which you can load independently of the services themselves.
* Other third-party modules, such as the NTFSSecurity module. This book makes use of several external modules whose authors may, or may not, have updated their modules to run on .NET Core.

Windows PowerShell included modules that provide basic PowerShell functionality. For example, the Microsoft.PowerShell.Management module contains the Get-Process cmdlet. With PowerShell 7, the development team re-implemented the fundamental Windows PowerShell commands to provide good fidelity with Windows PowerShell. A few Windows PowerShell commands made use of proprietary APIs, so they could not be included, but the number is relatively small.

These core modules (Microsoft.PowerShell.\*) now reside in the PowerShell installation folder, which simplifies side-by-side use of multiple versions of PowerShell. And through the use of the environment variable PSModulePath, when you call a cmdlet in PowerShell 7, you get the “right” cmdlet and supported underpinnings. It is important to note that these core modules were developed by the Windows PowerShell team initially and are not part of the open source PowerShell.

The Windows client and Windows Server also provide modules that enable you to manage your systems. Modules like the Active Directory module give you the tools to manage AD in your environment. These Windows modules live in the C:\Windows\System32\WindowsPowerShell\v1.0\modules folder.

The ownership of these operating system-related modules lies with the individual product teams, some of whom have not been motivated to update their modules to run with .NET Core. Like so many things, this is a work in progress. Some modules, such as the ActiveDirectory module, have been updated and are available natively in PowerShell 7. For a quick look at compatibility, view this document: https://docs.microsoft.com/en-us/powershell/scripting/whats-new/module-compatibility?view=powershell-7.1. It is worth noting that updating these OS modules can only be done by upgrading to a newer version of the OS, which is unfortunate.

The final set of commands to think about are third-party modules, possibly obtained from the PowerShell gallery, Spiceworks, or a host of other online sources. As the recipes in this book demonstrate, some third-party modules work fine natively in the latest versions of PowerShell 7, while others do not and may never work.

In the Exploring compatibility with Windows PowerShell recipe, you look specifically at the new commands you have available in PowerShell 7, as well as where PowerShell and Windows PowerShell find core commands.

Since many of the Windows product teams and other module providers might not update their modules to work natively, the PowerShell team devised a compatibility solution that enables you to import and use Windows PowerShell modules more or less seamlessly in PowerShell 7. In the Using the Windows PowerShell compatibility solution recipe, you investigate the workings of the Windows PowerShell compatibility solution.

## Incompatible modules

With the compatibility solution, the modules developed for Windows PowerShell work well, as demonstrated by many recipes in this book. However, there are some small and some more fundamental issues in a few cases. Some things do not and may never work inside PowerShell 7 and beyond. For instance, despite the compatibility solution, the BestPractices, PSScheduledJob, and WindowsUpdate modules will not work in PowerShell 7.

With Windows Powershell you use WindowsUpdate to manage **Windows Server Update Services** (**WSUS**). WSUS, the UpdateServices module, makes use of object methods. With the compatibility mechanism, these methods are not available in a PowerShell 7 session. Additionally, the module makes use of the SOAP protocol for which .NET Core provides no support (and is not likely to anytime soon). The other two modules make use of .NET types, which are not provided with .NET Core and are not likely to be updated.

Other Windows PowerShell features that are not going to work in Powershell 7 include:

* Windows PowerShell workflows: Workflows require the Windows Workflow Foundation component of .NET, which the .NET team has not ported to .NET Core.
* Windows PowerShell snap-ins: Snap-ins provided a way to add new cmdlets in PowerShell V2. Modules replaced snap-ins as an add-in mechanism in PowerShell V2, and are the sole method you use to add commands to PowerShell. You can convert some older snap-ins into modules by creating a module manifest, but most require the authors to re-engineer their modules.
* The WMI cmdlets: The CIMCmdlets module replaces the older WMI Cmdlets.
* Desired State Configuration: The core functions of DSC are not available to you from within PowerShell 7.
* The WebAdministration module’s IIS provider: This means many IIS configuration scripts do not work in PowerShell 7, even with the compatibility solution.
* The Add-Computer, Checkpoint-Computer, Remove-Computer, and Restore-Computer commands from the Microsoft.PowerShell.Management module.

While there are a few Windows PowerShell features and commands you cannot use in PowerShell 7, even with the help of the Windows PowerShell compatibility solution, you can always use Windows PowerShell.

As noted above, there are just three Microsoft authored modules that do not work in PowerShell 7, natively or via the compatibility mechanism. If you try to use these, you will receive unhelpful error messages. To avoid an awful user experience (and no doubt tons of support calls), the PowerShell team devised a list of modules that PowerShell does not load. If you try to use these modules, you get an error message, as you see in the *Exploring the module deny list* recipe.

One downside to the compatibility mechanism is that any format XML included with the module does not get loaded into the PowerShell 7 session. The result is that PowerShell 7 does not format objects as nicely. Fortunately, there is a way around this, as you see in the Importing format XML recipe.

The Windows PowerShell compatibility solution makes use of a specially created PowerShell remoting session. In Leveraging compatibility, you look at that session and learn how you can take advantage of it.

Despite compatability issues, it is more than worthwhile to move forward to PowerShell 7 and away from Windows PowerShell. One key reason is improved performance. You can use the ForEach-Object -Parallel construct to run script blocks in parallel without having to resort to workflows. If your script has to perform actions on a large number of computers, or you are processing a large array, the performance improvements in PowerShell 7 justify moving forward. On top of this, there are all the newly added features, which make life so much easier.

# Exploring compatibility with Windows PowerShell

When you invoke a cmdlet, PowerShell has to load the module containing the cmdlet and can then run that cmdlet. By default, PowerShell uses the paths on the environment variable $env:PSModulePath to discover the modules and the cmdlets contained in those modules.

As this recipe shows, the set of paths held by $env:PSModulePath has changed between Windows PowerShell 5.1 and PowerShell 7.

In this recipe, you examine the paths which PowerShell uses by default to load modules. You also look at the new commands now provided in PowerShell 7.

## Getting ready

You use SRV1 for this recipe after you have loaded PowerShell 7.1 and VS Code.

## How to do it...

1. Ensuring PowerShell remoting is fully enabled:

Enable-PSRemoting -Force -WarningAction SilentlyContinue |

  Out-Null

1. Getting session using endpoint for Windows PowerShell 5.1:

$SHT1 = @{

  ComputerName      = 'localhost'

  ConfigurationName = 'microsoft.powershell'

}

$SWP51 = New-PSSession @SHT1

1. Getting session using PowerShell 7.1 endpoint:

$CNFName = Get-PSSessionConfiguration |

             Where-Object PSVersion -eq '7.1' |

               Select-Object -Last 1

$SHT2 = @{

  ComputerName      = 'localhost'

  ConfigurationName = $CNFName.Name

}

$SP71    = New-PSSession @SHT2

1. Defining a script block to view default module paths:

$SBMP = {

  $PSVersionTable

  $env:PSModulePath -split ';'

}

1. Reviewing paths in Windows PowerShell 5.1:

Invoke-Command -ScriptBlock $SBMP -Session $SWP51

1. Reviewing paths in PowerShell 7.1:

Invoke-Command -ScriptBlock $SBMP -Session $SP71

1. Creating a script block to get commands in PowerShell:

$SBC = {

  $ModPaths = $ENv:PSModulePath -split ';'

  $CMDS = @()

  Foreach ($ModPath in $ModPaths) {

    if (!(Test-Path $Modpath)) {Continue}

    # Process modules found in an existing module path

    $Mods = Get-ChildItem -Path $ModPath -Directory

    foreach ($Mod in $Mods){

      $Name  = $Mod.Name

      $Cmds  += Get-Command -Module $Name

    }

  }

  $Cmds # return all commands discovered

}

1. Discovering all 7.1 cmdlets:

$CMDS71 = Invoke-Command -ScriptBlock $SBC -Session $SP71 |

            Where-Object CommandType -eq 'Cmdlet'

"Total commands available in PowerShell  7.1 [{0}]" -f $Cmds71.count

1. Discovering all 5.1 cmdlets:

$CMDS51 = Invoke-Command -ScriptBlock $SBC -Session $SWP51 |

            Where-Object CommandType -eq 'Cmdlet'

"Total commands available in PowerShell  5.1 [{0}]" -f $Cmds51.count

1. Creating arrays of just cmdlet names:

$Commands51 = $CMDS51 |

  Select-Object -ExpandProperty Name |

    Sort-Object -Unique

$Commands71 = $CMDS71 |

  Select-Object -ExpandProperty Name |

    Sort-Object -Unique

1. Discovering new commands in PowerShell 7.1:

Compare-Object $Commands51 $Commands71  |

  Where-Object sideindicator -match '^=>'

1. Creating a script block to check core PowerShell modules:

$CMSB = {

  $M = Get-Module -Name 'Microsoft.PowerShell\*' -ListAvailable

  $M

  "$($M.count) modules found in $($PSVersionTable.PSVersion)"

}

1. Viewing core modules in Windows PowerShell 5.1:

Invoke-Command -Session $SWP51 -ScriptBlock $CMSB

1. Viewing core modules in PowerShell 7.1:

Invoke-Command -Session $SP71 -ScriptBlock $CMSB

## How it works...

In step 1, you ensure that PowerShell remoting is enabled fully within PowerShell 7. This should not be necessary, but this step, which produces no output, makes sure that remoting is fully enabled and ensures that the endpoints you use later in this recipe exist.

In step 2, you create a new PowerShell remoting session to a Windows PowerShell 5.1 endpoint on the local computer. The endpoint’s configuration name, microsoft.powershell, is a well-known configuration name that runs Windows PowerShell 5.1.

With step 3, you then repeat the operation and create a new PowerShell remoting session to a PowerShell 7.1 endpoint. This step produces no output.

In step 4, you create a simple script block that displays the PowerShell version table and shows the endpoint’s module paths.

In step 5, you view the version of PowerShell running (inside the remoting session) and that version’s module paths. The output of this step looks like this:

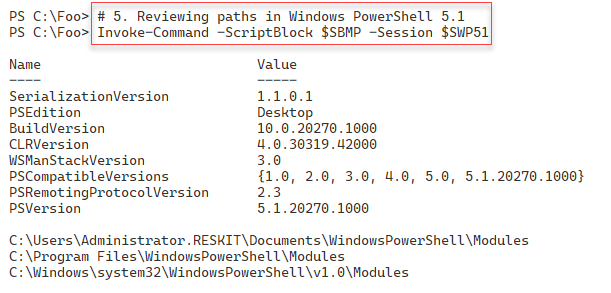


Figure 3.1: Inspecting the PowerShell 5.1 version and module paths

Insert image B42024\_03\_01.png

In step 6, you view the paths inside a PowerShell 7.1 endpoint, which looks like this:

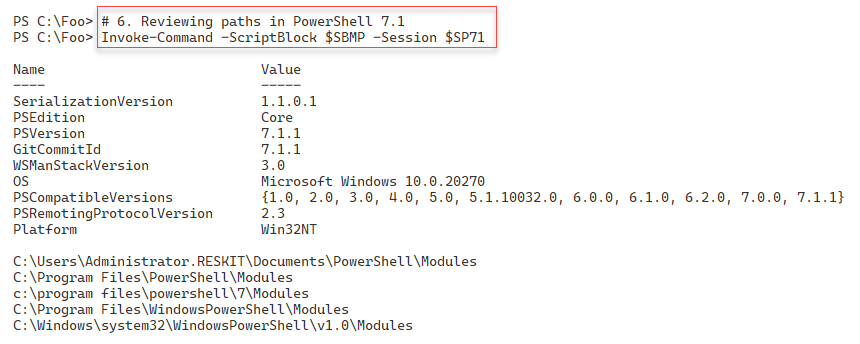


Figure 3.2: Inspecting the PowerShell 7.1 version and module paths

Insert image B42024\_03\_02.png

In step 7, you create a script block which gets the details of each module available in a given remoting endpoint. This step produces no output. In step 8, you run the script block in a PowerShell 7.1 remoting endpoint to discover the commands available, which looks like this:

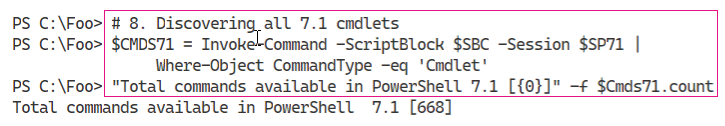


Figure 3.3: Discovering all cmdlets in PowerShell 7.1

Insert image B42024\_03\_03.png

In step 9, you run this script block inside the Windows PowerShell endpoint to produce output like the following:

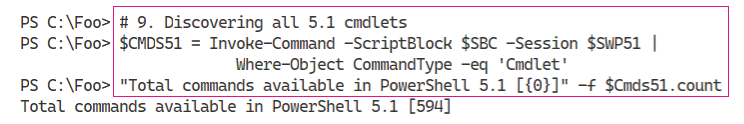


Figure 3.4: Discovering all cmdlets in PowerShell 5.1

Insert image B42024\_03\_04.png

In step 10, you create arrays to contain the name of the commands available. You compare these in step 11, to discover the commands in PowerShell 7.1 that are not in Windows PowerShell 5.1, which looks like this:



Figure 3.5: Discovering cmdlets that are in PowerShell 7.1 but not 5.1

Insert image B42024\_03\_05.png

In step 12, you create a script block that you can use to discover the core PowerShell modules and their file storage locations. This step creates no output. In step 13, you run this script block inside the Windows PowerShell remoting session you created earlier, which looks like this:

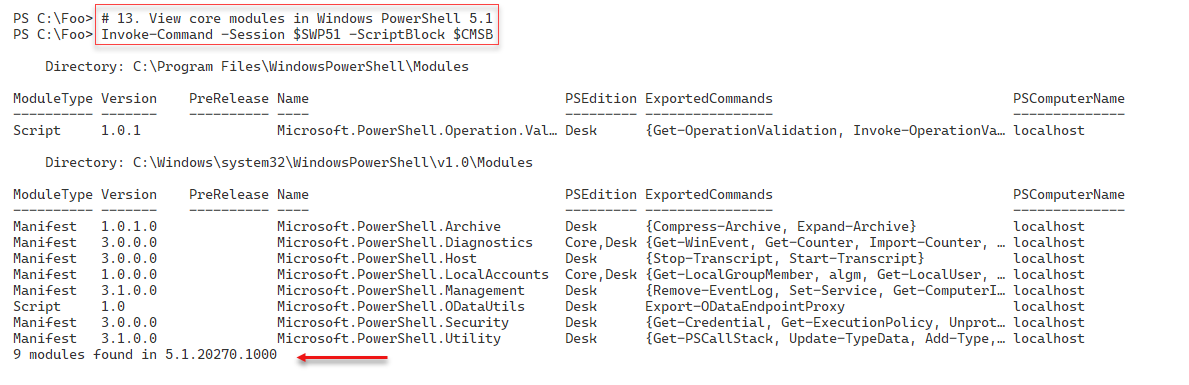


Figure 3.6: Viewing core modules in PowerShell 5.1

Insert image B42024\_03\_06.png

You then rerun this script block in a PowerShell 7.1 endpoint, in step 14. The output from this step looks like this:

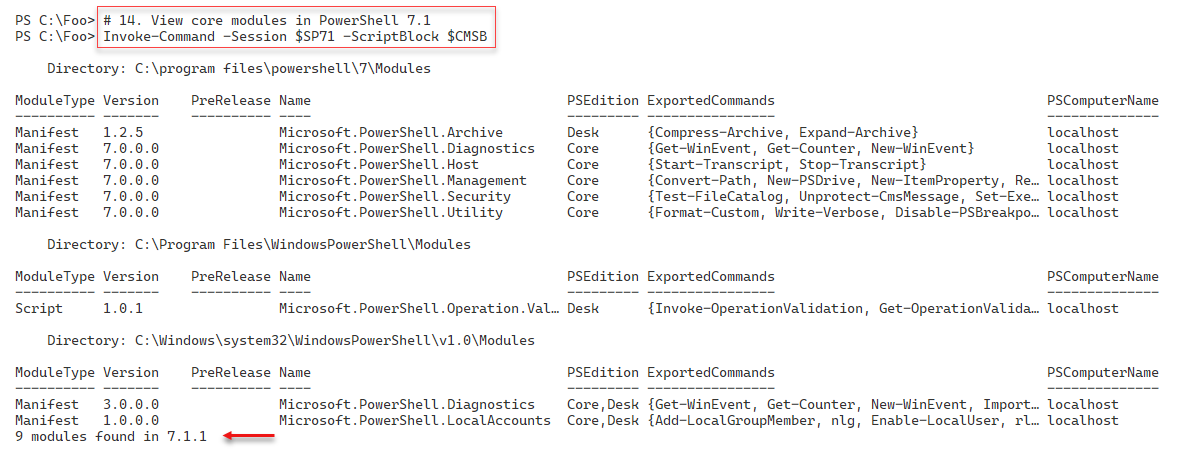


Figure 3.7: Viewing core modules in PowerShell 7.1

Insert image B42024\_03\_07.png

## There’s more...

The code in step 2 creates a PowerShell remoting endpoint to a well-known endpoint for Windows PowerShell 5. You cannot view Windows PowerShell remoting endpoint configurations from within PowerShell 7.1.

In step 3, you get a session that provides PowerShell 7.1 using a slightly different technique than you used in step 2. PowerShell 7.1 has multiple endpoints, especially if you have both a release version and a preview version loaded side-by-side. You can use Get‑PSSessionConfiguration to view the available remoting endpoint configurations in your specific PowerShell 7 host. Thus, you do not see the Windows PowerShell remoting endpoints inside PowerShell 7. And although the Get-PSSessionConfiguration cmdlet does not return any Windows PowerShell configuration details, the remoting endpoints do exist, and you can use them, as you see in step 2.

In steps 5 and 6, you discover the default module paths in Windows PowerShell and PowerShell 7.1. As you can see, in PowerShell 7, there are five default module paths, versus just three in Windows PowerShell 5.1. The first three module folders with PowerShell 7 enable PowerShell 7 to pick up commands from PowerShell 7.1-specific folders (if they exist), and the last two modules enable you to access older Windows PowerShell commands which you do not have with PowerShell 7.

In steps 13 and 14, you discover the core modules inside Windows PowerShell. These core modules contain the basic internal PowerShell commands, such as Get-ChildItem, found in the Microsoft.PowerShell.Management module.

# Using the Windows PowerShell compatibility solution

The PowerShell 7 Windows compatibility solution allows you to use older Windows PowerShell commands whose developers have not (yet) ported the commands to work natively in PowerShell 7. PowerShell 7 creates a special remoting session into a Windows PowerShell 5.1 endpoint, loads the modules into the remote session, then uses implicit remoting to expose proxy functions inside the PowerShell 7 session. This remoting session has a unique session name, WinPSCompatSession. Should you use multiple Windows PowerShell modules, PowerShell 7 loads them all into a single remoting session. Also, this session uses the “process” transport mechanism versus **Windows Remote Management** (**WinRM**). WinRM is the core transport protocol used with PowerShell remoting. The process transport is the transport used to run background jobs; it has less overhead than using WinRM, so is more efficient.

An example of the compatibility mechanism is using Get-WindowsFeature, a cmdlet inside the ServerManager module. You use the command to get details of features which are installed, or not, inside Windows Server. You use other commands in the ServerManager module to install and remove features. Unfortunately, the Windows Server team has not yet updated this module to work within PowerShell core. Via the compatibility solution, the commands in the ServerManager module enable you to add, remove, and view features. The Windows PowerShell compatibility mechanism allows you to use existing Windows PowerShell scripts in PowerShell 7, although with some very minor caveats.

When you invoke commands in PowerShell 7, PowerShell uses its command discovery mechanism to determine which module contains your desired command. In this case, that module is the ServerManager Windows PowerShell module. PowerShell 7 then creates the remoting session and, using implicit remoting, imports the commands in the module as proxy functions. You then invoke the proxy functions to accomplish your goal. For the most part, this is totally transparent. You use the module’s commands, and they return the object(s) you request. A minor caveat is that the compatibility mechanism does not import format XML for the Windows PowerShell module. The result is that the default output of some objects is not the same. There is a workaround for this, described in the Importing format XML recipe.

With implicit remoting, PowerShell creates a function inside a PowerShell 7 session with the same name and parameters as the actual command (in the remote session). You can view the function definition in the Function drive (Get-Item Function:Get-WindowsFeature | Format-List -Property \*). The output shows the proxy function definition which PowerShell 7 creates when it imports the remote module.

When you invoke the command by name, for example, Get‑WindowsFeature, PowerShell runs the function. The function then invokes the remote cmdlet using the steppable pipeline. Implicit remoting is a complex feature that is virtually transparent in operation. You can read more about implicit remoting at https://www.techtutsonline.com/implicit-remoting-windows-powershell/.

## Getting ready

You run this recipe on SRV1, after installing PowerShell 7 and VS Code.

## How to do it…

1. Discovering the ServerManager module:

Get-Module -Name ServerManager -ListAvailable

1. Discovering a command in the ServerManager module:

Get-Command -Name Get-WindowsFeature

1. Importing the module explicitly:

Import-Module -Name ServerManager

1. Discovering the commands inside the module:

Get-Module -Name ServerManager | Format-List

1. Using a command in the ServerManager module:

Get-WindowsFeature -Name TFTP-Client

1. Running the command in a remoting session:

$Session = Get-PSSession -Name WinPSCompatSession

Invoke-Command -Session $Session -ScriptBlock {

  Get-WindowsFeature -Name DHCP |

    Format-Table

}

1. Removing the ServerManager module from the current session:

Get-Module -Name ServerManager |

  Remove-Module

1. Installing a Windows feature using module autoload:

Install-WindowsFeature -Name TFTP-Client

1. Discovering the feature:

Get-WindowsFeature -Name TFTP-Client

1. Viewing output inside Windows PowerShell session:

Invoke-Command -Session $Session -ScriptBlock {

    Get-WindowsFeature -Name 'TFTP-Client' |

      Format-Table

}

## How it works...

In step 1, you use the Get-Module command to discover the commands in the ServerManager module. Despite using the -ListAvailable switch, since this module does not work natively in PowerShell 7, this command returns no output.

In step 2, you use Get-Command to discover the Get-WindowsFeature command. The output looks like this:

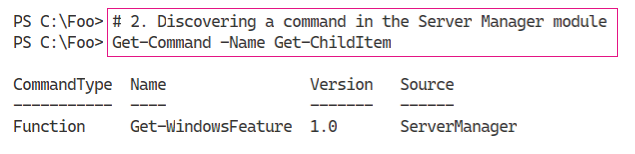


Figure 3.8: Discovering the Get-WindowsFeature command

Insert image B42024\_03\_08.png

In step 3, you import the ServerManager module explicitly, which looks like this:

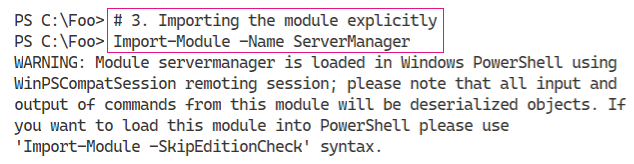


Figure 3.9: Importing the ServerManager module explictly

Insert image B42024\_03\_09.png

Now that PowerShell 7 has imported the module, you can now see the output from Get-Module. The output from step 4 looks like this:

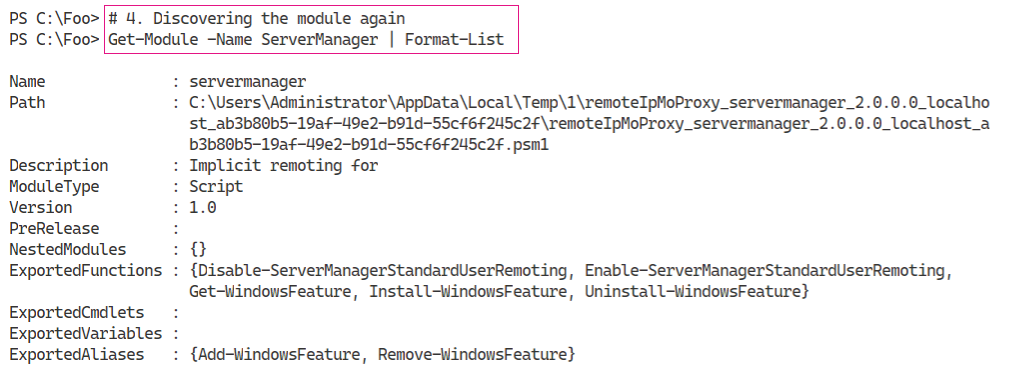


Figure 3.10: Discovering the ServerManager module

Insert image B42024\_03\_10.png

To illustrate that an older Windows PowerShell command works with PowerShell 7, in step 5, you invoke Get-WindowsFeature. This command discovers whether the TFTP Client feature is installed, which looks like this:

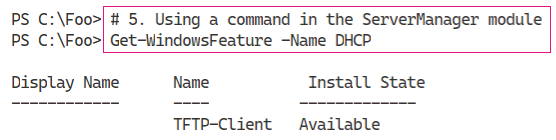


Figure 3.11: Invoking the Get-WindowsFeature command

Insert image B42024\_03\_11.png

In step 6, you rerun the Get-WindowsFeature command inside the Windows PowerShell remoting session, with output like this:

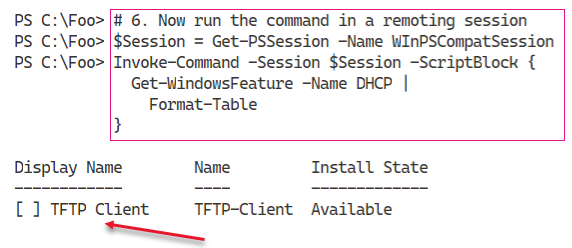


Figure 3.12: Invoking Get-WindowsFeature in a remoting session

Insert image B42024\_03\_12.png

In step 7, you remove the ServerManager module from the current PowerShell session, producing no output. In step 8, you install the TFTP-Client feature, which looks like this:

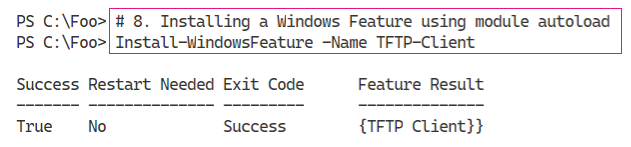


Figure 3.13: Installing TFTP-Client

Insert image B42024\_03\_13.png

In step 9, you rerun the Get-WindowsFeature command to check the state of the TFTP-Client feature, which looks like this:

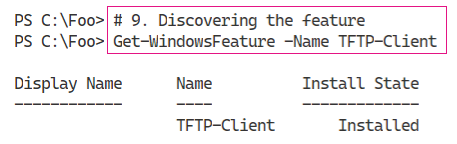


Figure 3.14: Checking the state of TFTP-Client

Insert image B42024\_03\_14.png

In step 10, you re-view the TFTP-Client feature inside the Windows PowerShell remoting session, which looks like this:

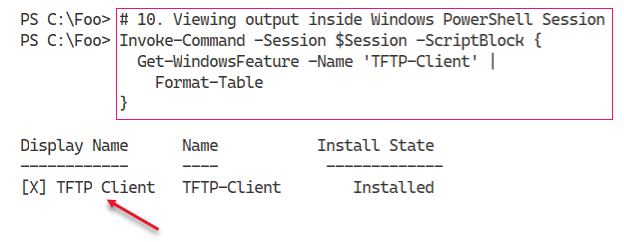


Figure 3.15: Viewing TFTP-Client inside the remoting session

Insert image B42024\_03\_15.png

## There’s more...

In step 1, you attempted to find the ServerManager module using the -ListAvailable switch. This module is not available natively in PowerShell 7, hence the lack of output. Even though the compatibility mechanism can find the module, by default, Get-Module does not display the module.

In step 2, you discover the Get-WindowsFeature command comes from the ServerManager module, which you then load explicitly in step 3. Loading the module using Import-Module generates the warning message you can see in the output. Note that in PowerShell 7, the Get-WindowsFeature command is a function, rather than a cmdlet. The implicit remoting process creates a proxy function (in the PowerShell 7 session), which then invokes the underlying command in the remote session. You can examine the function’s definition to see how the proxy function invokes the remote function.

In steps 9 and 10, you view the output from Get-WindowsFeature inside PowerShell 7 and then inside the remoting session to Windows PowerShell. In step 9, you see a different output from step 10, which is the result of PowerShell 7 not importing the format XML for this module. You can see how to get around this minor issue in *Importing format XML* later in this chapter.

The Windows PowerShell compatibility mechanism in PowerShell 7 does an excellent job of supporting otherwise incompatible Windows PowerShell modules. But even with this, there are some commands and modules that are just not going to work.

# Exploring compatibility solution limitations

In the previous recipe, you saw how you could use the Windows PowerShell compatibility mechanism built into PowerShell 7. This solution provides you with access to modules that their owners have not yet converted to run natively in PowerShell 7. This solution provides improved compatibility but does have some minor limitations.

The first limitation is discovery. You can’t easily discover unsupported commands. You cannot, for example, use PowerShell 7’s Get-Module to list the ServerManager module, even though you can use Get-Command to discover commands inside the module.

Another limitation of the compatibility solution is that PowerShell 7 does not import any display or type XML contained in the module. The result is that some commands may display output slightly differently. There are ways around this, including just running the entire command inside a remoting session.

Despite the compatibility solution, some Windows Server modules simply do not work at all in PowerShell 7. If you load one of these modules manually, the module may load, but some or all of the commands in the module do not work and often return error messages that are not actionable.

## Getting ready

Run this in a new PowerShell session on SRV1 after you have installed PowerShell 7 and VS Code.

## How to do it...

1. Attempting to view a Windows PowerShell module:

Get-Module -Name ServerManager -ListAvailable

1. Trying to load a module without edition check:

Import-Module -Name ServerManager -SkipEditionCheck

1. Discovering the Get-WindowsFeature Windows PowerShell command:

Get-Command -Name Get-WindowsFeature

1. Examining the Windows PowerShell compatibility remote session:

$Session = Get-PSSession

$Session | Format-Table -AutoSize

1. Examining Get-WindowsFeature in the remote session:

$SBRC = {Get-Command -Name Get-WindowsFeature}

Invoke-Command -Session $Session -ScriptBlock $SBRC

1. Invoking Get-WindowsFeature locally:

Invoke-Command $SBRC

## How it works...

In step 1, you use the Get-Module cmdlet to get details of the ServerManager module. However, since this module is not supported natively, Get-Module returns no output when you invoke it within a PowerShell 7 session, even though you use the -ListAvailable switch.

In step 2, you use the -SkipEditionCheck switch to instruct Import-Module to try to load the ServerManager module into the PowerShell 7 session, which looks like this:

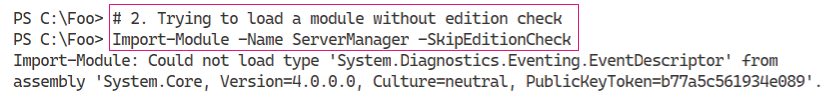


Figure 3.16: Loading ServerManager without edition check

Insert image B42024\_03\_16.png

As you can see in the output, Import-Module errors out when it attempts to load the module in PowerShell 7, with an error message stating that Import-Module was not able to load the .NET type, System.Diagnostics.Eventing.EventDescriptor. Since the EventDescriptor type is not available in .NET Core, you cannot use the ServerManager module natively. The only alternatives are for the Server Manager team to update their code, or for you to run the cmdlet in Windows PowerShell (either explicitly or via the compatibility mechanism).

In step 3, you use Get-Command to discover the Get-WindowsFeature command, which succeeds, as you can see in the output from this step:

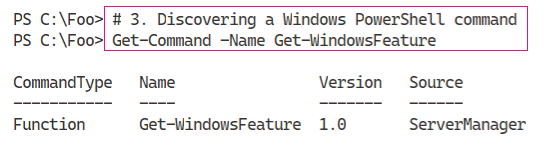


Figure 3.17: Discovering the Get-WindowsFeature command

Insert image B42024\_03\_17.png

In step 4, you use Get-PSSession to discover the Windows PowerShell compatibility remote session. The output of this step is as follows:

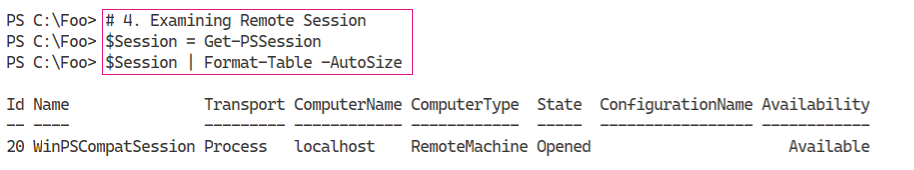


Figure 3.18: Discovering the compatibility remote session

Insert image B42024\_03\_18.png

In step 5, you invoke the Get-WindowsFeature command inside the remoting session to view the command details inside Windows PowerShell 5.1, like this:

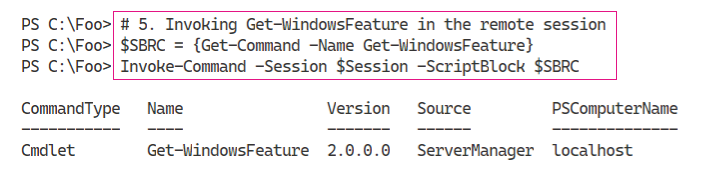


Figure 3.19: Invoking Get-WindowsFeature in the remote session

Insert image B42024\_03\_19.png

With step 6, you run Get-Command to discover the details about this command inside your PowerShell 7 console. The output of this step is as follows:

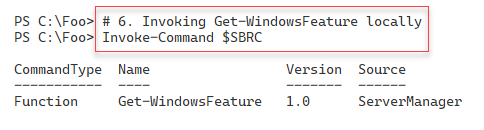


Figure 3.20: Invoking Get-WindowsFeature locally

Insert image B42024\_03\_20.png

## There’s more...

As you can see from the steps in this recipe, the Windows PowerShell compatibility solution imposes some minor restrictions on features Windows PowerShell users have grown accustomed to having. These limitations are relatively minor, and there are some workarounds.

As you can see in step 1, some modules are not directly discoverable in PowerShell 7. Attempting to force load a module into a PowerShell 7 session, as you attempt in step 2, fails (with a rather unhelpful message for the user).

In this case, when PowerShell 7 begins to load the module, PowerShell attempts to utilize a type (a .NET object) that does exist in the full .NET CLR, but not in .NET Core. This .NET type is not only not implemented in .NET core, but the .NET team have effectively deprecated it. To read through the discussions on this, see https://github.com/dotnet/core/issues/2933. The discussion in this post is an outstanding example of the transparency of open source software where you can see the problem and trace its resolution.

The ServerManager module is one of many Windows team modules that require updating before it can be used natively in PowerShell 7. Hopefully, future releases of Windows Server might address these modules. But in the meantime, the Windows PowerShell compatibility mechanism provides an excellent solution to most of the non-compatible modules.

In *steps 5* and *6*, you examine the details of the Get-WindowsFeature command. Inside Windows PowerShell 5.1, this command is a cmdlet. When, in *step 6*, you view the command inside the PowerShell 7.1 console, you can see the command is a (proxy) function. If you view the function definition (ls function:get-WindowsFeatue).definition), you can see how PowerShell uses the steppable pipeline to run the command in the remote session. For more background on the steppable pipeline, which has been a feature of Windows PowerShell since version 2, see https://livebook.manning.com/book/windows-powershell-in-action-third-edition/chapter-10/327.

# Exploring the module deny list

During the development of PowerShell 7, it became clear that a few Windows PowerShell modules did not work with PowerShell 7 despite the compatability solution. Worse, if you attempted to use them, the error messages that resulted were cryptic and non-actionable. One suggested solution was to create a list of modules that were known to not be usable within PowerShell 7. When Import-Module attempts to load any module on this list, the failure is more graceful with a cleaner message.

One possible issue that such a deny list might cause would be if the module owners were to release an updated module previously on the deny list that now works. To simplify this situation, PowerShell 7 stores the deny list in a configuration file in the $PSHOME folder.

In PowerShell 7.1, there are three modules which are in the deny list:

* PSScheduledJob: Windows PowerShell commands to manage the Windows Task Manager service. This module is installed in Windows Server by default.
* BestPractices: Windows Server commands to view, run, and view the results of best practice scans of core Windows Server services. This module is installed in Windows Server by default.
* UpdateServices: You use this module to manage the **Windows Server Update Service** (**WSUS**), which you can install along with WSUS, or as part of the **remote server administration** (**RSAT**) tools. This module makes use of object methods to manage the WSUS service, and these methods are not available via the compatibility mechanism. This module also makes use of the **Simple Object Access Protocol** (**SOAP**), which is also not implemented in .NET Core. For this reason, enabling you to manage WSUS natively in PowerShell 7 requires a significant development effort. Microsoft has not committed to undertake this work at this time.

It might be tempting to edit the deny list and to remove modules from it. However, such actions have no practical value. You can import the BestPractices module explicitly into a PowerShell 7 session, but the cmdlets in the module fail with more cryptic and non-actionable error messages.

In this recipe, you look at PowerShell’s module load deny list and discover how it works in practice.

## Getting ready

You run this recipe on SRV1, a Windows Server Datacenter Edition server with PowerShell 7 and VS Code installed.

## How to do it...

1. Getting the PowerShell configuration file:

$CFFile = "$PSHOME/powershell.config.json"

Get-Item -Path $CFFile

1. Viewing contents:

Get-Content -Path $CFFile

1. Attempting to load a module on the deny list:

Import-Module -Name BestPractices

1. Loading the module overriding edition check:

Import-Module -Name BestPractices -SkipEditionCheck

1. Viewing the module definition:

Get-Module -Name BestPractices

1. Attempting to use Get-BpaModel:

Get-BpaModel

## How it works...

In step 1, you locate and view file details of PowerShell’s configuration file, powershell.config.json. This file is in the PowerShell installation folder. You can see the output in the following screenshot:

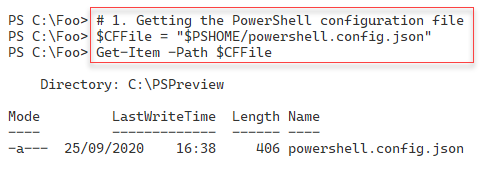


Figure 3.21: Getting the PowerShell configuration file

Insert image B42024\_03\_21.png

In step 2, you view the contents of this file, which looks like this:

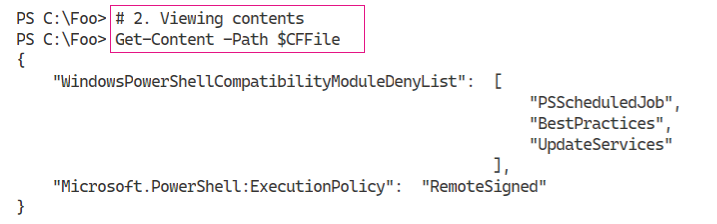


Figure 3.22: Viewing the contents of the configuration file

Insert image B42024\_03\_22.png

In step 3, you attempt to use Import-Module to import a module you can see (in the output from *step 2*) is on the module deny list. The output of this step looks like this:

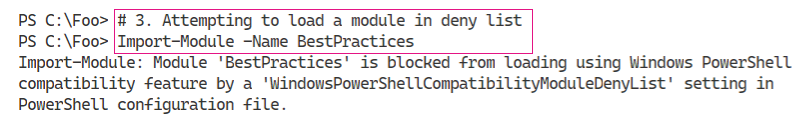


Figure 3.23: Attempting to load BestPractices, which is on the deny list

Insert image B42024\_03\_23.png

With PowerShell 7, you can force Import-Module to ignore the deny list and attempt to load the module in a PowerShell 7 session, as you can see in step 4. To do this, you specify the switch ‑SkipEditionCheck. This switch tells PowerShell to import the module without checking compatibility or attempting to load the module in the compatibility session.

With step 5, the import appears to work in that Import-Module does import the module, as you can see in the output from this step:

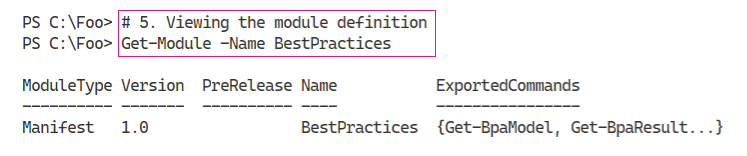


Figure 3.24: Viewing the BestPractices module definition

Insert image B42024\_03\_24.png

As you can see in step 6, running commands in modules on the deny list is likely to fail. In this case, the cmdlet Get-BpaModel makes use of a .NET type that does not exist in .NET Core, rendering the cmdlet unable to function:

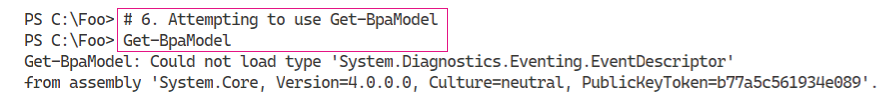


Figure 3.25: Attempting to use Get-BpaModel

Insert image B42024\_03\_25png

The compatibility mechanism means many older Windows PowerShell modules are going to be usable by your older scripts. However, some modules (just three, as it turns out) are not usable directly in PowerShell 7 or via the Windows PowerShell compatibility mechanism.

## There’s more...

In step 2, you view the configuration file. This file contains two sets of settings. The first creates the module deny list, and the other sets the starting execution policy. You can use the Set-ExecutionPolicy cmdlet to change the execution policy, in which case PowerShell writes an updated version of this file reflecting the updated execution policy. The module deny list in this configuration file contains the three module names whose contents are known not to work in PowerShell 7, as you saw in earlier recipes in this chapter.

As you can see in *steps 4* and *5*, you can import modules that are not compatible with .NET Core. The result is that, although you can import a non-compatible module, the commands in that module are not likely to work as expected. And when they fail, the error message is cryptic and not directly actionable.

In summary, the Windows PowerShell compatibility mechanism enables you to use a large number of additional commands in PowerShell 7. The mechanism is not perfect as it does impose a few limitations, but it is a functional approach in almost all cases. Furthermore, although you can, explicitly bypassing the compatibility mechanism is probably not very useful.

# Importing format XML

PowerShell, ever since the very beginning, has displayed objects automatically and with nice-looking output. By default, PowerShell display objects and properties for any object. It creates a table if the object to be displayed contains fewer than five properties, or it creates a list. PowerShell formats each property by calling the .ToString() method for each property.

You or the cmdlet developer can improve the output by using format XML. Format XML is custom-written XML that you store in a format.ps1XML file. The format XML file tells PowerShell precisely how to display a particular object type (as a table or a list), which properties to display, what headings to use (for tables), and how to display individual properties.

In Windows PowerShell, Microsoft included several format XML files which you can see in the Windows PowerShell home folder. You can view these by typing ls $PSHOME/\*.format.ps1xml.

In PowerShell 7, the default format XML is inside the code implementing each cmdlet, instead of being a separate file as in Windows PowerShell. The result is that there are, by default, no format.ps1XML files in PowerShell 7’s $PSHOME folder. As with Windows PowerShell, you can always write your own custom format XML files to customize how PowerShell displays objects by default.

The use of customized format XML is a great solution where you are displaying particular objects, typically from the command line, and the default display is not adequate for your purposes. You can implement format XML that tells PowerShell 7 to display objects as you want them displayed and import the format XML in your profile file.

An alternative to creating format XML is to pipe the objects to Format-Table or Format-List, specifying the properties that you wish to display. You can even use hash tables with Format-Table to define how properties are formatted. Depending on how often you display any given object, format XML can be useful to alter the information you see, especially at the console.

For more information about format XML, see https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about\_format.ps1xml.

Module developers often create format XML to help display objects generated by commands inside the module. The ServerManager module, for example, has format XML that enables the pretty output created with Get-WindowsFeature. This module is not directly supported by PowerShell 7. You can access the commands in the module thanks to the Windows PowerShell compatibility mechanism described earlier in this chapter. One restriction, by default, of the compatibility mechanism is that it does not import format XML (if it exists). The net result is that output can look different when you run scripts or issue console commands in PowerShell 7 versus Windows PowerShell.

The simple way around this is to import the format XML manually, as you see in this recipe.

## Getting ready

You run this recipe on SRV1, a workgroup server running Windows Server Datacenter edition and on which you have installed PowerShell 7 and VS Code.

## How to do it...

1. Importing the ServerManager module:

Import-Module -Name ServerManager

1. Checking a Windows feature:

Get-WindowsFeature -Name Simple-TCPIP

1. Running this command in the compatibility session:

$S = Get-PSSession -Name 'WinPSCompatSession'

Invoke-Command -Session $S -ScriptBlock {

    Get-WindowsFeature -Name Simple-TCPIP }

1. Running this command with formatting in the remote session:

Invoke-Command -Session $S -ScriptBlock {

                    Get-WindowsFeature -Name Simple-TCPIP |

                      Format-Table}

1. Getting path to Windows PowerShell modules:

$Paths = $env:PSModulePath -split ';'

foreach ($Path in $Paths) {

  if ($Path -match 'system32') {$S32Path = $Path; break}

}

"System32 path: [$S32Path]"

1. Displaying path to the format XML for the ServerManager module:

$FXML = "$S32path/ServerManager"

$FF = Get-ChildItem -Path $FXML\\*.format.ps1xml

"Format XML file: [$FF]"

1. Updating the format XML:

Foreach ($F in $FF) {

  Update-FormatData -PrependPath $F.FullName}

1. View the Windows Simple-TCPIP feature:

Get-WindowsFeature -Name Simple-TCPIP

1. Adding Simple-TCPIP Services:

Add-WindowsFeature -Name Simple-TCPIP

1. Examining the Simple-TCPIP feature:

Get-WindowsFeature -Name Simple-TCPIP

1. Using the Simple-TCPIP Windows feature:

Install-WindowsFeature Telnet-Client |

  Out-Null

Start-Service -Name simptcp

1. Using the Quote of the Day service:

Telnet SRV1 qotd

## How it works...

In step 1, inside a PowerShell 7.1 session, you import the ServerManager module, with output that looks like this:

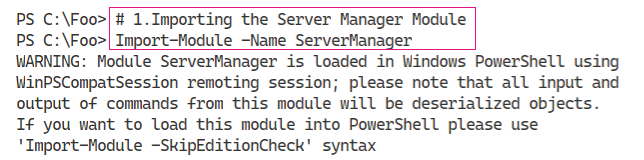


Figure 3.26: Importing the ServerManager module

Insert image B42024\_03\_26.png

Next, in step 2, you examine a Windows feature, the Simple-TCPIP services feature, using the Get‑WindowsFeature command, which produces output that looks like this:

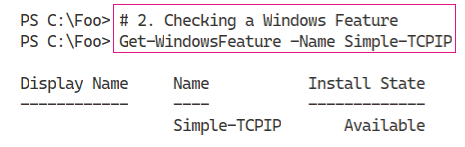


Figure 3.27: Examining the Simple-TCPIP feature

Insert image B42024\_03\_27.png

Note that, in the output, the Display Name column contains no information. PowerShell uses the format XML to populate the contents of this Display Name column. Since the compatibility mechanism does not import the format XML into the PowerShell session, you see a sub-optimal output.

In step 3, you use the Windows PowerShell compatibility remoting session to run the Get‑WindowsFeature cmdlet in the remoting session. The output of this step looks like this:

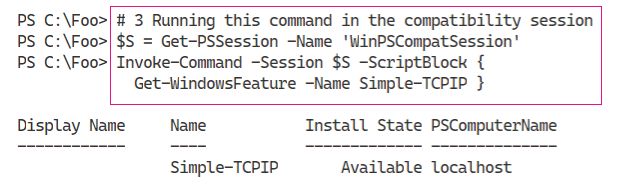


Figure 3.28: Running Get-WindowsFeature in the compatibility session

In both steps 2 and 3, the Get-WindowsFeature cmdlet produces an object which the PowerShell 7 session formats without the benefit of format XML.

Insert image B42024\_03\_28.png

In step 4, you run the Get-WindowsFeature command to retrieve the feature details then perform formatting in the remote session, with output like this:

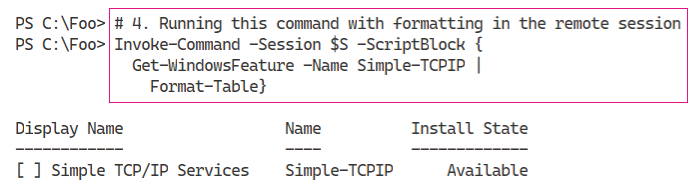


Figure 3.29: Running Get-WindowsFeature in the remote session

You see the nicely formatted Display Name filled, since Windows PowerShell performs the formatting of the output from Get-WindowsFeature cmdlet fully within the compatibility session, where the format XML exists.

Insert image B42024\_03\_29.png

In step 5, you parse the PSModulePath Windows environment variable to discover the default path to the Windows PowerShell modules. This path holds the modules added by core Windows services and includes the ServerManager module. The output of these commands looks like this:

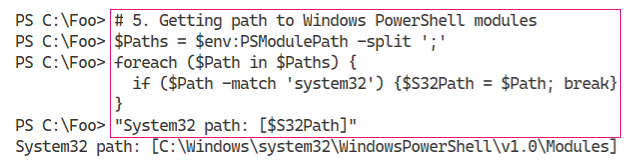


Figure 3.30: Getting the path to Windows PowerShell modules

Insert image B42024\_03\_30.png

In step 6, you discover the filename for the format XML for the ServerManager module, which produces output like this:

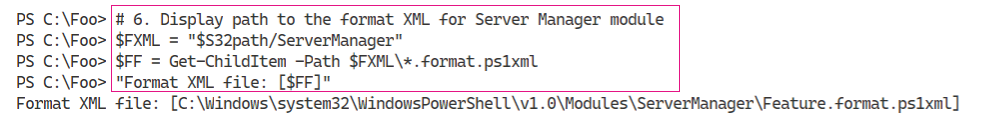


Figure 3.31: Retrieving the path to the format XML for ServerManager

Most, but not all, Windows modules have a single format XML file.

Insert image B42024\_03\_31.png

In step 7, which produces no output, you update the format data for the current Windows PowerShell session. If a module has multiple format XML files, this approach ensures they are all imported.

In step 8, you view a Windows feature, the Simple TCP/IP service, which has the feature name Simple-TCPIP. The output from this command looks like this:

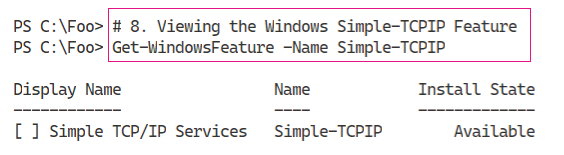


Figure 3.32: Viewing Simple-TCPIP

Insert image B42024\_03\_32.png

In step 9, you add the Simple-TCPIP Windows feature to SRV1, which produces output like this:

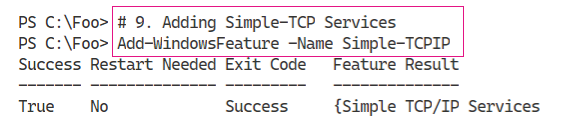


Figure 3.33: Adding Simple-TCPIP to SRV1

Insert image B42024\_03\_33.png

In the next step, step 10, you view the Simple-TCPIP Windows feature, which now looks like this:

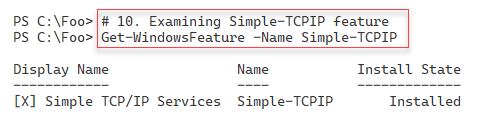


Figure 3.34: Viewing Simple-TCPIP

Insert image B42024\_03\_34.png

To test the Simple-TCPIP feature, you must install the Telnet client. Then you start the service. You perform both actions in *step 11*, which produces no output.

In step 12, you use the **Quote Of The Day** (**QOTD**) protocol. You can use QOTD for debugging connectivity issues, as well as measuring network performance. RFC 8965 defines the QOTD protocol; you can view the protocol definition at https://tools.ietf.org/html/rfc865 and read more about using Quote of the Day at https://searchcio.techtarget.com/tip/Quote-of-the-Day-A-troubleshooting-networking-protocol.

## There’s more...

In step 1, you import the ServerManager module. This ensures that the Windows compatibility remoting session is set up. If you have just run some of the earlier recipes in this chapter in the same PowerShell session, this step is redundant as the compatibility remoting session has already been set up and the ServerManager module is already loaded.

In step 2, you see that the output from Get-WindowsFeature does not populate the Display Name column. Likewise, in step 3. In both of these steps, PowerShell performs basic default formatting. In step 4, you get the details of the Windows feature and perform the formatting all in the remote session. As you can see, in the remote session, formatting makes use of format XML to produce superior output. In this case, the format XML populated the Display Name field and added an indication of whether the feature is installed (or not).

In steps 5 and 6, you discover the Windows PowerShell default modules folder and find the name of the format XML for this module. Having discovered the filename for the format XML, in step 7, you import this format information. With the format XML imported, in step 8, you run the Get‑WindowsFeature cmdlet to view a Windows feature. Since you have imported the format XML, the result is the output that is the same as you saw in *step 4*.

This recipe uses the Simple-TCPIP feature to demonstrate how you can use a command in PowerShell 7.1 and get the same output you are used to from using Windows PowerShell. The simple TCIP/IP services provided by this function are very old-school protocols, such as Quote of the Day. In production, these services are of potentially no value, and not adding them is a best practice.

# Leveraging compatibility

In this chapter so far, you have looked at the issue of compatibility between Windows PowerShell and PowerShell 7. You have examined the new features in PowerShell 7 and have looked at the Windows PowerShell compatibility mechanism.

The compatibility mechanism allows you to use incompatible Windows PowerShell cmdlets inside a PowerShell session. Incompatible Windows PowerShell cmdlets/modules rely on features which, while present in the full .NET CLR, are not available in .NET Core 5.0 (and are unlikely ever to be added to .NET core). For example, the Get-WindowsFeature cmdlet uses a .NET type System.Diagnostics.Eventing.EventDescriptor, as you saw earlier. Although the cmdlet cannot run natively in PowerShell 7, the compatibility mechanism allows you to make use of the cmdlet’s functionality.

When Import-Module begins loading an incompatible module, it checks to see if a remoting session with the name WinPSCompatSession exists. If that remoting session exists, PowerShell makes use of it. If the session does not exist, Import-Module creates a new remoting session with that name. PowerShell 7 then imports the module in the remote session and creates proxy functions in the PowerShell 7 session.

Once Import-Module has created the remoting session, PowerShell uses that single session for all future use, so loading multiple modules utilizes a single remoting session.

## Getting ready

You run this recipe on SRV1, on which you have installed PowerShell 7 and VS Code. SRV1 is a workgroup server running Windows Server Data Center Edition.

## How to do it...

1. Creating a session using the reserved name:

$S1 = New-PSSession -Name WinPSCompatSession -ComputerName SRV1

1. Getting loaded modules in the remote session:

Invoke-Command -Session $S1 -ScriptBlock {Get-Module}

1. Loading the ServerManager module in the remote session:

Import-Module -Name ServerManager -WarningAction SilentlyContinue |

  Out-Null

1. Getting loaded modules in the remote session:

Invoke-Command -Session $S1 -ScriptBlock {Get-Module}

1. Using Get-WindowsFeature:

Get-WindowsFeature -Name PowerShell

1. Closing remoting sessions and removing module from current PS7 session:

Get-PSSession | Remove-PSSession

Get-Module -Name ServerManager | Remove-Module

1. Creating a default compatibility remoting session:

Import-Module -Name ServerManager -WarningAction SilentlyContinue

1. Getting the new remoting session:

$S2 = Get-PSSession -Name 'WinPSCompatSession'

$S2

1. Examining modules in WinPSCompatSession:

Invoke-Command -Session $S2 -ScriptBlock {Get-Module}

## How it works...

In step 1, you create a new remoting session using the session name WinPSCompatSession. This step produces no output, but it does create a remoting session to a Windows PowerShell endpoint. PowerShell 7 holds the name of the endpoint which New-PSSession uses, unless you use the parameter -ConfigurationName to specify an alternate configuration name when you create the new remoting session.

In step 2, you run the Get-Module cmdlet to return the modules currently loaded in the remoting session you established in step 1. PowerShell generates no output for this step, indicating that there are no modules so far imported into the remoting session.

With step 3, you import the ServerManager module. Since this module is not compatible with PowerShell 7, PowerShell uses the Window PowerShell compatibility session you created earlier. Import-Module only checks to see if there is an existing remoting session with the name WinPSCompatSession. This step generates no output.

In step 4, you recheck the modules loaded into the remote session. As you can see in the output, this command discovers two modules loaded in the remoting session, one of which is the Windows PowerShell ServerManager module. The output from this step looks like this:

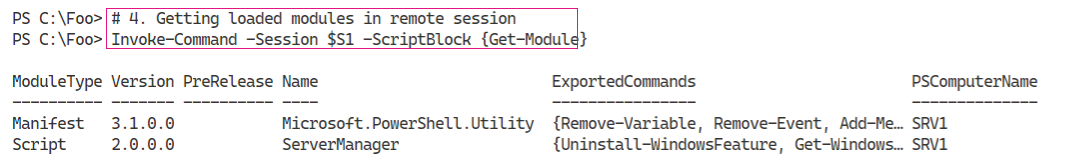


Figure 3.35: Getting loaded modules in the remote session

Insert image B42024\_03\_35.png

In step 5, you invoke Get-WindowsFeature to discover the PowerShell feature. As you can see in the output, this feature is both available and installed in SRV1:

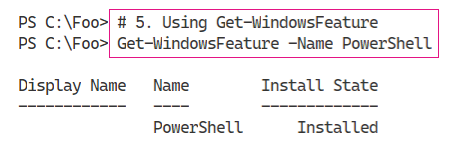


Figure 3.36: Invoking Get-WindowsFeature

Insert image B42024\_03\_36.png

In step 6, you close the remoting session that you created earlier in this recipe and remove all loaded modules. This step generates no output. In step 7, you import the (non-PowerShell 7 compatible) ServerManager module. As you have seen previously, this command creates a compatibility session, although there is no output from this step.

In step 8, you get and then display the remoting session, which produces output like this:

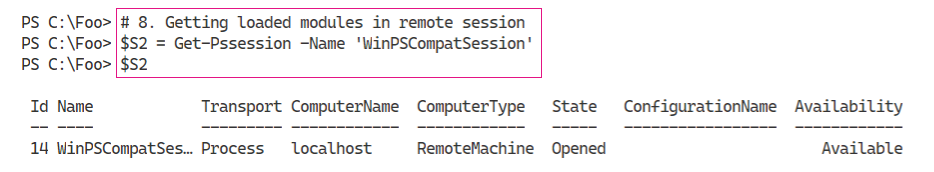


Figure 3.37: Getting loaded modules in the remote session

Insert image B42024\_03\_37.png

In step 9, you check to see the modules loaded in the Windows compatibility session (which you create in *step 7* by importing a module). The output from this step looks like this:

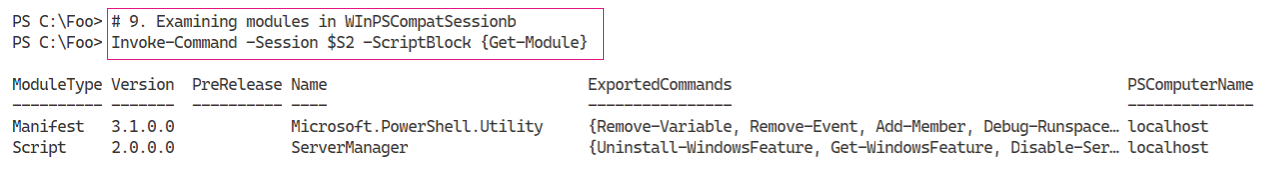


Figure 3.38: Checking modules loaded in the Windows compatibility session

Insert image B42024\_03\_38.png

## There’s more...

In this recipe, you create two PowerShell remoting sessions. In step 1, you create explicitly by using New-PSSession. Later, in step 7, you call Import-Module to import an incompatible module. Import-Module then creates a remoting session, since one does not exist.

As you can see in this recipe, so long as there is a remoting session with the name WinPSCompatSession in PowerShell 7, Import-Module uses that to attempt to load, in this case, the ServerManager module.

One potential use for the approach shown in this recipe might be where you wish to create a constrained endpoint for use with delegated administration that relies on older Windows PowerShell cmdlets. You could create the endpoint configuration details and then, in the scripts, create a remoting session using the customized endpoint and name the session WinPSCompatSession.