CHAPTER 1

Setting Up a PowerShell 7 Environment

The first versions of Windows PowerShell were provided via a user-installed download, initially for Windows XP and Windows Server 2008. Today, both Windows Server and Windows 10 come with Windows PowerShell version 5.1—which in this book I'll call simply Windows PowerShell to distinguish it from PowerShell 7 (and the Windows PowerShell Integrated Scripting Environment) installed and available by default. Windows PowerShell comes with a range of commands available for basic administration of Windows.

PowerShell 7 itself does not ship as part of Windows at the time of writing. At some point, the PowerShell team may ship PowerShell 7 as a Windows component, but until that time, you need to download and install it yourself.

The Windows PowerShell Integrated Scripting Environment (ISE) does not support PowerShell 7. IT pros who want a good interactive development environment for PowerShell can use Visual Studio Code (VS Code), a free tool you can also easily download and install. VS Code comes with an array of extensions that provide a much-improved development experience for IT pros (and others).

With earlier versions of PowerShell, the vast majority of commands came bundled into Windows or were added as part of installing an application (such as Exchange Server) or adding a Windows feature to your system. With PowerShell 7, the PowerShell Gallery has become a core source of modules/commands that you can use to perform various administrative tasks. To ensure that you can take advantage of the PowerShell Gallery, you need to be sure that the PowerShellGet module is up to date.

What Is New in PowerShell 7

PowerShell 7 is the latest version of PowerShell. The PowerShell development team released PowerShell 7.0 in March 2020. By the time you read this, the development team is certain to have released newer minor updates. PowerShell 7 has a number of key new features that IT pros can leverage.

If you are familiar with and can use Windows PowerShell to manage your Windows systems, almost all your knowledge is directly transferable to the new environment. Need to get help on a command? Just type `Get-Help` at the PowerShell command line. The basic architecture of PowerShell remains the same, with many internal changes, significant improvements, and a few breaking issues.

From the perspective of an IT professional with a working knowledge of managing Windows using Windows PowerShell, here are the key changes you can find:

- **Redeveloped cmdlets, based on .NET Core and open sourced via GitHub:** You can now read and even help to extend any cmdlet in PowerShell 7. This also means that the cmdlets were written to use .NET Core—which has created a few small compatibility issues.
- **A robust compatibility layer:** You use this to access Windows PowerShell modules that do not directly work on PowerShell 7. This means that all but a small number of Windows PowerShell 5.1 modules are available with and work under PowerShell 7. Chapter 2, "PowerShell 7 Compatibility with Windows PowerShell," describes this compatibility layer in more detail and notes how it works and its limitations as well as providing work-around solutions.
- **Significant performance enhancements:** In porting the Windows PowerShell modules to PowerShell 7, the development team was able to review the code and deliver performance enhancements. Processing large collections, for example using `Foreach`, is now a lot faster. The `Foreach-Object` cmdlet now has a `-Parallel` switch that allows you to run script blocks in parallel, which can provide substantially shorter run times, especially on larger multiprocessor and multicore servers.
- **New PowerShell language operators:** There are three new operator sets in PowerShell: the Ternary operator (`a ? b : c`), the Pipeline chain operators (`||` and `&&`), and the Null coalescing operators (`??` and `??=`). These operators were implemented in other shells such as Bash or Zsh, and you can now use them in PowerShell 7.
- **Simplified error views:** Windows PowerShell error messages were excellent and contained a lot of information. But in most cases the rich output was more than you normally need. Error messages in PowerShell 7 are now much more succinct. And when you *do* need that additional information,

you can use `Get-Error` to retrieve the full details of any error. You can set the `$ErrorView` variable to `NormalView` to view the older Windows PowerShell–style error messages or `CategoryView` to see just the error category.

- **Experimental features:** The PowerShell team has implemented a raft of new features that are at an experimental stage. You can opt in (or not) to these features. This gives you the opportunity to try new things and provide feedback.
- **Automatic new version notification:** At the time of writing, there is no support for PowerShell 7 within the Windows Store or via Windows Update. That means you need to manage the updates yourself, and these messages provide timely notification that a newer version of PowerShell exists for you to download.
- **`Set-Location` now supports a path of - and +:** When you use `Set-Location` to reset your current working directory, you can use `-Path "-"` to instruct `Set-Location` to move to the last folder. Having moved back, you can set location using + to move forward.
- **Ability to invoke a DSC resource directly:** PowerShell 7 does not support desired state configuration, so no pull/report servers, local configuration manager, and so on. You can, however, manually invoke DSC resources on a given host, which provides a partial solution.

The PowerShell 7 snippets in this book use and demonstrate most of these new features. For more information on any of these features, including use cases and examples, use your favorite search engine as the PowerShell community has produced a significant amount of content that describes the features. You can find numerous higher-level posts, such as the article at `https://www.thomasmaurer.ch/2020/03/whats-new-in-powershell-7-check-it-out`. There are also more detailed articles that cover specific new features such as `tfl09.blogspot.com/2020/03/introduction-and-background-welcome-to.html`, for example, which provides details on the new Pipeline Chain and Ternary operators.

Systems Used in This Book and Chapter

This book examines how you can use PowerShell 7 to carry out a wide range of tasks, including setting permissions on a file share, collecting and reporting on performance data, and installing and configuring Active Directory. To demonstrate these and many other tasks, this book uses a set of hosts and two domains: Reskit.Org and Kapoho.Com. You have options as to how you provision these systems.

Server VM Build Scripts

The scripts in this book assume you have a set of servers ready to configure. You could, if you choose, build each computer used in this book based on physical hardware. A simpler alternative is to build the necessary server VMs using Hyper-V using the build scripts you can find at `github.com/doctordns/ReskitBuildScripts`. This GitHub repository contains a `README.MD` file (`github.com/doctordns/ReskitBuildScripts/blob/master/README.md`) that explains how you can use these scripts to build your VM farm.

By way of background, these scripts are used to create VMs for a variety of training courses and other books. You do not need to create *all* the VMs. In the introduction to each chapter, you discover the specific VMs that the chapter uses.

These build scripts build VMs, but you need to take some care in terms of the order in which you build the VMs, where you store VMs and virtual hard disks, and so on.

The build scripts build VMs with basic networking (one NIC) although you can always add more should you wish. The scripts build the VMs you need for this book using a specific set of network addresses. The document `github.com/doctordns/ReskitBuildScripts/blob/master/ReskitNetwork.md` shows the details of the network hosts and IP addresses.

VM Internet Access

The VMs (or hosts if you choose to use physical computers) require Internet access. The VMs are all on the 10.10.10.0/24 IPv4 network implemented as an internal Hyper-V network, using an internal Hyper-V virtual switch. There are two broad mechanisms you can use to provide this.

First, you can configure each VM to have a second virtual NIC. You configure this NIC to use an external switch that you bind to your VM host's external NIC. This is a simple solution and can be set up quickly.

Another alternative is to set up a Windows Server VM running Routing and Remote Access. You configure the VM with two NICs (one internal, the other external) and configure routing between the 10.10.10.0/24 subnet used by the VMs in this book and the internet.

Systems in Use for This Chapter

In this chapter, you use PowerShell 7 to manage various networking aspects. The scripts in this chapter make use of one.

DC1: For the purposes of this chapter, DC1 is just a Windows Server 2019 host.

Figure 1.1 shows the systems in use in this chapter.

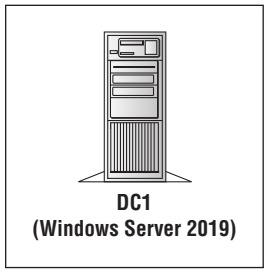

Figure 1.1: Systems used in this chapter

For the purposes of this chapter, `DC1` is a VM running Windows Server 2019 Datacenter. You can create this server using the build scripts noted earlier. In Chapter 3, "Managing Active Directory," you promote this server to be a domain controller.

Installing PowerShell 7

By default, Microsoft does not include PowerShell 7 within any version of Windows, including Window 10, Windows Server 2019, or any earlier supported versions of Windows client and server. To install and use PowerShell 7, you need to install it for your operating system.

The PowerShell team supports PowerShell 7, at the time of writing, on the following operating systems:

- Windows 7, 8.1, and 10
- Windows Server 2008 R2, 2012, 2012 R2, 2016, and 2019
- macOS 10.13+
- Red Hat Enterprise Linux (RHEL) / CentOS 7+
- Fedora 29+
- Debian 9+
- Ubuntu 16.04+
- openSUSE 15+
- Alpine Linux 3.8+

This list is constantly being reviewed and updated. The PowerShell product team will add distributions of Linux to the list and provide support for later versions of all platforms.

This book describes installing and using PowerShell 7 on the Windows platform. The chapters in this book leverage Windows Server features such as Active Directory, which have no counterpart on other platforms. Nevertheless, you can use PowerShell 7 on a non-Windows platform to manage Windows servers and clients.

In this book, you need to install PowerShell 7 on each host you use, since all the scripts shown in this book require PowerShell 7.

Before You Start

You run the code in this section on a Windows 2-++++-+--+-019 Server, `DC1`. You can also use the snippets on other hosts that you use to install VS Code on those hosts.

You begin by running the code snippets using the Windows PowerShell 5.1 (or the ISE), run in an elevated console. You can download the relevant script files, open them locally, and run them inside each VM in the ISE (initially). Once you have installed PowerShell 7, you use the PowerShell 7 console. (You use VS Code later in this chapter.)

Enabling the Execution Policy

By default, PowerShell does not allow you to run scripts. To do so, you must first set the execution policy.

```
# 1. Enable scripts to be run
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force
```

Here you set the execution policy to Unrestricted. On production systems you may want to set a more restrictive execution policy.

Installing the Latest Version of NuGet and PowerShellGet

PowerShell 7 comes with a large number of modules containing a useful set of commands, but it does not provide commands and modules to cover every situation. The PowerShell community has stepped up and created some outstanding added modules you can download and use, some of which are discussed in this book.

You use the PowerShell Gallery to download the PowerShell modules used in this book. For example, you use the NTFS Security module in later chapters to manage NTFS file and folder security.

You download and install a module from the PowerShell Gallery, or other module repositories, using `Import-Module`. To work with the PowerShell Gallery, you need to ensure you have the latest versions of the underlying tools that `Install-Module` uses to work against the PowerShell Gallery.

To ensure that you download the most up-to-date versions of the key modules from the PowerShell Gallery, use the following commands:

```
# 2. Install latest versions of Nuget and PowerShellGet
Install-PackageProvider Nuget -MinimumVersion 2.8.5.201 -Force |
```

```
  Out-Null
Install-Module -Name PowerShellGet -Force -AllowClobber
```

You use the commands in the PowerShellGet module to download content from the PowerShell Gallery. That, in turn, requires at least version 2.8.5.201 of NuGet.

For more details on the PowerShell Gallery, see `docs.microsoft.com/powershell/scripting/gallery/overview`. To view the commands in the PowerShellGet module, see `docs.microsoft.com/powershell/module/powershellget/`.

Creating the Foo Folder

Throughout this book, sample code snippets use the `C:\Foo` folder to hold various files that you use in the code snippets. You create it using the `New-Item` command.

```
# 3. Create local folder C:\Foo
$LFHT = @{
  ItemType    = 'Directory'
  ErrorAction = 'SilentlyContinue' # should it already exist
}
New-Item -Path C:\Foo @LFHT | Out-Null
```

These commands use a PowerShell technique known as *splatting*. First, you create a hash table containing parameter names and values. Then, you pass the hash table to a cmdlet, in this case `New-Item`. You pass the hash table instead of the parameters and their values. This book makes extensive use of this feature for a few reasons. First, it enables all the commands to fit within the width of the page, avoiding a single command that spans multiple lines. But for production code, this approach also makes it a bit easier to see and, as needed, to update scripts that contain commands with a larger number of parameters. For more information, you can type **`Get-Help about_splatting`** inside PowerShell or view the help file online at `docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_splatting`. Note that you need to run `Update-Help` within PowerShell 7 before you can view the splatting help file.

Downloading the PowerShell 7 Installation Script

There are a variety of ways you can install PowerShell 7. One simple way is to download and use an installation script created by the PowerShell team. The script is available via the Internet (from GitHub), and you download it as follows:

```
# 4. Download PowerShell 7 installation script
Set-Location C:\Foo
$URI = "https://aka.ms/install-powershell.ps1"
Invoke-RestMethod -Uri $URI |
  Out-File -FilePath C:\Foo\Install-PowerShell.ps1
```

These commands download the installation script from GitHub and store it in `C:\Foo`.

Viewing Installation File Help Information

It is always a good idea to examine any script you download to see how the developer intended it to be used. To that end, the installation script contains some basic help information, which you can view with `Get-Help`.

```
# 5. View Installation Script Help
Get-Help -Name C:\Foo\Install-PowerShell.ps1
```

You can see the output from this command in Figure 1.2.

```
PS C:\Foo> # 5. View Installation Script Help
Get-Help -Name C:\Foo\Install-PowerShell.ps1
Install-PowerShell.ps1 [-Destination <string>] [-Daily] [-DoNotOverwrite] [-AddToPath] [-Preview] [<CommonParameters>]
Install-PowerShell.ps1 [-UseMSI] [-Quiet] [-AddExplorerContextMenu] [-EnablePSRemoting] [-Preview] [<CommonParameters>]
```

Figure 1.2: Viewing help information

The installation script allows you to install PowerShell 7 by using an MSI file or by installing it by downloading a ZIP file and expanding it into a folder of your choice. You can install the current version of PowerShell 7, the latest Preview of the next version of PowerShell 7, or you can install PowerShell's daily build. You can install each version alongside the latest fully released version of PowerShell 7.

For most IT pros, using the MSI and installing silently is quick and easy. Evaluating future versions and bug fixes rolled into the daily build is for the brave. Obviously, you should proceed with all necessary caution when using an unsupported version of any product. With that being said, the advanced versions have been very stable and allow you early access to new functionality and bug fixes.

Installing PowerShell 7

To install PowerShell 7 on `DC1`, you run the installation script with this code snippet:

```
# 6. Install PowerShell 7
$EXTHT = @{
  UseMSI                = $true
  Quiet                 = $true
  AddExplorerContextMenu = $true
  EnablePSRemoting      = $true
}
C:\Foo\Install-PowerShell.ps1 @EXTHT
```

The installation script that downloads the PowerShell 7 MSI Installation package then runs this code silently. You should see no meaningful output. Because you are using the MSI, it installed PowerShell into a well-known location and updated the registry to indicate which versions of PowerShell are installed on this system.

Without using the MSI, for example when installing the build of the day, the installation script downloads the appropriate version as a ZIP file and expands it into a folder you specify.

When the `Install-PowerShell` script completes, you have installed PowerShell 7 on your system.

Examining the Installation Folder

Now that you have installed PowerShell 7, you can take a look at the installation folder with the following syntax:

```
# 7. Examine the installation folder
Get-Childitem -Path $env:ProgramFiles\PowerShell\7 -Recurse |
  Measure-Object -Property Length -Sum
```

You can see the output from these commands in Figure 1.3.

```
PS C:\Foo> # 7. Examine the installation folder
PS C:\Foo> Get-Childitem -Path $env:ProgramFiles\PowerShell\7 -Recurse |
             Measure-Object -Property Length -Sum

Count    : 986
Average  :
Sum      : 234332610
Maximum  :
Minimum  :
Property : Length
```

Figure 1.3: Examining the installation folder

As you can see in the figure, PowerShell 7's installation folder is different from that of Windows PowerShell (where, as a component of Windows, it is within the `C:\Windows\System32\WindowsPowerShell` folder).

If you examine the files in that folder carefully, you can see some differences in PowerShell 7. With PowerShell 7, there are significantly more files in the `$PSHome` folder (which can be confusing), and the name of the executable program you run to bring up the PowerShell 7 console is `pwsh.exe`. Another noticeable difference is that with PowerShell 7, there are no `.PS1XML` files. In Windows PowerShell, these XML files defined the default formatting for a wide range of objects and provided for IT pro–focused extensions to .NET Framework objects. PowerShell 7 brings the formatting XML inside PowerShell itself for improved performance.

The type extensions files were highly useful for ensuring consistency of property names across .NET classes, and PowerShell now has this functionality as well. You can still create your own type or format XML and extend PowerShell with updated types or default formatting that better meets your needs.

Viewing Module Folder Locations

In Windows PowerShell, commands you use are contained in modules. You can implicitly import a module before use to ensure it's available. Or you can make use of Windows PowerShell's module autoload feature. With module autoload, if you use a command that is not contained in modules already loaded, PowerShell searches all available modules to see if that command exists in some other module. If so, PowerShell loads the module and executes the command. PowerShell uses a built-in Windows environment variable to hold a semicolon-delimited list of paths. PowerShell searches each path in turn to discover any needed module. You can view the set of module folders in Windows PowerShell with this code:

```
# 8. View Module folders
#  View module folders for autoload
$I = 0
$env:PSModulePath -split ';' |
  Foreach-Object {
    "[{0:N0}]   {1}" -f $I++, $_}
```

You can see the output from these commands in Figure 1.4.

```
PS C:\Foo> # 8. View Module folders
PS C:\Foo> #  View module folders for autoload
PS C:\Foo> $I = 0
PS C:\Foo> $env:PSModulePath -split ';' |
             Foreach-Object {
               "[{0:N0}]   {1}" -f $I++, $_}

[0]   C:\Users\tfl\Documents\WindowsPowerShell\Modules
[1]   C:\Program Files\WindowsPowerShell\Modules
[2]   C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

Figure 1.4: Viewing module paths

As you can see, with Windows PowerShell there are just three module paths by default. If you add features or other applications/tools to your system(s), you may find installation programs add additional paths to the module file path variable.

To optimize performance, each time Windows PowerShell starts up, it spawns a low-priority thread that looks at the modules available and stores the details in a local cache. The module autoload makes use of this cache to discover the

modules that need to be imported before a command can be used. One side effect is that if different modules implement a given command name, the module in the higher-placed path is imported. This means you can create your own versions of `Get-Command`.

Viewing Profile File Locations

PowerShell defines four profile files.

- AllUsersAllHosts
- AllUsersCurrentHost
- CurrentUserAllHosts
- CurrentUserCurrentHost

Each profile file is associated with a well-known profile filename. PowerShell has a built-in variable, `$Profile`. At startup, PowerShell adds four properties to this variable that hold the well-known paths to each of the profile files. To see these properties, you need to use the `-Force` parameter explicitly.

These profile files allow you considerable flexibility with respect to startup profiles. Each profile file (which PowerShell runs as part of its startup) can create objects/variables, set environment options, send email, create a transcript, create PowerShell drives, and so on. In effect, profile files are a way of persisting a customized environment. You add commands to the profile that you want to have executed before you begin to work in a PowerShell session.

You can use PowerShell profile files for all users, perhaps to define some corporate or departmental aliases, create some "well-known" file locations for all users, or do more customized actions for just the current user. You have profiles for all PowerShell hosts (programs that host and use the PowerShell runtime) or separate profile files for different hosts. The ISE, for example, has the `$PSISE` variable, which allows you to control the ISE environment. This variable does not exist in either the PowerShell console or VS Code. Having different profiles for different PowerShell hosts allows you to customize each host using different techniques.

You can address a variety of deployment scenarios using different combinations of these four profile files as needed. Most IT pros just use the Current User Current Host profile, which is the value of `$Profile`.

You view the four profiles and their associated well-known filenames as follows:

```
# 9. View Profile File locations
# Inside the ISE
$PROFILE |
  Format-List -Property *Host* -Force
# from Windows PowerShell Console
powershell -Command '$Profile|   Format-List -Property *Host*' -Force
```

The output, which you can see in Figure 1.5, shows the profile file locations for both the ISE and the Windows PowerShell console.

```
PS C:\Foo> # 9. View Profile File locations
PS C:\Foo> # Inside the ISE
PS C:\Foo> $PROFILE |
             Format-List -Property *Host* -Force

AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    : C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1
CurrentUserAllHosts    : C:\Users\tfl\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\tfl\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1

PS C:\Foo> powershell -Command '$PROFILE |
             Format-List -Property *Host*' -Force

AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    : C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\tfl\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\tfl\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
```

Figure 1.5: Viewing profile file locations

Starting PowerShell 7

Now that you have installed PowerShell 7, you can click the Windows Start button, enter **pwsh**, and hit Enter to open a PowerShell 7 console. After you open the PowerShell 7 console, verify the version by viewing the `$PSVersionTable` variable.

```
# 10. Run PowerShell 7 console and then...
$PSVersionTable
```

As you can see in Figure 1.6, I was running PowerShell 7 when I captured the output. There are almost certainly going to be newer versions released by the time you read this (such as 7.0.1 or 7.0.2), so you may well see a slightly later version for PowerShell 7.

```
PS C:\Users\tfl> # 10. Run PowerShell 7 console and then...
PS C:\Users\tfl> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.0.0
PSEdition                      Core
GitCommitId                    7.0.0
OS                             Microsoft Windows 10.0.18363
Platform                       Win32NT
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

Figure 1.6: Viewing help information

Viewing New Locations for Module Folders

In the previous section "Viewing Module Folder Locations" you viewed the folders where Windows PowerShell looked to find modules. With PowerShell 7, you repeat this as follows:

```
# 11. View Modules folders
$ModFolders = $Env:PSModulePath -split ';'
$I = 0
$ModFolders |
  ForEach-Object {"[{0:N0}]   {1}" -f $I++, $_}
```

You can see the output from these commands in Figure 1.7.

```
PS C:\Foo> # 11. View Modules folders
PS C:\Foo> $ModFolders = $Env:PSModulePath -split ';'
PS C:\Foo> $I = 0
PS C:\Foo> $ModFolders |
             ForEach-Object {"[{0:N0}]   {1}" -f $I++, $_}

[0]   C:\Users\tfl\Documents\PowerShell\Modules
[1]   C:\Program Files\PowerShell\Modules
[2]   c:\program files\powershell\7\Modules
[3]   C:\Program Files\WindowsPowerShell\Modules
[4]   C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

Figure 1.7: Viewing PowerShell 7 module paths

In the output, notice that you now have five module file paths.

Viewing New Locations for Profile Files

As in Windows PowerShell, you have multiple profile files with PowerShell 7, although these are now in a slightly different place on the disk. You can view the locations for the four PowerShell 7 profile files like this:

```
# 12. View Profile Locations
$PROFILE | Format-List -Property *Host* -Force
```

As you can see in Figure 1.8, there are four profiles files, each at a location separate from Windows PowerShell.

```
PS C:\Foo> # 12. View Profile Locations
PS C:\Foo> $PROFILE | Format-List -Property *Host* -Force

AllUsersAllHosts       : C:\Program Files\PowerShell\7\profile.ps1
AllUsersCurrentHost    : C:\Program Files\PowerShell\7\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\tfl\Documents\PowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\tfl\Documents\PowerShell\Microsoft.PowerShell_profile.ps1
```

Figure 1.8: Viewing PowerShell profile file locations

Creating a Current User/Current Host Profile

To demonstrate how PowerShell profile files work with PowerShell 7, you can download a sample profile from the Internet and create the Current User Current Host Profile using the following commands:

```
# 13. Create Current user/Current host profile
$URI = 'https://raw.githubusercontent.com/doctordns/Wiley20/master/' +
       'Goodies/Microsoft.PowerShell_Profile.ps1'
$ProfileFile = $Profile.CurrentUserCurrentHost
New-Item $ProfileFile -Force -WarningAction SilentlyContinue |
   Out-Null
(Invoke-WebRequest -Uri $uri -UseBasicParsing).Content |
  Out-File -FilePath  $ProfileFile
```

You can obtain the scripts for this book both from Wiley and from the author's GitHub repository. The author's repository contains a Goodies folder that has a sample profile file to illustrate the use of profiles with the PowerShell 7 console.

This sample sets some useful defaults and aliases, creates some variables, configures the console heading, and sets the current working folder to the eponymous `C:\Foo` folder. Feel free to adjust this profile to suit your working style. Once you run these commands, you need to restart PowerShell to have the new profiles take effect.

Installing and Configuring VS Code

The PowerShell ISE is an interactive development environment for Windows PowerShell. The ISE allows you to edit, manage, and run scripts within a single program. While there have been other IDEs available for PowerShell, the ISE is free and built in and has good functionality for the IT pro. The screenshots taken for this book that show PowerShell 7 code are shown running in VS Code.

Microsoft has indicated it has no plans to update the ISE to support PowerShell 7. A recommended alternative is VS Code. VS Code is a free, Microsoft-created, open source, cross-platform source code editor. VS Code runs as a desktop application on Windows, macOS, and Linux and provides great support for PowerShell. VS Code is an excellent tool for managing not only PowerShell source code but also documents containing Perl, Python, Markdown, and a host of other formats.

Although VS Code is cross-platform, this chapter deals with VS Code on Windows. For more information on VS Code, see `code.visualstudio.com/`.

VS Code supports a rich set of extensions that further enhance the development experience. You can install a spell checker, for example. If you work with Markdown (`en.wikipedia.org/wiki/Markdown`), the community has built VS Code

extensions to help you. You can use VS Code itself to add new extensions to your environment, and you can install specific extensions when you install VS Code.

To install VS Code on your computer, you download and run an installation script from Microsoft's Visual Studio site that enables you to install VS Code and any required extensions in a single operation.

If you work with more than just PowerShell, visit the Visual Studio marketplace at `marketplace.visualstudio.com`, where you can find a large selection of extensions. Some extensions are free, while others are commercial (usually with a free trial).

Once you have installed VS Code, you can easily alter the font it uses. VS Code can use any of the fonts installed on your system, and you can change the font via the VS Code GUI or via a JSON user settings file.

Microsoft recently published a new font, Cascadia Code. This font looks good when you are using VS Code. All the screenshots in this book containing PowerShell code use this font.

Finally, if you plan to use VS Code and PowerShell 7, you might like to add shortcuts to your taskbar.

NOTE **The screenshots in this book were taken using VS. You can use VS Code in all your VMs to test the scripts contained in this book, although in some environments this may not be allowed or may not be considered best practice.**

Before You Start

The code in this section runs on a VM running Windows Server 2019 Datacenter, `DC1`. You can also use the snippets on other hosts that you use to install VS Code and test the scripts in this book.

You run the scripts in this section using PowerShell 7, which you installed in the previous section "Installing PowerShell 7."

Downloading the VS Code Installation Script

The VS Code team has created an installation script to enable you to install VS Code and uploaded this script to the PowerShell Gallery. You download the script as follows:

```
# 1. Download the VS Code Installation Script
$VSCPATH = 'C:\Foo'
Save-Script -Name Install-VSCode -Path $VSCPATH
Set-Location -Path $VSCPATH
```

This snippet retrieves the `Install-VSCode` script and saves it to the `C:\Foo` folder.

Installing VS Code and Extensions

You install VS Code using the `Install-VSCode.ps1` file you just downloaded. You can specify certain VS Code extensions that the installer should install along with VS Code itself, as follows:

```
# 2. Now run it and add in some popular VSCode Extensions
$Extensions =  "Streetsidesoftware.code-spell-checker",
               "yzhang.markdown-all-in-one"
$InstallHT = @{
  BuildEdition          = 'Stable-System'
  AdditionalExtensions = $Extensions
  LaunchWhenDone        = $true
}
.\Install-VSCode.ps1 @InstallHT
```

You can see the output generated by the VS Code installation script in Figure 1.9.

```
PS C:\Foo> # 2. Now run it and add in some popular VSCode Extensions
PS C:\Foo> $Extensions =  "Streetsidesoftware.code-spell-checker",
                          "yzhang.markdown-all-in-one"
PS C:\Foo> $InstallHT = @{
             BuildEdition         = 'Stable-System'
             AdditionalExtensions = $Extensions
             LaunchWhenDone       = $true
             }
PS C:\Foo> .\Install-VSCode.ps1 @InstallHT

Installing extension ms-vscode.PowerShell...
Installing extensions...
Installing extension 'ms-vscode.powershell' v2020.4.0...
Extension 'ms-vscode.powershell' v2020.4.0 was successfully installed.

Installing extension Streetsidesoftware.code-spell-checker...
Installing extensions...
Installing extension 'streetsidesoftware.code-spell-checker' v1.8.0...
Extension 'streetsidesoftware.code-spell-checker' v1.8.0 was successfully installed.

Installing extension yzhang.markdown-all-in-one...
Installing extensions...
Installing extension 'yzhang.markdown-all-in-one' v2.8.0...
Extension 'yzhang.markdown-all-in-one' v2.8.0 was successfully installed.

Installation complete, starting Visual Studio Code (64-bit)...
```

Figure 1.9: Installing VS Code on *DC1*

This snippet retrieves and installs the latest stable version of VS Code from the Internet. As part of the installation two extensions are added. The first is a good spellchecker, and the second is helpful if you are editing Markdown files.

Depending on what your workload consists of, there are other extensions you might find useful. The community is active in developing, extending, and maintaining a wide variety of extensions.

These commands also start VS Code, so as additional output you see a VS Code window. You can use this window to run code snippets in the rest of this chapter.

Creating a Sample Personal Profile File

PowerShell profile files are quite powerful as a way of persisting customizations to PowerShell sessions. You can create new PowerShell drives, create and populate custom variables, define useful functions, and much more. By default, PowerShell 7 ships without profile files. As with Windows PowerShell, you create an (empty) profile file by running the following commands in the VS Code window:

```
# 3. Create a Sample Profile File
$SAMPLE = 'https://raw.githubusercontent.com/doctordns/Wiley20/master/' +
          'Goodies/Microsoft.VSCode_profile.ps1'
(Invoke-WebRequest -Uri $Sample).Content |
  Out-File $Profile
```

This snippet downloads a sample VS Code profile and saves it as the Current Host/Current User profile. Depending on your organization, this sample may be all you need. If you use both the PowerShell 7 console and VS Code, you need to maintain two separate versions of that profile—one for each host. You should review the two sample profiles and amend them as needed.

Downloading the Cascadia Code Font

Along with VS Code, Microsoft has also developed a new fixed-width font, named Cascadia Code, for use with VS Code (and any other Windows applications that use fixed-width fonts, including the PowerShell and Windows PowerShell console, Microsoft Office Word, and more). The Cascadia Code font's developers ship released versions of this font via GitHub (as well as via the Microsoft Store). There are several versions of the font, including the basic font (`Cascadia.ttf`). You can also download a variant that supports Powerline symbols (`CascadiaPL.ttf`). To download the latest version of the basic font, use the following code:

```
# 4. Download Cascadia Code font from GitHub
# Get File Locations
$CascadiaFont    = 'Cascadia.ttf'    # font file name
```

Continues

continued

```
$CascadiaRelURL  = 'https://github.com/microsoft/cascadia-code/releases'
$CascadiaRelease = Invoke-WebRequest -Uri $CascadiaRelURL # Get&all of them
$CascadiaPath    = "https://github.com" + ($CascadiaRelease.Links.href |
                       Where-Object { $_ -match "($CascadiaFont)" } |
                         Select-Object -First 1)
$CascadiaFile    = "C:\Foo\$CascadiaFont"
# Download Cascadia Code font file
Invoke-WebRequest -Uri $CascadiaPath -OutFile $CascadiaFile
```

Installing the Cascadia Code Font in Windows

There is no direct support in PowerShell to install fonts. To install the font you have just downloaded, you make use of the `Shell.Application` COM object, as shown here:

```
# 5. Install Cascadia Code font
$FontShellApp = New-Object -Com Shell.Application
$FontShellNamespace = $FontShellApp.Namespace(0x14)
$FontShellNamespace.CopyHere($CascadiaFile, 0x10)
```

Although you downloaded the font file previously, you only now install it. As long as you have not previously loaded this font, these commands produce no output.

Updating VS Code User Settings

VS Code is highly configurable. Many of the individual configuration preferences are stored in a JSON file that you can modify as you choose. Here is how you can update your user settings for VS Code:

```
# 6. Update Local User Settings for VS Code
#    This step in particular needs to be run in PowerShell 7!
$JSON = @'
{
  "workbench.colorTheme": "PowerShell ISE",
  "powershell.codeFormatting.useCorrectCasing": true,
  "files.autoSave": "onWindowChange",
  "files.defaultLanguage": "powershell",
  "editor.fontFamily": "'Cascadia Code',Consolas,'Courier New'",
  "workbench.editor.highlightModifiedTabs": true,
  "window.zoomLevel": 1
}
'@
$JHT = ConvertFrom-Json -InputObject $JSON -AsHashtable
$PWSH = "C:\\Program Files\\PowerShell\\7\\pwsh.exe"
```

```
$JHT += @{
  "terminal.integrated.shell.windows" = "$PWSH"
}
$Path = $Env:APPDATA
$CP   = '\Code\User\Settings.json'
$Settings = Join-Path  $Path -ChildPath $CP
$JHT |
  ConvertTo-Json  |
    Out-File -FilePath $Settings
```

This code snippet sets several VS Code settings for the current user, including using the newly added Cascadia Code font, setting the VS Code color theme to the PowerShell ISE, and more. Note that this snippet overwrites any user settings you may have. As you continue using VS Code and updating settings, the contents of the JSON file are likely to change.

The snippet also shows how you can manage JSON documents by using a new feature in PowerShell 7 that converts a JSON document to a PowerShell hash table. In the snippet, you import the JSON document and convert it to a PowerShell hash table. You can add a new value—in this case the name of the PowerShell 7 executable file. Once you have finished adding the user settings, you convert the hash table back to JSON and write it away.

Note that if you run this snippet in VS Code, you see the color theme and font change once these commands complete execution.

Creating a Shortcut to VS Code

You next create a shortcut to VS Code (which you use later in this section), as follows:

```
# 7. Create a short cut to VS Code
$SourceFileLocation  = "$env:ProgramFiles\Microsoft VS Code\Code.exe"
$ShortcutLocation    = "C:\foo\vscode.lnk"
# Create a  new wscript.shell object
$WScriptShell        = New-Object -ComObject WScript.Shell
$Shortcut            = $WScriptShell.CreateShortcut($ShortcutLocation)
$Shortcut.TargetPath = $SourceFileLocation
#Save the Shortcut to the TargetPath
$Shortcut.Save()
```

These steps create a shortcut in the `C:\Foo` folder to VS Code.

Creating a Shortcut to the PowerShell 7 Console

You can also create a shortcut to the PowerShell 7 console in a similar manner.

```
# 8. Create a shortcut to PowerShell 7
$SourceFileLocation  = "$env:ProgramFiles\PowerShell\7\pwsh.exe"
```

Continues

continued

```
$ShortcutLocation    = 'C:\Foo\pwsh.lnk'
# Create a  new wscript.shell object
$WScriptShell        = New-Object -ComObject WScript.Shell
$Shortcut            = $WScriptShell.CreateShortcut($ShortcutLocation)
$Shortcut.TargetPath = $SourceFileLocation
#Save the Shortcut to the TargetPath
$Shortcut.Save()
```

These steps create a shortcut in the `C:\Foo` folder to the PowerShell 7 console.

Building Layout.XML

If you are going to use VS Code and PowerShell a lot, it's useful to add a shortcut to these two tools in the Windows taskbar. To do that, you first create an XML file that tells Windows to add the two recently created shortcuts to the taskbar, as follows:

```
# 9. Build Updated Layout XML
$XML = @'
<?xml version="1.0" encoding="utf-8"?>
<LayoutModificationTemplate
  xmlns="http://schemas.microsoft.com/Start/2014/LayoutModification"
  xmlns:defaultlayout=
    "http://schemas.microsoft.com/Start/2014/FullDefaultLayout"
  xmlns:start="http://schemas.microsoft.com/Start/2014/StartLayout"
  xmlns:taskbar="http://schemas.microsoft.com/Start/2014/TaskbarLayout"
  Version="1">
<CustomTaskbarLayoutCollection>
<defaultlayout:TaskbarLayout>
<taskbar:TaskbarPinList>
 <taskbar:DesktopApp DesktopApplicationLinkPath="C:\Foo\vscode.lnk"/>
 <taskbar:DesktopApp DesktopApplicationLinkPath="C:\Foo\pwsh.lnk"/>
</taskbar:TaskbarPinList>
</defaultlayout:TaskbarLayout>
</CustomTaskbarLayoutCollection>
</LayoutModificationTemplate>
'@
$XML | Out-File -FilePath C:\Foo\Layout.Xml
```

This snippet creates a new `Layout.XML` file and stores it in the `C:\Foo` folder.

Importing the New Layout.XML File

To add these shortcuts to the taskbar, you need to import the start layout XML file you just created, as follows:

```
# 10. Import the start layout XML file
#     You get an error if this is not run in an elevated session
Import-StartLayout -LayoutPath C:\Foo\Layout.Xml -MountPath C:\
```

As noted in the snippet, you must run this final command in an elevated console or run VS Code elevated. Elevated mode allows you to run commands with administrative privileges. You do this by clicking Start, typing **Code**, right-clicking the VS Code icon, and then clicking Run As Administrator.

To see the two new shortcuts on the taskbar, you sign out of Windows and then sign back in. After you sign in again, your desktop should resemble Figure 1.10.

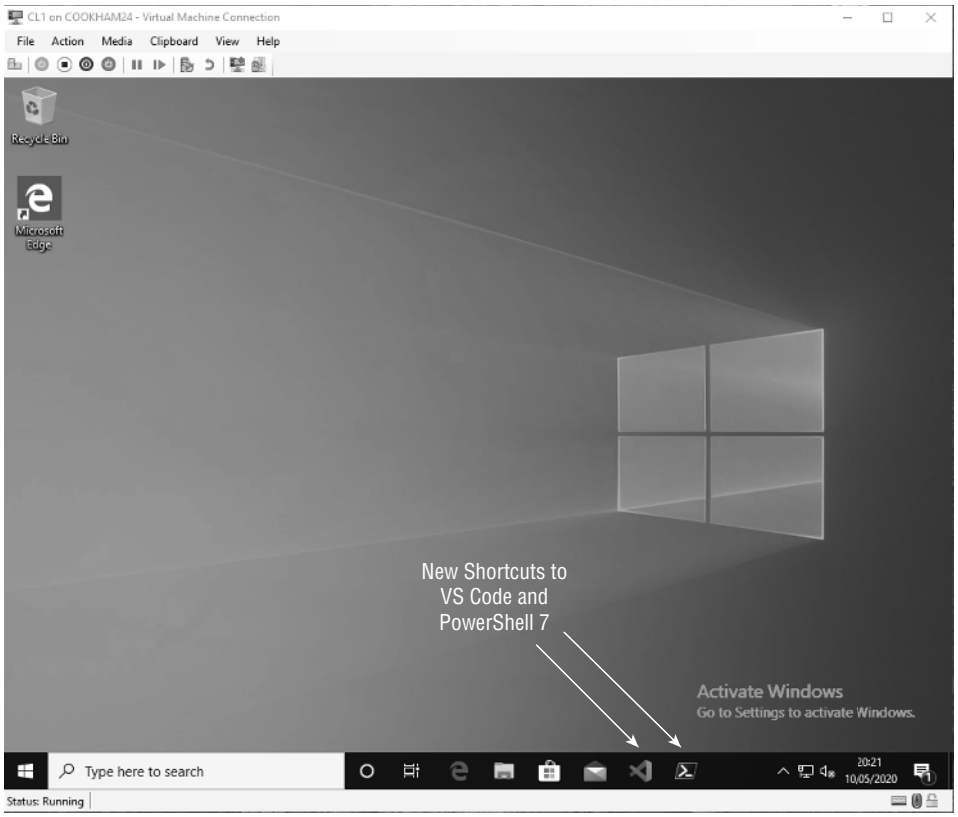

Figure 1.10: Updated taskbar

You may see a slightly different desktop depending on the media you used to install Windows 10 (for example, evaluation or fully licensed versions).

For the more adventurous, you might install both the daily build and the latest Preview build with taskbar shortcuts to all of the versions of PowerShell (and Windows PowerShell) on your computer.

Using the PowerShell Gallery

Almost since the beginning of the PC, IT pros have had to find and deploy additional software and tools. For PowerShell users, Microsoft's PowerShell

Gallery (`www.powershellgallery.com/`) is a repository of PowerShell add-ins including additional modules, scripts, and more. You used the PowerShell Gallery in "Installing and Configuring VS Code" to download the script to install VS Code, for example.

To discover, download, install, and update these add-ins, you use the commands in the PowerShellGet module. This module is installed by default. Given the pace of change in the PowerShell world and the importance of security, you need to ensure that you have the latest versions of this module (which is true of any module you download from the Gallery). You did that in the section "Installing PowerShell 7."

The PowerShell Gallery is run by Microsoft, and a number of the items in the PowerShell Gallery were created by Microsoft employees and Microsoft product teams. There are also a variety of excellent tools created by the community. This book shows how to use a number of these modules. The snippets in this section show how you can leverage and use the PowerShell Gallery.

Before You Start

You run the snippets in this section on a Windows Server host, `DC1`. You must have installed PowerShell 7, and optionally VS Code, on this host using the scripts earlier in this chapter.

Discovering PowerShell Gallery Modules

To discover modules available from the PowerShell Gallery, you can use the `Find-Module` command.

```
# 1. Get Details of all PS Gallery Modules
$PGSM = Find-Module -Name *
"There are {0:N0} Modules in the PS Gallery" -f $PGSM.count
```

This snippet downloads details of all modules available from the PowerShell Gallery and displays a count of how many are available, as you can see in Figure 1.11.

```
PS C:\Foo> # 1. Get Details of all PS Gallery Modules
PS C:\Foo> $PGSM = Find-Module -Name *
PS C:\Foo> "There are {0:N0} modules in the PS Gallery" -f $PGSM.Count
There are 5,526 modules in the PS Gallery
```

Figure 1.11: Count of modules available in the PowerShell Gallery

The number of available modules changes constantly but was more than 5,000 at the time of writing. While many of these modules are useful, some may

be less than helpful, incomplete, or not working. Care is needed when using third-party modules.

Determining the Modules That Support .NET Core

Some, but not all, of the modules support .NET Core and thus should work natively within PowerShell 7. To report on how many modules support .NET Core, you can run these commands:

```
# 2. Get Details of packages tagged with 'PSEdition_Core'
$PGSMC = Find-Module -Name * -Tag 'PSEdition_Core'
"There are {0:N0} modules supporting PowerShell Core" -f $PGSMC.Count
```

As you can see in Figure 1.12, there are about 800 modules that support .NET core, although this number, too, is increasing.

```
PS C:\Foo> # 2. Get Details of packages tagged with 'PSEdition_Core'
PS C:\Foo> $PGSMC = Find-Module -Name * -Tag 'PSEdition_Core'
PS C:\Foo> "There are {0:N0} modules supporting PowerSell Core" -f $PGSMC.Count
There are 798 modules supporting PowerShell Core
```

Figure 1.12: Count of available modules that support .NET core

Finding NTFS Modules

As an example of how to search for modules in the PowerShell Gallery, you might want to search for a module that works with the Windows NTFS filesystem. You can search for possible modules with this code:

```
# 3. Find NTFS Modules
$PGSM | Where-Object Name -match 'NTFS'
```

This command searches the set of modules you previously obtained and displays those that have "NTFS" in the module name. Figure 1.13 shows the output.

```
PS C:\Foo> # 3. Find NTFS Modules
PS C:\Foo> $PGSM | Where-Object Name -match 'NTFS'

Version  Name                   Repository   Description
-------  ----                   ----------   -----------
4.2.6    NTFSSecurity           PSGallery    Windows PowerShell Module for managing file and folder security …
1.4.1    cNtfsAccessControl     PSGallery    The cNtfsAccessControl module contains DSC resources for NTFS ac…
1.0      NTFSPermissionMigration PSGallery   This module is used as a wrapper to the popular icacls utility t…
```

Figure 1.13: NTFS-related modules

As you can see in the figure, three modules are available. One of those is the NTFSSecurity module, which you can use to manage NTFS access control lists

(ACLs) and inheritance (you use this module in several chapters in this book to adjust file ACLs).

Installing the NTFSSecurity Module

You install the NTFSSecurity module by using the `Install-Module` command.

```
# 4. Install the NTFSSecurity module
Install-Module -Name NTFSSecurity
```

Viewing Available Commands

Now that you have downloaded the module, you can use `Get-Command` to view the commands available within the NTFSSecurity module.

```
# 5. View Commands in the NTFSSecurity module
Get-Command -Module NTFSSecurity
```

You can see the output of this command in Figure 1.14.

Creating a Local PowerShellGet Repository

Public repositories such as the PowerShell Gallery are great resources for IT professionals. They provide you with a wide range of useful modules, scripts, and other resources. Modules such as NTFSSecurity make it easier to administer NTFS permissions and inheritance, for example. This book makes use of a number of modules you download from the PowerShell Gallery.

At the same time as you can leverage external modules, you can also develop modules for your own use and create your own internal module repository. You can create a module to manage users in your Active Directory, for example, that takes into account your specific business requirements. Once it is created, you can store this module on an internal repository for use by other IT professionals, or even end users, within your organization.

You have several alternative methods of creating an internal PowerShell repository. A simple implementation of a PowerShell repository would be an SMB share within your organization where you publish packages for others to use.

Before You Start

You run the code in this section on a Windows Server 2019 host, `DC1`. This is a Windows Server 2019 host on which you have installed PowerShell 7 and, optionally, VS Code.

```
PS C:\Foo> # 5. View Commands in the NTFS Security module
PS C:\Foo> Get-Command -Module NTFSSecurity

CommandType   Name                              Version  Source
-----------   ----                              -------  ------
Cmdlet        Add-NTFSAccess                    4.2.6    NTFSSecurity
Cmdlet        Add-NTFSAudit                     4.2.6    NTFSSecurity
Cmdlet        Clear-NTFSAccess                  4.2.6    NTFSSecurity
Cmdlet        Clear-NTFSAudit                   4.2.6    NTFSSecurity
Cmdlet        Copy-Item2                        4.2.6    NTFSSecurity
Cmdlet        Disable-NTFSAccessInheritance     4.2.6    NTFSSecurity
Cmdlet        Disable-NTFSAuditInheritance      4.2.6    NTFSSecurity
Cmdlet        Disable-Privileges                4.2.6    NTFSSecurity
Cmdlet        Enable-NTFSAccessInheritance      4.2.6    NTFSSecurity
Cmdlet        Enable-NTFSAuditInheritance       4.2.6    NTFSSecurity
Cmdlet        Enable-Privileges                 4.2.6    NTFSSecurity
Cmdlet        Get-ChildItem2                    4.2.6    NTFSSecurity
Cmdlet        Get-DiskSpace                     4.2.6    NTFSSecurity
Cmdlet        Get-FileHash2                     4.2.6    NTFSSecurity
Cmdlet        Get-Item2                         4.2.6    NTFSSecurity
Cmdlet        Get-NTFSAccess                    4.2.6    NTFSSecurity
Cmdlet        Get-NTFSAudit                     4.2.6    NTFSSecurity
Cmdlet        Get-NTFSEffectiveAccess           4.2.6    NTFSSecurity
Cmdlet        Get-NTFSHardLink                  4.2.6    NTFSSecurity
Cmdlet        Get-NTFSInheritance               4.2.6    NTFSSecurity
Cmdlet        Get-NTFSOrphanedAccess            4.2.6    NTFSSecurity
Cmdlet        Get-NTFSOrphanedAudit             4.2.6    NTFSSecurity
Cmdlet        Get-NTFSOwner                     4.2.6    NTFSSecurity
Cmdlet        Get-NTFSSecurityDescriptor        4.2.6    NTFSSecurity
Cmdlet        Get-NTFSSimpleAccess              4.2.6    NTFSSecurity
Cmdlet        Get-Privileges                    4.2.6    NTFSSecurity
Cmdlet        Move-Item2                        4.2.6    NTFSSecurity
Cmdlet        New-NTFSHardLink                  4.2.6    NTFSSecurity
Cmdlet        New-NTFSSymbolicLink              4.2.6    NTFSSecurity
Cmdlet        Remove-Item2                      4.2.6    NTFSSecurity
Cmdlet        Remove-NTFSAccess                 4.2.6    NTFSSecurity
Cmdlet        Remove-NTFSAudit                  4.2.6    NTFSSecurity
Cmdlet        Set-NTFSInheritance               4.2.6    NTFSSecurity
Cmdlet        Set-NTFSOwner                     4.2.6    NTFSSecurity
Cmdlet        Set-NTFSSecurityDescriptor        4.2.6    NTFSSecurity
Cmdlet        Test-Path2                        4.2.6    NTFSSecurity
```

Figure 1.14: Commands in the NTFSSecurity module

In Chapter 3, you convert the `DC1` host to be a domain controller. However, for the purposes of this section, `DC1` is just a Windows Server 2019 host.

Creating the Repository Folder

The repository you create in this section is based on a simple SMB file share. You start by creating the underlying folder.

```
# 1. Create Repository Folder
$LPATH = 'C:\RKRepo'
New-Item -Path $LPATH -ItemType Directory | Out-Null
```

In production, you would most likely protect this folder from being updated with ACLs. You can use the techniques covered in Chapter 4 , "Managing Networking," for this.

Sharing the Repository Folder

You create the repository share by using the `New-SMBShare` command.

```
# 2. Share the Repository Folder
$SMBHT = @{
  Name        = 'RKRepo'
  Path        = $LPATH
  Description = 'Reskit Repository'
  FullAccess  = 'Everyone'
}
New-SmbShare @SMBHT
```

These commands create a new SMB share named RKREPO on the `DC1` host. This share is the basis for the repository.

Creating a Module Working Folder

While you're developing any module, prior to publishing it and using it in production, you can store it in any folder. In this case, you create a new folder, `C:\HW`, that is to hold the module as you develop it.

```
# 3. Create a Working Folder for a Module
New-Item C:\HW -ItemType Directory | Out-Null
```

In production, you might want to create the working module in a separate volume. You would also want to put your module under source code control, for example using Git for this purpose.

Creating a Simple Module

The simplest module is a `.PSM1` file with just a function definition (and in this case an alias).

```
# 4. Create a simple module
$HS = @'
Function Get-HelloWorld {'Hello World'}
Set-Alias GHW Get-HelloWorld
'@
$HS | Out-File C:\HW\HW.psm1
```

These commands create a PowerShell module with one function and an alias. You save this "module" in your module working folder (`C:\HW`).

Loading and Testing the Module

Before you upload the module to a repository, you should test it. A simple way to do this is to load the module from the module's working folder and then use the `GHW` alias you added to the `.PSM1` file.

```
# 5. Load and Test the Module
Import-Module -Name C:\HW -Verbose
GHW
```

You can see the output from these two commands in Figure 1.15.

```
PS C:\Foo> # 5. Load and Test the Module
PS C:\Foo> Import-Module -Name C:\HW -Verbose
VERBOSE: Loading module from path 'C:\HW\HW.psm1'.
VERBOSE: Exporting function 'Get-HelloWorld'.
VERBOSE: Exporting alias 'GHW'.
VERBOSE: Importing function 'Get-HelloWorld'.
VERBOSE: Importing alias 'GHW'.
PS C:\Foo> GHW
Hello World
```

Figure 1.15: Testing the Hello World module

In the figure, you first see the verbose output from `Import-Module`. You can see in that output that PowerShell loads the module, adds a function to your PowerShell session, and then adds an alias to the function. Once imported, you can use the module's GHW alias.

Creating a Module Manifest

PowerShell repository and commands such as `Install-Module` are based on NuGet. Each item in your repository is a NuGet package. For more information on NuGet, see `docs.microsoft.com/en-us/nuget/what-is-nuget`.

NuGet requires all packages to have a module manifest (a `.PSD1` file that you add to the folder holding the module). You can create one using `New-ModuleManifest`.

```
# 6. Create a Module Manifest for this module
$NMHT = @{
  Path              = 'C:\HW\HW.psd1'
  RootModule        = 'HW.psm1'
  Description       = 'Hello World module'
  Author            = 'DoctorDNS@Gmail.com'
  FunctionsToExport = 'Get-HelloWorld'
  ModuleVersion      = '1.0.0'
}
New-ModuleManifest @NMHT
```

These commands create a module manifest, `HW.psd1`, in the module folder. You could re-import the module, using the `-Verbose` switch parameter, to see that PowerShell now loads the module via the manifest.

Trusting the Repository

If you use a repository that is not trusted, attempting to use commands such as `Install-Module` results in a prompt asking whether to use an untrusted repository. You can trust a repository to avoid this warning. Note that a trusted repository is just a NuGet repository that a given system trusts. To trust any repository, you use the `Register-PSRepository` command.

```
# 7. Create the repository as trusted
#    Repeat on every host that uses this repository
$Path = '\\DC1 \RKRepo'
$REPOHT = @{
  Name               = 'RKRepo'
  SourceLocation     = $Path
  PublishLocation    = $Path
  InstallationPolicy = 'Trusted'
}
Register-PSRepository @REPOHT
```

These commands make the new `RKRepo` repository trusted from `DC1`. Note that if you want other hosts to trust this repository, you need to run this command on those hosts.

Viewing Configured Repositories

You use the `Get-PSRepository` command to view the currently configured repositories.

```
# 8. View configured repositories
Get-PSRepository
```

You can see the output from this command in Figure 1.16.

```
PS C:\Foo> # 8. View configured repositories
PS C:\Foo> Get-PSRepository

Name       InstallationPolicy  SourceLocation
----       ------------------  --------------
PSGallery  Untrusted           https://www.powershellgallery.com/api/v2
RKRepo     Trusted             \\DC1\RKRepo
```

Figure 1.16: Viewing configured repositories

As shown in the figure, DC1 currently has two repositories you can use. The PowerShell 7 installation process created the first, the PowerShell Gallery, although the installation process creates it as untrusted. You can configure PowerShell to trust the PowerShell Gallery should you wish.

Publishing a Module

To publish a module to your repository, you use the Publish-Module command.

```
# 9. Publish the module to the repository
Publish-Module -Path C:\HW -Repository RKRepo -Force
```

Publish-Module requires an up-to-date version of the NuGet provider. You added this explicitly to DC1 in the section "Installing PowerShell 7." By using the parameter -Force, you instruct PowerShell to download an appropriate version or the NuGet provider, if needed, before completing the publishing process.

If you plan to make use of repositories, whether public or private, as part of your deployment of PowerShell 7, you should install the latest versions of all modules (and have a process in place to ensure that you keep the downloaded modules up to date on each host in your infrastructure).

Viewing the Repository Folder

You can view the NuGet packages in your RKRepo repository using Get-ChildItem.

```
# 10. View the repository folder
Get-ChildItem -Path C:\RKRepo
```

Figure 1.17 shows the output from this command. You can see that there is just one package in the repository.

```
PS C:\Foo> # 10. View the repository folder
PS C:\Foo> Get-ChildItem -Path C:\RKRepo

    Directory: C:\RKRepo

Mode        LastWriteTime  Length Name
----        -------------  ------ ----
-a--- 11/05/2020    09:20    3463 HW.1.0.0.nupkg
```

Figure 1.17: Viewing the repository folder

Finding a Module

With the module published to your private repository, you can use `Find-Module` to view the module.

```
# 11. Find the module in the RKRepo repository
Find-Module -Repository RKRepo
```

You can see basic module detail in Figure 1.18.

```
PS C:\Foo> # 11. Find the module in the RKRepo repository
PS C:\Foo> Find-Module -Repository RKRepo

Version  Name   Repository   Description
-------  ----   ----------   -----------
1.0.0    HW     RKRepo       Hello World module
```

Figure 1.18: Viewing the RKRepo modules

In this section, you created, trusted, and used a PowerShell repository. In production, you would need policies and procedures to govern who can upload packages to the repository and how to set them up.

Creating a Code-Signing Environment

PowerShell has the ability to control the execution of digitally signed scripts, via settings of the PowerShell execution policy. If you set the execution policy to All Signed, for example, PowerShell does not run any script that is not signed or whose signature is untrusted. If you are developing PowerShell code for customers, you may find it useful to sign your code to ensure that the code you ship is the code the customer has received and to flag any unauthorized changes.

For script signing, PowerShell requires two X.509 digital certificates. The first, the signing certificate, is the certificate you use to sign a script using `Set-AuthenticodeSignature`. The second, the CA certificate, tells Windows and PowerShell to trust the actual signing certificate.

In most organizations that use digitally signed PowerShell scripts, there would be at least one certificate authority (CA) deployed along with the necessary procedures to issue signing keys and to ensure that other systems trust those keys. The instructions for setting up a CA and issuing certificates are outside the scope of this book.

As an alternative to a full-blown CA, you can use self-signed certificates to test a code-signing environment. A self-signed certificate is one that is signed by itself. You can create one with `New-SelfSignedCertifcate`. You can copy this

certificate to the trusted root certificate store on the computer to enable Windows to trust the signing certificate.

The script fragments in this section show you how to create and use a self-signed certificate. Although using self-signed certificates works, in production you should use certificates issued by your organization or from public CAs.

Before You Start

This section uses the Windows Server 2019 host, `DC1`, that you used previously in this chapter. For this section, `DC1` is a Windows 2019 Server (prior to being promoted to be a domain controller in Chapter 3).

Creating a Self-Signed Certificate

You use the `New-SelfSignedCertificate` cmdlet to create a new self-signed code-signing certificate.

```
# 1. Create a self-signed certificate
Import-Module PKI -WarningAction SilentlyContinue
$CERTHT = @{
  Subject           = 'Sign.Reskit.Org'
  Type              = "CodeSigningCert"
  CertStoreLocation = "Cert:\CurrentUser\my"
}
$SignCert = New-SelfSignedCertificate @CERTHT
```

The `New-SelfSignedCertificate` cmdlet adds a new code-signing certificate in the current user's personal certificate store.

Viewing the Certificate

You can view the new certificate.

```
# 2. View Certificate
$SignCert
```

You see the output of this command in Figure 1.19.

```
PS C:\Foo> # 2. View Certificate
PS C:\Foo> $SignCert

   PSParentPath: Microsoft.PowerShell.Security\Certificate::CurrentUser\my

Thumbprint                                Subject              EnhancedKeyUsageList
----------                                -------              --------------------
017D2305332E66C1E109EED14DDC2A9E1379E491  CN=Sign.Reskit.Org   Code Signing
```

Figure 1.19: Viewing the certificate

Each certificate has a unique thumbprint. When you create a new certificate, the cmdlet creates a new thumbprint for you.

Creating a Simple Script

To demonstrate signing PowerShell scripts, create a simple script and store it in the `C:\Foo` folder.

```
# 3. Create a simple .PS1 File
$File = @"
# A script to be signed
"Hello World"
"@
$SignedFile = "C:\Foo\HelloWorld.ps1"
$File |
  Out-File -FilePath $SignedFile -Force
```

Setting Execution Policy

To test the requirement to use a signed script, you set the execution policy to All Signed as follows:

```
# 4. Set Execution Policy to ALL Signed
Set-ExecutionPolicy -ExecutionPolicy AllSigned
```

Attempting to Run the Script

You can try to run the script as follows:

```
# 5. Attempt to Run the script (pre-signing)
& $SignedFile
```

You can see the output from this command in Figure 1.20.

```
PS C:\Foo> # 5. Attempt to Run the File (pre-signing)
PS C:\Foo> & $SignedFile
&: File C:\Foo\HelloWorld.ps1 cannot be loaded. The file C:\Foo\HelloWorld.ps1
is not digitally signed. You cannot run this script on the current system.
For more information about running scripts and setting execution policy, see
about_Execution_Policies at https://go.microsoft.com/fwlink/?LinkID=135170.
```

Figure 1.20: Attempting to run the script

As expected, PowerShell does not run the script file, since it is unsigned and you previously set the execution policy to All Signed.

Signing the Script

You next attempt to sign the script using `Set-AuthenticodeSignature`, as follows:

```
# 6. Sign the script with the $SignCert certificate
Set-AuthenticodeSignature -FilePath $SignedFile -Certificate $SignCert
```

You can see the output from this command in Figure 1.21.

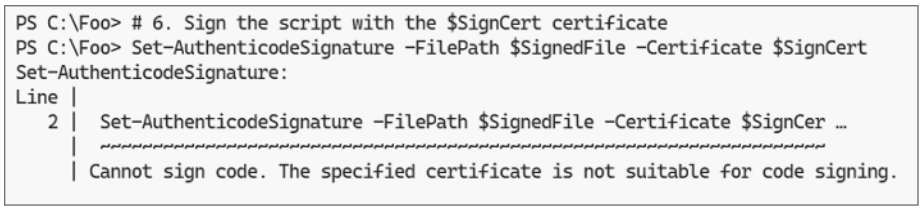

Figure 1.21: Signing a PowerShell script

As you can see, PowerShell declined to sign the script. This is because the signing certificate is not trusted.

Copying a Certificate to the Trusted Publisher Certificate and Trusted Root Stores

One way to trust the self-signed certificate is to copy the certificate into the local machine's Trusted Root certificate store. You also need to ensure that PowerShell trusts code that you sign with a code-signing certificate, which you do by also copying the certificate into the local machine's Trusted Publisher certificate store, as follows:

```
# 7. Copy the cert to the Trusted Root Cert store of Local Machine
#    And to the Trusted Publisher cert store
# local Machine Trusted Root store
$CertStore = 'System.Security.Cryptography.X509Certificates.X509Store'
$CertArgs  = 'Root','LocalMachine'
$Store     = New-Object -TypeName $CertStore -ArgumentList $CertArgs
$Store.Open('ReadWrite')
$Store.Add($SignCert)
$Store.Close()
# Local Machine Trusted Publisher store
$CertStore = 'System.Security.Cryptography.X509Certificates.X509Store'
$CertArgs  = 'TrustedPublisher','LocalMachine'
$Store     = New-Object -TypeName $CertStore -ArgumentList $CertArgs
$Store.Open('ReadWrite')
$Store.Add($SignCert)
$Store.Close()
```

The PKI module you use to create a self-signed certificate has no commands to copy a certificate into either the Local Machine Trusted Publisher store or the Local Machine Trusted Root certificate store. You use the .NET Framework directly to perform the copy.

Signing the Script Again

Now that your signing certificate is trusted, you can attempt to sign the script again, like this:

```
# 8. Re-Sign the script
$SignCert = Get-ChildItem -Path Cert:\CurrentUser\my -CodeSigningCert
Set-AuthenticodeSignature -FilePath $SignedFile -Certificate $SignCert |
    Format-Table -AutoSize -Wrap
```

These commands first retrieve the signing certificate from the current user's personal certificate store. You retrieve a signing cert by using `Get-ChildItem` with the `-CodeSigningCert` parameter. If you had more than one code signing certificate, you would need to amend the snippet to select the correct one.

You can see the output from these commands in Figure 1.22.

```
PS C:\Foo> # 8. Re-Sign the script
PS C:\Foo> $SignCert = Get-ChildItem -Path Cert:\CurrentUser\my -CodeSigningCert
PS C:\Foo> Set-AuthenticodeSignature -FilePath $SignedFile -Certificate $SignCert |
            Format-Table -AutoSize -Wrap

    Directory: C:\Foo

SignerCertificate                        Status StatusMessage       Path
-----------------                        ------ -------------       ----
017D2305332E66C1E109EED14DDC2A9E1379E491 Valid  Signature verified. HelloWorld.ps1
```

Figure 1.22: Signing a PowerShell script with a trusted certificate

Running the Script

Now that the script is signed, you can run it.

```
# 9. Run the script
& $SignedFile
```

You can see the output from running the signed script in Figure 1.23.

As you can see in the output, this script runs successfully.

```
PS C:\Foo> # 9. Run the script
PS C:\Foo> & $SignedFile
Hello World
```

Figure 1.23: Running a signed PowerShell script

Testing the Script's Digital Signature

After you sign a script, you can test the digital signature by using the `Get-AuthenticodeSignature` cmdlet, like this:

```
# 10. Test the script's digital signature
Get-AuthenticodeSignature -FilePath $SignedFile |
  Format-Table -AutoSize&
```

You can see the output from this command in Figure 1.24, which shows that the signature is trusted.

```
PS C:\Foo> # 10. Test the script's digital signature
Get-AuthenticodeSignature -FilePath $SignedFile |
  Format-Table -AutoSize

    Directory: C:\Foo

SignerCertificate                         Status StatusMessage       Path
-----------------                         ------ -------------       ----
017D2305332E66C1E109EED14DDC2A9E1379E491 Valid  Signature verified. HelloWorld.ps1
```

Figure 1.24: Testing a script's digital signature

In a production environment using signed scripts, certificates may be revoked or expire over time. If you rely on signed PowerShell scripts, a best practice would be to test all script signatures regularly or after any changes to the signing certificates. Of course, if you are using signed scripts, ensure that you have a certificate authority set up within your organization or use third-party certificates from firms such as Digicert.

Now that you have tested script signing, you should use `Set-ExecutionPolicy` to reset the execution policy on `DC1` back to Unrestricted. If you want to retain an All Signed execution policy, you must also digitally sign your profile files.

Summary

In this chapter, you learned how you can set up a PowerShell 7 environment. You installed PowerShell 7 and VS, and then you configured PowerShell (and

VS Code) with a profile file. You then used the PowerShell gallery and saw how you can create your own trusted code repository. You finished by looking at signing PowerShell scripts by using self-signed certificates.

With the introduction of PowerShell 7, you should consider how best to deploy PowerShell and associated tools such as VS Code and the new Cascadia Code font. You also need to consider whether you need to use digitally signed scripts, and if so, you need to plan to deploy a code-signing environment.