14

Troubleshooting With PowerShell

In this chapter, we cover the following recipes:

* Checking network connectivity using Get-NetView
* Using the PowerShell Script Analyzer
* Using the Best Practices Analyzer
* Network troubleshooting
* Exploring PowerShell script debugging

# Introduction

You can think about debugging as being the art and science of removing bugs from your PowerShell scripts. You may find your script does not always do what you or your users want, both during the development of a script and later, after you have used the script in production. Troubleshooting is a process you go through to determine why your script is not doing what you want (and helps you resolve your issues with the script).

There are three broad classes of problems that you encounter:

* Syntax errors
* Logic errors

Runtime errors

Syntax errors are all too 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 until you resolve your syntax errors, your script cannot run successfully. 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.

Another way to reduce at least some typos or other 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 for you.

Logic errors are bits of code that do not do what you want or what you expect. There are a myriad number of 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 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 you run these services in the best possible way.

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 the network path from your user to the relevant servers is working as required. But you also need to ensure your networking configuration itself is correct.

Network issues, over and above basic connectivity, can be challenging to resolve since there are so many potential issues that might arise. Many issues are simple to diagnose and resolve. You can see this in the Network Troubleshooting recipe. For more challenging issues, you may need to delve deeper into the networking stack to gather more information. The Get-NetView module and cmdlet is useful in obtaining a huge amount of troubleshooting information that a network professional can use to resolve problems.

Finally, an important tool for finding script errors is PowerShell's script debugging features. As we see in the final recipe, using these features makes it easier to find and remove errors in your scripts.

# Checking network connectivity using Get-NetView

Get-NetView is a tool that collects details about your network environment which can help you troubleshoot network issues.

The Get-NetView module contains a single function, Get-NetView. When you run the command, it pulls together a huge range of network details and creates a zip file containing a wealth of details about your network. By default, Get-NetView creates this output on our desktop.

Get-NetView output includes the following details:

* Get-NetView metadata
* The host environment (including OS, hardware, domain, 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 useful to you. However, if there is an issue in your network, this information is going to help you do the troubleshooting.

## 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 installed version of Get-NetView

Get-Module -Name Get-NetView -ListAvailable

1. Importing the Get-NetView module

Import-Module -Name Get-NetView -Force

1. Creating new folder

$OF = 'C:\NetViewOutput'

New-Item -Path $OF -ItemType directory | Out-Null

1. Running Get-NetView

Get-NetView -OutputDirectory $OF

1. Viewing the output folder using Get-ChildItem

$OFF = Get-ChildItem $OF

$OFF

1. Viewing the output folder contents using Get-ChildItem

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

Get-ChildItem -Path $Results

1. Viewing 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:

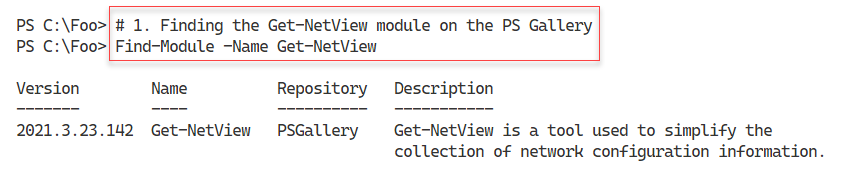


Figure 14.1: Finding the Get-NetView module

**Insert image B42024\_14\_01.png**

In step 2, you 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:

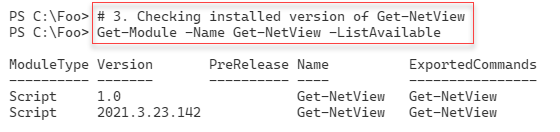


Figure 14.2: Checking the installed version(s) of Get-NetView

**Insert image B42024\_14\_02.png**

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 Get-NetView generates. These two steps produce no output.

In step 6, you run Get-NetView. At each step taken, the command logs some network information and outputs a running commentary. This command generates a lot of console output, a subset of which looks like this:

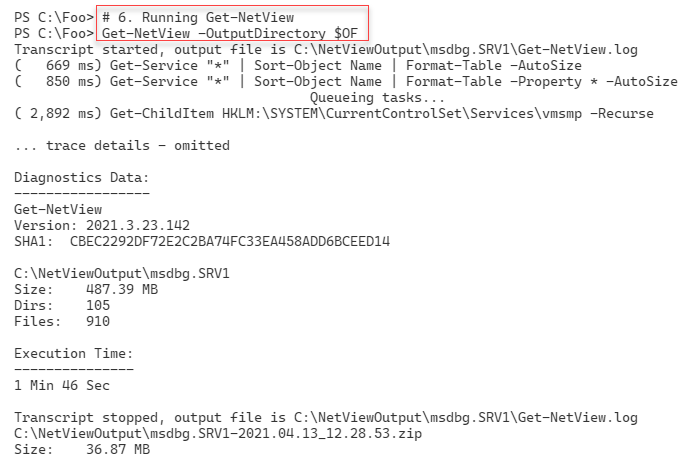


Figure 14.3: Running Get-NetView

**Insert image B42024\_14\_03.png**

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

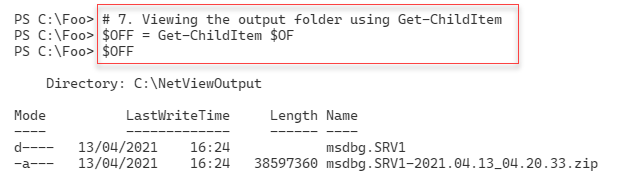


Figure 14.4: Viewing the Get-NetView output folder

**Insert image B42024\_14\_04.png**

In step 8, you view the detailed information created by Get-NetView, with output like this:

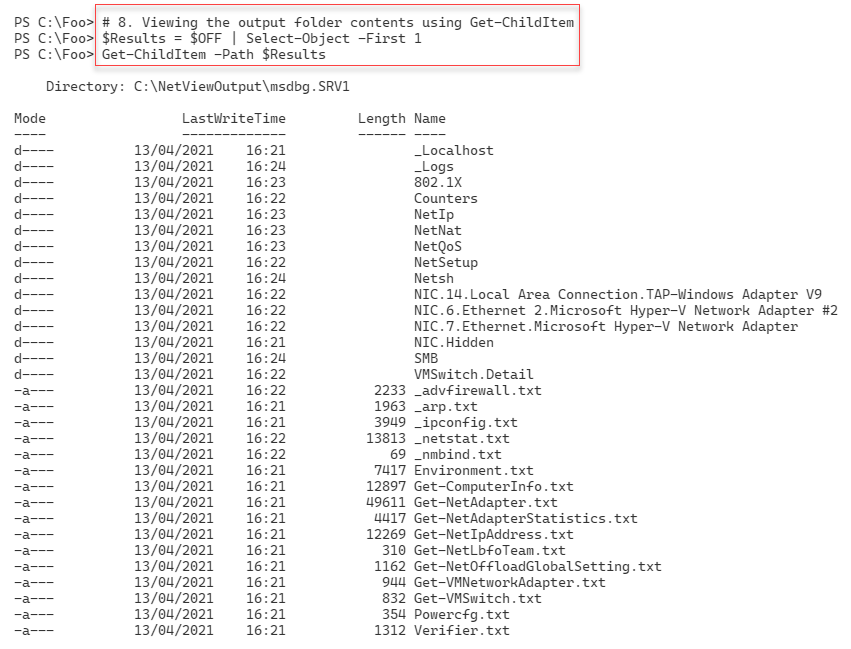


Figure 14.5: Viewing the Get-NetView output folder contents

**Insert image B42024\_14\_05.png**

In step 9, you examine the IP configuration generated by Get-NetView, which looks like this:

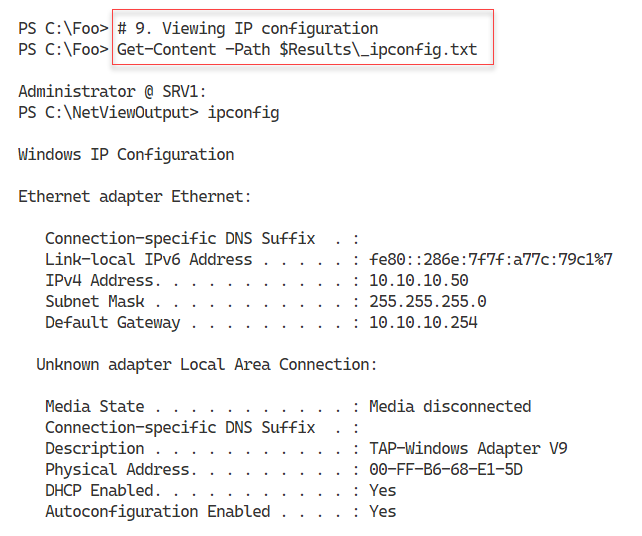


Figure 14.6: Viewing IP configuration

**Insert image B42024\_14\_06.png**

## There's more...

In step 3, you check the version(s) of the Get-NetView module on your system. In this case, you can see version 1.0, which shipped with the edition of Windows Server (running on SRV1) as well as a later version you obtained from the PowerShell Gallery. As always, you may see a later version of this module on the PowerShell Gallery. In this step, you install the module rather than updating it. You cannot upgrade the inbox versions of any module unless and until you install it first explicitly. Then, going forward, you can use Update-Module to obtain any updates.

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 you can send to a network technician for resolution.

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. Interestingly, the developers used ipconfig.exe to create this information without using the /all switch. This means you cannot see from this output the configured DNS server IP addresses.

# Using the PowerShell 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.

If you are using the VS Code editor to develop your code, you should know that the Script Analyzer is built-in to VS Code. So as you are developing your PowerShell script, VS code highlights any errors which 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 have numerous settings you can configure to tell you how the Script Analyzer should reformat your code.

## Getting ready

This recipe uses SRV1, a domain-joined Windows Server 2022 host.

## How to do it...

1. Discovering the PowerShell Script Analyzer module

Find-Module -Name PSScriptAnalyzer |

  Format-List Name, 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 more nicely

$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 the 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:

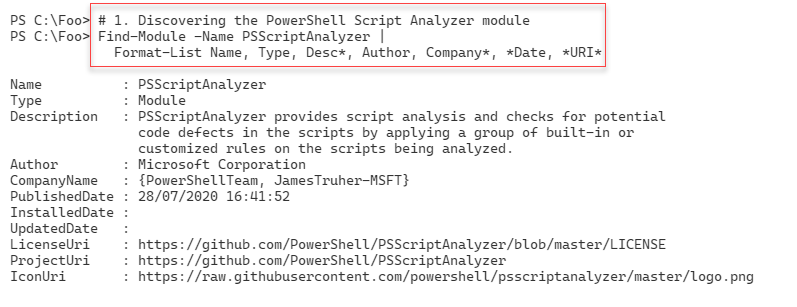


Figure 14.7: Finding the PowerShell Script Analyzer module

**Insert image B42024\_14\_07.png**

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

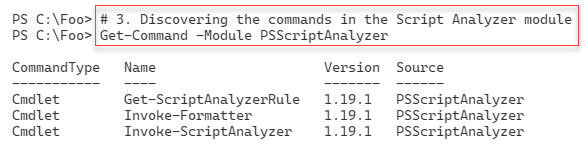


Figure 14.8: Getting the commands in the Script Analyzer module

**Insert image B42024\_14\_08.png**

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:

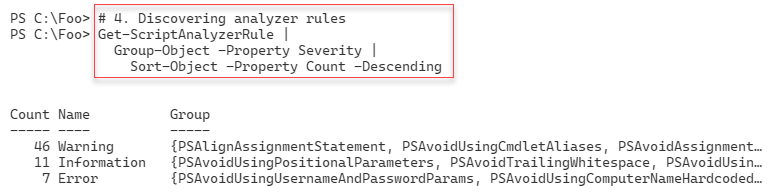


Figure 14.9: Examining Script Analyzer rules

**Insert image B42024\_14\_09.png**

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

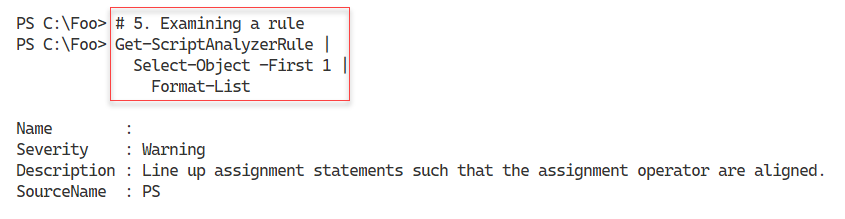


Figure 14.10: Examining a Script Analyzer rule

**Insert image B42024\_14\_10.png**

In step 6, which generates no console output, you create a script file with issues that Script Analyzer can detect and provide you with problems to resolve. In step 7, you check on the newly created script file, with output like this:

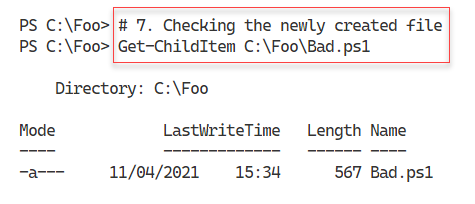


Figure 14.11: Checking the script file

**Insert image B42024\_14\_11.png**

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:

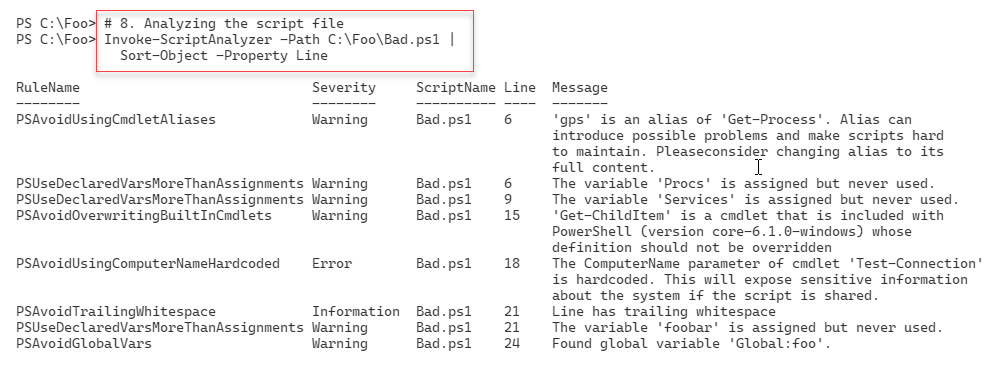


Figure 14.12: Analyzing the script

**Insert image B42024\_14\_12.png**

The Script Analyzer's secondary function is to reformat a script file to improve the script's layout. In step 9, you define a simple function that you later reformat. This step generates no console output.

Views on what constitutes a good code layout vary. The PS Script’s Rule Analyzer feature allows you to specify settings that govern how you wish the code to be formatted. In step 10, you specify some rule settings, which generates no output.

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

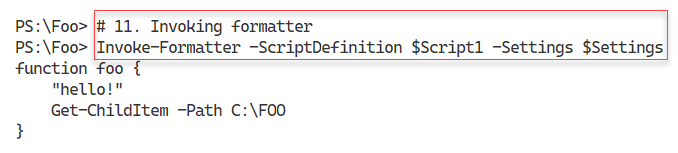


Figure 14.13: Invoking the script formatter

**Insert image B42024\_14\_13.png**

In step 12, you change the rule's value that places an open brace character on the same line or separate line as, say, the function name. The output of this step looks like this:

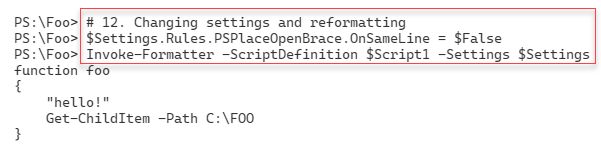


Figure 14.14: Changing settings and reformatting the script file

**Insert image B42024\_14\_14.png**

## There's more...

In step 1, you view details about the PSScriptAnalyzer module. Note that the author of this module was a long-time PowerShell team member, Jim Truher. Interestingly, Jim did some of the demonstrations the very first time Microsoft displayed Monad (as PowerShell was then named) at the PDC in the autumn of 2003.

In the final part of this recipe, you use the formatting feature to format a script file. The goal is to create easier to read code. That can be very useful for long production scripts. A consistent layout makes it easier to find issues as well as simplifying any later 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. You always have options. There are other formatting 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/1.19.1/Content/Settings%5CCodeFormatting.psd1.

# Using the 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 having two DHCP servers in a failover relationship means you can experience numerous issues in these core services and still deploy a reliable end-user service. While you may still need to do troubleshooting 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 how you should deploy a Windows infrastructure. Some product teams, such as Exchange, publish extensive guidance and have developed a self-contained tool.

The Windows Server Best Practices Analyzer (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 environment. The AD team have built a BPA model for AD, 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.

## 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. 1. Creating a remoting session to Windows PowerShell on DC1

$BPAS = New-PSSession -ComputerName DC1

2. Discovering the BPA module on DC1

$SB1 = {

  Get-Module -Name BestPractices -List |

    Format-Table -AutoSize

}

Invoke-Command -Session $BPAS -ScriptBlock $SB1

1. Discovering the commands in the BPA module

$SB2 = {

    Get-Command -Module BestPractices  |

      Format-Table -AutoSize

}

Invoke-Command -Session $BPAS -ScriptBlock $SB2

1. Discovering all available BPA models on DC1

$SB3 = {

  Get-BPAModel  |

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

}

Invoke-Command -Session $BPAS -ScriptBlock $SB3

1. Running the BPA DS model on DC1

$SB4 = {

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

    Format-Table -AutoSize

}

Invoke-Command -Session $BPAS -ScriptBlock $SB4

1. Getting BPA results from DC1

$SB5 = {

    Get-BpaResult -ModelID Microsoft/Windows/DirectoryServices  |

      Where-Object Resolution -ne $null|

        Format-List -Property Problem, Resolution

}

Invoke-Command -Session $BPAS -ScriptBlock $SB5

## How it works...

In step 1, you create a PowerShell remoting session with your DC, 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:

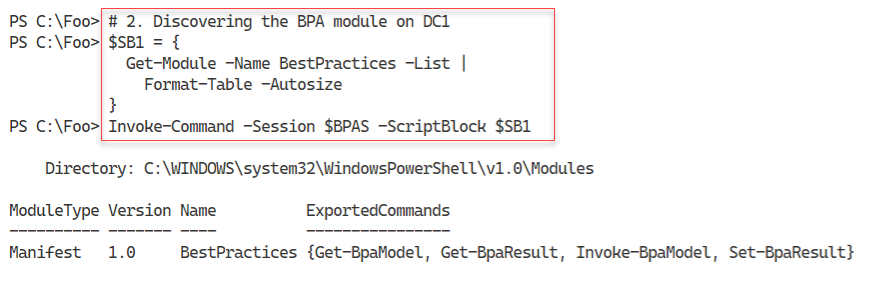


Figure 14.15: Viewing the Best Practices Analyzer module on DC1

**Insert image B42024\_14\_15.png**

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

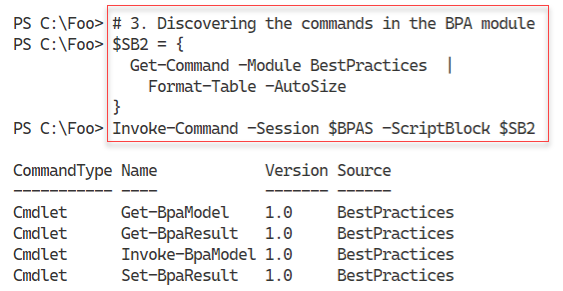


Figure 14.16: Discovering the commands inside the BPA module

**Insert image B42024\_14\_16.png**

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

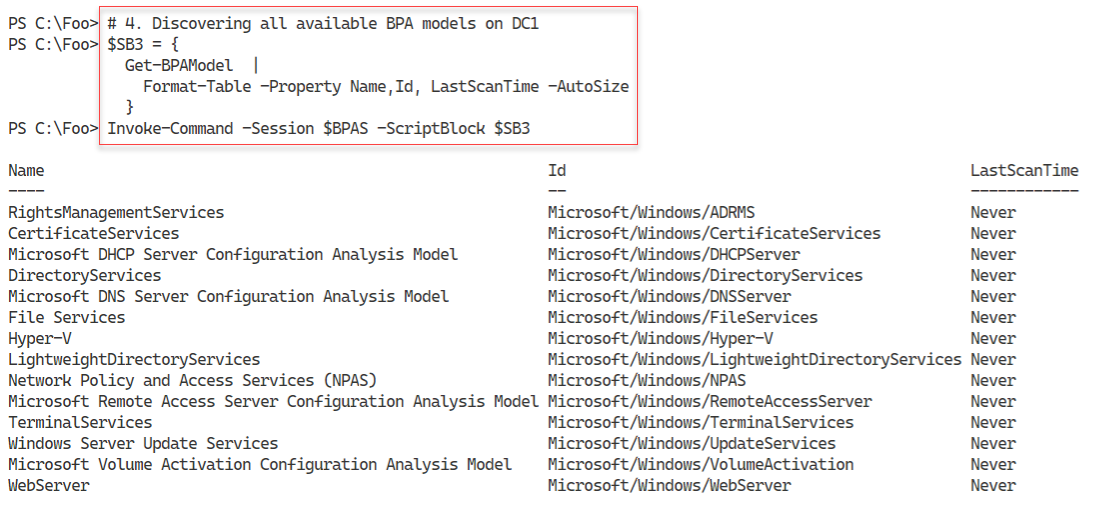


Figure 14.17: Discovering the BPA models available on DC1

**Insert image B42024\_14\_17.png**

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

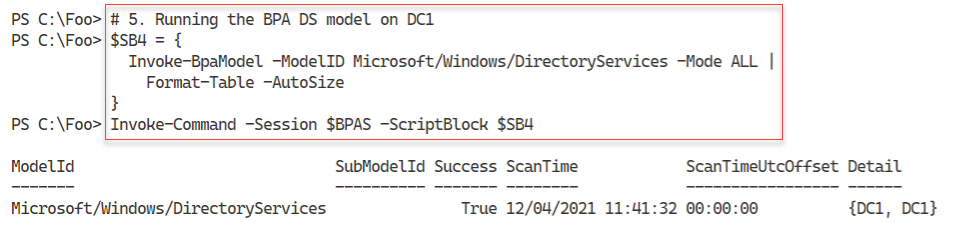


Figure 14.18: Invoking a BPA scan on DC1

**Insert image B42024\_14\_18.png**

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:

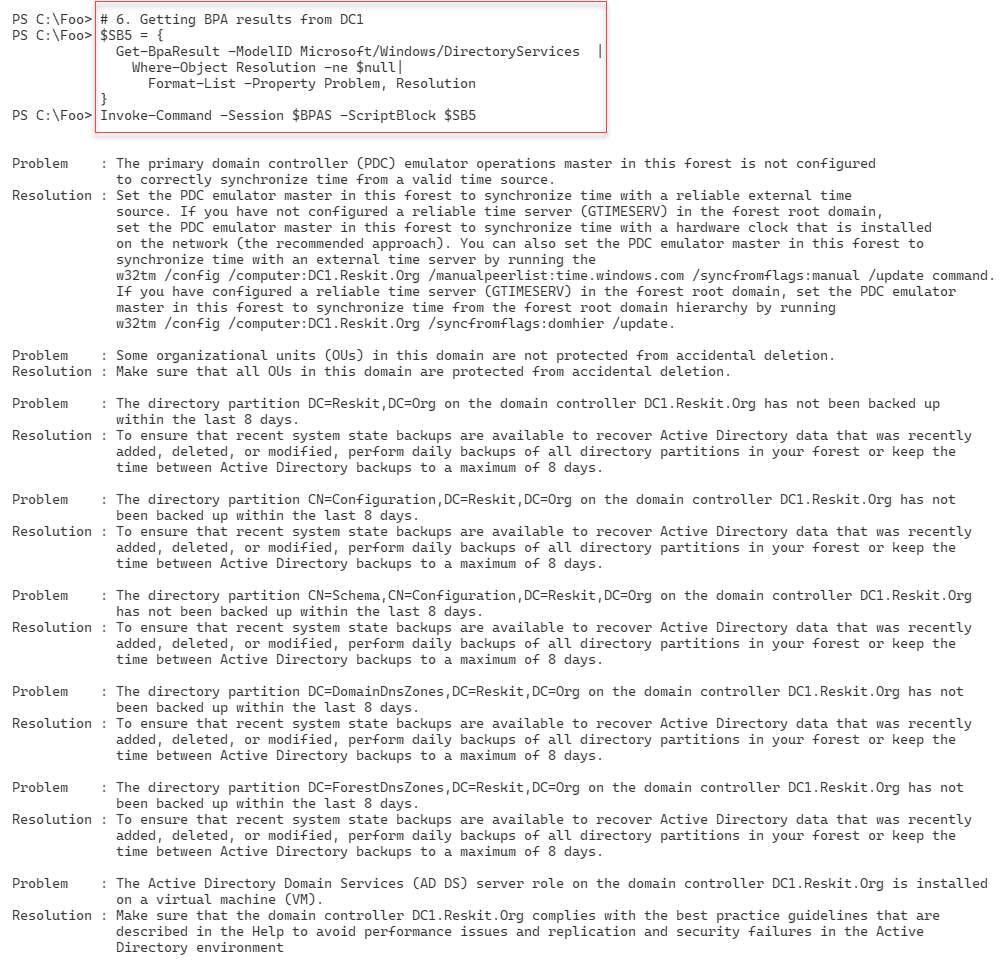


Figure 14.19: Obtaining BPA scan results

**Insert image B42024\_14\_19.png**

## There's more...

BPA results include details of successful and unsuccessful tests. The unsuccessful results, where BPA finds that your deployment does not implement a best practice, are the ones you most likely 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 the real-world time, possibly leading to issues later on. See https://blogs.msmvps.com/acefekay/tag/pdc-emulator-time-configuration/ for more information on how to configure your DCs with a reliable time source.
* You have not backed up your AD environment. Even with multiple DCs, it is a best practice to perform regular backups. 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 for the most part, 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. In a test environment, having a backup of your AD is not needed. 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.

# Network troubleshooting

In the recipe Checking network connectivity using Get-NetView, you used the Get-NetView command to gather a large amount of network-related information to diagnose and resolve network issues. For some issues, this level of detail is fundamental in helping you to resolve network issues. But in some cases, it can be overkill. Often some simpler steps may help you resolve your more common issues or point you towards 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 theory is that any network problem is due to DNS (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 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 both port 389 (LDAP) and 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

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

## How to do it...

1. 1. Getting the DNS name of this host

$DNSDomain = $Env:USERDNSDOMAIN

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

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 default gateway

$NIC    = Get-NetIPConfiguration -InterfaceAlias Ethernet

$DG     = $NIC.IPv4DefaultGateway.NextHop

$TestDG = Test-NetConnection $DG

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

"Default Gateway for [$($NIC.Interfacealias) is [$DG] - $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 553

$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. This step creates no output.

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:

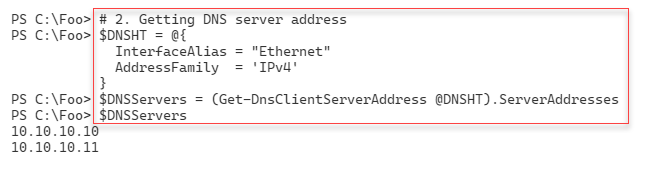


Figure 14.20: Obtaining IP addresses configured on SRV1

**Insert image B42024\_14\_20.png**

In step 3, you check whether the configured DNS servers are available, with output like this:

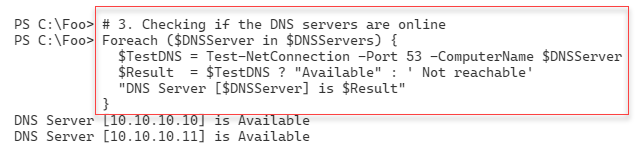


Figure 14.21: Checking reachability of each configured DNS server

**Insert image B42024\_14\_21.png**

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

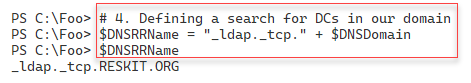


Figure 14.22: Defining an RR name for DC SRV records

**Insert image B42024\_14\_22.png**

In step 5, you retrieve the SRV resource records for DCs in your 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:

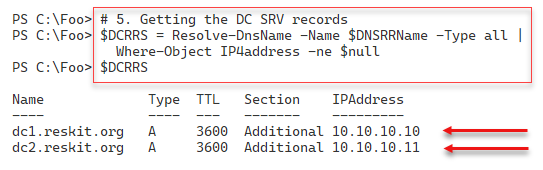


Figure 14.23: Querying for DNS RRs for DCs

**Insert image B42024\_14\_23.png**

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

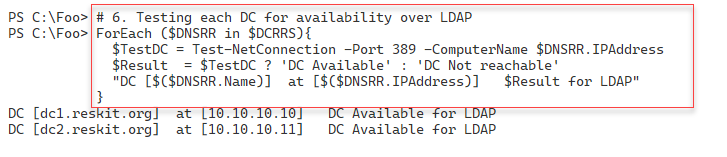


Figure 14.24: Testing LDAP connectivity to domain controllers

**Insert image B42024\_14\_24.png**

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

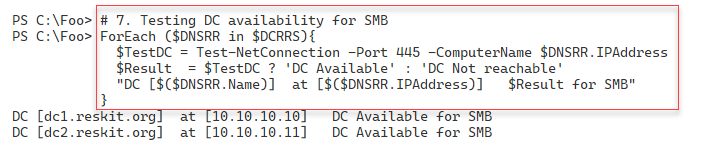


Figure 14.25: Testing SMB connectivity to domain controllers

**Insert image B42024\_14\_25.png**

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

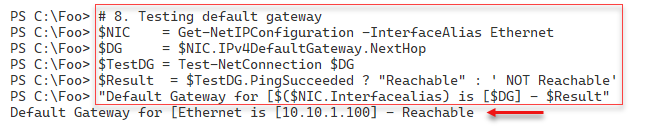


Figure 14.26: Testing the default gateway

**Insert image B42024\_14\_26.png**

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:

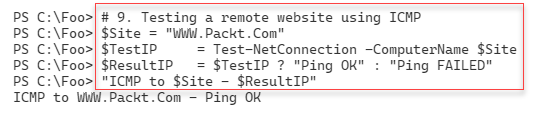


Figure 14.27: Pinging an Internet site

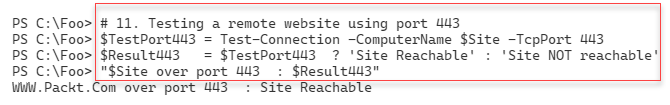
**Insert image B42024\_14\_27.png**

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 14.28: Testing connectivity over port 80

**Insert image B42024\_14\_28.png**

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 14.29: Testing connectivity over port 443

**Insert image B42024\_14\_29.png**

## There's more...

In step 1, you create a variable to hold the FQDN of the host. You may want to use this to ensure that your host has properly registered your host's FQDN in the 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 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, 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 likely to fail due to a default gateway issue.

In step 9, step 10, and step 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 messages output by those steps. These steps provide a good example of using this operator, which you would have read about in Chapter 2 in the Exploring new operators recipe.

# Exploring PowerShell script debugging

PowerShell, both Windows PowerShell and PowerShell 7, contain some great debugging features. Using these features makes it easier to find and remove errors in your scripts. You can set breakpoints in a script – when you run your script, PowerShell stops execution at the breakpoint. For example, you can set a breakpoint to stop at a particular line, any time your script writes to a particular variable, or any time PowerShell calls a particular 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 so far and run additional commands 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. At that point, you could 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 and VS Code on this host.

## How to do it...

1. 1. Creating a script to debug

$SCR = @'

# 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

$ScrFile = 'C:\Foo\Breakpoint.ps1'

$SCR | Out-File -FilePath $ScrFile

1. Executing the script

& $ScrFile

1. Adding a breakpoint at a line in the script

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

    Out-Null

1. Adding a breakpoint on the script whenever the script uses a specific command

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

  Out-Null

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

Set-PSBreakpoint -Script $SCRFile -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

& $ScrFile

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 from the DBG prompt until the execution of Get-CimInstance

continue

1. Continuing script execution from the DBG prompt until the end

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 create no output.

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

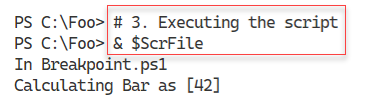


Figure 14.30: Executing the script

**Insert image B42024\_14\_30.png**

In step 4, you set a breakpoint in the script at a specific line. In step 5, you set another breakpoint, this time whenever your script calls a specific 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:

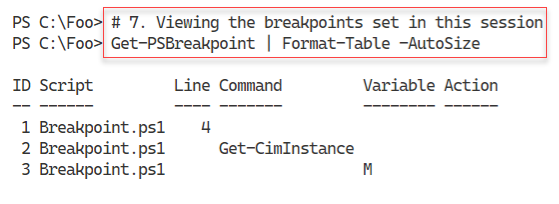


Figure 14.31: Viewing breakpoints

**Insert image B42024\_14\_31.png**

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), with output like this:

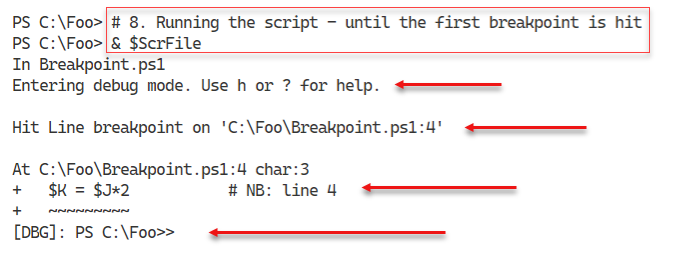


Figure 14.32: Running the script until PowerShell hits the first breakpoint

**Insert image B42024\_14\_32.png**

From the DBG prompt, you can enter any PowerShell command, for example, to view the value of $J, which you do in step 9. It looks like this:

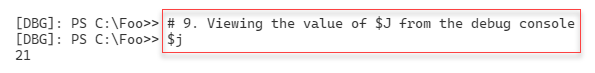


Figure 14.33: Viewing the value of the $J variable

**Insert image B42024\_14\_33.png**

In step 10, you view the value of the $K variable. Since PowerShell stopped execution before it could write a value to this variable, this step displays no value, as you can see here:

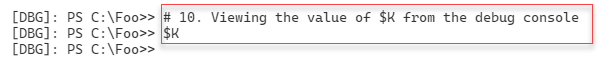


Figure 14.34: Viewing the value of the $K variable

**Insert image B42024\_14\_34.png**

To continue the execution of the script, in step 11, you type continue, which generates output like this:

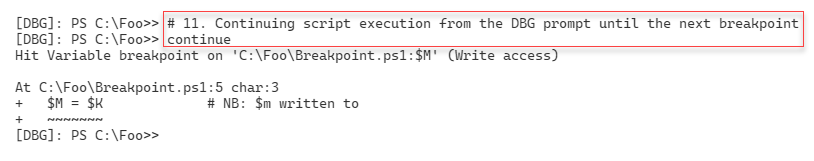


Figure 14.35: Running the script until PowerShell hits the second breakpoint

**Insert image B42024\_14\_35.png**

As in the previous step, you can examine the script's actions thus far. You continue the script, in step 12, by typing continue like this:

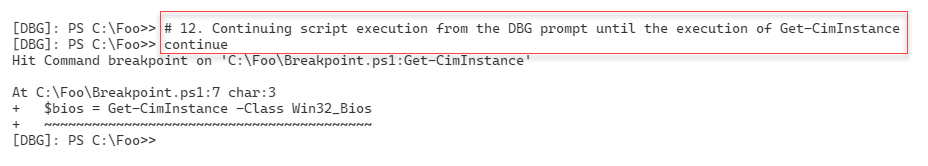


Figure 14.36: Running the script until PowerShell hits the third and final breakpoint

**Insert image B42024\_14\_36.png**

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

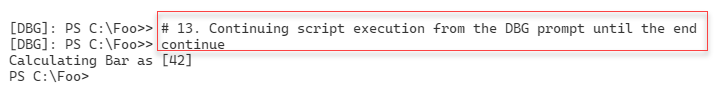


Figure 14.37: Running the script until the end of the script

**Insert image B42024\_14\_37.png**

## 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 specific 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.

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. When debugging, you have to ask yourself if the value you see is the value you expected to see. In step 10, you also view the value of $K. Since PowerShell has not yet processed the assignment stage, 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 output 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 debugging, after making changes to the underlying script.