8

Implementing Enterprise Security

In this chapter, we cover the following recipes:

* Implementing Just Enough Administration (JEA)
* Examining the Applications and Services logs
* Discovering logon events in event log
* Deploying PowerShell group policies
* Using PowerShell script block logging Configuring AD password policies
* Managing Windows Defender antivirus

# Introduction

Security within every organization is vital, and with the near-constant threats, more so now than ever. With today's near constant threats from any number of attackers, , you need to ensure every aspect of your organization is secure, from physical security to the security of your network and computer infrastructure.

Since the earliest times, security-savvy folks have preached the gospel of security in depth. Having as many layers as possible and realistic is just a good thing. As the theory goes – the bad guys have to defeat all your layers to defeat you, while you only need to hold one to stay safe.

PowerShell is a powerful tool for IT professionals wanting to be secure and stay secure. There is so much you can do with PowerShell to help your organization deploy excellent security over your network and computer infrastructure. In this chapter, we look at several ways to use PowerShell to improve your Windows infrastructure's security.

Just Enough Administration is a feature that enable you to implement fine-grained administration, giving users just enough power to enable them to do their job and not more. A core objective of JEA is to reduce the number of users who are members of very high privilege groups, including the local Administrators, Domain Admins, and Enterprise Admins groups.

In Windows, just about any component that does anything logs information to Windows's Event Logs. These include the classic logs (first implemented with Windows NT 3.1), plus the Application and Services logs Microsoft added to Windows Vista. Theyprovide a massive amount of information to help you manage your systems. One particular event that can be of interest is Logon events – who logged on and when. You can use this information to track unusual or suspicious logons.

You can manage certain aspects of PowerShell 7, like Windows PowerShell, using group policy or manually setting policy registry keys. With attackers increasingly using file-less PowerShell attacks, script block logging is one way of detecting suspicious behaviour. You can use these event log entries for active detection by deploying a security information and event management (SIEM) tool, such as Solar Windows Security Event Manager, or RSA NetWitness. Or you can store the events for manual review.

A critical security consideration for any sized organization is your password policy. You have a considerable amount of flexibility over your Windows password policiesWindows 10 and Windows Server 2022 have a default password policy which you can change. You can set a default domain assword policy if you want longer or shorter passwords, complex or non-complex passwords. For those cases where you wish to have a different password policy for specific users, you can use AD's fine-grained password feature that enable you to set a password policy for an OU.

Windows Server 2022 and Windows 10 come with a built-in antivirus and antimalware product, Microsoft Defender Antivirus or MDA. This was formerly just Microsoft Defender. MDA is part of a more extensive suite of products under the umbrella name of Microsoft Defender for Endpoint. See https://www.microsoft.com/microsoft-365/security/endpoint-defender for more information. Windows 10 and Windows Server come with a Defender module to help you manage Defender on a server.

# Implementing Just Enough Administration (JEA)

Just Enough Administration, also known as JEA, is a security framework providing you with the ability to implement fine-grained administrative delegation. With JEA, you enable a user to have just enough administrative power to do their job, and no more. JEA is a more secure alternative to just adding users to the Domain Administrator or Enterprise Administrator groups.

With JEA, you could, for example, enable a junior administrator the rights to access your domain controllers to administer the DNS service on the DC. JEA allows you to constrain what the user can do on the protected server. For example, you could allow the user to stop and start the DNS service (using Stop-Service and Start-Service) but no other services.

JEA makes use of three objects:

* **JEA role capabilities file** (**.psrc**): This file defines a role in terms of its capabilities. You would configure the JEA role RKDnsAdmins to define a limited set of cmdlets that the role has access to on the Domain Controller, namely those related to administering DNS on a DC.
* **JEA session configuration file** (**.pssc**): This file defines who can access a PowerShell remoting session and what they can do within the session. You could allow anyone in the RKDnsAdmins domain security group to access the server using a JEA endpoint. The session configuration file defines the JEA session's actions by reference to the role capabilities file. A JEA-protected remoting session can only be used by certain people who can do whatever the role capabilities file dictates.
* **A PowerShell remoting endpoint**: Once you have the role capabilities and session configuration files created, you register the JEA endpoint to the server you are protecting with JEA.

Once the JEA endpoint is registered, a user who is a member of the domain security group, RKDnsAdmins, can use Invoke-Command or Enter-PsSession, specifying the remote server and the JEA-protected endpoint to access the protected server. Once inside the remoting session, the user can only do what the role capabilities file allows.

The following diagram shows the components of JEA:

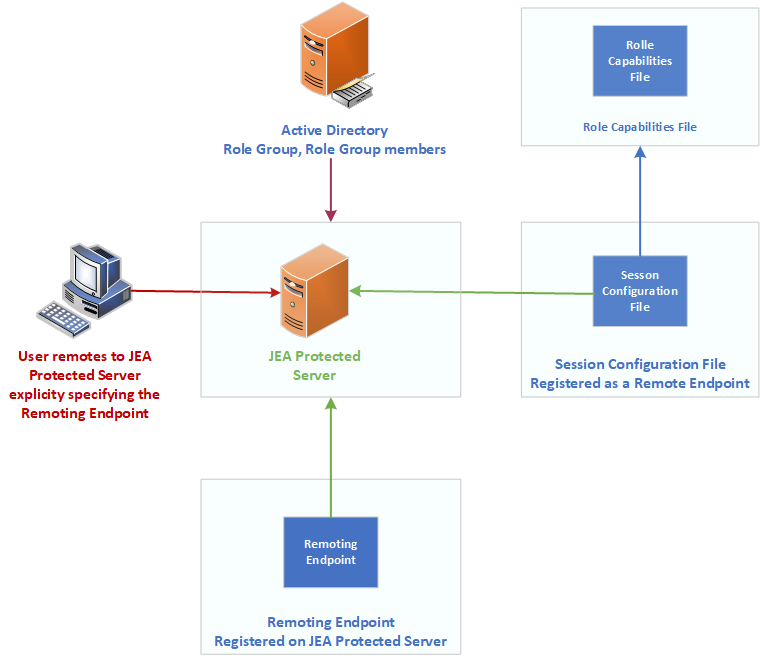


Figure 8.1: Components of JEA

1. Insert image B16762\_08\_01.png

## Getting ready

This recipe uses DC1, a domain controller in the Reskit.Org domain on which you set up JEA for inbound connections. You installed DC1 as a domain controller and configured users, groups, and OUs, both in Chapter 5, Exploring .NET. You run the first part of this recipe on DC1.

You would typically use a client computer to access the DC to manage DNS in production. For this recipe, adding an extra client host is replaced by using DC1 to test JEA without requiring an additional host. Of course, in production, you should test JEA on a client host.

## How to do it...

1. Creating a transcripts folder

New-Item -Path C:\JEATranscripts -ItemType Directory |

  Out-Null

1. Creating a role capabilities folder

$JEACF = "C:\JEACapabilities"

New-Item -Path $JEACF -ItemType Directory |

  Out-Null

1. Creating a JEA session configuration folder

$SCF = 'C:\JEASessionConfiguration'

New-Item -Path $SCF -ItemType Directory |

  Out-Null

1. Creating DNSAdminsJEA as a global security group

$DNSGHT = @{

  Name          = 'DNSAdminsJEA'

  Description   = 'DNS Admins for JEA'

  GroupCategory = 'Security'

  GroupScope    = 'Global'

}

New-ADGroup @DNSGHT

Get-ADGroup -Identity 'DNSAdminsJEA' |

  Move-ADObject -TargetPath 'OU=IT, DC=Reskit, DC=Org'

1. Adding JerryG to the DNS Admins group

$ADGHT = @{

  Identity  = 'DNSAdminsJEA'

  Members   = 'JerryG'

}

Add-ADGroupMember @ADGHT

1. Creating a role capabilities file

$RCF = Join-Path -Path $JEACF -ChildPath "DnsAdmins.psrc"

$RCHT = @{

  Path            = $RCF

  Author          = 'Reskit Administration'

  CompanyName     = 'Reskit.Org'

  Description     = 'DnsAdminsJEA role capabilities'

  AliasDefinition = @{Name='gh';Value='Get-Help'}

  ModulesToImport = 'Microsoft.PowerShell.Core','DnsServer'

  VisibleCmdlets  = (@{ Name        = "Restart-Computer";

                        Parameters  = @{Name = "ComputerName"}

                        ValidateSet = 'DC1, DC2'},

                       'DNSSERVER\\*',

                     @{ Name        = "Stop-Service";

                        Parameters  = @{Name = "DNS"}},

                     @{ Name        = "Start-Service";

                        Parameters  = @{Name = "DNS"}}

                     )

  VisibleExternalCommands = ('C:\Windows\System32\whoami.exe',

                             'C:\Windows\System32\ipconfig.exe')

  VisibleFunctions = 'Get-HW'

  FunctionDefinitions = @{

    Name = 'Get-HW'

    Scriptblock = {'Hello JEA World'}}

}

New-PSRoleCapabilityFile @RCHT

1. Creating a JEA session configuration file

$P   = Join-Path -Path $SCF -ChildPath 'DnsAdmins.pssc'

$RDHT = @{

  'DnsAdminsJEA' =

      @{'RoleCapabilityFiles' =

        'C:\JEACapabilities\DnsAdmins.psrc'}

}

$PSCHT= @{

  Author              = 'DoctorDNS@Gmail.Com'

  Description         = 'Session Definition for DnsAdminsJEA'

  SessionType         = 'RestrictedRemoteServer'   # ie JEA!

  Path                = $P       # Role Capabilties file

  RunAsVirtualAccount = $true

  TranscriptDirectory = 'C:\JeaTranscripts'

  RoleDefinitions     = $RDHT     # tk role mapping

}

New-PSSessionConfigurationFile @PSCHT

1. Testing the session configuration file

Test-PSSessionConfigurationFile -Path $P

1. Enabling remoting on DC1

Enable-PSRemoting -Force |

  Out-Null

1. Registering the JEA session configuration remoting endpoint

$SCHT = @{

  Path  = $P

  Name  = 'DnsAdminsJEA'

  Force =  $true

}

Register-PSSessionConfiguration @SCHT

1. Viewing remoting endpoints

Get-PSSessionConfiguration  |

  Format-Table -Property NAME, PSVERSION, Run\*Account

1. Verifying what the user can do

$SCHT = @{

  ConfigurationName = 'DnsAdminsJEA'

  Username          = 'Reskit\JerryG'

}

Get-PSSessionCapability  @SCHT |

  Sort-Object -Property Module

1. Creating credentials for user JerryG

$U    = 'JerryG@Reskit.Org'

$P    = ConvertTo-SecureString 'Pa$$w0rd' -AsPlainText -Force

$Cred = [PSCredential]::New($U,$P)

1. Defining three script blocks and an invocation splatting hash table

$SB1   = {Get-Command}

$SB2   = {Get-HW}

$SB3   = {Get-Command -Name  '\*-DNSSERVER\*'}

$ICMHT = @{

  ComputerName      = 'DC1.Reskit.Org'

  Credential        = $Cred

  ConfigurationName = 'DnsAdminsJEA'

}

1. Getting commands available within the JEA session

Invoke-Command -ScriptBlock $SB1 @ICMHT |

  Sort-Object -Property Module |

    Select-Object -First 15

1. Invoking a JEA-defined function in a JEA session as JerryG

Invoke-Command -ScriptBlock $SB2 @ICMHT

1. Getting DNSServer commands available to JerryG

$C = Invoke-Command -ScriptBlock $SB3 @ICMHT

"$($C.Count) DNS commands available"

1. Examining the contents of the transcripts folder

Get-ChildItem -Path $PSCHT.TranscriptDirectory

1. Examining a transcript

Get-ChildItem -Path $PSCHT.TranscriptDirectory |

  Select-Object -First 1  |

    Get-Content

## How it works...

In step 1, you create a new folder which you use to hold JEA transcripts. In step 2, you create a folder to hold role capabilities files. In step 3, you create a folder to hold JEA session configuration files. These three steps produce no output.

In step 4, you create a global security group you use with JEA, and in step 5, you add the user JerryG to that group. Neither step produces output. By default, Windows creates this new group in the Users container in the AD, which may not be ideal. To ensure the account is in the right OU, you use Move-ADGroup to move it to the correct place.

In step 6, you create a role capabilities file and store it in the role capabilities folder you created in step 2. The role capabilities file contains metadata (including author and organization), aliases a user can access, modules to import in the JEA session, the visibile cmdlets, console commands, and functions that the JEA server can access. In short, the role capabilities file allows you to customize precisely what the JEA user can do in a JEA session. For more information on the JEA role capabilities and the role capabilities file, see <https://docs.microsoft.com/powershell/scripting/learn/remoting/jea/role-capabilities>.

The JEA role capabilities file contains, in this recipe, a new function, Get-HW. This pretty simple function shows how you can define additional functions for the JEA user to access, possibly related to their specific role.

In step 7, you create a JEA session configuration file, which you store in the folder you created in step 3. These two steps produce no output.

In step 8, you use the Test-PSSessionConfigurationFile command to test the session configuration file. The command produces the following output:

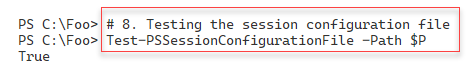


Figure 8.2: Testing the session configuration file

1. Insert image B16762\_08\_02.png

In step 9, you use the Enable-PSRemoting command to ensure that you have configured DC1 for WinRM remoting. This step creates no output.

In step 10, you complete the setup by registering a JEA session configuration remoting endpoint, producing no output.

In step 11, you use the Get-PSSessionConfiguration command to view the remoting endpoints available on DC1, with output like this:

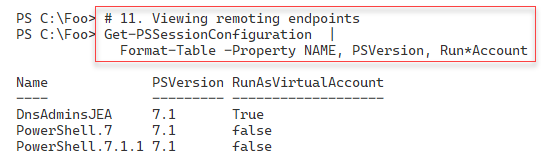


Figure 8.3: Viewing remoting endpoints available on DC1

1. Insert image B16762\_08\_03.png

In step 12, you verify the commands that the JEA session configuration allows the user, JerryG, to access within the DNSAdminsJEA remoting endpoint, which produces the following output:

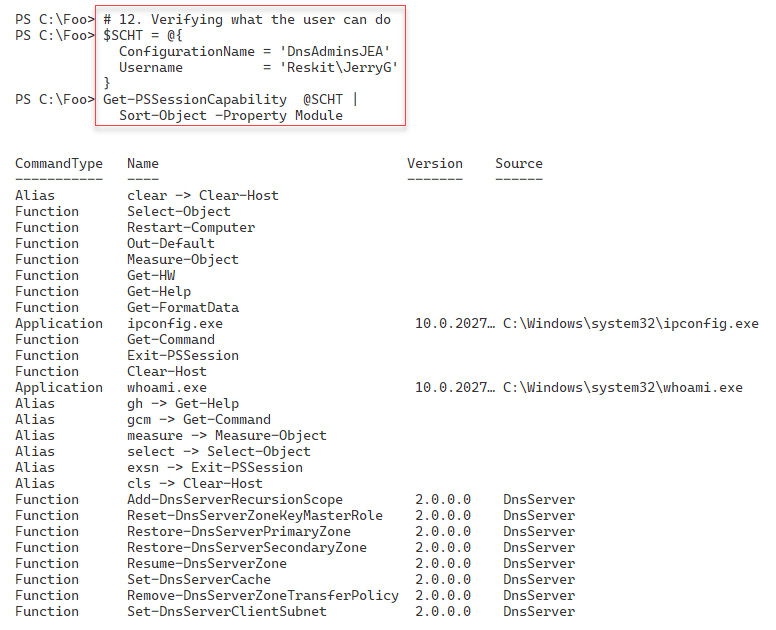


Figure 8.4: Verifying what the user JerryG can do

1. Insert image B16762\_08\_04.png

In step 13, you create a PowerShell credential object for JerryG. In step 14, you create a Windows credential object for JerryG@Reskit.Org. These two steps create no output.

In step 14, you create three script blocks and an invocation hash table for use in later steps, producing no output. In step 15, you invoke the $SB1 script block inside a JEA session, with output (truncated) like this:

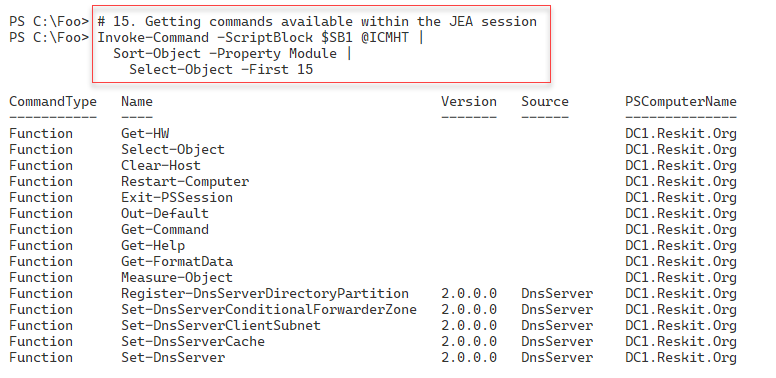


Figure 8.5: Getting commands available within the JEA session

1. Insert image B16762\_08\_05.png

In step 16, you invoke the $SB2 script block to call the Get-HW function defined in the role capabilities file.

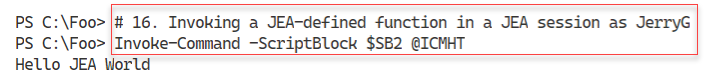


Figure 8.6: Invoking a JEA-defined function in a JEA session as JerryG

1. Insert image B16762\_08\_06.png

In step 17, you invoke the $SB3 script block which counts the number of commands available in the DNS server module which the user JerryG has permissions to use. The output is like this:

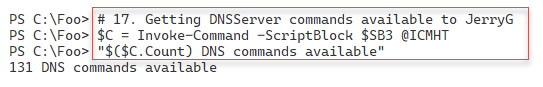


Figure 8.7: Counting number of DNS commands available to JerryG

1. Insert image B16762\_08\_07.png

When you set up JEA, you indicated JEA should create a transcript for each JEA session. In step 18, you examine the transcripts in the transcript folder, with output like this.

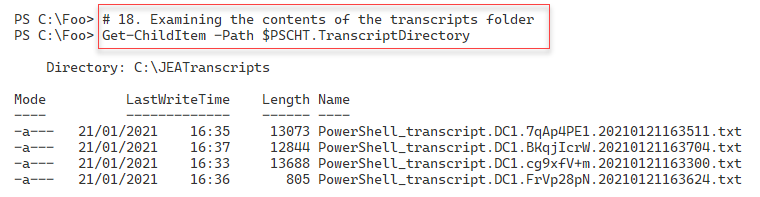


Figure 8.8: Examining the transcripts folder contents

1. Insert image B16762\_08\_08.png

In the final step, step 19, you examine the first transcript in the transcripts folder, with output (truncated for publishing) that should look like this:



1. Figure 8.9: Examining a transcriptInsert image B16762\_08\_09.png

## There's more...

In step 6, you create a JEA role capabilities file. In this file, you specify what actions a user can perform within a JEA session. This file specifies the commands a user can run, modules they can load, specific JEA session-specific functions they can access, and more. You can, for example, specify that the JEA user can run a cmdlet that takes parameters, but you only allow specific parameter values. For example, you could allow the user to start a service using Start-Service, but only the DNS service and not other services within the JEA session. If you are deploying JEA in an enterprise, you might have many servers and services managed with JEA sessions and a disparate user base. In such cases, you may wish to create a platform for your deployment to simplify creating the role definitions and your JEA deployment in general.

In step 9, you use the Enable-PSRemoting command. This command ensures you enable WinRM remoting and creates two standard PowerShell 7 remoting endpoints, in addition to the one you create in step 10.

In step 15, you run $SB1 inside a JEA session on DC1. This script invokes Get-Command to list all the commands available to any member of the DNSAdminsJEA group. The output is truncated in the figure to take up a bit less space for publishing. The complete output lists all the commands available.

In step 16, you use the Get-HW function defined in the JEA role capabilities file. This function only exists within the JEA session.

In step 18, you examine the transcripts in the JEA transcripts folder. Depending on what you have done so far, you may see a different number of transcripts. Each transcript represents one use of a JEA session. The transcriptcontains full details of the user's commands inside the session and documents that the user initiated the session and when. The transcript provides valuable information for subsequent analysis where needed.

In the final step, step 19, you examine one of the JEA session transcripts. In Figure 8.9, you see the transcript generated by step 15. You also need to manage the transcripts folder, including archiving or removing older transcripts. And if you are going to be implementing JEA widely, you may wish to develop some summary reporting based on each transcript's contents, including which users used any JEA session and when.

# Examining Applications and Services logs

Since the first version of Windows NT in 1993, anytime anything happens on a Windows, the component responsible writes details to an event log. In the earlier versions of Windows Server, there were four different Windows logs:

* Application - holds events related to software you have installed on the server
* Security - holds events related to the security of your server
* Setup - holds events related to Knowledge Base (KB) installation and events that occurred during installation
* System - holds events that relate to this system, such as system start and system shut down.

In addition to these logs, other applications and features can add additional logs. You can see the classic logs and the additiousing the Windows Powershell Get-Eventlog cmdlet.

With Window Vista, Microsoft made some significant improvements to the event logging features. A substantial improvement was adding the Applications and Services Logs, which contain over four hundred individual logs. These extra logs allow Windows components to write to application-specific logs rather than the System or Application classic event logs, making it easier to find the events on a given host. There are hundreds of these application and services logs that provide application-specific or service-specific event entries, but Windows does not enable all the logs by default. With PowerShell 7, you use Get-WinEvent to work with all of the event logs, including these newer ones.

In this recipe, you examine the logs and how to get log event details.

## Getting ready

You run this recipe on SRV1, a domain-joined Windows Server. You also need DC1, a domain controller in the Reskit.Org domain. You have installed PowerShell 7 and VS Code on each system.

## How to do it...

1. Registering PowerShell event log provider

& $PSHOME\RegisterManifest.ps1

1. Discovering classic event logs on SRV1

Get-EventLog -LogName \*

1. Discovering and measuring all event logs on this host

$Logs = Get-WinEvent -ListLog \*

"There are $($Logs.count) total event logs on SRV1"

1. Discovering and measuring all event logs on DC1

$SB1     = {Get-WinEvent -ListLog \*}

$LogsDC1 = Invoke-Command -ComputerName DC1 -ScriptBlock $SB1

"There are $($LogsDC1.count) total event logs on DC1"

1. Discovering log member details

$Logs | Get-Member

1. Measuring enabled logs on SRV1

$Logs |

  Where-Object IsEnabled |

    Measure-Object |

      Select-Object -Property Count

1. Measuring enabled logs on DC1

$LogsDC1 |

  Where-Object IsEnabled |

    Measure-Object |

      Select-Object -Property Count

1. Measuring enabled logs that have records on SRV1

$Logs |

  Where-Object IsEnabled |

    Where-Object Recordcount -gt 0 |

      Measure-Object |

        Select-Object -Property Count

1. Discovering PowerShell-related logs:

$Logs |

  Where-Object LogName -match 'powershell'

1. Examining PowerShellCore event log:

Get-Winevent -LogName 'PowerShellCore/Operational' |

  Select-Object -First 10

## How it works...

In step 1, you ensure that indows has had the PowerShell event log provider registered. This step creates no output.

In step 2, you use Get-EventLog to discover the classic event logs on SRV1, with output like this:

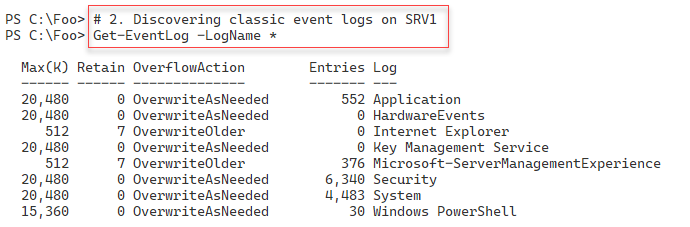


Figure 8.10: Discovering classic event logs on SRV1

1. Insert image B16762\_08\_10.png

In step 3, you use the Get-WinEvent cmdlet to discover all of the event logs on SRV1. The output looks like this:

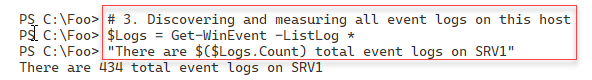


Figure 8.11: Discovering and counting all event logs on SRV1

1. Insert image B16762\_08\_11.png

In step 4, you discover and measure the number of event logs on the domain controller DC1, with output that looks like this:

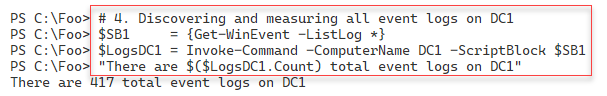


Figure 8.12: Discovering and counting all event logs on DC1

1. Insert image B16762\_08\_12.png

In step 5, you use Get-Member to discover the properties of event logs you can use when querying. The output looks like this:

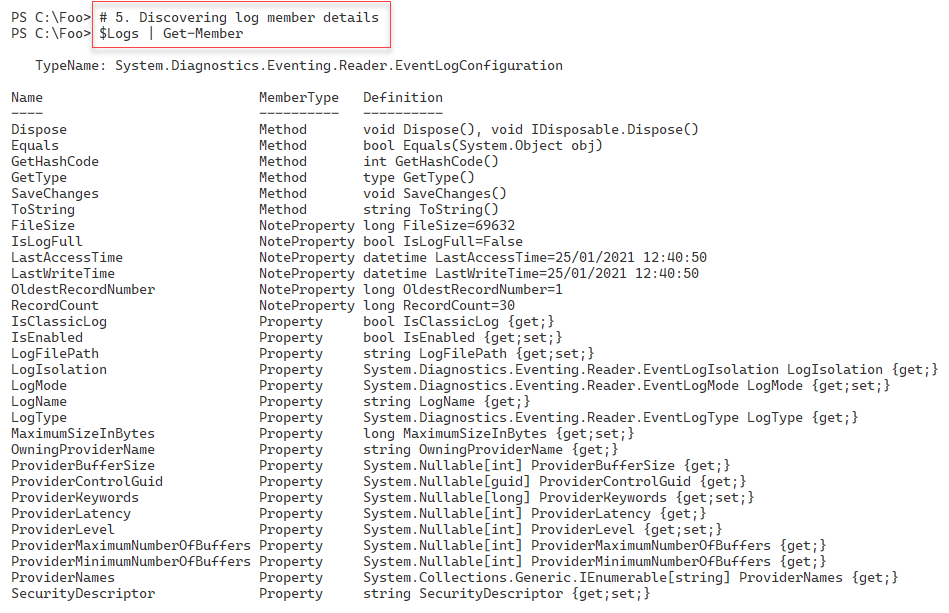


Figure 8.13: Discovering log member details

1. Insert image B16762\_08\_13.png

Windows does not enable all event logs by default. In step 6, you discover the enabled logs on SRV1. The output looks like this:

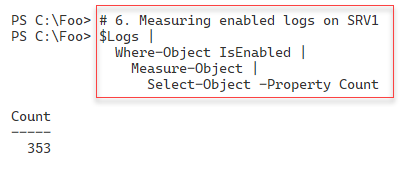


Figure 8.14: Measuring enabled logs on SRV1

1. Insert image B16762\_08\_14.png

In step 7, you measure the enabled event logs on DC1, with output like this:

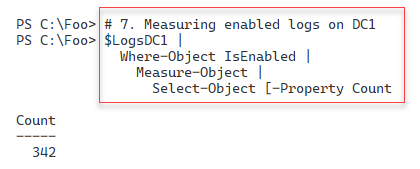
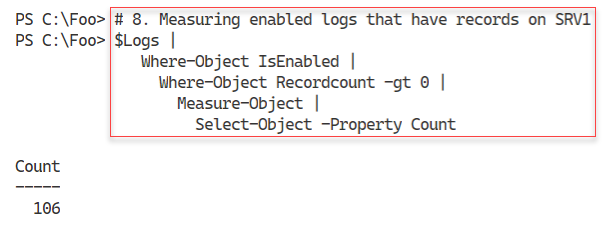
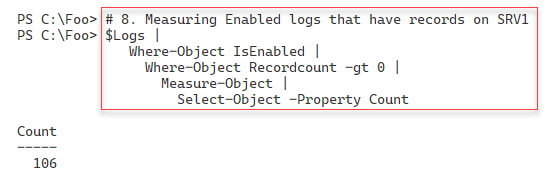


Figure 8.15: Measuring enabled logs on DC1

1. Insert image B16762\_08\_15.png

With step 8, you count the number of event logs you have enabled on SRV1 containing event log entries. The output looks like this:

1. Insert image B16762\_08\_16.png

In step 9, you discover which event logs could contain events for Windows PowerShell or PowerShell 7 (aka PowerShell Core). The output is as follows:

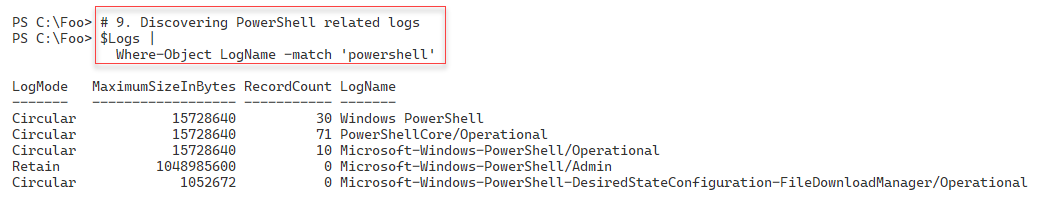


Figure 8.17: Discovering PowerShell related logs

1. Insert image B16762\_08\_17.png

In the final step in this recipe, step 10, you examing the PowerShellCore event log, with output like this:

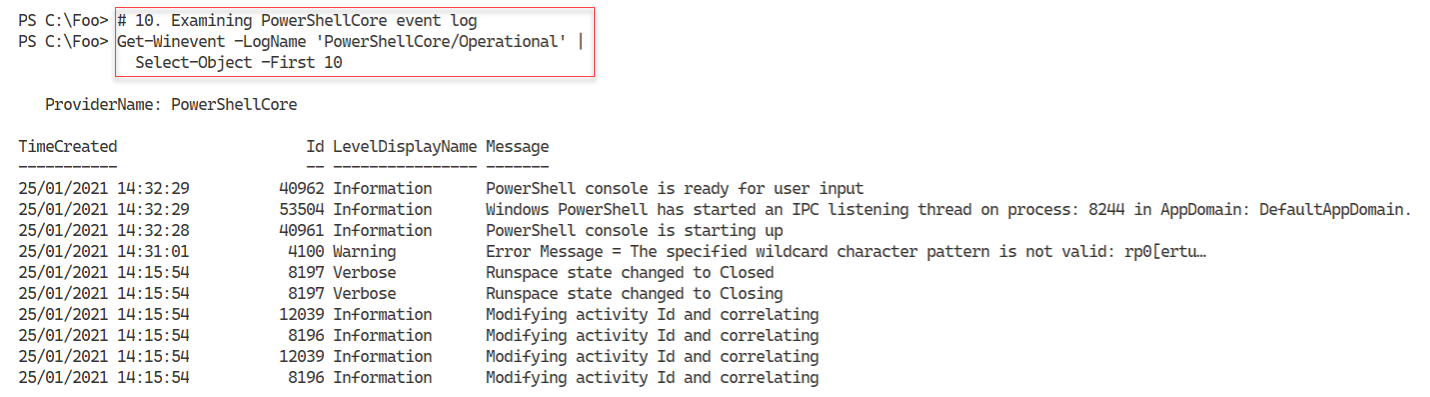


Figure 8.17b: Discovering PowerShell related logs

1. Insert image B16762\_08\_17b.png

## There's more...

In step 3 and step 4, you get a count of the number of event logs on SRV1 and DC1. As you can see, the number of logs differs. Different Windows features and applications can add additional event logs for your use. In step 6 and step 7, you also see the number of enabled logs on both systems. And with step 8, you see how many enabled logs (on SRV1) actually contain event log entries.

In step 9, you see the event logs on SRV1 related to Windows PowerShell and PowerShe, "Logging PowerShell Activity".

# Discovering logon events in event log

When you attempt to logon, whether you are successful or not, Windows logs the attempt. These log events can help you determine who logged into a computer and when.

In Windows, there are several different logon types. A logon type of 2 indicates a local console logon (that is, logging on to a physical host), while a logon type of 10 indicates logon over RDP. Other logon types include service logon (type 5), batch or scheduled task (type 4), and console unlock (type 7).

You can read more detail in this article: https://docs.microsoft.com/previous-versions/windows/it-pro/windows-server-2003/  
cc787567(v=ws.10). Note that this document is somewhat outdated and Microsoft has not updated it for later versions of Windows, although the information continues to be correct.

In this recipe, you use PowerShell to examine the Security event log and look at the logon events.

## Getting ready

You run this recipe on DC1, a domain controller in the Reskit.Org forest.

## How to do it...

1. Getting security log events:

$SecLog = Get-WinEvent -ListLog Security

"Security Event log entries:    [{0,10:N0}]" -f $Seclog.RecordCount

1. Getting all Windows Security log event details:

$SecEvents = Get-WinEvent -LogName Security

"Found $($SecEvents.count) security events on DC1"

1. Examining Security event log event members:

$SecEvents |

  Get-Member

1. Summarizing security events by event ID:

$SecEvents |

  Sort-Object -Property Id |

    Group-Object -Property ID |

      Sort-Object -Property Name |

        Format-Table -Property Name, Count

1. Getting all successful logon events on DC1:

$Logons = $SecEvents | Where-Object ID -eq 4624   # logon event

"Found $($Logons.Count) logon events on DC1"

1. Getting all failed logon events on DC1:

$FLogons = $SecEvents | Where-Object ID -eq 4625   # failed logon event

"Found $($FLogons.Count) failed logon events on DC1"

1. Creating a summary array of successful logon events:

$LogonEvents = @()

Foreach ($Logon in $Logons) {

  $XMLMSG = [xml] $Logon.ToXml()

  $Text = '#text'

  $HostName   = $XMLMSG.Event.EventData.data.$Text[1]

  $HostDomain = $XMLMSG.Event.EventData.data.$Text[2]

  $Account    = $XMLMSG.Event.EventData.data.$Text[5]

  $AcctDomain = $XMLMSG.Event.EventData.data.$Text[6]

  $LogonType  = $XMLMSG.Event.EventData.data.$Text[8]

  $LogonEvent = New-Object -Type PSCustomObject -Property @{

     Account   = "$AcctDomain\$Account"

     Host      = "$HostDomain\$Hostname"

     LogonType = $LogonType

     Time      = $Logon.TimeCreated

  }

  $LogonEvents += $logonEvent

}

1. Summarizing successful logon events on DC1:

$LogonEvents |

  Group-Object -Property LogonType |

    Sort-Object -Property Name |

      Format-Table Name, Count

1. Creating a summary array of failed logon events on DC1:

$FLogonEvents = @()

Foreach ($FLogon in $FLogons) {

  $XMLMSG = [xml] $FLogon.ToXml()

  $Text = '#text'

  $HostName   = $XMLMSG.Event.EventData.data.$Text[1]

  $HostDomain = $XMLMSG.Event.EventData.data.$Text[2]

  $Account    = $XMLMSG.Event.EventData.data.$Text[5]

  $AcctDomain = $XMLMSG.Event.EventData.data.$Text[6]

  $LogonType  = $XMLMSG.Event.EventData.data.$Text[8]

  $LogonEvent = New-Object -Type PSCustomObject -Property @{

     Account   = "$AcctDomain\$Account"

     Host      = "$HostDomain\$Hostname"

     LogonType = $LogonType

     Time      = $FLogon.TimeCreated

  }

  $FLogonEvents += $LogonEvent

}

1. Summarizing failed logon events on DC1:

$FLogonEvents |

  Group-Object -Property Account |

    Sort-Object -Property Name |

      Format-Table Name, Count

## How it works...

In step 1, you use the Get-WinEvent cmdlet to retrieve details about the Security log on DC1. Then you display the number of events in the log. The output looks like this:

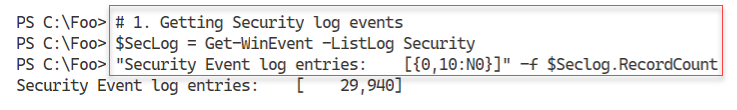


Figure 8.18: Getting security log events

1. Insert image B16762\_08\_18.png

In step 2, you use Get-WinEvent to retrieve all events from the Security log and display a count of the events returned, with output like this:

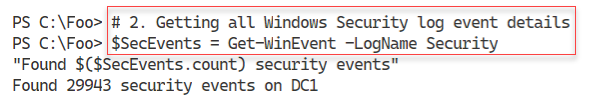


Figure 8.19: Getting all Windows Security log event details

1. Insert image B16762\_08\_19.png

The Get-Winevent cmdlet returns objects that contain individual event log entries. Each object is of the type System.Diagnostics.Eventing.Reader.EventLogRecord. In step 3, you view the members of this .NET object class, with output like this:

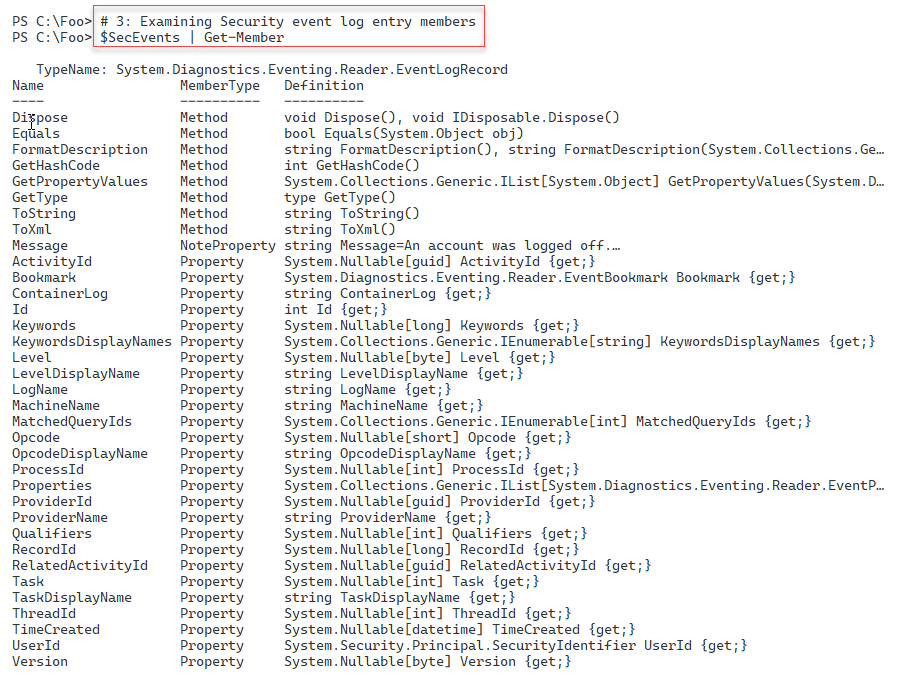


Figure 8.20: Examining Security event log entry members

1. Insert image B16762\_08\_20.png

Once you have retrieved the events in the security log, you can examine the different security event types, held in the ID field of each log record. In step 4, you view and count the different event IDs in the security log, which looks like this.

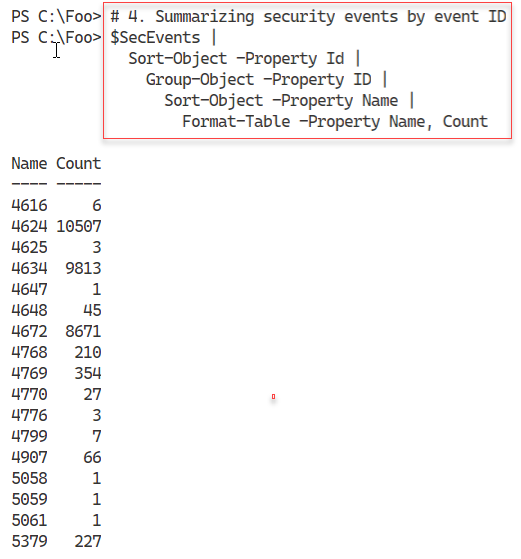


Figure 8.21: Summarizing security events by event ID

1. Insert image B16762\_08\_21.png

There are two related logon events you can track. Log entries with an event ID of 4624 represent successful logon events, while 4625 represents failed logons. In step 5, you get ALL the successful logon events, with output like this:

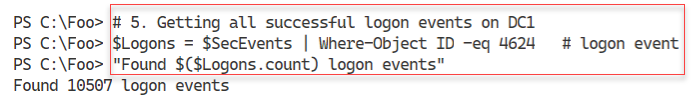


Figure 8.22: Getting all successful logon events on DC1

1. Insert image B16762\_08\_22.png

In step 6, you count the number of logon failures on DC1, which looks like this:

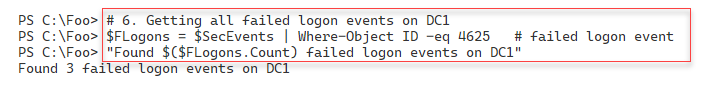


Figure 8.23: Getting all failed logon events on DC1

1. Insert image B16762\_08\_23.png

In step 7, you create a summary array of all the successful logons. This step produces no output. In step 8, you summarize the logon events, with output like this:

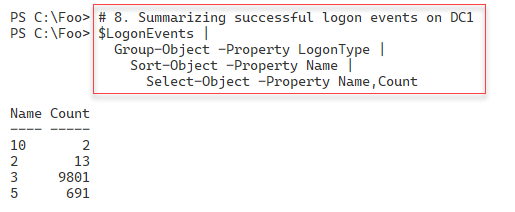


Figure 8.24: Summarizing successful logon events on DC1

1. Insert image B16762\_08\_24.png

In step 9, you summarize the failed logon events on DC1. You display the details of unsuccessful logins with step 10, which looks like this:

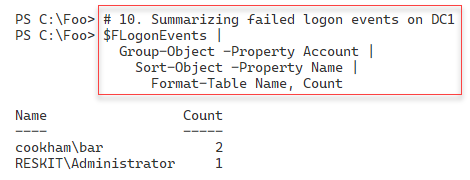


Figure 8.25: Summarizing failed logon events on DC1

1. Insert image B16762\_08\_25.png

## There's more...

In step 1, you retrieve a summary of the events in the security log and display the number of events in the log. In step 2, you retrieve and count the number of entries. As you can see in the figures above, the counts do not match. The event counts may differ since Windows is constantly logging additional events to the security log. The additional events are events generated by background tasks or services. This minor discrepancy is not unexpected and is harmless.

In step 3, you view the members of log event objects. You can discover more about the members of the class at https://docs.microsoft.com/dotnet/api/system.diagnostics.eventing.reader.eventlogrecord.

In step 6, you obtain unsuccessful logon events. To obtain unsuccessful logons, you need to ensure you have attempted to log on to DC1 but with invalid credentials. As you see in the output of step 10 in Figure 8.25, there were two users involved with the three unsuccessful logon attempts on DC1. Depending on which user you have attempted to login to this server (and failed), the results you see in this step may differ from the above figure.

# Deploying PowerShell group policies

Group policies are groups of policies you can deploy that control a user or computer environment. The policies define what a given user can and cannot do on a given Windows computer. For example, you can create a Group Policy Object (GPO) to set policies that define what screen saver to use, allow the user to see the Control Panel, or specify a default PowerShell execution policy. There are over 2500 individual settings that you can deploy.

After you create a GPO and specify the policies to deploy, you can apply it to any Organizational Unit (OU) in your domain. An OU is a container object within the AD that can contain both other OUs and leaf objects such as AD user, computer, or group objects. You use OUs to support both deployment of GPOs and delegation of AD administration.

You can apply a GPO to an OU, but also to the domain or to an AD site. Additionally, you can specify whether policies within a given GPO are to apply to users, computers, or both. GPOs provide you with considerable flexibility in how you restrict what users can do on a workstation or a server.

With Windows PowerShell 5.1, Microsoft includes a set of five Group Policy settings. The PowerShell team has extended the policies you can use in PowerShell 7. By default, the installation of PowerShell 7, even on a DC, does not install the necessary GPO administrative template files.

In the PowerShell home folder ($PSHOME), you can find the policy template files and a script to install them. After installing PowerShell on your domain controller, you run the installation script in the $PSHOME folder and install the policy definitions. You either do this on all DCs, or to the central policy store if you use one.

For some details on PowerShell 7's group policies, see <https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_group_policy_settings?view=powershell-7.1>.

In this recipe, you discover the files necessary to add PowerShell 7 GPO support, run the installer, then create a GPO to deploy a set of PowerShell related policies.

## Getting ready

You run this recipe on DC1 after you install PowerShell 7 and Visual Studio. DC1 is a domain controller in the Reskit.Org domain that you have used in earlier chapters.

## How to do it...

1. Discovering the GPO-related files

Get-ChildItem -Path $PSHOME -Filter \*Core\*Policy\*

1. Installing the PowerShell 7 group policy files

$LOC = 'C:\Program Files\PowerShell\7\' +         # $PSHome

       'InstallPSCorePolicyDefinitions.ps1'       # Script

& $LOC -VERBOSE

1. Creating and displaying a new GPO  for the IT group

$PshGPO = New-GPO -Name 'PowerShell GPO for IT'

1. Enabling module logging

$GPOKEY1 =

  'HKCU\Software\Policies\Microsoft\PowerShellCore\ModuleLogging'

$GPOHT1 = @{

  DisplayName    = $PshGPO.DisplayName

  Key            = $GPOKEY1

  Type           = [Microsoft.Win32.RegistryValueKind]::DWord

  ValueName      = 'EnableModuleLogging'

  Value          = 1

}

Set-GPRegistryValue @GPOHT1 | Out-Null

1. Configuring module names to log

$GPOHT2 = @{

  DisplayName    = $PshGPO.DisplayName

  Key            = "$GPOKEY1\ModuleNames"

  Type           = [Microsoft.Win32.RegistryValueKind]::String

  ValueName      = 'ITModule1', 'ITModule2'

  Value          = 'ITModule1', 'ITModule2'

 }

Set-GPRegistryValue @GPOHT2 | Out-Null

1. Enabling script block logging

$GPOKey3 =

  'HKCU\Software\Policies\Microsoft\PowerShellCore\ScriptBlockLogging'

$GPOHT3  = @{

    DisplayName    = $PshGPO.DisplayName

    Key            = $GPOKEY3

    Type           = [Microsoft.Win32.RegistryValueKind]::DWord

    ValueName      = 'EnableScriptBlockLogging'

    Value          = 1

   }

Set-GPRegistryValue @GPOHT3 | Out-Null

1. Enabling Unrestricted Execution Policy

$GPOKey4 =

  'HKCU\Software\Policies\Microsoft\PowerShellCore'

# create the key value to enable

$GPOHT4 =  @{

    DisplayName    = $PshGPO.DisplayName

    Key            = $GPOKEY4

    Type           = [Microsoft.Win32.RegistryValueKind]::DWord

    ValueName      = 'EnableScripts'

    Value          = 1

  }

  Set-GPRegistryValue @GPOHT4 | Out-Null

# Set the default execution policy

$GPOHT4 = @{

  DisplayName    = $PshGPO.DisplayName

  Key            = "$GPOKEY4"

  Type           = [Microsoft.Win32.RegistryValueKind]::String

  ValueName      = 'ExecutionPolicy'

  Value          = 'Unrestricted'

}

Set-GPRegistryValue @GPOHT4

1. Assigning GPO to IT OU:

$Target = "OU=IT, DC=Reskit, DC=Org"

New-GPLink -DisplayName $PshGPO.Displayname -Target $Target |

   Out-Null

1. Creating an RSOP report:

$RSOPHT = @{

  ReportType = 'HTML'

  Path       = 'C:\Foo\GPOReport.Html'

  User       = 'Reskit\JerryG'

}

Get-GPResultantSetOfPolicy @RSOPHT

& $RSOPHT.Path

## How it works...

In step 1, you discover the PowerShell 7 GPO files, with output like this:

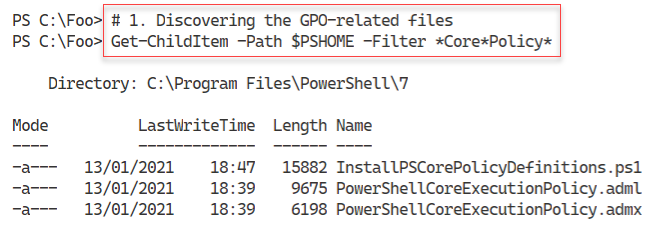


Figure 8.26: Discovering the GPO-related files

1. Insert image B16762\_08\_26.png

In step 2, you first create a string that holds the location of the GPO installation file,. Then you run this file to install the GPO files which looks like this:

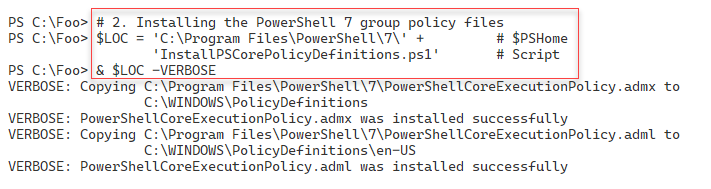


Figure 8.27: Installing the PowerShell 7 group policy files

1. Insert image B16762\_08\_27.png

In step 3, you create a new GPO, which creates the following output:

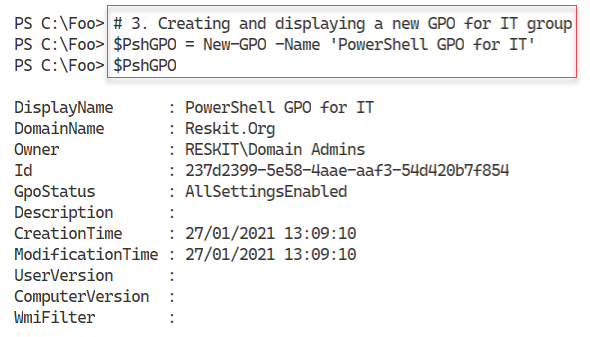


Figure 8.28: Creating and displaying a new GPO object for IT group

1. Insert image B16762\_08\_28.png

In step 4, you configure the GPO to enable module logging, and in step 5, you configure the module names to log. In step 6, you enable script block logging, and in step 7, you configure the GPO to enable an Unrestricted PowerShell execution policy. These 4 steps produce no output.

In step 8, you assign this GPO to the IT OU, creating no output. In the final step, step 9, you create and view a resultant set of policies report, which looks like this:

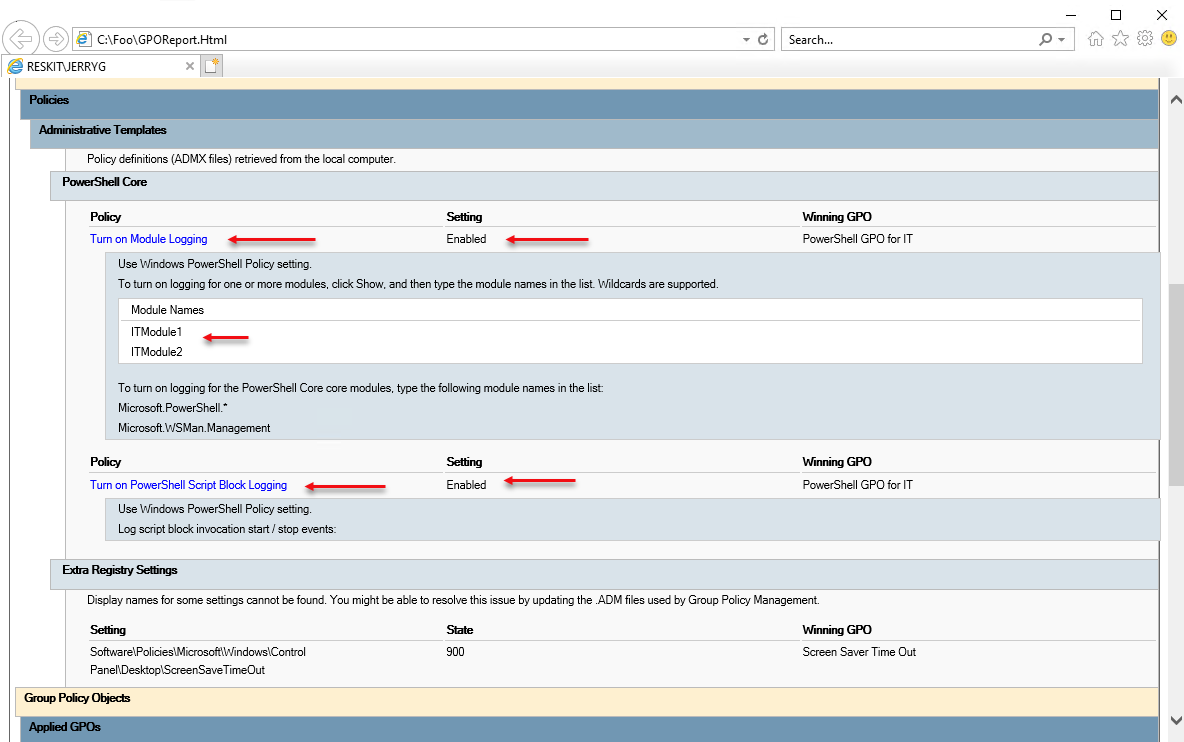


Figure 8.29: Policies report

1. Insert image B16762\_08\_29.png

## There's more...

This recipe creates a new GPO, configures the object with specific policy values, and then assigns to the IT OU in the Reskit.Org domain. When any user in the IT group logs on, PowerShell performs the specified logging and uses an Unrestricted execution policy. You can see in the RSOP report, produced in step 9, which policy settings PowerShell applies.

In step 9, you create an RSOP report. In order to ensure you get sensible output, you need to ensure that the user JerryG has logged into the DC. That way, you can generate a meaningful RSOP report.

# Using PowerShell script block logging

In the "Deploying PowerShell group policies" recipe, you saw how you could deploy policies related to PowerShell 7. One of these policies, Script Block Logging, causes PowerShell 7 to generate log events whenever you cause the execution of a script block that PowerShell deems noteworthy.

In addition to using Group Policy to invoke script block logging, you can also configure the local registry. In effect, this mimics using the Local Group Policy editor. Using the editor provides a nice interface to the policies but you can’t really automate a GUI. If you are making a single change to a single policy, then the GUI may be more convenient,. But if you are making changes to or creating more policies, using a PowerShell script may be more productive.

## Getting ready

You run this recipe on DC1, a domain controller in the Reskit.Org domain. You must log in as a Reskit\Administrator, a member of the domain administrators group.

## How to do it...

1. Clearing PowerShell Core operational log

WevtUtil cl 'PowerShellCore/Operational'

1. Enabling script block logging for the current user:

$SBLPath = 'HKCU:\Software\Policies\Microsoft\PowerShellCore' +

           '\ScriptBlockLogging'

if (-not (Test-Path $SBLPath))  {

        $null = New-Item $SBLPath -Force

    }

Set-ItemProperty $SBLPath -Name EnableScriptBlockLogging -Value '1'

1. Examining the PowerShell Core event log for 4104 events:

Get-Winevent -LogName 'PowerShellCore/Operational' |

  Where-Object Id -eq 4104

1. Examining logged event details:

Get-Winevent -LogName 'PowerShellCore/Operational' |

  Where-Object Id -eq 4104  |

    Select-Object -First 1 |

      Format-List -Property ID, LogName, Message

1. Creating another script block that Powershell does not log:

$SBtolog = {Get-CimInstance -Class Win32\_ComputerSystem | Out-Null}

$Before = Get-WinEvent -LogName 'PowerShellCore/Operational'

Invoke-Command -ScriptBlock $SBtolog

$After = Get-WinEvent -LogName 'PowerShellCore/Operational'

1. Comparing the events before and after you invoke the command

"Before:  $($Before.Count) events"

"After :  $($After.Count) events"

1. Removing registry policy entry:

Remove-Item -Path $SBLPath

## How it works...

In step 1, you use the wevtutil.exe console application to clear the PowerShell Core operational log. In step 2, you update the current user's registry to enable script block logging (for the currently logged on user Reskit\Administrator). These steps produce no output.

In step 3, you examine the PowerShell Core log for 4104 events, with output like this:

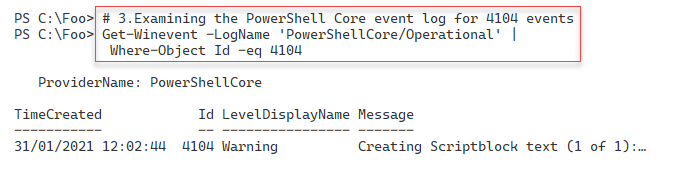


Figure 8.30: Examining the PowerShell Core event log for 4104 events

1. Insert image B16762\_08\_30.png

In step 4, you view the details of the event log entry you saw in the previous step, with output that looks like this:

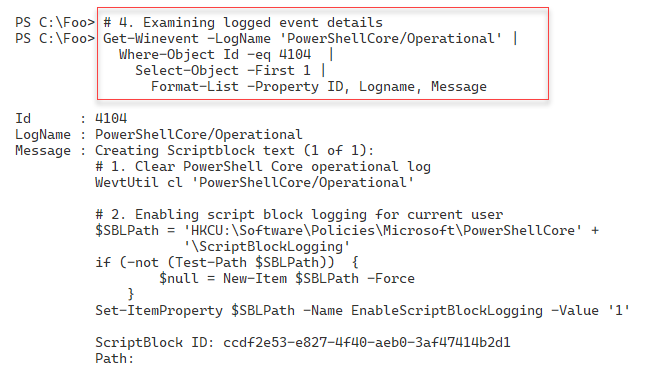


Figure 8.31: Examining logged event details

1. Insert image B16762\_08\_31.png

In step 5, you create and execute another script block, one that script block logging does not consider important enough to log. This step creates a count of the total number of event log entries before and after invoking the script block. This step produces no output, but in step 6, you view the before and after counts, like this:

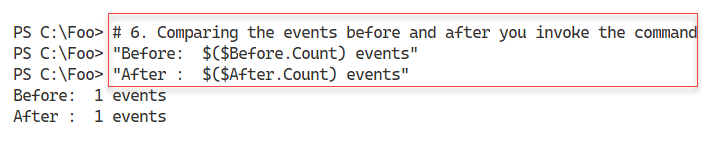


Figure 8.32: Comparing before and after

1. Insert image B16762\_08\_32.png

In the final step, step 7, you remove the policy entry from then registry, producing no output.

## There's more...

In step 1, you use the wevtutil.exe console application to clear an event log. With Windows PowerShell, you have the Clear-EventLog cmdlet that you can use to clear an event log. This cmdlet does not exist in PowerShell 7, which is why you usea Win32 console application to clear the log.

In step 6, you can see that ther script block you executed did *not* result in PowerShell logging the script block. That is to be expected, and it does reduce the amount of logging carried out.

# Configuring AD password policies

Passwords are essential for security as they help ensure that a person is whom they say they are and thus are allowed to perform some action such as log on to a host, or edit a file. Password policies allow you to define your password attributes, including minimum length and whether complex passwords are required. You can also set the number of times a user enters an invalid password before that user is locked out (and a lockout duration). For more details on improving authentication security, see https://www.microsoftpressstore.com/articles/article.aspx?p=2224364&seqNum=2.

In AD, you can apply a default domain password policy. This policy applies to all users in the domain. In most cases, this is adequate for the organization. But in some cases, you wish to apply a more stringent password policy to certain users or groups of users. You use AD's fine-grained password policy to manage these more restrictive passwords. A ‘fine-grained’ policy is one you can apply to just a single user as opposed to every user in the domain.

## Getting ready

You run this recipe on DC1, a domain controller in the Reskit.Org domain. You must login as a domain administrator.

## How to do it...

1. Discovering the current domain password policy

Get-ADDefaultDomainPasswordPolicy

1. Discovering if there is a fine-grained password policy for JerryG

Get-ADFineGrainedPasswordPolicy -Identity 'JerryG'

1. Updating the default password policy

$DPWPHT = [Ordered] @{

    LockoutDuration             = '00:45:00'

    LockoutObservationWindow    = '00:30:00'

    ComplexityEnabled           = $true

    ReversibleEncryptionEnabled = $false

    MinPasswordLength           = 6

}

Get-ADDefaultDomainPasswordPolicy -Current LoggedOnUser |

  Set-ADDefaultDomainPasswordPolicy @DPWPHT

1. Checking updated default password policy

Get-ADDefaultDomainPasswordPolicy

1. Creating a fine-grained password policy

$PD = 'DNS Admins Group Fine-grained Password Policy'

$FGPHT = @{

  Name                     = 'DNSPWP'

  Precedence               = 500

  ComplexityEnabled        = $true

  Description              = $PD

  DisplayName              = 'DNS Admins Password Policy'

  LockoutDuration          = '0.12:00:00'

  LockoutObservationWindow = '0.00:15:00'

  LockoutThreshold         = 4

}

1. Assigning the policy to DNS Admins

$DNSADmins = Get-ADGroup -Identity DNSAdmins

$ADDHT = @{

  Identity  = 'DNSPWP'

  Subjects  = $DNSADmins

}

Add-ADFineGrainedPasswordPolicySubject  @ADDHT

1. Assigning the policy to JerryG

$Jerry = Get-ADUser -Identity JerryG

Add-ADFineGrainedPasswordPolicySubject -Identity DNSPWP -Subjects $Jerry

1. Checking on policy application for the group

Get-ADGroup 'DNSAdmins' -Properties \* |

  Select-Object -Property msDS-PSOApplied

1. Checking on policy application for the user

Get-ADUser JerryG -Properties \* |

  Select-Object -Property msDS-PSOApplied

1. Getting DNS Admins policy

Get-ADFineGrainedPasswordPolicy -Identity DNSPWP

1. Checking on JerryG's resultant password policy

Get-ADUserResultantPasswordPolicy -Identity JerryG

## How it works...

In step 1, you retrieve the default AD password policy, which looks like this:

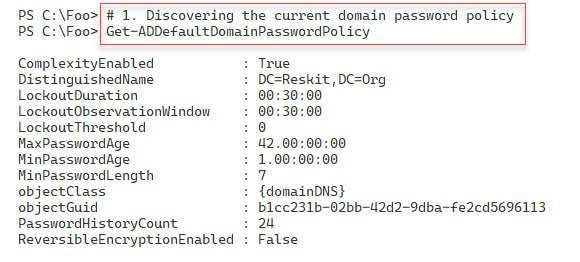


Figure 8.33: Discovering the current domain password policy

1. Insert image B16762\_08\_33.png

In step 2, you check to see if there are any fine-grained password policies for the user JerryG, which looks like this:

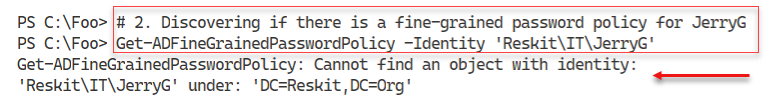


Figure 8.34: Checking for fine-grained password policies

1. Insert image B16762\_08\_34.png

In step 3, you update the default password policy for the domain, changing a few settings. This produces no output. In step 4, you review the updated default password policy which looks like this:

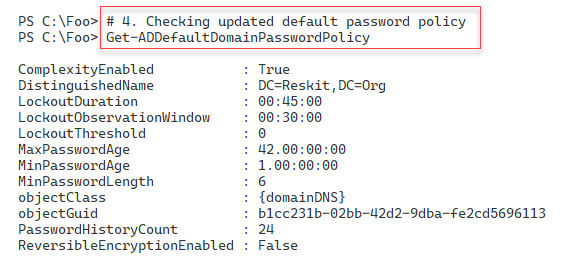


Figure 8.35: Checking updated default password policy

1. Insert image B16762\_08\_35.png

In step 5, you create a new fine-grained password policy with some overrides to the default domain policy you looked at above. In step 6, you assign the policy to the DNS Admins group, and in step 7, you apply this policy explicitly to the user JerryG. These three steps create no output.

In step 8, you check on the policy application for the DNS Admins group, which looks like this:

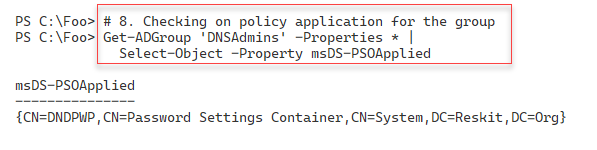


Figure 8.36: Checking on policy application for the DNS Admins group

1. Insert image B16762\_08\_36.png

In step 9, you check on the password policy applied to the user JerryG, which looks like this:

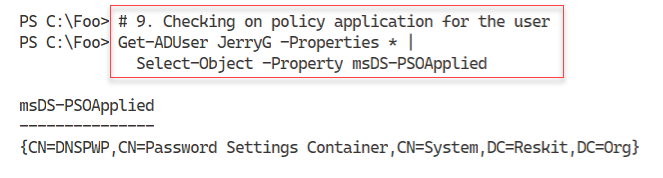


Figure 8.37: Checking on policy application for the user

1. Insert image B16762\_08\_37.png

In step 10, you examine the DNS Admins password policy, with output like this:

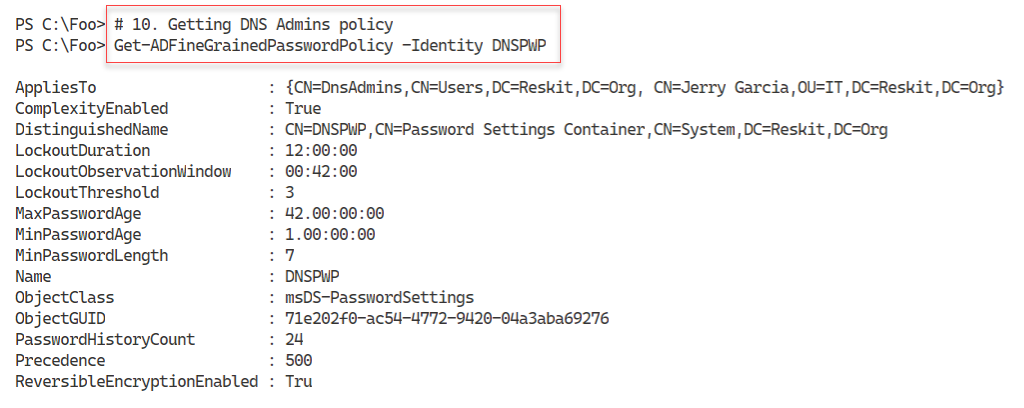


Figure 8.38: Getting DNS Admins password policy

1. Insert image B16762\_08\_38.png

In the final step, step 11, you examine the resulting password policy for the user JerryG, which looks like this:

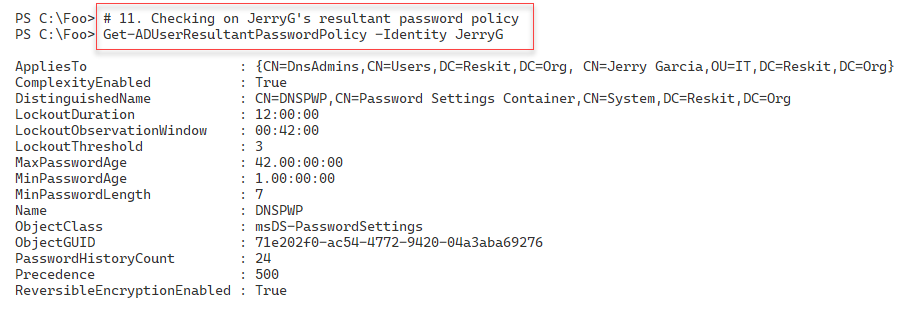


Figure 8.39: Checking JerryG's resultant password policy

1. Insert image B16762\_08\_39.png

## There's more...

In step 1, you view the existing default domain password policy. The settings you see in this step were created by the installation process when you installed Windows Server on DC1.

In step 2, you attempt to find a fine-grained password policy that would apply to the user JerryG, which does not exist.

In step 5, you create a new fine-grained password policy that you assign to the DNS Admins group (in step 6) and JerryG (in step 7). This assignment ensures the policy applies to JerryG, whether or not he is a DNS Admins group member.

In step 11, you see the password policy settings for the user JerryG. These settings derive from the default domain policy plus the settings you specified in the DNSPWP policy. In theory, you could have a user with the effective password policy settings coming for multiple policy objects (eg a GPO for the domain, one for an OU, etc), although you should probably avoid such complexity.

# Managing Windows Defender antivirus

Microsoft Defender Antivirus is the next-generation protection component of Microsoft Defender for Endpoint. Defender Antivirus provides antivirus and antimalware facilities. The product also does some packet analysis to detect network-level attacks.

The Windows installation process installs Defender on both Windows 10 and Windows Server 2022, by default, although you can remove the feature should you wish. For more details on Defender in Windows Server, see <https://docs.microsoft.com/en-gb/windows/security/threat-protection/microsoft-defender-antivirus/microsoft-defender-antivirus-on-windows-server-2016>.

Testing any antivirus or antimalware application can be difficult. You want to ensure that the product, Defender, in this case, is working. But at the same time, you don't want to infect a server. One solution is to create a test file. The European Institute for Computer Anti-Virus Research (EICAR) has created a simple set of test files you can use to ensure your antivirus product works. EICAR have created several versions of this file, including both a text file and an executable. These files are harmless, but as you see, triggers Defender.

## Getting ready

You run this recipe on DC1, a domain controller in the Reskit.Org domain.

## How to do it...

1. Ensuring Defender and associated tools are installed:

$DHT = @{

  Name                   =  'Windows-Defender'

  IncludeManagementTools = $true

}

$Defender = Install-WindowsFeature @DHT

If ($Defender.RestartNeeded -eq 'Yes') {

  Restart-Computer

}

1. Discovering the cmdlets in the Defender module:

Import-Module -Name Defender

Get-Command -Module Defender

1. Checking the Defender service status

Get-Service  -Name WinDefend

1. Checking the operational status of Defender on this host

Get-MpComputerStatus

1. Getting and counting threat catalog:

$ThreatCatalog = Get-MpThreatCatalog

"There are $($ThreatCatalog.count) threats in the catalog"

1. Viewing five threats in the catalog

$ThreatCatalog |

  Select-Object -First 5 |

    Format-Table -Property SeverityID, ThreatID, ThreatName

1. Setting key settings

# Enable real-time monitoring

Set-MpPreference -DisableRealtimeMonitoring 0

# Enable Cloud-DeliveredProtection

Set-MpPreference -MAPSReporting Advanced

# Enable sample submission

Set-MpPreference -SubmitSamplesConsent Always

# Enable checking signatures before scanning

Set-MpPreference -CheckForSignaturesBeforeRunningScan 1

# Enable email scanning"

Set-MpPreference -DisableEmailScanning 0

1. Creating a false positive threat:

$TF = 'C:\Foo\FalsePositive1.Txt'

$FP = 'X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-' +

'STANDARD-ANTIVIRUS-TEST-FILE!$H+H\*'

$FP | Out-File -FilePath $TF

Get-Content -Path $TF

1. Running a quick scan on C:\Foo:

$ScanType = 'QuickScan'

Start-MpScan -ScanType $ScanType -ScanPath C:\Foo

1. Viewing detected threats:

Get-MpThreat

## How it works...

In step 1, you use the Install-WindowsFeature command to ensure that you have installed both Defender and the tools you can use to manage Defender . This step may require a reboot. If so, this step reboots DC1 without producing any output.

In step 2, you look at the Defender module to discover the cmdlets contained in the module. The output looks like this:

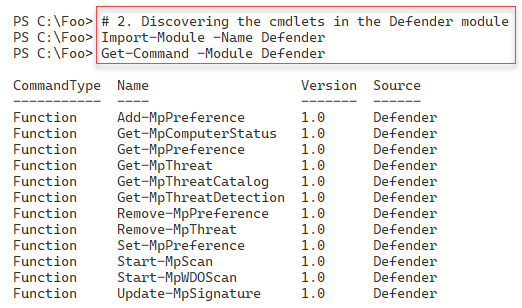


Figure 8.40: Discovering the cmdlets in the Defender module

1. Insert image B16762\_08\_40.png

In step 3, you check the status of the WinDefend service. You should see the following output:

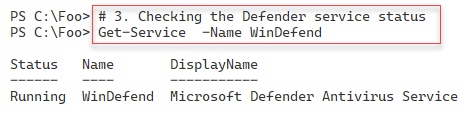


Figure 8.41: Checking the Defender service status

1. Insert image B16762\_08\_41.png

You use the Get-MpComputerstatus cmdlet to get the status of Defender on the local computer in step 4. The output looks like this:

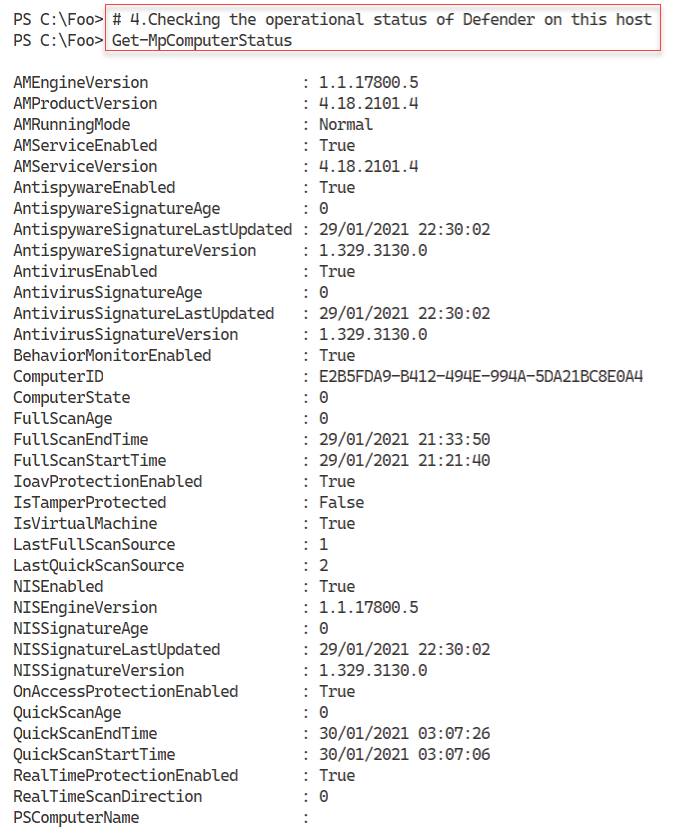


Figure 8.42: Checking Defender status

1. Insert image B16762\_08\_42.png

Defender uses details of individual threats that it stores in a threat catalog. Windows Update regularly updates this catalog as needed. In step 5, you produce a count of the number of threats in the catalog, which looks like this:

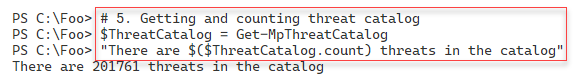


Figure 8.42: Getting and counting threat catalog

1. Insert image B16762\_08\_42.png

In step 6, you examine the first five threats in the Defender threat catalog, which looks like this:

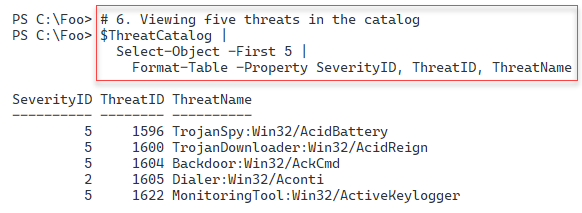


Figure 8.43: Viewing first five threats in the catalog

1. Insert image B16762\_08\_43.png

In step 7, you configure four important Defender settings. You can use the Set‑MpPreference cmdlet to configure a range of preference settings for Windows Defender scans and updates. You can modify exclusion file name extensions, paths, or processes, and specify the default action for high, moderate, and low threat levels. You can view more details at: https://docs.microsoft.com/powershell/module/defender/set-mppreference.

In step 8, you attempt to create a file which Defender regards as a threat. This file comes from the EICAR and as you can see is a benign text file. When you run this step, you get no output, although you may notice a Defender popup warning you that it has discovered a threat. If you run this step from the PowerShell console, you should get an error message.

In step 9, you run a quick scan on the C:\Foo folder where you attempted to create the test threat file. This step also produces no output.

In step 10, you view all the threats detected by Defender, which looks like this:

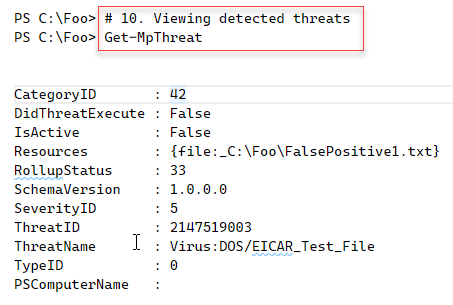


Figure 8.44: Viewing all detected threats

1. Insert image B16762\_08\_44.png

## There's more...

In step 5, you get a count of the number of threats of which Defender is aware as of the time of writing. When you run this step, you should see a higher number, reflecting newly discovered threats.

With step 8, you attempt to create a file which Defender recognizes as a threat. This file is the EICAR test file which is harmless, but you can use it to test the basic functioning of Defender. In step 10, you view the threats Defender detected, and you can see it is the file identified as an EICAR\_Test\_File.