12

Debugging and Troubleshooting Windows Server

This chapter covers the following recipes:

* Using PS Script Analyzer
* Checking networking connectivity
* Using Best Practices Analyzer
* Exploring PowerShell script debugging
* Performing networking troubleshooting

Diagnosing network issues

# Introduction

You can think of debugging as the art and science of removing bugs from your PowerShell scripts. A script may not do what you or your users want, both during the development of a script and after you deploy the script into production. Troubleshooting is a process you go through to determine why your script is not doing what you want, and then it helps you resolve your issues.

There are three broad classes of problems that you encounter:

* Syntax errors
* Logic errors

Runtime errors

Syntax errors are very common – especially if your typing is less than perfect. It is so easy to type Get-ChildTiem as opposed to Get-ChildItem. The good news is that your script won't run successfully until you resolve your syntax errors. There are several ways to avoid syntax errors and to simplify the task of finding and removing them. One simple way is to use a good code editor, such as VS Code. Just like Microsoft Word, VS Code highlights potential syntax errors to help you identify, fix, and eliminate them as you make them.

Another way to reduce typos or syntax issues is to use tab completion in the PowerShell console or the VS Code editor. You type some of the necessary text, hit the tab key, and PowerShell does the rest of the typing.

Logic errors, on the other hand, are bits of code that do not do what you want or expect. There are many reasons why code could have a logic error. One issue many IT pros encounter is defining a variable but not using it later or typing the variable name incorrectly. Tools such as the PowerShell Script Analyzer can analyze your code and help you track down potential issues in your code.

You can also encounter runtime errors. For example, your script to add and configure a user in your AD could encounter a runtime problem. The AD service on a DC may have crashed, the NIC in your DC might have failed, or the network path from a user to the DC might have a failed router or one with an incorrect routing table. Checking network connectivity ensures that the network path from your user to the relevant servers is working as required. But you also need to confirm that your networking configuration itself is correct.

Troubleshooting network issues in large and complex networks can be a challenge. The Get-NetView module (and cmdlet) enables you to gather important information about networking on a given host, which can help resolve issues. Windows and PowerShell do not install this module by default, but it is readily available from the PowerShell gallery.

The Get-NetView command produces a vast amount of detail that can be hard to use. In many cases, there are a few very common networking issues you may face. You will examine some of these problems and their solution in the Checking Network Connectivity recipe.

You may have a working system or service that, in some cases, could become problematic if you are unlucky. The Best Practices Analyzer enables you to examine core Windows services to ensure that you run these services in the best possible way.

Windows comes with a best practices analyzer (BPA) that can help you analyze your hosts to check whether you have deployed using best practices. You can use the built-in analyzer to assess your systems and possibly implement best practice recommendations, such as always having at least two domain controllers. Some BPA recommendations may not be appropriate for your environment. Running BPA regularly to avoid configuration creep is a great approach.

## The systems used in the chapter

This chapter primarily uses the server SRV1, a domain-joined server in the Reskit.Org domain. You have used this server, and the domain, in previous chapters of this book. You also need access to the domain’s two DCs (DC1 and DC2). Additionally, you should ensure that SRV1 has access to the internet.

The hosts used in this chapter are as shown here:

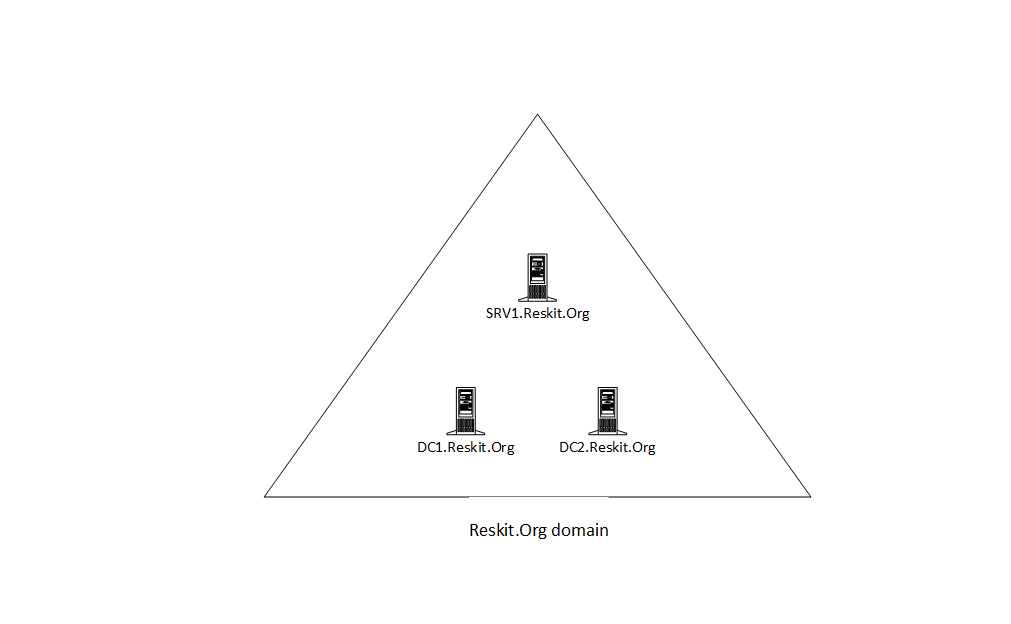


Figure 12.1: Host in use for this chapter

# Using PS Script Analyzer

The PowerShell Script Analyzer is a PowerShell module produced by the PowerShell team that analyzes your code and provides opportunities to improve. You can download the latest version of the module from the PowerShell Gallery. Like many PowerShell modules, the Script Analyzer is subject to constant updating. You can always download the newest version of this module directly from GitHub.

Using the VS Code editor to develop your code, you should know that the Script Analyzer is built into VS Code. So as you are creating your PowerShell script, VS Code highlights any errors the Script Analyzer finds. VS Code, therefore, helps you to write better code straightaway.

Another feature of the PowerShell Script Analyzer is the ability to reformat PowerShell code to be more readable. You can configure numerous settings to define how the Script Analyzer should reformat your code.

## Getting ready

## How to do it...

1. Discovering the Powershell Script Analyzer module

Find-Module -Name PSScriptAnalyzer |

  Format-List Name, Version, Type, Desc\*, Author, Company\*, \*Date, \*URI\*

1. Installing the script analyzer module

Install-Module -Name PSScriptAnalyzer -Force

1. Discovering the commands in the Script Analyzer module

Get-Command -Module PSScriptAnalyzer

1. Discovering analyzer rules

Get-ScriptAnalyzerRule |

  Group-Object -Property Severity |

    Sort-Object -Property Count -Descending

1. Examining a rule

Get-ScriptAnalyzerRule |

  Select-Object -First 1 |

    Format-List

1. Creating a script file with issues

@'

# Bad.ps1

# A file to demonstrate Script Analyzer

#

### Uses an alias

$Procs = gps

### Uses positional parameters

$Services = Get-Service 'foo' 21

### Uses poor function header

Function foo {"Foo"}

### Function redefines a built in command

Function Get-ChildItem {"Sorry Dave I cannot do that"}

### Command uses a hard-coded computer name

Test-Connection -ComputerName DC1

### A line that has trailing white space

$foobar = "foobar"

### A line using a global variable

$Global:foo

'@ | Out-File -FilePath "C:\Foo\Bad.ps1"

1. Checking the newly created script file

Get-ChildItem C:\Foo\Bad.ps1

1. Analyzing the script file

Invoke-ScriptAnalyzer -Path C:\Foo\Bad.ps1 |

  Sort-Object -Property Line

1. Defining a function to format better

$Script1 = @'

function foo {"hello!"

Get-ChildItem -Path C:\FOO

}

'@

1. Defining formatting settings

$Settings = @{

  IncludeRules = @("PSPlaceOpenBrace", "PSUseConsistentIndentation")

  Rules = @{

    PSPlaceOpenBrace = @{

      Enable = $true

      OnSameLine = $true

    }

    PSUseConsistentIndentation = @{

      Enable = $true

    }

  }

}

1. Invoking formatter

Invoke-Formatter -ScriptDefinition $Script1 -Settings $Settings

1. Changing settings and reformatting

$Settings.Rules.PSPlaceOpenBrace.OnSameLine = $False

Invoke-Formatter -ScriptDefinition $Script1 -Settings $Settings

## How it works...

In step 1, you use the Find-Module command to find the PSScriptAnalyzer module in the PowerShell Gallery. The output of this step looks like this:

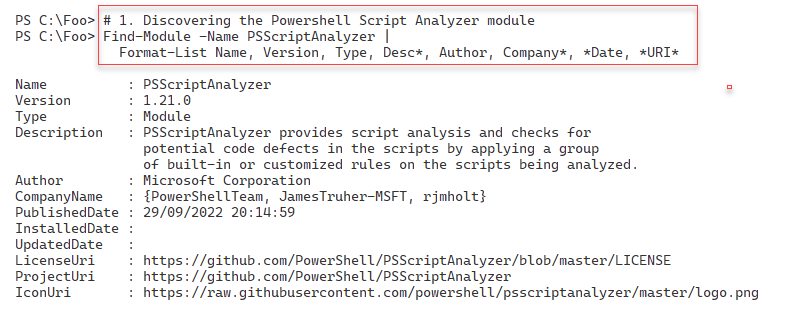


Figure 12.2: Finding the PowerShell Script Analyzer module

In step 2, you install the PSScriptAnalyzer module, which generates no console output. In step 3, you use the Get-Command cmdlet to discover the commands inside the PSScriptAnalyzer module, with output like this:

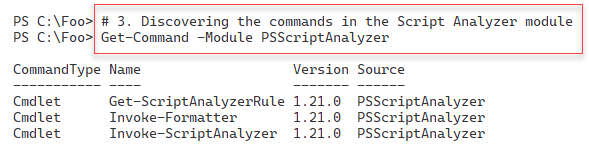


Figure 12.3: Getting the commands in the Script Analyzer module

The PowerShell Script Analyzer uses a set of rules that define potential problems with your scripts. In step 4, you use Get-ScriptAnalyzerRule to examine the types of rules available, with output like this:

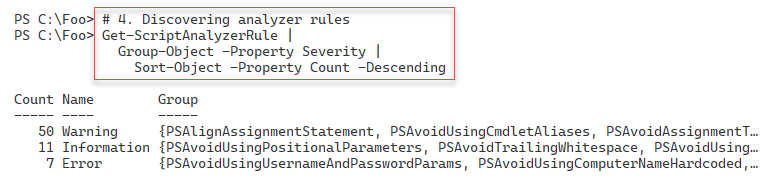


Figure 12.4: Examining Script Analyzer rule types

You can view one of the Script Analyzer rules using the Get-ScriptAnalyzerRule, as shown in step 5, with output like this:

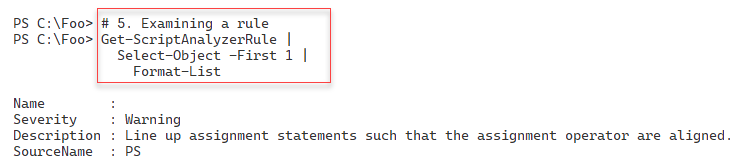


Figure 12.5: Examining a Script Analyzer rule

In step 6, which generates no console output, you create a new script file (bad.ps1). This script has issues that possibly need attention. In step 7, you check on the newly created script file with output like this:

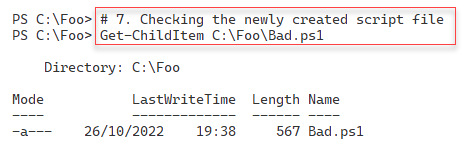


Figure 12.6: Checking the newly created script file

In step 8, you use the Invoke-ScriptAnalyzer command to check the C:\Foo\Bad.ps1 file for potential issues. The output from this step looks like this:

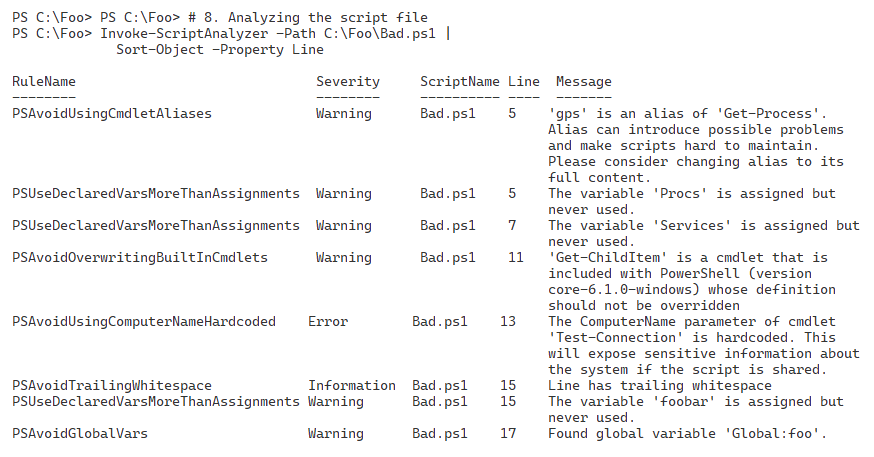


Figure 12.7: Analyzing the script

A second and useful feature of Script Analyzer is to reformat a script file to improve the script's layout. Reformatting can be useful, for example, if you are cutting and pasting code from various internet sources (each with its own unique formatting styles). Here, you define a simple PowerShell with no formatting applied. This step generates no console output.

IT pros may never agree on what constitutes a good code layout. Script Analyzer lets you specify exactly how you want it to format your code. In step 10, you specify a set of formatting rules, which generates no output.

In step 11, you invoke the script formatter using the settings specified in the previous step. The output of this step is as follows:

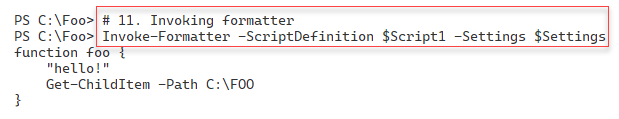


Figure 12.8: Using the script formatter and formatting rules

In step 12, you update the formatting rules. These changes ask the formatter to place the start of a function definition's script block. You have the option to have the formatter put the open brace character ({) that follows the function keyword and function name on the same line or, as in this case, on a separate line. The output of this step looks like this:

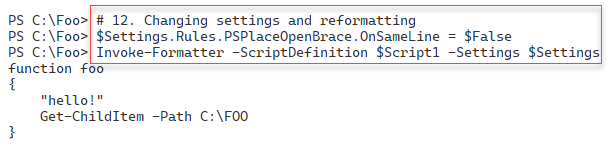


Figure 12.9: Changing formatter rules reformatting the function definition

## There's more...

In step 1, you view details about the PSScriptAnalyzer module. The version shown in the output may differ from what you see if you test the recipe. The module developers update this module regularly. Microsoft regularly posts details about what is new in an updated module version. For example, you can read details about the latest (at the time of writing!) update to the module here: https://devblogs.microsoft.com/powershell/psscriptanalyzer-pssa-1-21-0-has-been-released/.

Note that the author of this module was a long-time PowerShell team member, Jim Truher. Interestingly, Jim did some demonstrations the first time Microsoft displayed Monad (as PowerShell was then named) at the PDC in the autumn of 2003. This book’s author was in the room!

In the final part of this recipe, you use the formatting feature to format a small function. The goal of the formatter is to help you create easier-to-read code. That can be very useful for long production scripts or when you have hundreds of scripts that you want to format consistently. A consistent layout makes it easier to find issues, as well as simplifies subsequent script maintenance.

Step 12 sets a rule that makes the Script Analyzer's formatter put a script block's opening brace on a separate line. Opinion varies as to whether this is a good approach. Therefore, the formatting rules provide you with options such as lining up the “=” sign in a set of assignment statements, and many more. The documentation on these rules is not particularly helpful, but you can start here: <https://www.powershellgallery.com/packages/PSScriptAnalyzer/1.21.0/Content/Settings%5CCodeFormatting.psd1>. If Microsoft issues an update to this module, adjust this URL to point to the latest version.

And for the curious, you can look at the module and its contents on the project’s GitHub repository at https://github.com/PowerShell/PSScriptAnalyzer.

# Performing BASIC Network Troubleshooting

For many common network problems, some simple steps may help you resolve your more common issues or point you toward a solution.

In this recipe, you carry out some basic troubleshooting on a local SRV1, a domain-joined host running Windows Server 2022. A common adage amongst many IT pros is that the problem is DNS, irrespective of the problem (until you prove otherwise). You start this recipe by getting the host's fully qualified domain name (FQDN) and the IPv4 address of the DNS server, and then you check whether the DNS server(s) is online.

You then use the configured DNS server to determine the names of the DCs in your domain and ensure you can reach each DC over TCP port 389 (LDAP) and TCP port 445 (for GPOs). Next, you test the default gateway's availability. Finally, you test the ability to reach a remote host over port 80 (HTTP) and port 443 (HTTP over SSL/TLS).

In most cases, the simple tests in this recipe, run on the afflicted host, should help you find some of the more common problems,

## Getting ready

You run this recipe on SRV1, a domain-joined host. You must have both DC1 and DC2 running, providing a DNS service for the domain. You should also ensure you have configured SRV1 to point to these two DCs for DNS.

## How to do it...

1. Getting and displaying the DNS name of this host

$DNSDomain = $Env:USERDNSDOMAIN

$FQDN      = "$Env:COMPUTERNAME.$DNSDomain"

"Host FQDN: $FQDN"

1. Getting the DNS server address

$DNSHT = @{

  InterfaceAlias = "Ethernet"

  AddressFamily  = 'IPv4'

}

$DNSServers = (Get-DnsClientServerAddress @DNSHT).ServerAddresses

$DNSServers

1. Checking if the DNS servers are online

Foreach ($DNSServer in $DNSServers) {

  $TestDNS = Test-NetConnection -Port 53 -ComputerName $DNSServer

  $Result  = $TestDNS ? "Available" : ' Not reachable'

  "DNS Server [$DNSServer] is $Result"

}

1. Defining a search for DCs in our domain

$DNSRRName = "\_ldap.\_tcp." + $DNSDomain

$DNSRRName

1. Getting the DC SRV records

$DCRRS = Resolve-DnsName -Name $DNSRRName -Type all |

    Where-Object IP4address -ne $null

$DCRRS

1. Testing each DC for availability over LDAP

ForEach ($DNSRR in $DCRRS){

  $TestDC = Test-NetConnection -Port 389 -ComputerName $DNSRR.IPAddress

  $Result  = $TestDC ? 'DC Available' : 'DC Not reachable'

  "DC [$($DNSRR.Name)]  at [$($DNSRR.IPAddress)]   $Result for LDAP"

}

1. Testing DC availability for SMB

ForEach ($DNSRR in $DCRRS){

  $TestDC =

    Test-NetConnection -Port 445 -ComputerName $DNSRR.IPAddress

  $Result  = $TestDC ? 'DC Available' : 'DC Not reachable'

  "DC [$($DNSRR.Name)]  at [$($DNSRR.IPAddress)]   $Result for SMB"

}

1. Testing the default gateway

$NIC     = Get-NetIPConfiguration -InterfaceAlias Ethernet

$DGW     = $NIC.IPv4DefaultGateway.NextHop

$TestDG = Test-NetConnection $DGW

$Result  = $TestDG.PingSucceeded ? "Reachable" : ' NOT Reachable'

"Default Gateway for [$($NIC.Interfacealias) is [$DGW] - $Result"

1. Testing a remote website using ICMP

$Site = "WWW.Packt.Com"

$TestIP   = Test-NetConnection -ComputerName $Site

$ResultIP = $TestIP ? "Ping OK" : "Ping FAILED"

"ICMP to $Site - $ResultIP"

1. Testing a remote website using port 80

$TestPort80 = Test-Connection -ComputerName $Site -TcpPort 80

$Result80   = $TestPort80  ? 'Site Reachable' : 'Site NOT reachable'

"$Site over port 80   : $Result80"

1. Testing a remote website using port 443

$TestPort443 = Test-Connection -ComputerName $Site -TcpPort 443

$Result443   = $TestPort443  ? 'Site Reachable' : 'Site NOT reachable'

"$Site over port 443  : $Result443"

## How it works...

In step 1, you create a variable to hold the FQDN of the host. Then, you display the value with output like this:

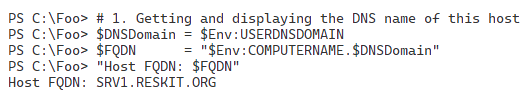


Figure 12.10: Displaying the FQDN of this host

In step 2, you use Get-DNSClientServerAddress to get the IP addresses of the DNS servers that you (or DHCP) have configured on the host. The output looks like this:

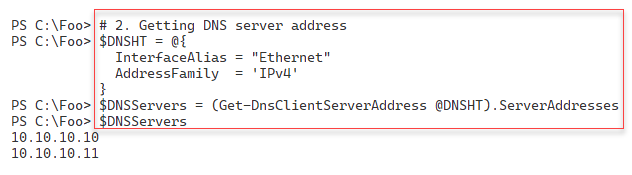


Figure 12.11: Discovering configured DNS servers

In step 3, you check whether each configured DNS server is available, with output like this:

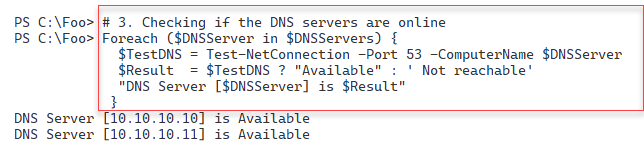


Figure 12.12: Checking the reachability of each configured DNS server

In step 4, you determine the DNS resource record (RR) name for the SRV records registered by active DCs for a given domain. The output looks like this:

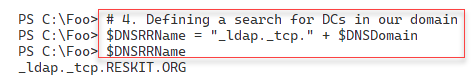


Figure 12.13: Defining an RR name for DC SRV records

In step 5, you retrieve the SRV RRs for the DCs in the Reskit.Org domain. Each RR represents a server that can act as a DC in the Reskit.Org domain. The output of this step looks like this:

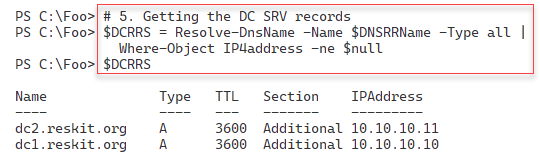


Figure 12.14: Querying for DNS RRs for DCs

In step 6, you test each discovered DC for LDAP connectivity, with output like this:

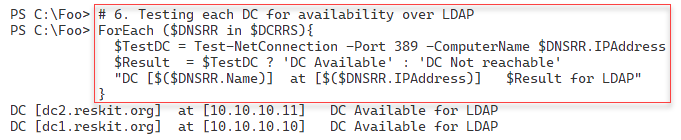


Figure 12.15: Testing LDAP connectivity to domain controllers

For each host's Group Policy agent to download GPOs from a DC, the host uses an SMB connection to the SYSVOL share on a DC. In step 7, you check connectivity to each DC's SMB port (port 445). The output of this step looks like this:

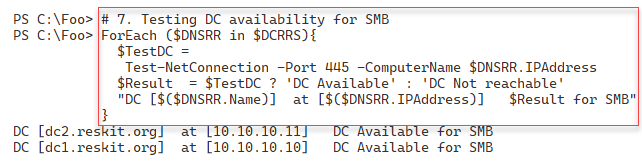


Figure 12.16: Testing SMB connectivity to domain controllers

In step 8, you check whether your host can reach its configured default gateway. The output of this step looks like this:

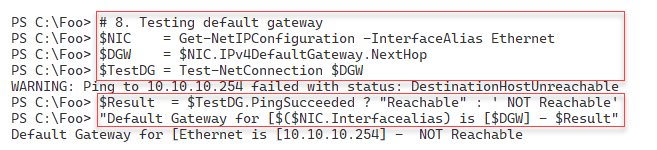


Figure 12.17: Testing the default gateway

In step 9, you check to see if you can reach an external internet-based host using ICMP (aka ping). The output of this step looks like this:

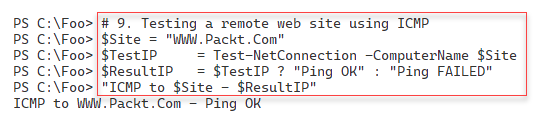


Figure 12.18: Testing connectivity to an internet site

In step 10, you check to see whether you can reach the same server, via the HTTP port, port 80, with output like this:

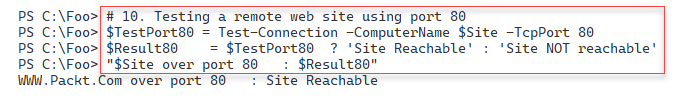


Figure 12.19: Testing connectivity over port 80

Finally, in step 11, you check to see whether you can reach the same server via HTTP over SSL/TLS, port 443, with output like this:

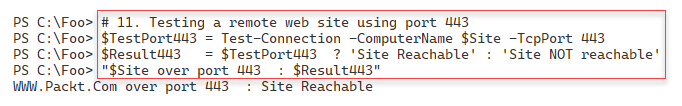


Figure 12.20: Testing connectivity over port 443

## There's more...

In step 1, you create a variable to hold the FQDN of the host and then display this value at the console. You should ensure you have configured the hostname correctly and want to use this to ensure that your host has properly registered your host's FQDN in any configured DNS server. If DNS misregistration is causing problems, you may wish to adapt this script to check for correct DNS resource record registration.

In step 3, you use the ternary operator. This operator is a new feature in PowerShell 7.0. For more details on the ternary operator, see https://learn.microsoft.com/ powershell/module/microsoft.powershell.core/about/about\_if?view=powershell-7.3.

In step 4, you create a DNS RR name, which you then use in step 5 to query for any SRV records of that name. AD uses DNS as a locator service – each DC registers SRV records to advertise its ability to serve as a DC. The SRV record contains the FQDN name of the advertised DC. The approach taken by these two steps is similar to how any domain client finds a domain controller. The Windows Netlogon service on a DC registers all the appropriate SRV records each time the service starts or every 24 hours. One troubleshooting technique is to use Restart-Service to restart the Netlogon service on each DC.

If you have a large routed network, you may wish to move the default gateway check, performed here in step 8, earlier in your production version of this recipe, and possibly before step 3. If you can't reach your default gateway and your DNS server and your DCs are on different subnetworks, the earlier steps are going to fail due to a default gateway issue. For the remaining steps to work, you need to resolve the problem with the default gateway not being reachable (e.g., by turning on the default gateway VM).

In steps 9, 10, and 11, you test connectivity to a remote host via ICMP and ports 80 and 443. A host or an intermediate router may drop ICMP traffic yet allow port 80/443 traffic in many cases. So just because a ping has not succeeded does not necessarily suggest a point of failure at all – it may be a deliberate feature and by design.

In some of the steps in this recipe, you used the PowerShell 7 ternary operator to construct the message output. These steps provide a good example of this operator. For more details on the ternary operator, see https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about\_operators.

# Using Get-NetView to Diagnose Network issues

In most cases, network issues are relatively easy to resolve. But in some cases, especially in larger networks, problems can be much more complex. Getting a resolution can require a lot of information. As shown in the Performing Basic Network Troubleshooting recipe, a few steps can point you to the issue’s cause.

Get-NetView is a tool that collects details about your network environment, which can help you troubleshoot network issues. This tool gathers everything you might want to know about a host to enable you to resolve complex issues.

The Get-NetView module contains a single function, Get-NetView. The command pulls together a range of network details. The command creates a set of text files for you to peruse and a ZIP file containing those same details. You can view the text files directly on the host or email the ZIP file to a central network team for deeper analysis.

Get-NetView output includes the following details:

* Get-NetView metadata
* The host environment (including OS, hardware, domain, and hostname)
* Physical, virtual, and container NICs
* Network configuration (including IP addresses, MAC addresses, neighbors, and IP routes)
* Physical switch configuration, including QoS policies
* Hyper-V VM configuration
* Hyper-V virtual switches, bridges, and NATs
* Windows device drivers
* Performance counters

System and application events

The output provided by Get-NetView, as the above list suggests, is voluminous. To help troubleshoot a given issue, only a very small amount of the information is likely to be useful to you. However, if there is an issue in your network, this information can help you troubleshoot.

## Getting ready

This recipe uses SRV1, a domain-joined Windows Server 2022 host. You have installed PowerShell 7 and VS Code on this host.

## How to do it...

1. Finding the Get-NetView module on the PS Gallery

Find-Module -Name Get-NetView

1. Installing the latest version of Get-NetView

Install-Module -Name Get-NetView -Force -AllowClobber

1. Checking the installed version of Get-NetView

Get-Module -Name Get-NetView -ListAvailable | ft -au

1. Importing Get-NetView

Import-Module -Name Get-NetView -Force

1. Creating a new folder for NetView output

$FolderName = 'C:\NetViewOutput'

New-Item -Path $FolderHName -ItemType Directory | Out-Null

1. Running Get-NetView

Get-NetView -OutputDirectory $FolderHName

1. Viewing the output folder using Get-ChildItem

$OutputDetails = Get-ChildItem -Path $FolderName

$OutputDetails

1. Viewing the output folder contents using Get-ChildItem

$Results = $OutputDetails | Select-Object -First 1

Get-ChildItem -Path $Results

1. Viewing the IP configuration

Get-Content -Path $Results\\_ipconfig.txt

## How it works...

In step 1, you find the Get-NetView module on the PowerShell Gallery. The output from this step looks like this:

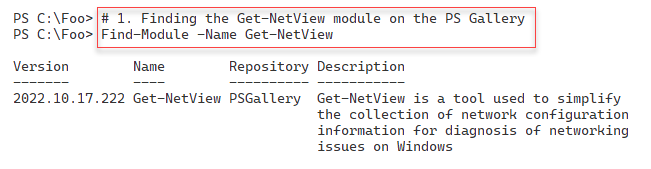


Figure 12.21: Finding the Get-NetView module

In step 2, you download and install the latest version of this module, which generates no output. In step 3, you check which version (or versions) of the Get-NetView module are on SRV1, with output like this:

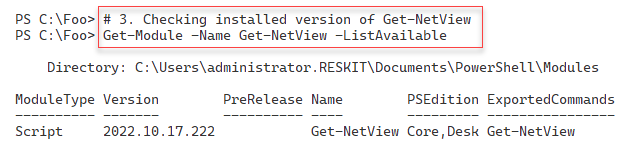


Figure 12.22: Checking the installed version of the Get-NetView module

In step 4, you import the Get-NetView module. In step 5, you create a new folder on the C:\ drive to hold the output that Get-NetView generates. These two steps produce no output.

In step 6, you run Get-NetView. As you can see, as it runs, the command logs a series of network configuration details and outputs a running commentary. This command generates a lot of console output, a subset of which looks like this:

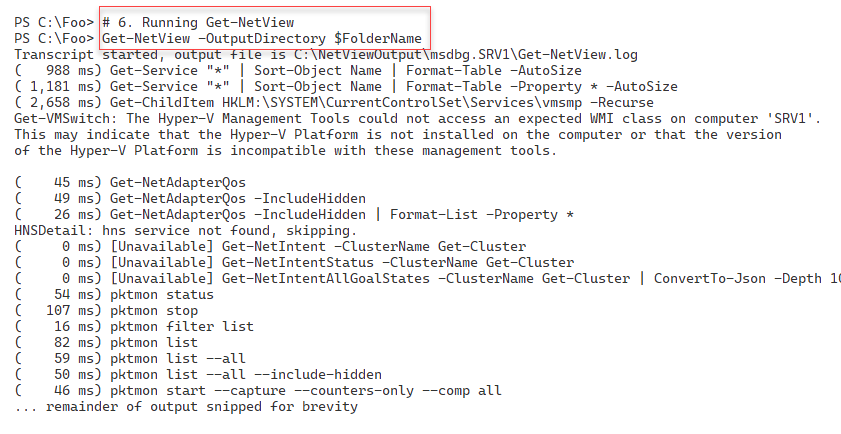


Figure 12.23: Running Get-NetView

In step 7, you view the output folder to view the files created by Get-NetView, with output like this:

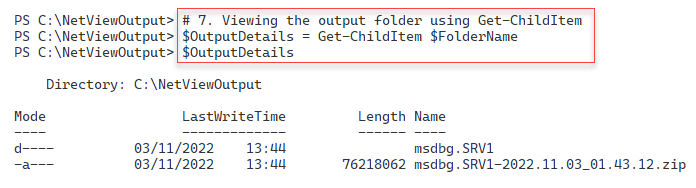


Figure 12.24: Viewing the Get-NetView output folder

In step 8, you view the output folder that Get-NetView populates with detailed network configuration information, with output like this:

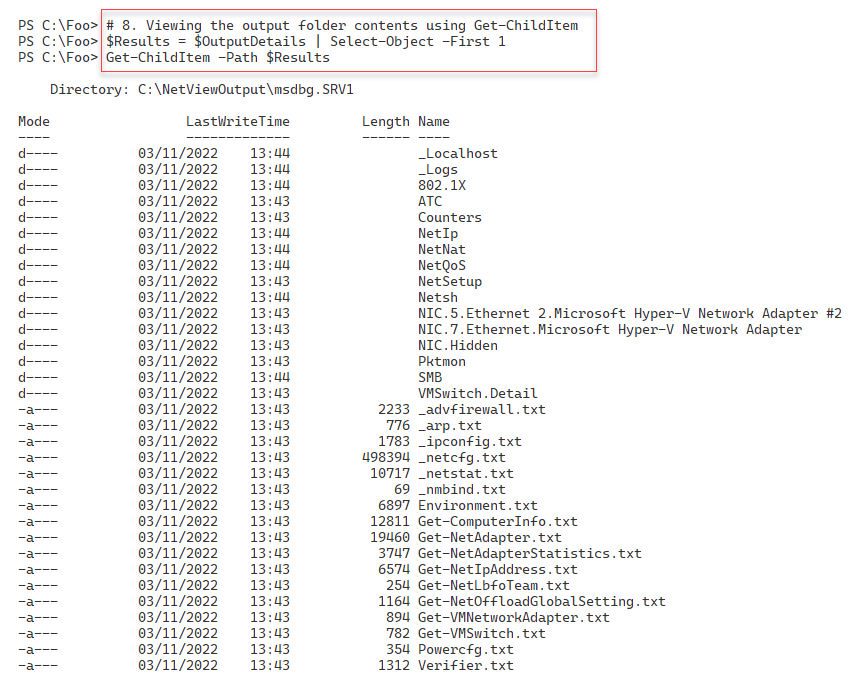


Figure 12.25: Viewing the Get-NetView output folder

In step 9, you examine one of the files created by Get-NetView. This file contains details of the IP configuration of the server, with output that looks like this:

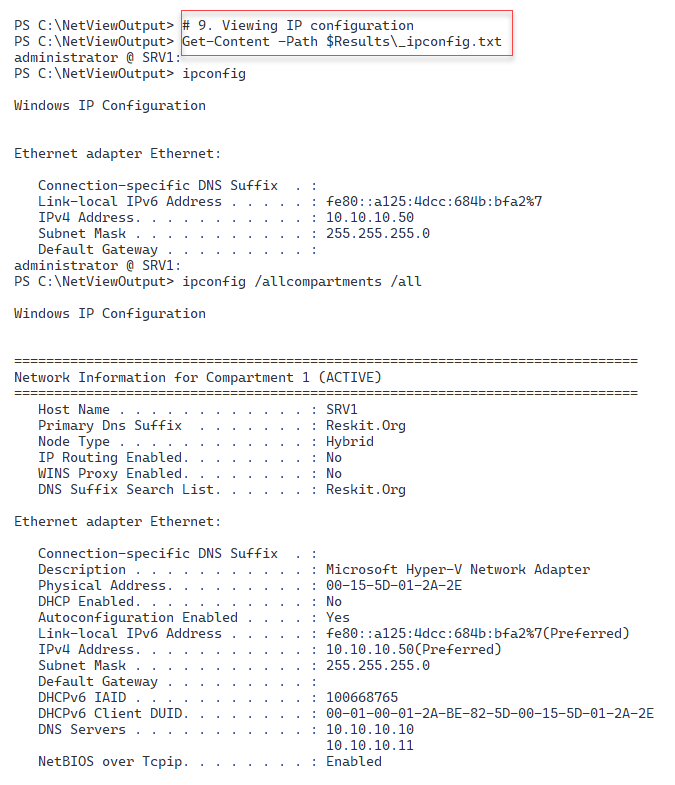


Figure 12.26: Viewing IP configuration

## There's more...

In *step 3*, you check the version or versions of the Get-NetView module on your system. You may see a later version of this module.

In *step 7*, you view the files output by Get-NetView. As you can see, there is a folder and a ZIP archive file in the output folder. Get-NetView adds all the network information to separate files in the output folder and then compresses all that information into a single archive file that you can send to a network technician for resolution. This approach allows the remote technician to view the ZIP file's contents, even if they do not have access to the host (and can read the files in the output folder).

In *step 9*, you view one of the many bits of information created by Get-NetView. In this case, you look at the IP configuration information, including the IP address, subnet mask, and the default gateway, as well as the configured DNS servers.

# Using Best Practices Analyzer

One way to avoid needing to perform troubleshooting is to deploy your services in a more trouble-free, or at least trouble-tolerant, manner. There are many ways to deploy and operate your environment, and some methods are demonstrably better than others. For example, having two DCs, two DNS servers with AD-integrated zones, and two DHCP servers in a failover relationship means that you can experience numerous issues in these core services and still deploy a reliable end-user service. While you may still need to troubleshoot to resolve any issue, your services are running acceptably, with your users unaware that there is an issue.

Along with industry experts, MVPs, and others, Microsoft product teams have identified recommendations for deploying a Windows infrastructure. Some product teams, such as Exchange, publish extensive guidance and have developed a self-contained tool.

The Windows Server BPA is a built-in Windows Server tool that analyzes your on-premises servers for adherence to best practices. A best practice is a guideline that industry experts agree is the best way to configure your servers. For example, most AD experts recommend you have at least TWO domain controllers for each domain. But for a test environment, that may be overkill. So while best practices are ones to strive for, sometimes they may be inappropriate for your needs. It is, therefore, important to use some judgment when reviewing the results of BPA.

Important note: BPA does not work natively in PowerShell 7 on any supported Windows Server version, including (at the time of writing) Windows Server 2022. There is, however, a way around this that involves using PowerShell remoting and running the BPA in Windows PowerShell, as you can see from this recipe.

BPA with Windows Server 2022 comes with 14 BPA models. Each model is a set of rules that you can use to test your AD environment. The AD team, for example, has built a BPA model for Active Directory, Microsoft/Windows/DirectoryServices, which you can run to determine issues with AD on a domain controller.

In this recipe, you create a PowerShell remoting session with DC1. You use the Invoke-Command cmdlet to run the BPA cmdlets, allowing you to analyze, in this recipe, the Active Directory model. In effect, you run the actual cmdlets in Windows PowerShell via remoting.

## Getting ready

This recipe uses SRV1, a domain-joined Windows 2022 server in the Reskit.Org domain. You also need the domain controllers in the Reskit.Org (DC1 and DC2) online for this recipe.

## How to do it...

1. Creating a remoting session to Windows PowerShell on DC1

$BPASession = New-PSSession -ComputerName DC1

1. Discovering the BPA module on DC1

$ScriptBlock1 = {

  Get-Module -Name BestPractices -List |

    Format-Table -AutoSize

}

Invoke-Command -Session $BPASession -ScriptBlock $ScriptBlock1

1. Discovering the commands in the BPA module

$ScriptBlock2 = {

    Get-Command -Module BestPractices  |

      Format-Table -AutoSize

}

Invoke-Command -Session $BPASession -ScriptBlock $ScriptBlock2

1. Discovering all available BPA models on DC1

$ScriptBlock3 = {

  Get-BPAModel  |

    Format-Table -Property Name, Id, LastScanTime -AutoSize

}

Invoke-Command -Session $BPASession -ScriptBlock $ScriptBlock3

1. Running the BPA DirectoryServices model on DC1

$ScriptBlock4 = {

  Invoke-BpaModel -ModelID Microsoft/Windows/DirectoryServices -Mode ALL |

    Format-Table -AutoSize

}

Invoke-Command -Session $BPASession -ScriptBlock $ScriptBlock4

1. Getting BPA results from DC1

$ScriptBlock5 = {

  Get-BpaResult -ModelID Microsoft/Windows/DirectoryServices  |

    Where-Object Resolution -ne $null|

      Format-List -Property Problem, Resolution

}

Invoke-Command -Session $BPASession -ScriptBlock $ScriptBlock5

## How it works...

In step 1, you create a PowerShell remoting session with your domain controller, DC1. This step creates no output. In step 2, you run the Get-Module command on DC1 using the remoting session. The output of this step looks like this:

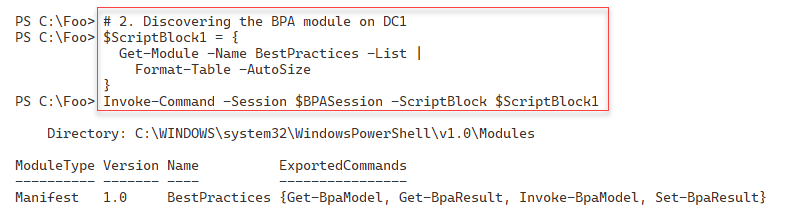


Figure 12.27: Viewing the BPA module on DC1

In step 3, you discover the commands contained in the BPA module (on DC1), with output like this:

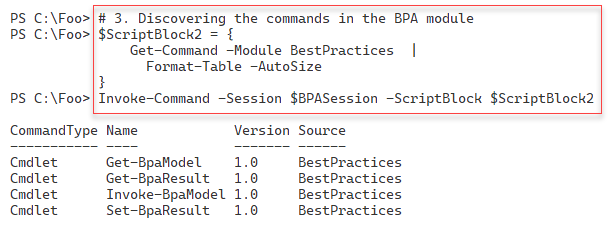


Figure 12.28: Discovering the commands inside the BPA module

In step 4, you discover the BPA models, which are available on DC1. The output looks like this:

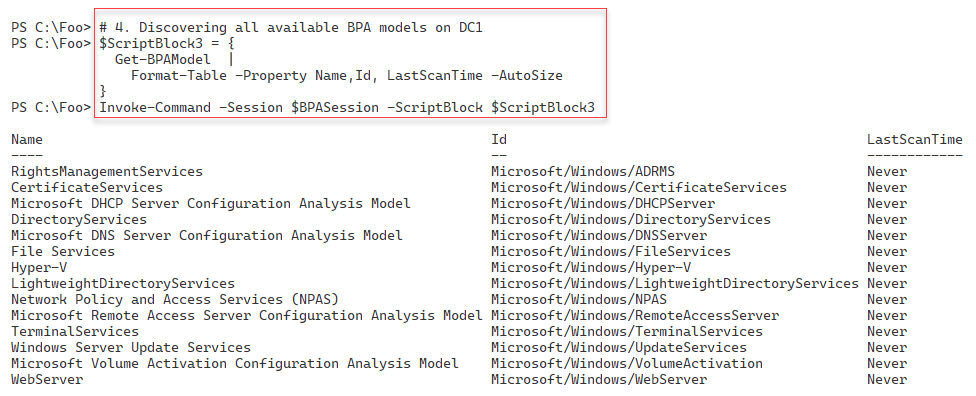


Figure 12.29: Discovering the BPA models available on DC1

In step 5, you use the Invoke-BpaModel command to run the Directory Services BPA model on DC1. Invoking the model produces minimal output, as follows:

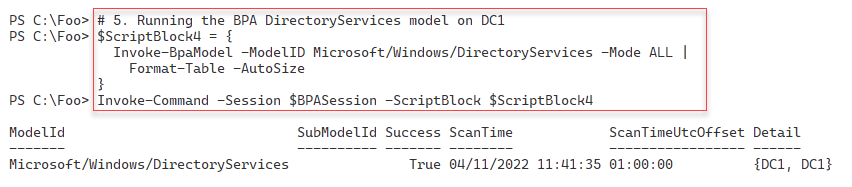


Figure 12.30: Running the Directory Services BPA

To obtain the detailed results of the BPA scan, you use the Get-BpaResult command, as you can see in step 6, which produces the following output:

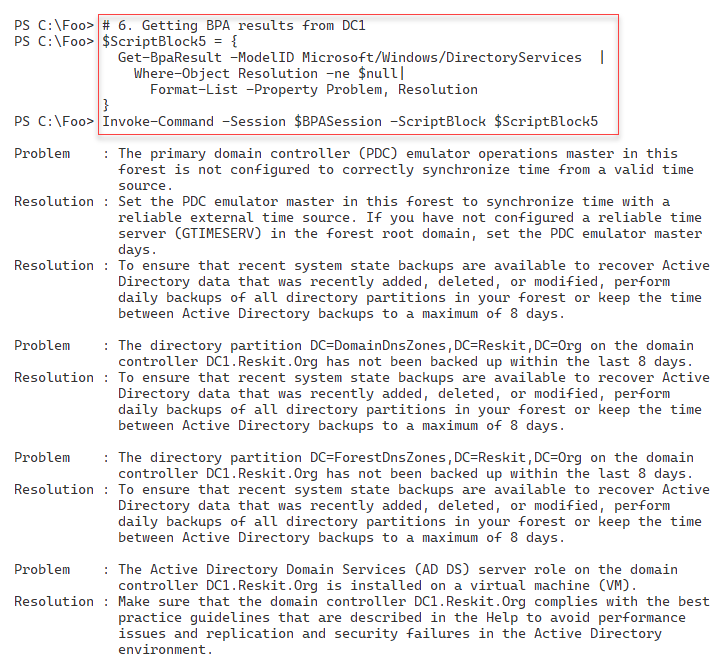


Figure 12.31: Viewing BPA results

## There's more...

BPA results include details of unsuccessful tests. The unsuccessful results, where BPA finds that your deployment does not implement a best practice, are the ones you may need to review and take action.

In step 6, you retrieve the results of the BPA scan you ran in the previous step. The results show three fundamental issues:

* You have not synchronized time on the DC holding the PDC emulator FSMO role with some reliable external source. This issue means that time on your hosts could "wander" from real-world time, possibly leading to problems later on. See https://blogs.msmvps.com/acefekay/tag/pdc-emulator-time-configuration for more information on configuring your DCs with a reliable time source.
* You have not backed up your AD environment. Even with multiple DCs, performing regular backups is best practice. See https://docs.microsoft.com/en-us/windows/win32/ad/backing-up-and-restoring-an-active-directory-server for more information on backing up and restoring a DC.

DC1 is a DC you are running in a VM. While Microsoft supports such a deployment, there are some best practices you should adhere to to ensure the reliable running of your AD service. See https://docs.microsoft.com/windows-server/identity/ad-ds/get-started/virtual-dc/virtualized-domain-controllers-hyper-v for more details on virtualizing DCs using Hyper-V.

For a test environment, these issues are inconsequential, and you can probably ignore them. If you are using Hyper-V for test VMs, you can configure Hyper-V to update the VMs' local time, at least for the DCs you run in a VM. A backup of your AD is unnecessary in a test environment. And running a domain controller in a Hyper-V, at least for a testing environment, is not an issue with the latest, supported Windows Server versions.

# Exploring PowerShell script debugging

PowerShell 7, as well as Windows PowerShell, contains some great debugging features. You use these debugging tools to find and remove errors in your scripts. You can set breakpoints in a script. When you run the script, PowerShell stops execution at the breakpoint. You can set a breakpoint to stop at a particular line in the script, any time the script reads or writes a PowerShell variable, or any time PowerShell calls a named cmdlet.

When PowerShell encounters a breakpoint, it suspends processing and presents you with a debugging prompt, as you see in this recipe. You can then examine the results that your script has produced. When you hit a breakpoint, PowerShell enters a debug terminal from which you can run additional commands. This helps to ensure your script produces the output and results you expect. If your script adds a user to the AD and then performs an action on that user (adding the user to a group, for example). You could stop the script just after the Add‑ADUser command completes. You could then use Get-AdUser or other commands to check whether your script has added the user as you expected. You can then use the continue statement to resume your script. PowerShell then resumes running your script until it either completes or hits another breakpoint.

## Getting ready

This recipe uses SRV1, a domain-joined host controller in the Reskit.Org domain. You have installed PowerShell 7 on this host. You run this script from the PowerShell console.

## How to do it...

1. Creating a script to debug

$Script = @'

# Script to illustrate breakpoints

Function Get-Foo1 {

  param ($J)

  $K = $J\*2           # NB: line 4

  $M = $K             # NB: $M written to

  $M

  $BIOS = Get-CimInstance -Class Win32\_Bios

}

Function Get-Foo {

  param ($I)

  (Get-Foo1 $I)      # Uses Get-Foo1

}

Function Get-Bar {

  Get-Foo (21)}

# Start of ACTUAL script

  "In Breakpoint.ps1"

  "Calculating Bar as [{0}]" -f (Get-Bar)

'@

1. Saving the script

$FileName = 'C:\Foo\Breakpoint.ps1'

$Script| Out-File -FilePath $FileName

1. Executing the script

& $FileName

1. Adding a breakpoint at a line in the script

Set-PSBreakpoint -Script $FileName -Line 4 |  # breaks at line 4

    Out-Null

1. Adding a breakpoint on the script using a specific command

Set-PSBreakpoint -Script $FileName -Command "Get-CimInstance" |

  Out-Null

1. Adding a breakpoint when the script writes to $M

Set-PSBreakpoint -Script $FileName -Variable M -Mode Write |

  Out-Null

1. Viewing the breakpoints set in this session

Get-PSBreakpoint | Format-Table -AutoSize

1. Running the script until the first breakpoint is hit

& $FileName

1. Viewing the value of $J from the debug console

$J

1. Viewing the value of $K from the debug console

$K

1. Continuing script execution from the DBG prompt until the next breakpoint

continue

1. Continuing script execution until the execution of Get-CimInstance

continue

1. Continuing script execution until the end of the script

continue

## How it works...

In step 1, you create a script to allow you to examine PowerShell script debugging. In step 2, you save this file to the C: drive. These steps generate no output.

In step 3, you execute the script, which produces some output to the console like this:

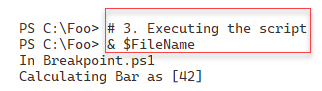


Figure 12.32: Executing the script

In step 4, you set a breakpoint in the script at a specific line. In step 5, you set another breakpoint whenever your script calls a particular command (Get-CimInstance). In step 6, you set a breakpoint to stop whenever you write to a specific variable ($M). Setting these three breakpoints produces no output (since you piped the command output to Out-Null).

In step 7, you view the breakpoints you have set thus far in the current PowerShell session. The output looks like this:

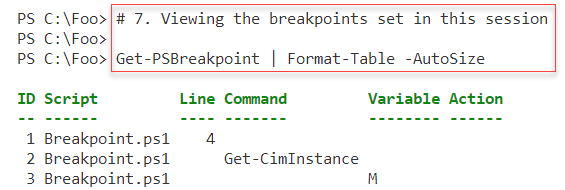


Figure 12.33: Viewing the breakpoints

Having set three break points in the script, in step 8, you run the script. PowerShell stops execution when it reaches the first breakpoint (in line 4 of the script file), with output like this:

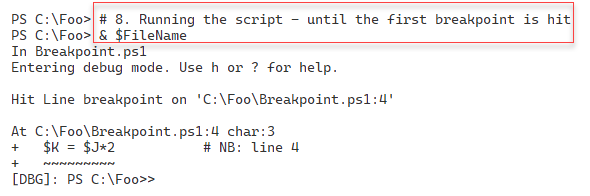


Figure 12.34: Running the script until PowerShell hits the first breakpoint

From the DBG prompt, you can enter any PowerShell command, for example, to view the value of $J, which you do in step 9. This step produces the following console output:



Figure 12.35: Viewing the value of the $J variable

In step 10, you attempt to view the value of the $K variable. Since PowerShell stopped execution *before* the line executes and before your script can create and write a value to this variable, this step displays no output, as you can see here:

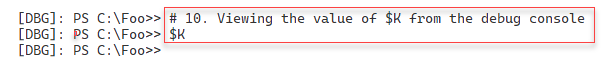


Figure 12.36: Attempting to view the value of the $K variable

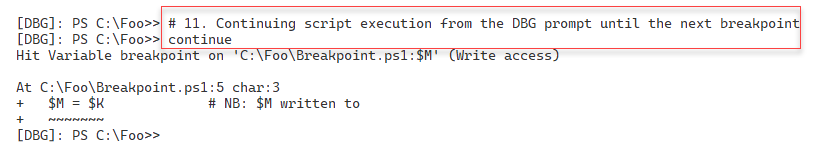
To continue the script execution, in step 11, you type continue to have PowerShell continue running the script until it hits the next breakpoint. The console output looks like this:

Figure 12.37: Running the script until PowerShell hits the second breakpoint

As in the previous step, at the debug console, you can examine the script's actions thus far. Then, you can continue script execution, in step 12, by typing continue, which produces output like this:

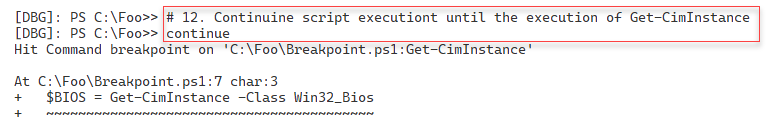


Figure 12.38: Running the script until PowerShell hits the third and final breakpoint

In step 13, you continue running the script, which now completes, with output like this:

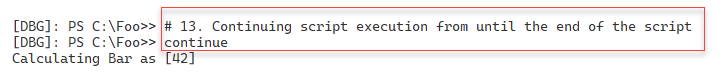


Figure 12.39: Running the script to completion

## There's more...

In step 4, you set a line breakpoint, instructing PowerShell to stop execution once it reaches a specific line (and column) in our script. In step 5, you set a command breakpoint, telling PowerShell to break whenever the script invokes a particular command, in this case, Get-CimInstance. In step 6, you set a variable breakpoint – you tell PowerShell to stop whenever your script reads from or writes to a specific variable.

When debugging, whenever you reach a breakpoint, you should check to see if the value of variables is what you expect to see. In step 8, you run this instrumented script – which breaks at the first breakpoint. From the debug console, as you see in step 9, you can view the value of any variable, such as $J. In step 10, you also view the value of $K. Since PowerShell has not yet processed the assignment, this variable has no value.

In step 11, you continue execution until PowerShell hits the second breakpoint. As before, you could examine the values of key variables.

After continuing again, your script hits the final breakpoint just before PowerShell invokes Get-CimInstance and assigns the output to $BIOS. From the debug console, you could invoke the cmdlet to check what the result would be.

Finally, in step 13, you continue to complete the execution of the script. Note that you now see the normal PowerShell prompt.

If you have an especially complex script, you could set the breakpoints using another script similar to this recipe. You would use Set-PSBreakpoint whenever your script executes important commands, writes to specific variables, or reaches a key line in the script. You could later use that script file when you perform subsequent debugging.