15

Managing With Windows Management Instrumentation

In this chapter, we cover the following recipes:

* Exploring WMI in Windows
* Exploring WMI namespaces
* Exploring WMI classes
* Getting local and remote WMI objects
* Using WMI methods
* Using WMI events
* Implementing permanent WMI eventing

# Introduction

Windows Management Instrumentation (WMI) is a Windows component you use to help manage Windows systems. WMI is Microsoft's proprietary implementation of the standards-based of Web-Based Enterprise Management (WBEM), an open standard promulgated by the Distributed Management Task Force. WBEM aims to utilize standards-based Internet technologies to unify the management of distributed computing environments.

In addition to WMI for Windows, there are other implementations of WBEM, including OpenWBEM. You can read more about the DMTF and WBEM at https://www.dmtf.org/about/faq/wbem\_faq, and check out OpenWBEM over at http://openwbem.sourceforge.net/.

Microsoft first introduced WMI as an add-on component for Window NT 4. They later integrated WMI as an essential component of the Windows client, from Windows XP onwards, and Windows Server versions since Windows Server 2000. Subsequently, several feature teams inside the Windows group made heavy use of WMI. Both the storage and networking stacks within Windows, for example, use WMI. Many of the cmdlets, such as Get-NetAdapter, are based directly on WMI.

The WBEM standards originally specified that the components of WMI communicate using HTTP. To improve performance, Microsoft instead implemented the components of WMI to communicate using the Common Object Model (COM), which was a popular development technology at that time. WMI itself remains based on COM. The PowerShell cmdlets use .NET to access the features of WMI.

PowerShell 1.0 came with a set of cmdlets that you could use to access WMI. These were basic cmdlets that work in all subsequent versions of Windows PowerShell. However, there is no support for these cmdlets directly in PowerShell 7. You can invoke older WMI cmdlet-based scripts on a machine using PowerShell remoting if you really need to.

With PowerShell V3, Microsoft did some significant work on WMI, resulting in a new module, CimCmdlets. There were several great reasons behind both the new module and the associated updates to some of the WMI internals to assist developers. In this chapter, you make use of the CimCmdlets module to access the features of WMI. You can read more about why the team decided to build new cmdlets in a blog post at https://devblogs.microsoft.com/powershell/introduction-to-cim-cmdlets/. If you have scripts that use the older WMI cmdlets, consider upgrading them to use the later CimCmdlets module instead. These newer cmdlets are faster, which is always a nice benefit.

## WMI architecture

Before looking at CimCmdlets and what you can do with them, it is useful to understand the WMI architecture within Windows. The run-time architecture of WMI is the same in Windows 10 and Windows Server 2022. The following diagram shows the WMI conceptual architecture:



Figure 15.1: WMI Architecture

**Insert image B42024\_15\_01.png**

As an IT Pro, you use the CIMcmdlets module's cmdlets to access WMI. These cmdlets use .NET to communicate, via transport protocol, with the WMI core and the CIM Object manager on a local or remote host. The transport protocols include DCOM and WSMan. The core components of WMI, particularly the CIM Object Manager (CIMOM), are COM components you find on every Windows host.

The CIMOM stores information in the WMI repository, sometimes referred to as the Common Information Model (CIM) or the CIM database. This database is, in effect, a subset of an ANSI-SQL database. The CIM cmdlets enable you to access the information within the database. The CIM database organizes the information into namespaces of classes. .NET also uses namespaces to organize classes. However, .NET classes include the namespace name as part of the class name. You can create a new email message. The email message class name is System.Net.Mail.Mailmessage, where the namespace is System.Net.Mail. With WMI, namespace names and class names are separate and supplied to the CIM cmdlets using different parameters.

WMI classes contain data instances that hold relevant management data. For example, the WMI namespace Root\CimV2 contains the class Win32\_Share. Each instance within this class represents one of the SMB shares within your host. With PowerShell, you would normally use the SMB cmdlets to manage SMB shares. There is useful information contained in other WMI classes for which there is no cmdlet support.

When you retrieve instances of, for example, the Win32\_Share class, .NET gets the instance information and returns it in a .NET wrapper object. Strictly speaking, a WMI object instance is an instance of a specialized .NET class with data returned in WMI. For this reason, you treat WMI objects using the same methods you employ with other .NET objects.

Many WMI classes have methods you can invoke which perform some operation on either a given WMI instance or statically based on the class. The Win32\_Share class, for example, has a static Create() method you can use to create a new share. Each instance of that class has a dynamic Delete() method, which deletes the SMB share.

An important architectural feature of WMI is the WMI provider. A WMI provider is an add-in to WMI that implements WMI classes inside a given host. The Win32 WMI provider, for example, implements hundreds of WMI classes, including Win32\_Share and Win32\_Bios. A provider also implements class methods and class events. For example, the Win32 provider is responsible for performing both the Delete() method to delete an SMB share and the Create() method to create a new SMB share.

In production, you are more likely to manage SMB shares using the SMB cmdlets and less likely to use WMI directly. Since SMB shares should be very familiar, they make a great example to help you understand more about WMI, and this chapter's recipes make use of the class.

WMI and WMI providers offer a rich eventing system. WMI and WMI provider classes can implement events to which you can subscribe. When the event occurs, the eventing system notifies you, and you can take some action to handle the event occurrence. For example, you can register for a WMI event that occurs when someone changes the membership of an AD group. When this happens, WMI eventing allows you to take some actions, such as emailing a security administrator informing them of the group's membership change. WMI also implements permanent WMI events. This feature allows you to configure WMI to trap and handle events with no active PowerShell session running. Permanent events even survive a reboot of the host, extremely powerful in lights-off environments.

There is a lot more detail about WMI than can fit in this chapter. For more details about WMI and how you can interact with it in more detail, consult with Richard Siddaway's PowerShell and WMI book (© Manning, Aug 2012 - https://www.manning.com/books/powershell-and-wmi). Richard's book goes into great detail about WMI, but all the code samples use the older WMI cmdlets. You should be able to translate the samples to use the CIM cmdlets. A key value of the book is the discussion of WMI features and how they work. The basic functioning of WMI has not changed significantly since that book was published.

# Exploring WMI in Windows

Windows installs WMI during the installation of the OS. The installation process puts most of the WMI components, including the repository, tools, and the providers, into a folder C:\Windows\System32\WBEM.

Inside a running Windows host, WMI runs as a service, the winmgmt service (WinMgmt.exe). Windows runs this service inside a share service process (svchost.exe). In the early versions of WMI in Windows, WMI loaded all the WMI providers into the winmgmt service. The failure of a single provider could cause the entire WMI service to fail. Later, with Windows XP and beyond, Microsoft improved WMI to load providers in a separate process, Wmiprvse.exe.

In this recipe, you examine the contents of the WBEM folder, the WMI service, and run-time components of WMI.

## Getting ready

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

## How to do it...

1. Viewing the WBEM folder

$WBEMFOLDER = "$Env:windir\system32\wbem"

Get-ChildItem -Path $WBEMFOLDER |

  Select-Object -First 20

1. Viewing the WMI repository folder

Get-ChildItem -Path $WBEMFOLDER\Repository

1. Viewing the WMI service details

Get-Service -Name Winmgmt  |

  Format-List -Property \*

1. Getting process details

$S = tasklist.exe /svc /fi "SERVICES eq winmgmt" |

       Select-Object -Last 1

$P = [int] ($S.Substring(30,4))

Get-Process -Id $P

1. Examining DLLs loaded by the WMI service process

Get-Process -Id $P |

  Select-Object -ExpandProperty modules |

    Where-Object ModuleName -match 'wmi' |

      Format-Table -Property FileName, Description, FileVersion

1. Discovering WMI providers

Get-ChildItem -Path $WBEMFOLDER\\*.dll |

  Select-Object -ExpandProperty Versioninfo |

    Where-Object FileDescription -match 'prov' |

      Format-Table -Property Internalname,

                             FileDescription,

                             ProductVersion

1. Examining the WmiPrvSE process

Get-Process -Name WmiPrvSE

1. Finding the WMI event log

$Log = Get-WinEvent -ListLog \*wmi\*

$Log

1. Looking at the event types in the WMI log

$Events = Get-WinEvent -LogName $Log.LogName

$Events | Group-Object -Property LevelDisplayName

1. Examining WMI event log entries

$Events |

  Select-Object -First 5 |

    Format-Table -Wrap

1. Viewing executable programs in WBEM folder

$Files = Get-ChildItem -Path $WBEMFOLDER\\*.exe

"{0,15}  {1,-40}" -f 'File Name','Description'

Foreach ($File in $Files){

 $Name = $File.Name

 $Desc = ($File |

          Select-Object -ExpandProperty VersionInfo).FileDescription

"{0,15}  {1,-40}" -f $Name,$Desc

}

## How it works...

The WMI service and related files in the Windows installation folder's System32\WBEM folder. In step 1, you view part of the contents of that folder, with output like this:

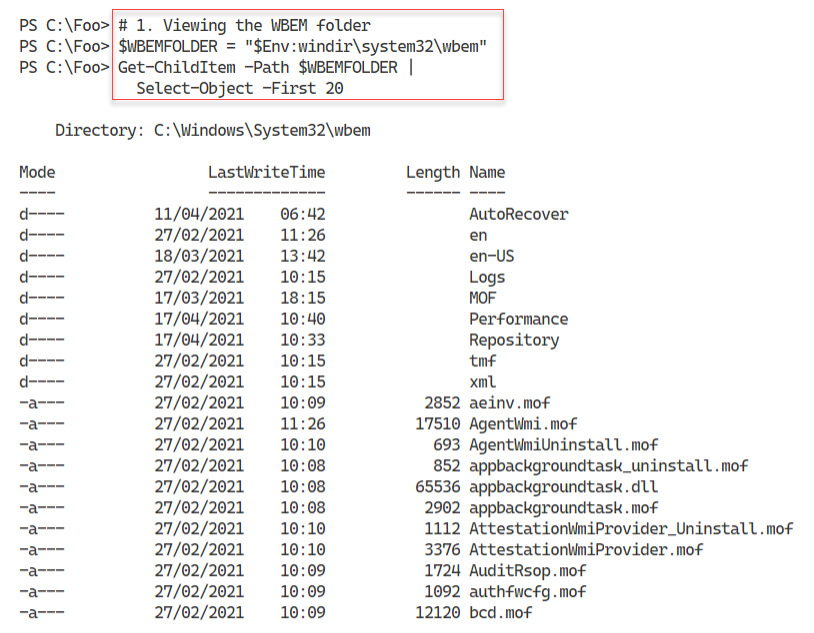


Figure 15.2: Examining the WBEM folder

**Insert image B42024\_15\_02.png**

WMI stores the CIM repository in a separate folder. In step 2, you examine the files that make up the database, with output like this:

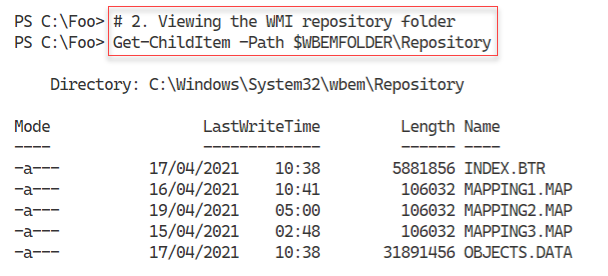


Figure 15.3: Examining the files making up the CIM repository

**Insert image B42024\_15\_03.png**

In step 3, you use Get-Service to examine the WMI service, with output that looks like this:

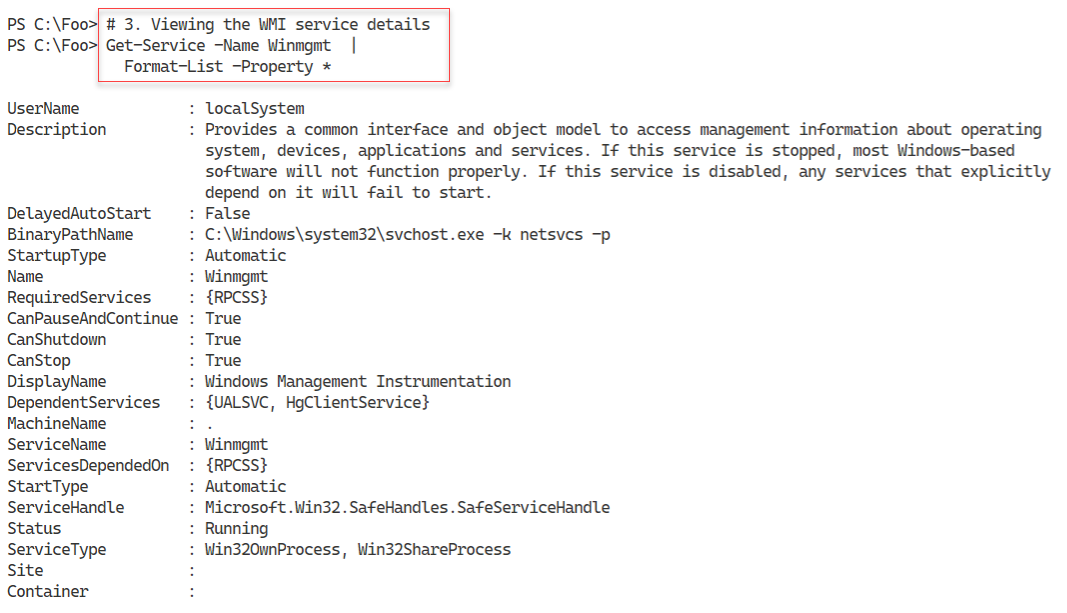


Figure 15.4: Viewing the WMI service

**Insert image B42024\_15\_04.png**

In step 4, you examine the Windows process that runs the WMI service, with output like this:

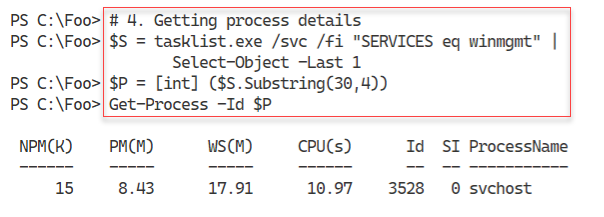


Figure 15.5: Viewing the WMI service

**Insert image B42024\_15\_05.png**

In step 5, you look at the DLLs loaded by the WMI service process, with the following output:

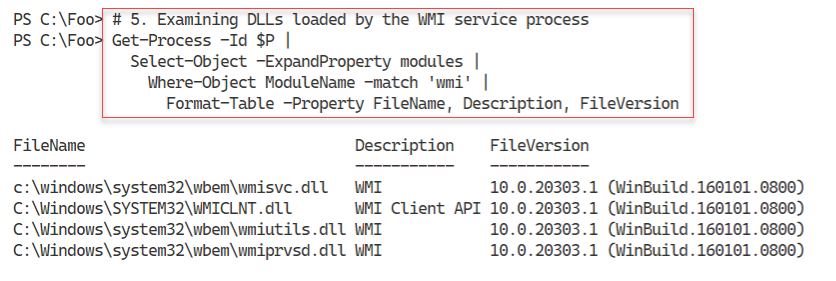


Figure 15.6: Viewing the DLLs loaded by the WMI service process

**Insert image B42024\_15\_06.png**

Each WMI provider is a DLL which the WMI service can use. In step 6, you look at the WMI providers on SRV1, with output like this:

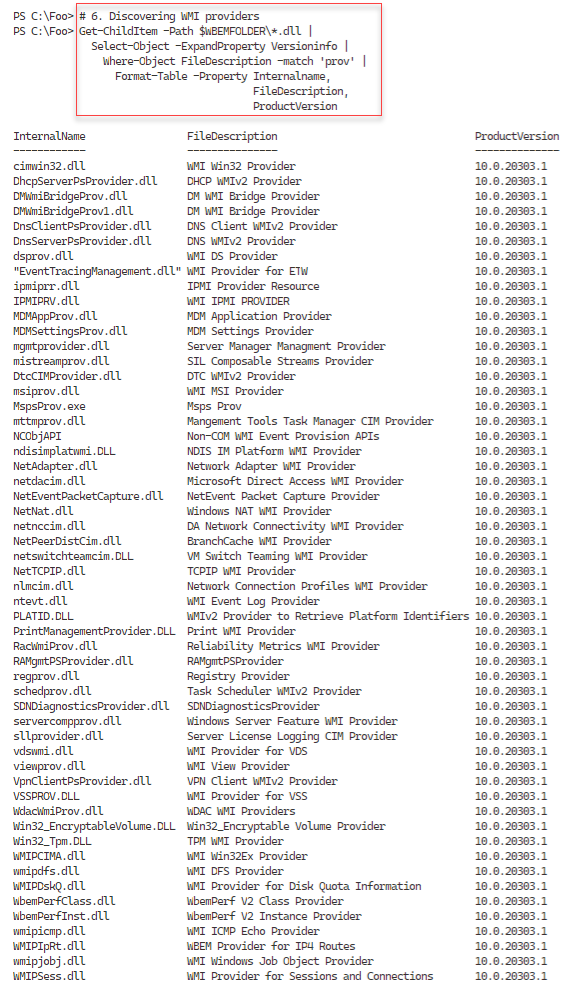


Figure 15.7: Viewing WMI provider DLLs

**Insert image B42024\_15\_07.png**

In step 7, you examine the WmiPrvSE process, with output like this:

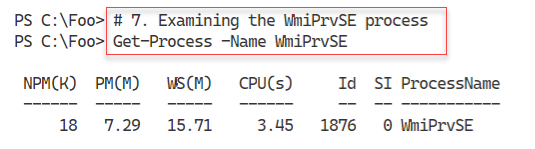


Figure 15.8: Viewing the WmiPrvSE process

**Insert image B42024\_15\_08.png**

Like other Windows services, WMI logs events to an event log, which can help troubleshoot WMI issues. In step 8, you look for any WMI related event logs with output like this:

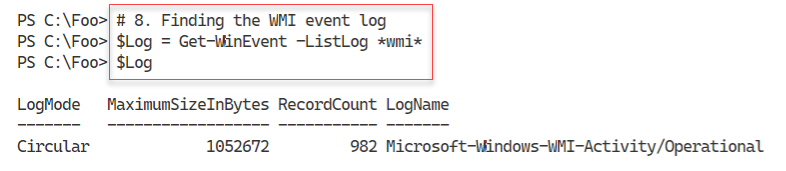


Figure 15.9: Viewing WMI-related event logs

**Insert image B42024\_15\_09.png**

In step 9, you get the events from the log to view the different log levels, with output like this:

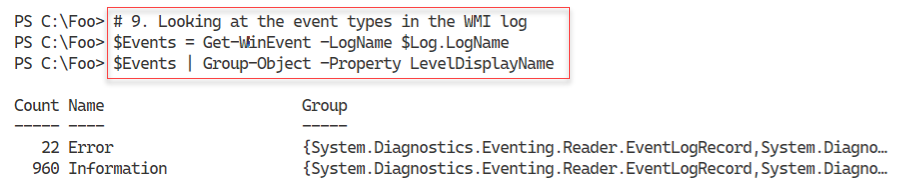


Figure 15.10: Discovering WMI event types

**Insert image B42024\_15\_10.png**

In step 10, you view the first five WMI event log entries on SRV1. The output looks like this:



Figure 15.11: Viewing WMI event log entries

**Insert image B42024\_15\_11.png**

In step 11, you view the executable programs in the WBEM folder, with output like this:

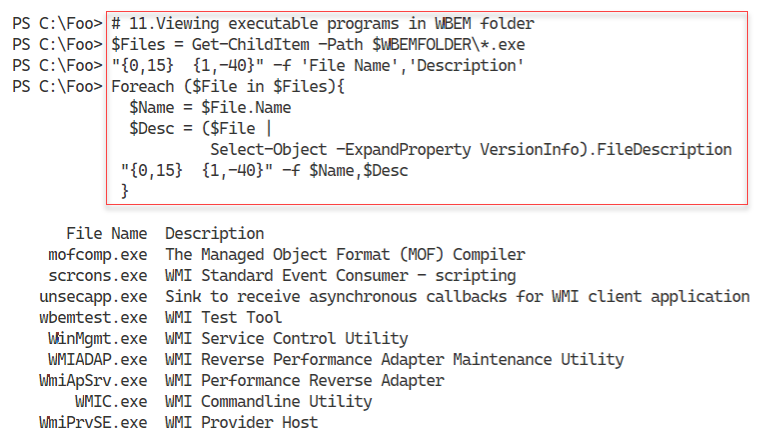


Figure 15.12: Viewing the executable programs in the WBEM folder

**Insert image B42024\_15\_12.png**

With PowerShell 7 (and optionally with Windows PowerShell), you access WMI's functionality using the cmdlets in the CimCmdlets module. You installed this module as part of installing PowerShell 7. The Windows installation program installed a version of this module when you installed the host OS. In step 12, you examine the properties of this module, with output like this:

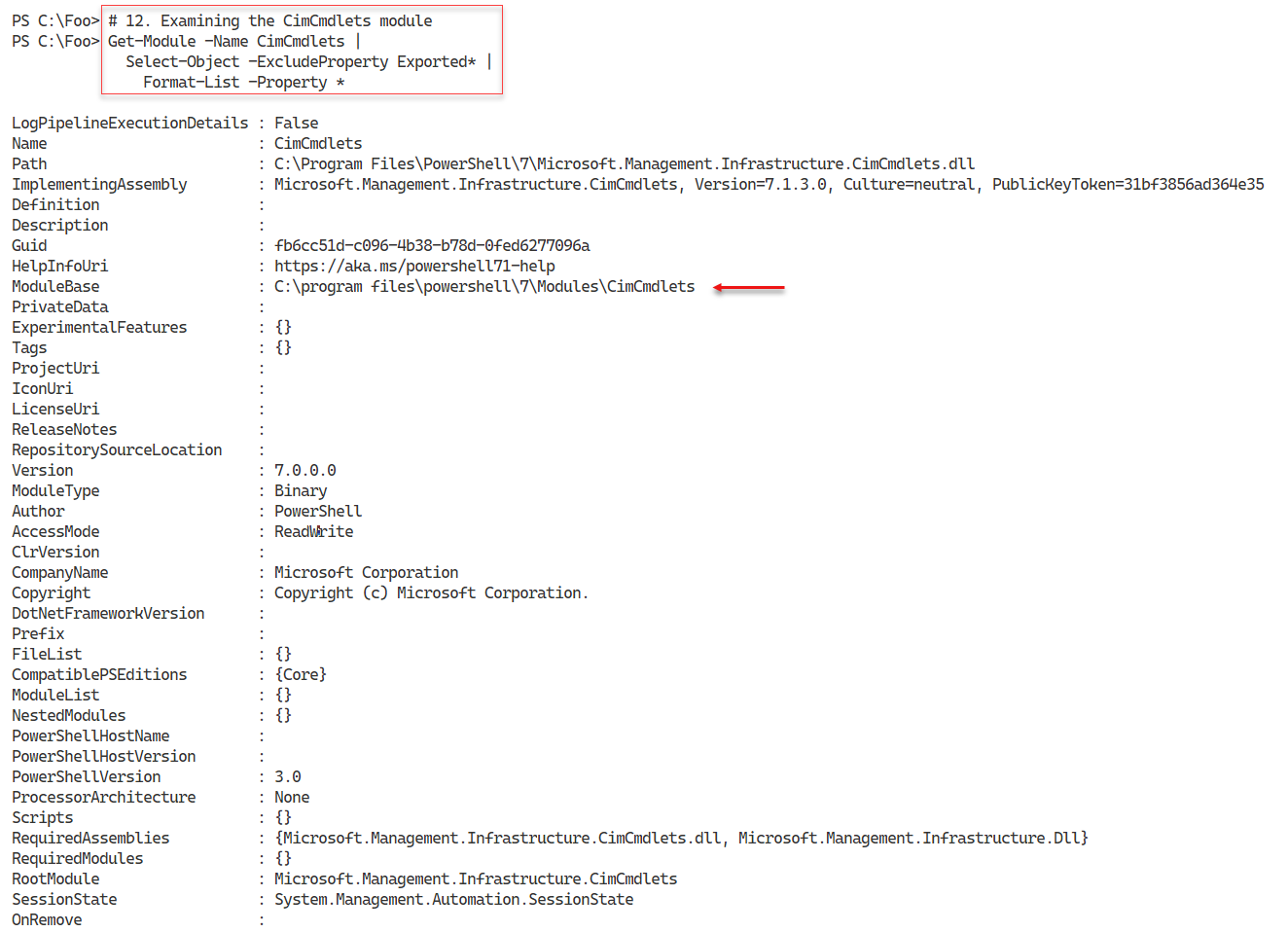


Figure 15.13: Viewing the CimCmdlets module details

**Insert image B42024\_15\_13.png**

In step 13, you use Get-Command to discover the cmdlets within the CimCmdlets module, which looks like this:

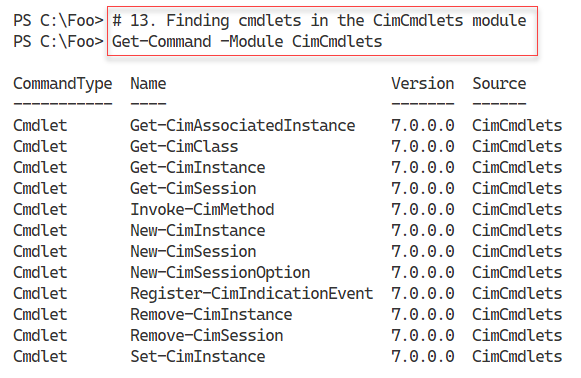


Figure 15.14: Viewing the cmdlets in the CimCmdlets module

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

In step 14, you examine the properties of an object returned from WMI after using the Get‑CimInstance command. The output from this step looks like this:

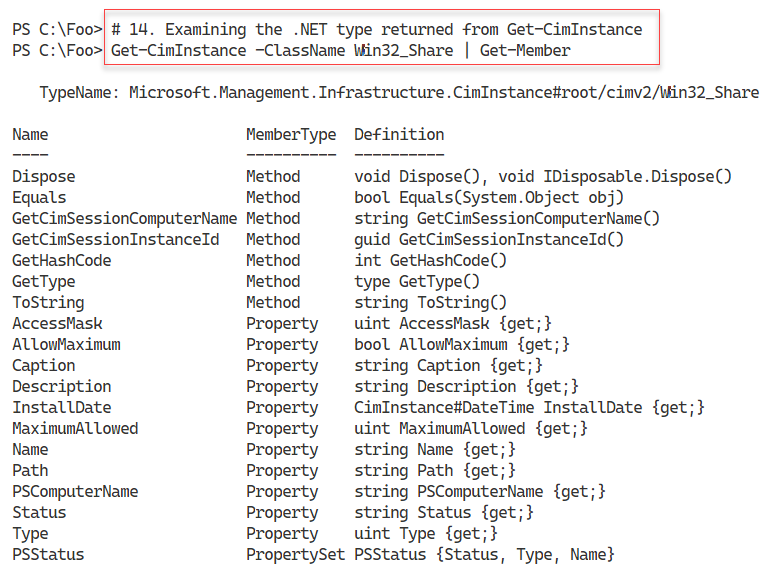


Figure 15.15: Examining the output from Get-CimInstance

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

## There's more...

In step 1, you viewed the first 20 files/folders in the WBEM folder. There are a lot more files than you see in the figure. These include the DLL files for the WMI providers available on your system.

In step 7, you view the WmiPrvSE process. This process hosts WMI providers. Depending on the actions WMI is currently doing, you may see zero, one, or more occurrences of this process on your hosts.

In step 9 and step 10, you discover and examine the WMI system event log. On SRV1, you can see there are both Error and Information event log entries. As you can see in step 10, the information entries are mostly indications that WMI has loaded and invoked a particular provider. In most cases, the Error event messages you see are transient or benign.

In step 14, you looked at the data returned by Get-CimInstance. As you can see from the output, the cmdlet returns the data obtained from the WMI class. This data is wrapped in a .NET object and has a class of Microsoft.Management.Infrastructure.CimInstance, with a suffix indicating the path to the WMI class, in this case, the Win32\_Share class in the ROOT/CIMV2 namespace. As you can see from the output, the returned object and its contents differ from that returned from Get-WMIObject.

# Exploring WMI namespaces

The PowerShell CIM cmdlets enable you to retrieve, update, and remove information from the CIM database and subscribe to and handle WMI events. The CIM database organizes its information into sets of classes within a hierarchical set of namespaces. A namespace is, in effect, a container holding WMI classes.

The name of the root WMI namespace is ROOT, although WMI is not overly consistent with regards to capitalization, as you may notice. A namespace can contain classes as well as additional child namespaces. For example, the root namespace has a child namespace, CIMV2, which you refer to as ROOT\CIMV2. This namespace also has child namespaces.

Every namespace in the CIM DB, including ROOT, has a special system class called \_\_NAMESPACE. This class contains the names of child namespaces within the current namespaces. Thus, in the namespace ROOT, the \_\_NAMESPACE class contains an instance for the CIMV2 child namespace. Since this class exists inside every namespace, it is straightforward to discover all the namespaces on your system.

There are many namespaces and classes within WMI on any given system. The specific namespaces and classes depend on what applications and Windows features you run on a host. Additionally, not all the namespaces or classes are useful to the IT pro. The classes in the ROOT\CIMV2 namespace, for example, implemented by the WIN32 provider, implements hundreds of OS and host-related WMI classes that can be very useful to an IT professional. Other classes or namespaces are typically only useful for developers implementing WMI components or providers.

Another less commonly used namespace is ROOT\directory\LDAP, which contains classes related to the Active Directory. While you perform most of the AD management using the AD cmdlets, there are features of this namespace, specifically eventing, that are not available with the AD cmdlets and that you may find useful.

## Getting ready

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

## How to do it...

1. Viewing WMI classes in the root namespace

Get-CimClass -Namespace 'ROOT' |

  Select-Object -First 10

1. Viewing the \_\_NAMESPACE class in ROOT

Get-CimInstance -Namespace 'ROOT' -ClassName \_\_NAMESPACE |

  Sort-Object -Property Name

1. Getting and counting classes in ROOT\CIMV2

$Classes = Get-CimClass -Namespace 'ROOT\CIMV2'

"There are $($Classes.Count) classes in ROOT\CIMV2"

1. Discovering all the namespaces on SRV1

Function Get-WMINamespaceEnum {

  [CmdletBinding()]

  Param($NS)

  Write-Output $NS

  Get-CimInstance "\_\_Namespace" -Namespace $NS -ErrorAction SilentlyContinue |

  ForEach-Object { Get-WMINamespaceEnum "$ns\$($\_.name)"   }

}  # End of function

$Namespaces = Get-WMINamespaceEnum 'ROOT' |

  Sort-Object

"There are $($Namespaces.Count) WMI namespaces on SRV1"

1. Viewing the first 25 namespaces on SRV1

$Namespaces |

  Select-Object -First 25

1. Creating a script block to count namespaces and classes

$SB = {

 Function Get-WMINamespaceEnum {

   [CmdletBinding()]

   Param(

     $NS

    )

   Write-Output $NS

$EAHT = @{ErrorAction = SilentlyContinue}

   Get-CimInstance "\_\_Namespace" -Namespace $NS @EAHT |

     ForEach-Object { Get-WMINamespaceEnum "$ns\$($\_.name)"   }

   }  # End of function

   $Namespaces = Get-WMINamespaceEnum 'ROOT' | Sort-Object

   $WMIClasses = @()

   Foreach ($Namespace in $Namespaces) {

   $WMIClasses += Get-CimClass -Namespace $Namespace

  }

 "There are $($Namespaces.count) WMI namespaces on $(hostname)"

 "There are $($WMIClasses.count) classes on $(hostname)"

}

1. Running the script block locally on SRV1

Invoke-Command -ComputerName SRV1 -ScriptBlock $SB

1. Running the script block on SRV2

Invoke-Command -ComputerName SRV2 -ScriptBlock $SB

1. Running the script block on DC1

Invoke-Command -ComputerName DC1 -ScriptBlock $SB

## How it works...

In step 1, you view the WMI classes in the WMI root namespace on SRV1, with output like this:

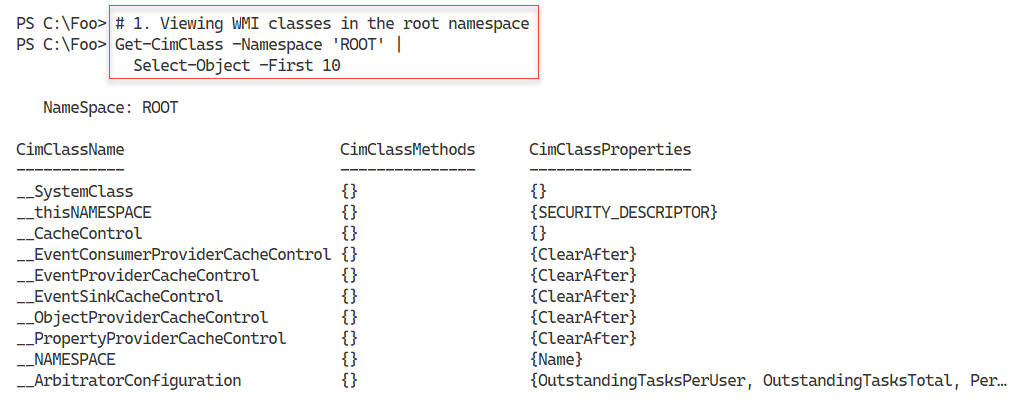


Figure 15.16: Examining WMI classes in the root namespace

**Insert image B42024\_15\_16.png**

In step 2, you examine the instances of the \_\_NAMESPACE class in the root WMI namespace. The output looks like this:

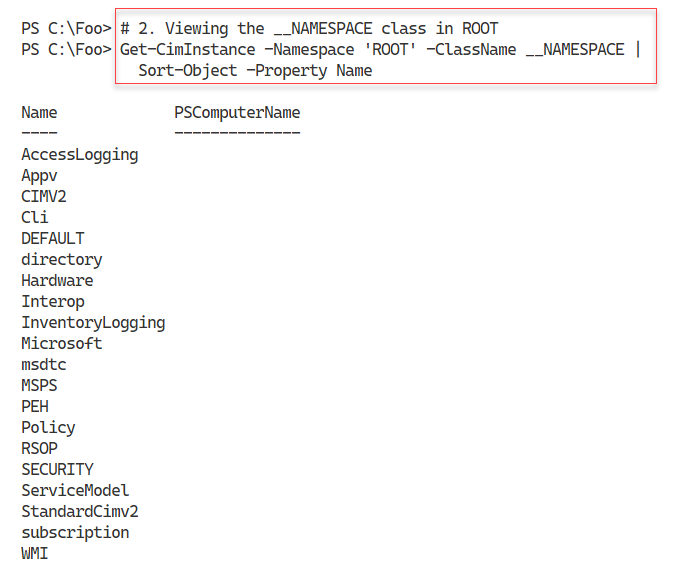


Figure 15.17: Examining the \_\_NAMESPACE class in the root namespace

**Insert image B42024\_15\_17.png**

With step 3, you get and count the classes in the ROOT\CIMV2 namespace, with output like this:

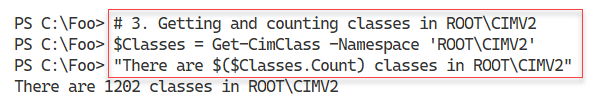


Figure 15.18: Counting the classes in the ROOT\CIMV2 namespace

**Insert image B42024\_15\_18.png**

In step 4, you define and then use a function to discover all the namespaces in WMI on this host, sorted alphabetically. The output of this step looks like this:

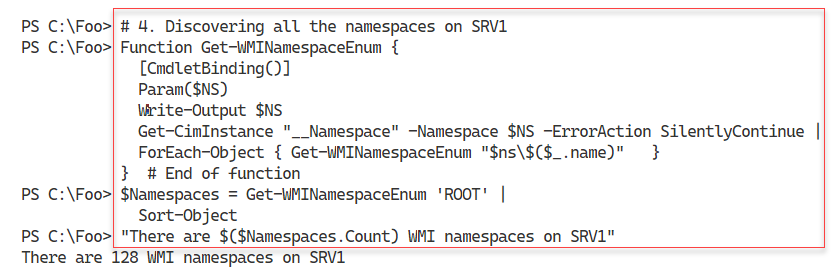


Figure 15.19: Discovering all the WMI namespaces on SRV1

**Insert image B42024\_15\_19.png**

In step 5, you view the first 25 namespace names, with output like this:

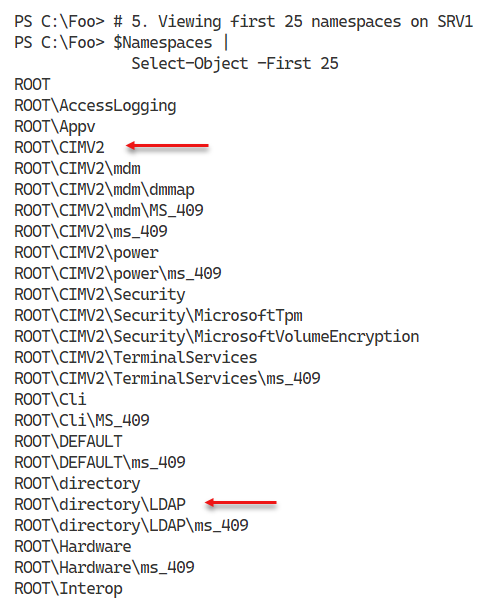


Figure 15.20: Listing the first 25 namespaces in WMI on SRV1

**Insert image B42024\_15\_20.png**

In step 6, you create a script bock that counts WMI namespaces and classes. This step generates no console output. In step 7, you run this function against SRV1 with output like this:

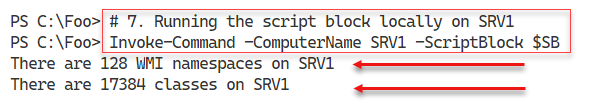


Figure 15.21: Counting WMI namespaces and classes on SRV1

**Insert image B42024\_15\_21.png**

In step 8, you run the script block on SRV2, with output like this:

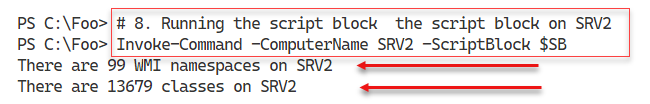


Figure 15.22: Counting WMI namespaces and classes on SRV2

**Insert image B42024\_15\_22.png**

In the final step, step 9, you run the script block on a domain controller, DC1. The output of this step is as follows:

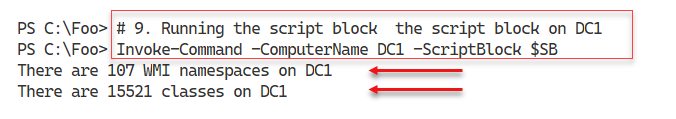


Figure 15.23: Counting WMI namespaces and classes on DC1

**Insert image B42024\_15\_23.png**

## There's more...

In step 2, you determine the child namespaces of the root namespaces. Each instance contains a string with the child's namespace name. The first entry is AccessLogging. Therefore the namespace name of this child namespace is ROOT\AccessLogging.

In step 3, you count the classes in ROOT\CIMV2. As mentioned before, not all of these classes are useful to an IT pro, although many are. You can use your search engine to find classes that might be useful.

In step 4, you define a recursive function. When you call this function, specifying the ROOT namespace, the function retrieves the child namespace names from the \_\_NAMESPACE class in the root namespace. Then, for each child's namespace of the root, the function calls itself with a child namespace name. Eventually, this function returns the names of every namespace in WMI. You then sort this alphabetically by namespace name. Note that you can sort the output of GET-WMINamespaceEnum without specifying a property – you are sorting on the contents of the strings returned from the function.

In step 5, you view some of the namespaces in WMI on SRV1. Two important namespaces are the ROOT\CIMV2 and the ROOT\directory\LDAP. The former contains classes provided by the Win32 WMI provider, containing details about software and hardware on your system, including the bios, the OS, files, and a lot more.

Step 7, step 8, and step 9 run the function (defined in step 6) remotely. These steps count and display a count of the namespaces and classes on all three systems. For this reason, you should expect that the number of classes and namespaces differs.

# Exploring WMI classes

A WMI class defines a WMI-managed object. All WMI classes live within a namespace and contain members that include properties, methods, and events. An example class is Win32\_Share which you find in the root\CIMV2 namespace. This class defines an SMB share on a Windows host. Within WMI, the Win32 provider implements this class (along with multiple other OS and host-related classes).

As mentioned, you typically use the SMB cmdlets to manage SMB shares (as discussed in Chapter 10, Managing Shared Data, including the Creating and securing SMB shares recipe). Likewise, you carry out most AD management activities using AD cmdlets rather than accessing the information via WMI. Nevertheless, you can do things with WMI, such as event handling, that can be very useful to the IT professional.

A WMI class contains one or more properties which are attribute of an occurrence of a WMI class. For example, the Win32\_Share class contains a Name property that holds the share name for that share. Each WMI property has a data type, such as integer or string. The Name that a WMI object can take when requested. The Win32\_Share class, for example, has a Create() method to create a new SMB share and a Delete() method to remove a specific share. A method can be dynamic (instance-based) or static (class-related). The Win32\_Share's Delete() method is a dynamic method you use to delete a particular SMB share. The Create() method is a static method that the class can perform to create a new SMB share.

## Getting ready

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

## How to do it...

1. Viewing the Win32\_Share class

Get-CimClass -ClassName Win32\_Share

1. Viewing Win32\_Share class properties

Get-CimClass -ClassName Win32\_Share |

  Select-Object -ExpandProperty CimClassProperties |

    Sort-Object -Property Name |

      Format-Table -Property Name, CimType

1. Getting methods of Win32\_Share class

Get-CimClass -ClassName Win32\_Share |

  Select-Object -ExpandProperty CimClassMethods

1. Getting classes in a non-default namespace

Get-CimClass -Namespace root\directory\LDAP |

  Where-Object CimClassName -match '^ds\_group'

1. Viewing the instances in the ds\_group class

Get-CimInstance -Namespace root\directory\LDAP -Classname 'DS\_Group' |

  Format-Table -Property DS\_name, DS\_Member

## How it works...

In step 1, you view a specific class, the Win32\_Share class, with output like this:

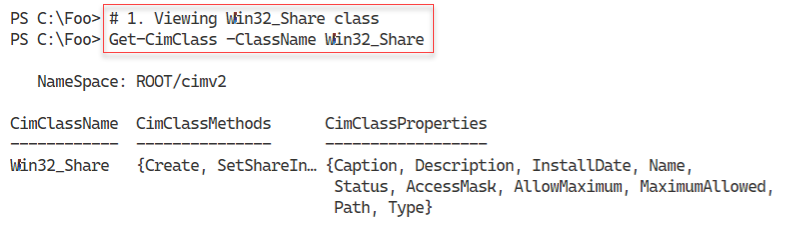


Figure 15.24: Viewing the Win32\_Share WMI class

**Insert image B42024\_15\_24.png**

In step 2, you view the properties of the Win32\_Share class. The output of this step looks like this:

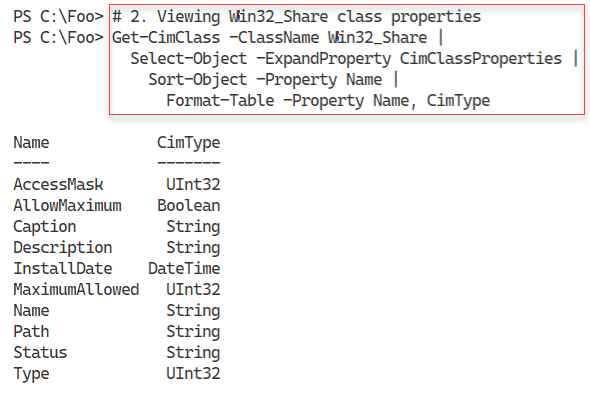


Figure 15.25: Viewing the Win32\_Share class properties

**Insert image B42024\_15\_25.png**

In step 3, you use the Get-CimClass cmdlet to view the methods available with the Win32\_Share class, with output like this:

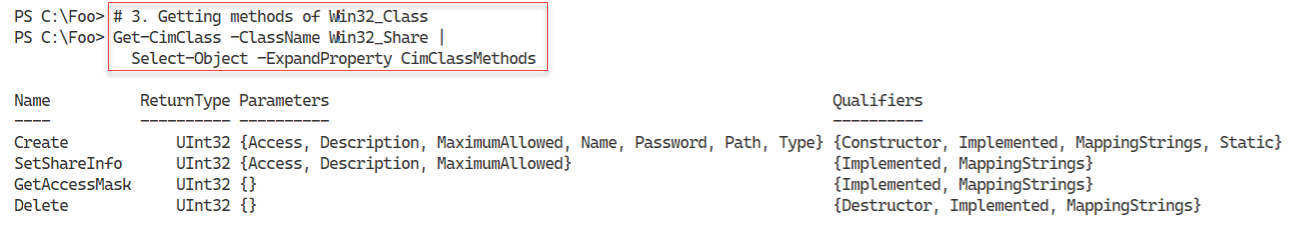


Figure 15.26: Viewing methods in the Win32\_Share class

**Insert image B42024\_15\_26.png**

In step 4, you get group-related classes in the ROOT\directory\LDAP namespace. The step returns just those classes that have the name of ds\_group. As you can see, this matches a few of the classes in this namespace, as follows:

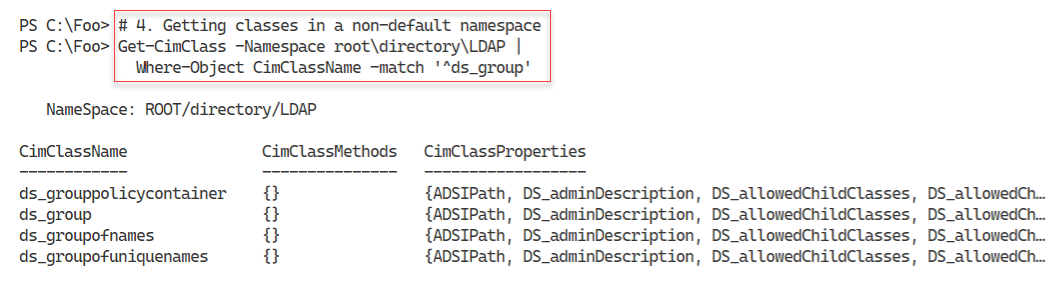


Figure 15.27: Finding AD Group related classes in the LDAP namespace

**Insert image B42024\_15\_27.png**

In step 5, you get the first twenty instances of the ds\_group WMI class. The output, shown here, includes both the AD group's name and the current members of each AD group. The output of this step looks like this:

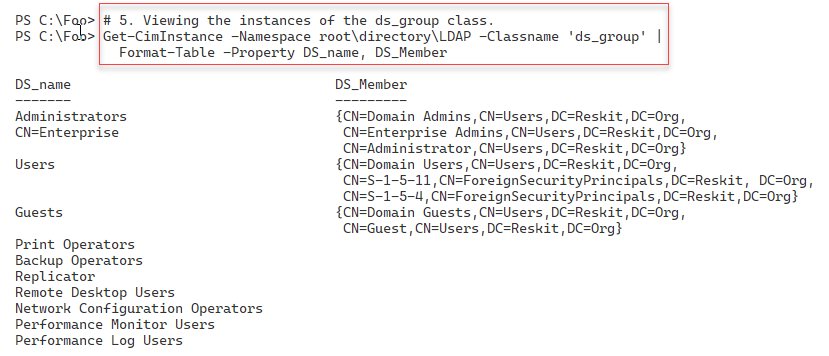


Figure 15.28: Finding AD Groups and members

**Insert image B42024\_15\_28.png**

## There's more...

In step 3, you view the methods in the Win32\_Share class using Get-CimClass. Since you did not specify a namespace, the WMI cmdlet assumes you are interested in the ROOT\CIV2 namespace. Note that, in this step, the Create() method has two important qualifiers – constructor and status. These two qualifiers tell you that you use the static Create() method to construct a new instance of this class (and a new SMB share). Also, you can see in the output that a provider. In this case, the Win32 provider has implemented all four of the methods shown. You may find that some classes have no implementation, although this is not common.

In step 5, you view the first instances in the ds\_group class. This class contains an instance for every group in the Reskit.Org domain. It contains more information for each group returned by your use of the Get-ADGroup cmdlet.

# Obtaining local and remote WMI objects

In the Exploring WMI classes recipe, you discovered that WMI provides a large number (over 100) of namespaces on each host along with thousands of WMI classes. You use the Get-Ciminstance cmdlet to return the instances of a WMI class on either the local or a remote host, as you can see in the recipe. This cmdlet returns the WMI instances for a specified WMI class wrapped in a .NET object.

With WMI, you have three ways you can use Get-CimInstance:

* The first way is to use the cmdlet to return all class occurrences and return all class properties.
* The second way is to use the -Filter parameter to specify a WMI filter. When used with Get-Ciminstance, the filter instructs the Get-CimInstance command to return some, and not all, instances of the desired class.
* The third method uses a WMI query. A WMI query is in effect a SQL statement that instructs WMI to return some (or all) properties of some (or all) occurrences of the specified WMI class.

When you use WMI across the network, specifying a filter or a full WMI query can reduce the amount of data transiting the wire and improve performance.

## Getting ready

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

## How to do it...

1. Using Get-CimInstance in the default namespace

Get-CimInstance -ClassName Win32\_Share

1. Getting WMI objects from a non-default namespace

$GCIMHT1 = @{

    Namespace = 'ROOT\directory\LDAP'

    ClassName = 'ds\_group'

}

Get-CimInstance @GCIMHT1 |

  Sort-Object -Property Name |

    Select-Object -First 10 |

      Format-Table -Property DS\_name, DS\_distinguishedName

1. Using a WMI filter

$Filter = "ds\_Name LIKE '%operator%' "

Get-CimInstance @GCIMHT1  -Filter $Filter |

  Format-Table -Property DS\_Name

1. Using a WMI query

$Q = @"

  SELECT \* from ds\_group

    WHERE ds\_Name like '%operator%'

"@

Get-CimInstance -Query $q -Namespace 'root\directory\LDAP' |

  Format-Table DS\_Name

1. Getting a WMI object from a remote system (DC1)

Get-CimInstance -CimSession DC1 -ClassName Win32\_ComputerSystem |

  Format-Table -AutoSize

## How it works...

In step 1, you use Get-CimInstance to return all the instances of the Win32\_Share class on SRV1, with output like this:

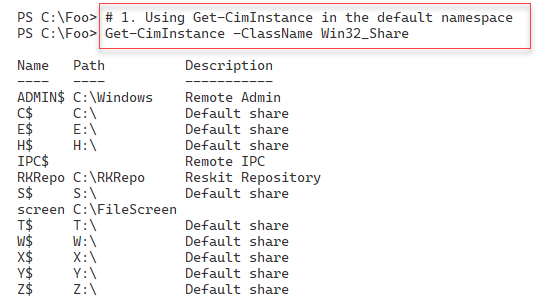


Figure 15.29: Retrieving Win32\_Share class instances on SRV1

**Insert image B42024\_15\_29.png**

In step 2, you use Get-CimInstance to retrieve instances of a class in a non-default namespace that you name explicitly, with output like this:

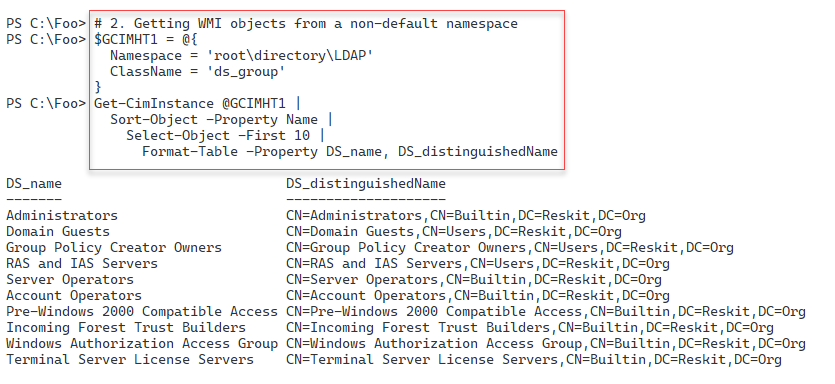


Figure 15.30: Retrieving WMI objects in an explicitly named namespace

**Insert image B42024\_15\_30.png**

In step 3, you use a WMI filter, specified with the -Filter parameter to the Get-CimInstance cmdlet. The output looks like this:

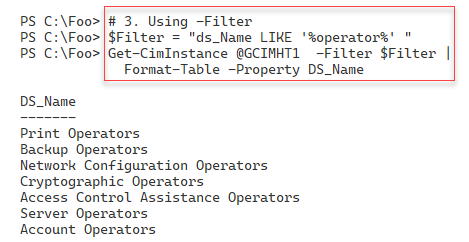


Figure 15.31: Retrieving WMI objects using a WMI filter

**Insert image B42024\_15\_31.png**

In step 4, you use a full WMI query that contains the namespace/class you wish to retrieve and details of which properties and which instances WMI should return.

In step 5, you retrieve a WMI object from a remote host, DC1. The class retrieved by this step, Win32\_ComputerSystem, holds details of the host such as hostname, domain name, and total physical memory, as you can see in the following output:

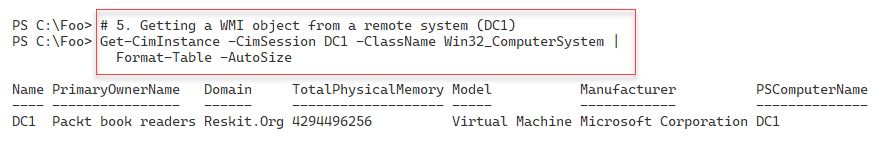


Figure 15.32: Retrieving WMI information from DC1

**Insert image B42024\_15\_32.png**

## There's more...

In step 4, you create a WMI query. This query returns all properties on any instance of the class whose name contains the characters "operator", using WMI's wildcard syntax. This query returns all properties in the groups that include printer operators and server operators, as you can see in the output from this step.

In step 5, you return details of the DC1 host. You used Get-Ciminstance to return the single occurrence of the Win32\_ComputerSystem class. The output shows that the DC1 host has 4 GB of memory. If you are using virtualization to implement this host, you may see a different value depending on how you configured the VM.

# Using WMI methods

In many object-oriented programming languages, a method is some action that an object can carry out. WMI also provides class methods. For example, the Win32\_Share class has a Delete() method to delete a given SMB share. The class also has the Create() static method that creates a new SMB share.

In many cases, WMI methods duplicate what you can do with other PowerShell cmdlets. You could, for example, use the New-SMBShare cmdlet to create a new SMB share rather than using the Create() static method of the Win32\_Share class.

As mentioned previously, WMI methods include instance methods and static methods. A dynamic or instance method operates on a specific instance – for example, deleting a specific SMB share. Classes also provide static methods, and these do not need a reference to any existing class instances. For example, you can use the Create() static method to create a new SMB share (and a new occurrence in the Win32\_Share class).

## Getting ready

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

## How to do it...

1. Reviewing methods of Win32\_Share class on SRV1

Get-CimClass -ClassName Win32\_Share |

  Select-Object -ExpandProperty CimClassMethods

1. Reviewing properties of Win32\_Share class

Get-CimClass -ClassName Win32\_Share |

  Select-Object -ExpandProperty CimClassProperties |

    Format-Table -Property Name, CimType

1. Creating a new SMB share using the Create() static method

$NSHT = @{

  Name        = 'TestShare1'

  Path        = 'C:\Foo'

  Description = 'Test Share'

  Type        = [uint32] 0 # disk

}

Invoke-CimMethod -ClassName Win32\_Share -MethodName Create -Arguments $NSHT

1. Viewing the new SMB share

Get-SMBShare -Name 'TestShare1'

1. Viewing the new SMB share using Get-CimInstance

Get-CimInstance -Class Win32\_Share -Filter "Name = 'TestShare1'"

1. Removing the share

Get-CimInstance -Class Win32\_Share -Filter "Name = 'TestShare1'" |

  Invoke-CimMethod -MethodName Delete

## How it works...

In step 1, you use Get-CimClass to retrieve and display the methods provided by the Win32\_Share WMI class. The output is as follows:

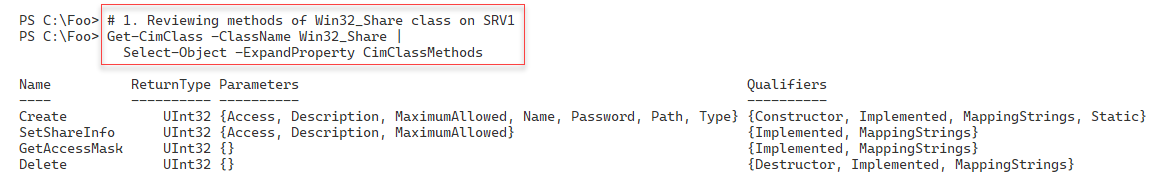


Figure 15.33: Reviewing the methods contained in the Win32\_Share WMI class

**Insert image B42024\_15\_33.png**

In step 2, you use the Get-Cimclass cmdlet to get the properties of each instance of the Win32\_Share class, producing the following:

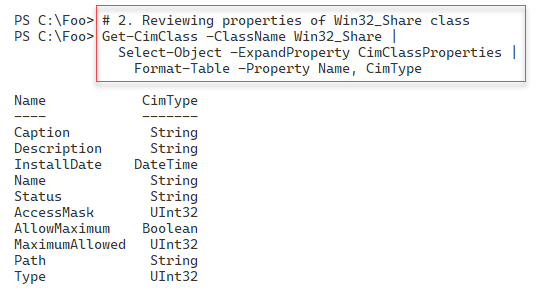


Figure 15.34: Reviewing the properties of an instance of the Win32\_Share class

**Insert image B42024\_15\_34.png**

With step 3, you use the Invoke-CimMethod cmdlet to invoke the Create() method of the Win32\_Share class and create a new SMB share on SRV1, with output like this:

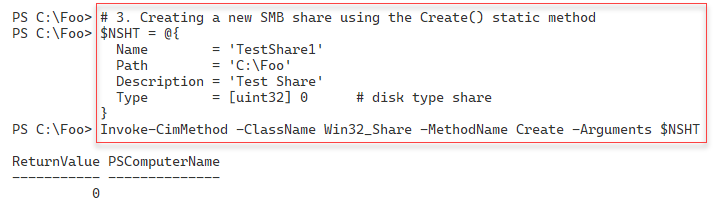


Figure 15.35: Creating a new SMB share using WMI

**Insert image B42024\_15\_35.png**

In step 4, you use the Get-SMBShare cmdlet to get the SMB share information for the share you created in the previous step, producing output like this:

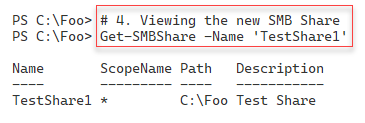


Figure 15.36: Viewing the newly created SMB share using Get-SMBShare

**Insert image B42024\_15\_36.png**

In step 5, you use Get-CimInstance to view the details of the share via WMI. This step produces the following output.

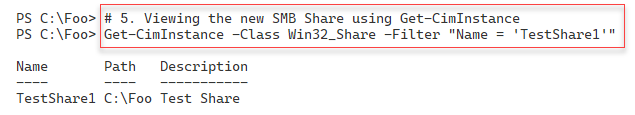


Figure 15.37: Viewing the newly created SMB share using Get-CimInstance

**Insert image B42024\_15\_37.png**

In the final step in this recipe, step 6, you use Invoke-CimMethod to delete a specific share (the one you created in step 3).

## There's more...

In step 3, you create a new SMB share using the Invoke-CimMethod cmdlet. This cmdlet takes a hash table containing the properties and property values for the new SMB share. The cmdlet returns an object containing a ReturnCode property. A return code of 0 indicates success – in this case, WMI created the new share. For other values of the return code, you need to consult the documentation. For the Win32\_Share class, you can find more online documentation at <https://docs.microsoft.com/en-us/windows/win32/cimwin32prov/create-method-in-class-win32-share>. This page shows the return codes that the Create() method could generate and what those return codes indicate. For example, a return code of 8 would indicate that you attempted to create a share whose name already exists. If you plan to use WMI in production scripting, consider testing for non-zero return codes and handling common errors gracefully.

In step 6, you use a WMI method, Delete(), to delete the previously created SMB share. You delete this share by first using the Get-CimInstance with a WMI filter to retrieve the share(s) to be deleted. You then pipe these share objects to the Invoke-CimMethod cmdlet and invoke the Delete() method on the instance passed in the pipeline. The approach taken in step 6 is a common way to remove WMI class instances of this class and, for that matter, any WMI class.

# Managing WMI events

A key feature of WMI is its event handling. There are thousands of events that can occur within a Windows system that might be of interest. For example, you might want to know if someone adds a new member to a high privilege AD group such as Enterprise Admins. You can tell WMI to notify you when such an event occurs, then take whatever action is appropriate. For example, you might just print out a lot of when the group membership changes occurred. You could even check a list of users who should be members of the group and take some action if the user added is not authorized.

Events are handled both by WMI itself and by WMI providers. WMI itself can signal an event should a change be detected in a CIM class – that is, any new, updated, or deleted class instance. You can detect changes too to entire classes or namespaces. WMI calls these events intrinsic events. One common intrinsic event would occur when you (or Windows) starts a new process and, by doing so, WMI adds a new instance to the Win32\_Process class (contained in the ROOT/CIMV2 namespace).

WMI providers can also implement events. These are known as extrinsic WMI events. The AD WMI provider, for example, implements an event that fires any time the membership of an AD group changes. The Windows registry provider also provides an extrinsic event that detects changes to the registry, such as a new registry key or an updated registry value.

To make use of WMI event management, you first create an event subscription. The event subscription tells WMI which event you want it to track. Additionally, you can define an event handler that tells WMI what you want to do if the event occurs. For example, if a new process starts, you may wish to display the event's details. If an AD group membership changes, you might want to check to see if any group members are not authorized and report the fact or possibly even delete the invalid group member.

For more details on how you can receive WMI events, see https://docs.microsoft.com//windows/win32/wmisdk/receiving-a-wmi-event.

For information around the types of events to receive, see https://docs.microsoft.com/windows/win32/wmisdk/determining-the-type-of-event-to-receive.

There are two types of WMI eventing you can utilize. In this recipe, you create and handle temporary WMI events which work within a PowerShell session. If you close a session, WMI stops tracking all the events you registered for in that session. In the Implementing permanent WMI eventing recipe, you look at creating event subscripts independent of the current PowerShell session.

When you register for a temporary event, you can provide WMI with a script block that you want WMI to execute when the event occurs. WMI executes this script block in the background, inside a PowerShell job.

When you register for a WMI event, PowerShell creates this job in which it runs the action script. As with all PowerShell jobs, you use Receive-Job to view any output generated by the script. If your script block contains Write-Host statements, PowerShell sends any output directly to the console (and not the background job). You can also register for a WMI event without specifying an action block. In that case, WMI queues the events, and you can use Get-WinEvent to retrieve the event details.

When WMI detects an event, it generates an event record that contains the details of the event. These event records can be useful in helping you with more details of the event, but the records are not a complete snapshot of the event. You can register for a WMI event should the membership of an AD group change and receive details such as the new member. However, the event record does not contain details of the user who modified the group's membership, or the IP address of the host used to effect the unauthorized change.

## Getting ready

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

## How to do it...

1. Registering an intrinsic event

$Query1 = "SELECT \* FROM \_\_InstanceCreationEvent WITHIN 2

          WHERE TargetInstance ISA 'Win32\_Process'"

$CEHT = @{

  Query            = $Query1

  SourceIdentifier = 'NewProcessEvent'

}

Register-CimIndicationEvent @CEHT

1. Running Notepad to trigger the event

notepad.exe

1. Getting the new process event

$Event = Get-Event -SourceIdentifier 'NewProcessEvent' |

           Select-Object -Last 1

1. Displaying event details

$Event.SourceEventArgs.NewEvent.TargetInstance

1. Unregistering the event

Unregister-Event -SourceIdentifier 'NewProcessEvent'

1. Registering an event query based on the registry provider

New-Item -Path 'HKLM:\SOFTWARE\Packt' | Out-Null

$Query2 = "SELECT \* FROM RegistryValueChangeEvent

            WHERE Hive='HKEY\_LOCAL\_MACHINE'

              AND KeyPath='SOFTWARE\\Packt' AND ValueName='MOLTUAE'"

$Action2 = {

  Write-Host -Object "Registry Value Change Event Occurred"

  $Global:RegEvent = $Event

}

Register-CimIndicationEvent -Query $Query2 -Action $Action2 -Source RegChange

1. Creating a new registry key and setting a value entry

$Q2HT = [ordered] @{

  Type  = 'DWord'

  Name  = 'MOLTUAE'

  Path  = 'HKLM:\Software\Packt'

  Value = 42

}

Set-ItemProperty @Q2HT

Get-ItemProperty -Path HKLM:\SOFTWARE\Packt

1. Unregistering the event

Unregister-Event -SourceIdentifier 'RegChange'

1. Examining details

$RegEvent.SourceEventArgs.NewEvent

1. Creating a WQL event query

$Group = 'Enterprise Admins'

$Query1 = @"

  Select \* From \_\_InstanceModificationEvent Within 5

   Where TargetInstance ISA 'ds\_group' AND

         TargetInstance.ds\_name = '$Group'

"@

1. Creating a temporary WMI event registration

$Event = @{

  Namespace =  'ROOT\directory\LDAP'

  SourceID  = 'DSGroupChange'

  Query     = $Query1

  Action    = {

    $Global:ADEvent = $Event

    Write-Host 'We have a group change'

  }

}

Register-CimIndicationEvent @Event

1. Adding a user to the enterprise admin group

Add-ADGroupMember -Identity 'Enterprise Admins' -Members Malcolm

1. Viewing the newly added user

$ADEvent.SourceEventArgs.NewEvent.TargetInstance |

  Format-Table -Property DS\_sAMAccountName,DS\_Member

1. Unregistering the event

Unregister-Event -SourceIdentifier 'DSGroupChange'

## How it works...

In step 1, you register for an intrinsic event that occurs whenever Windows starts a process. The registration does not include an action block. In step 2, you run Notepad.exe to trigger the event. In step 3, you use Get-WinEvent to retrieve details of the event. These three steps produce no console output.

In step 4, you view details of the process startup event, with output like this:

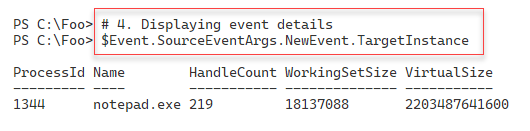


Figure 15.38: Displaying event details

**Insert image B42024\_15\_38.png**

In step 5, you de-register for the process start event. This step generates no output. In step 6, you register a new event subscription using a WMI query that targets the WMI provider, with output like this:

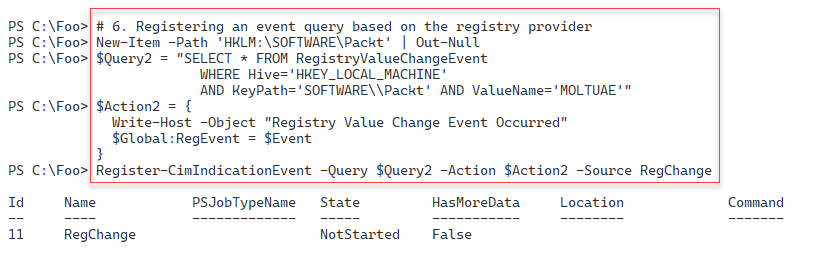


Figure 15.39: Registering for a registry provider based event

**Insert image B42024\_15\_39.png**

With the event registration complete, in step 7, you create a new registry key and set a registry key value to test the event subscription. The output of this step looks like this:

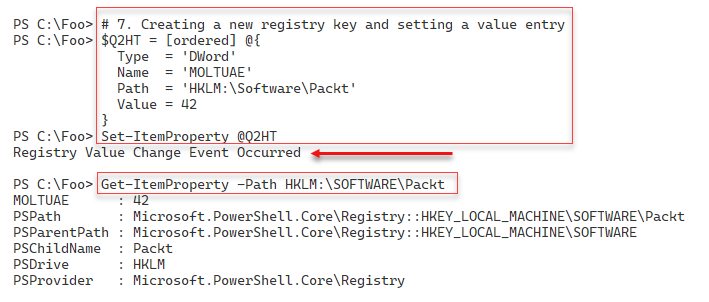


Figure 15.40: Invoking the WMI registry event

**Insert image B42024\_15\_40.png**

In step 8, you un-register the registry event to avoid more event handling and event output. In step 9, you examine the output WMI generated based on the registry changes you made in step 7. The event details look like this:

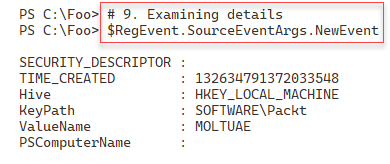


Figure 15.41: Examining a registry change WMI event

**Insert image B42024\_15\_41.png**

In step 10, you create a WQL query that captures changes to the Enterprise Admins AD group generating no output. In step 11, you use the query to create a temporary WMI event that fires when the group membership changes. The output looks like this:

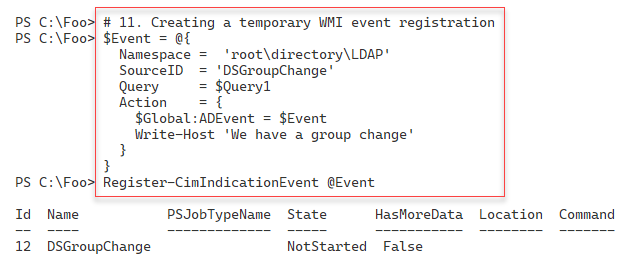


Figure 15.42: Creating a temporary WMI event registration

**Insert image B42024\_15\_42.png**

To test this new directory change event, in step 12, you add a user to the Enterprise Admins AD group, generating output like this:

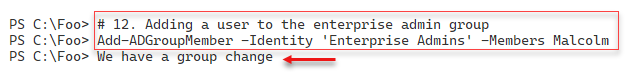


Figure 15.43: Triggering an AD group membership change event

**Insert image B42024\_15\_43.png**

In step 13, you examine the details of the event you generated in the previous step, with output like this:

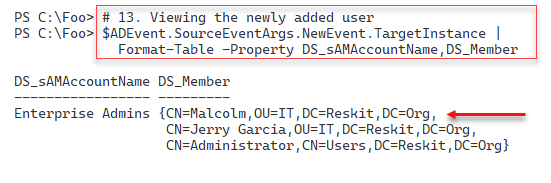


Figure 15.44: Examining AD group membership change event details

**Insert image B42024\_15\_44.png**

In the final step in this recipe, step 14, you unregister for the AD group membership change. This step generates no output.

## There's more...

In step 7, you test the registry event handling that includes an event action script block that you want PowerShell to execute when the event occurs. Since the script block you specified in step 6 contains a Write‑Host statement, you see the output on the PowerShell console.

In step 9, you examine the details WMI generated when the WMI registry change event occurred. As with other events, the event details omit potentially critical information. For example, the event does not tell you which user made this change or provide the new value of the registry value. You can examine the Windows Security event log to discover the user logged on that system (and therefore the user who made the change). And you can use Get-ItemProperty to determine the new value for this registry key property.

In step 14, you explicitly de-register the AD change event. As an alternative, you could have closed the current PowerShell session, removing the event subscriptions.

# Implementing permanent WMI eventing

In the Managing WMI events recipe, you used PowerShell's WMI event handling capability and used temporary event handling – the event handlers were active only as long as the PowerShell session was active and a user was logged into the host. In that recipe, you created an event subscription and handled the events as your system generated them. This temporary event handling is a great troubleshooting tool that works well as long as you are logged in and are running PowerShell.

WMI also provides permanent event handling. You configure WMI to subscribe and handle events as they occur without using an active and open session. With permanent event handling, you configure WMI to subscribe to a specific event, for instance, adding a new member to a high privilege AD group such as Enterprise Admins. You can also configure WMI to perform a predefined action when that event occurs, such as creating a report or sending an email message to report on the event if/when it occurs.

WMI in Windows defines several different types of permanent event consumers you can use to set a permanent event:

* Active Script Consumer: You use this to run a specific VBS script.
* Log File Consumer: This handler writes details of events to event log files.
* NT Event Log Consumer: This consumer writes event details into the Windows event log.
* SMTP Event Consumer: You can use this consumer to send an SMTP email message when an event occurs.
* Command Line Consumer: With this consumer, you can run a program, such as PowerShell 7, and pass a script filename. When the event occurs, the script has access to the event details and can do pretty much anything you can do in PowerShell.

Microsoft developed the Active Script consumer in the days of Visual Basic and VBS scripts. Unfortunately, this consumer does not support PowerShell scripts. The Command Line WMI permanent event handler, on the other hand, enables you to run any programs you wish when the event handler detects an event occurrence. In this recipe, you ask WMI to run pwsh.exe and to run a specific script file when the event fires.

Managing permanent event handling is similar to the temporary WMI events you explored in the Managing WMI events recipe. You tell WMI which event to trap and what to do when that event occurs. You add a WMI class occurrence to three WMI classes, as you see in the recipe, to implement a permanent event handler as follows:

* Define an event filter. The event filter specifies the specific event that WMI should handle. You do this by adding a new instance to the specific event class you want WMI to detect. This event filter is essentially the same as in the previous recipe, Managing WMI events.
* Define an event consumer. In this step, you define the action you want WMI to take when the event occurs.
* Bind the event filter and event consumer. With this step, you add a new occurrence to an event binding class. This occurrence directs WMI to take some action (invoke the event consumer) whenever WMI detects that a specific WMI event (specified in the event filter) has occurred.

The AD WMI provider implements a wide range of AD-related events to which you can subscribe. WMI namespaces typically contain specific event classes that can detect when anything changes within the namespace. The namespace ROOT/Directory/LDAP has a system class named \_\_InstanceModificationEvent. For a permanent event handler, you add an occurrence to this class.

A small word of caution is appropriate. You need to be very careful when working with WMI permanent event handling. A best practice is to understand how you remove the objects related to the permanent event handler. Note that unless you remove these records explicitly, WMI continues to monitor your host for the events, which can unnecessarily consume host resources.

This recipe also demonstrates a useful approach – creating PowerShell functions to display the event subscription and then to remove the subscription fully. Finally, be careful when changing an event filter's refresh time (specified in the WMI event filter). Decreasing the event refresh time can consume additional CPU and memory. For the most part, a refresh rate of once per second or even every 5 seconds is possibly overly excessive. Checking every 10 seconds is more than adequate for most WMI events.

## Getting ready

1. This recipe uses SRV1, a domain-joined host. You have installed PowerShell 7 and VS code on this host.

## How to do it...

1. Creating a list of valid users for the Enterprise Admins AD group

$OKUsersFile = 'C:\Foo\OKUsers.Txt'

$OKUsers  =  @'

Administrator

JerryG

'@

$OKUsers |

  Out-File -FilePath $OKUsersFile

1. Defining helper functions to get/remove permanent events

Function Get-WMIPE {

  '\*\*\* Event Filters Defined \*\*\*'

  Get-CimInstance -Namespace root\subscription -ClassName \_\_EventFilter  |

    Where-Object Name -eq "EventFilter1" |

     Format-Table Name, Query

  '\*\*\*Consumer Defined \*\*\*'

  $NS = 'ROOT\subscription'

  $CN = 'CommandLineEventConsumer'

  Get-CimInstance -Namespace $ns -Classname  $CN |

    Where-Object {$\_.name -eq "EventConsumer1"}  |

     Format-Table Name, Commandlinetemplate

  '\*\*\*Bindings Defined \*\*\*'

  Get-CimInstance -Namespace root\subscription -ClassName \_\_FilterToConsumerBinding |

    Where-Object -FilterScript {$\_.Filter.Name -eq "EventFilter1"} |

      Format-Table Filter, Consumer

}

Function Remove-WMIPE {

  Get-CimInstance -Namespace root\subscription \_\_EventFilter |

    Where-Object Name -eq "EventFilter1" |

      Remove-CimInstance

  Get-CimInstance -Namespace root\subscription CommandLineEventConsumer |

    Where-Object Name -eq 'EventConsumer1' |

      Remove-CimInstance

  Get-CimInstance -Namespace root\subscription \_\_FilterToConsumerBinding  |

    Where-Object -FilterScript {$\_.Filter.Name -eq 'EventFilter1'}   |

      Remove-CimInstance

}

1. Creating an event filter query

$Group = 'Enterprise Admins'

$Query = @"

  SELECT \* From \_\_InstanceModificationEvent Within 10

   WHERE TargetInstance ISA 'ds\_group' AND

         TargetInstance.ds\_name = '$Group'

"@

1. Creating the event filter

$Param = @{

  QueryLanguage =  'WQL'

  Query          =  $Query

  Name           =  "EventFilter1"

  EventNameSpace =  "root/directory/LDAP"

}

$IHT = @{

  ClassName = '\_\_EventFilter'

  Namespace = 'root/subscription'

  Property  = $Param

}

$InstanceFilter = New-CimInstance @IHT

1. Creating Monitor.ps1

$MONITOR = @'

$LogFile   = 'C:\Foo\Grouplog.Txt'

$Group     = 'Enterprise Admins'

"On:  [$(Get-Date)]  Group [$Group] was changed" |

  Out-File -Force $LogFile -Append -Encoding Ascii

$ADGM = Get-ADGroupMember -Identity $Group

# Display who's in the group

"Group Membership"

$ADGM | Format-Table Name, DistinguishedName |

  Out-File -Force $LogFile -Append  -Encoding Ascii

$OKUsers = Get-Content -Path C:\Foo\OKUsers.txt

# Look at who is not authorized

foreach ($User in $ADGM) {

  if ($User.SamAccountName -notin $OKUsers) {

    "Unauthorized user [$($User.SamAccountName)] added to $Group"  |

      Out-File -Force $LogFile -Append  -Encoding Ascii

  }

}

"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*`n`n" |

Out-File -Force $LogFile -Append -Encoding Ascii

'@

$MONITOR | Out-File -Path C:\Foo\Monitor.ps1

1. Creating a WMI event consumer

# The consumer runs PowerShell 7 to execute C:\Foo\Monitor.ps1

$CLT = 'Pwsh.exe -File C:\Foo\Monitor.ps1'

$Param =[ordered] @{

  Name                = 'EventConsumer1'

  CommandLineTemplate = $CLT

}

$ECHT = @{

  Namespace = 'root/subscription'

  ClassName = "CommandLineEventConsumer"

  Property  = $param

}

$InstanceConsumer = New-CimInstance @ECHT

1. Binding the filter and consumer

$Param = @{

  Filter   = [ref]$InstanceFilter

  Consumer = [ref]$InstanceConsumer

}

$IBHT = @{

  Namespace = 'root/subscription'

  ClassName = '\_\_FilterToConsumerBinding'

  Property  = $Param

}

$InstanceBinding = New-CimInstance   @IBHT

1. Viewing the event registration details

Get-WMIPE

1. Adding a user to the Enterprise Admins group

Add-ADGroupMember -Identity 'Enterprise admins' -Members Malcolm

1. Viewing the Grouplog.txt file

Get-Content -Path C:\Foo\Grouplog.txt

1. Tidying up

Remove-WMIPE # invoke this function you defined above

$RGMHT = @{

 Identity = 'Enterprise admins'

 Member   = 'Malcolm'

 Confirm  = $false

}

Remove-ADGroupMember @RGMHT

Get-WMIPE # ensure you have removed the event handling

## How it works...

In step 1, you create a text file containing the SAMAccountName of users you have specified should be a member of the Enterprise Admins group. In step 2, you create two helper functions to view and delete the WMI class instances that handle the event. In step 3, you create an event filter query which, in step 4, you add to WMI. These steps produce no ouput.

When the permanent WMI event occurs and the group membership changes, you want WMI to run a specific PowerShell script. In step 5, you create a file, C:\Foo\Monitor.ps1, creating no console output.

In step 6, you create a new event consumer, telling WMI to run the monitor script to detect the event. Then in step 7, you bind the event consumer and the event filter to complete setting up a permanent event handler. These two steps also produce no output.

In step 8, you use the Get-WMIPE function you defined in step 2 to view the event filter details. The output of this step is as follows:

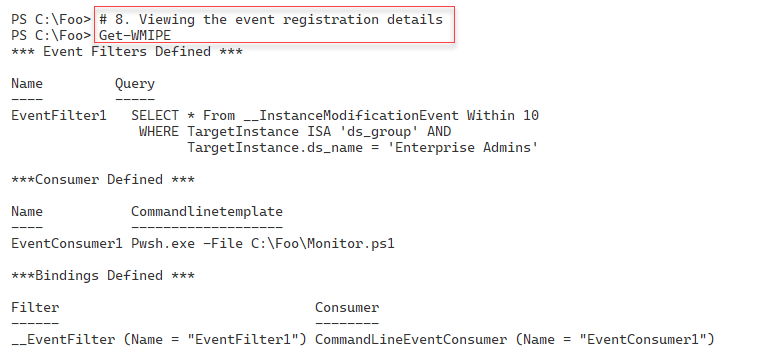


Figure 15.44: Examining details of an AD group membership change event

**Insert image B42024\_15\_44.png**

In step 9, you test the permanent event handling by adding a new user (Malcolm) to the Enterprise Admins group. This step does not generate console output because you added no Write-Host statements to Monitor.ps1.

In step 10, you view the Grouplog.txt file with output like this:

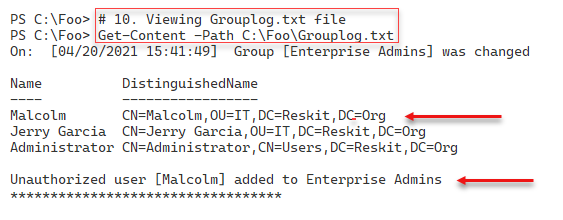


Figure 15.45: Viewing Grouplog.txt (generated by Monitor.ps1)

**Insert image B42024\_15\_45.png**

In the final step in this recipe, step 11, you tidy up and call the Remove-WMIPE function you defined in step 2 to remove the event details from WMI. At the end of this step, you run the Get-WMIPE function to ensure you have deleted all the event subscription class instances. The output of this step looks like this:

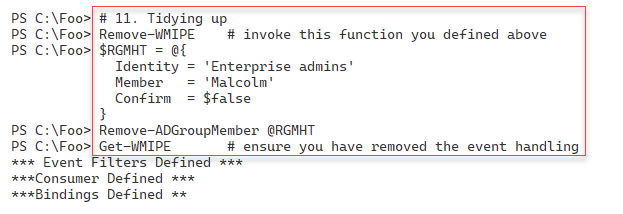


Figure 15.46: Tidying up

**Insert image B42024\_15\_46.png**

## There's more...

In step 5, you create a script that you want WMI to run any time the membership of the Enterprise Admins group changes. This script writes details to a text file (Grouplog.txt) containing the time the event occurred, the new membership, and whether this group now contains any unauthorized users. You could add to Monitor.ps1 to send an email to an administrative mailbox or just remove the unauthorized user. You could also look in the Windows Security event log to find the most recent user to log on to the server.

In this recipe, you create two helper functions, Get-WMIPE and Remove-WMIPE, to view and delete the permanent event handling details. You also call these functions at the end of the recipe to ensure you have not left the permanent event fully or partly configured.