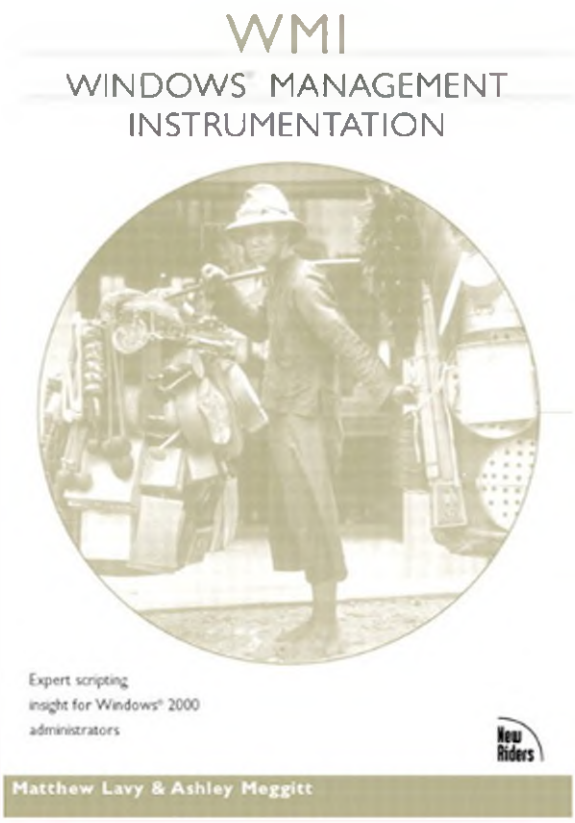
BOOK

Windows Management Instrumentation (WMI)

Write the first review

By Matthew M. Lavy, Ashley J. Meggitt



Continue

TIME TO COMPLETE:
11h 42m

TOPICS:
Windows

PUBLISHED BY:
Sams

PUBLICATION DATE:
October 2001

PRINT LENGTH:
432 pages

Windows Management Instrumentation (WMI) tells readers everything they need to know in order to harness the power of WMI to automate vast parts of Windows 2000 workstation and server management. The book starts with the absolute basics, teaching the fundamentals of WMI terminology and the VBScript programming language. By the time readers complete the book, the readers will not only have the ability to use the scripts supplied, but also have a good understanding of both WMI and VBScript.

Trend Micro, Inc.
EX1017-1



Windows Management Instrumentation (WMI)

Matthew M. Lavy, Ashley J. Meggitt

Published by Sams
Windows

5% complete

Approx. 11 hours left

What Readers Are Saying About Windows Management Instrumentation (WMI)

About The Authors

About The Technical Editors

Acknowledgments

Tell Us What You Think

Introduction

Introduction To WMI

8% complete

Using VBScript

Examining The Filesystem With WMI

Remote Administration

52% complete

Connecting to a Remote Machine

Handling Errors

Representation of Processes in WMI

Summary

WMI Security

Logs And Reports

Trend Micro, Inc.
EX1017-2

Representation of Processes in WMI

Having discussed the principles and techniques involved in implementing scripts that invoke WMI on remote machines, it seems appropriate to illustrate situations in which this can be useful, and develop some scripting techniques that can help with successful remote administration of multiple machines. As an example of a facet of a computer system that could usefully be managed remotely, we take a process—a program running on a remote computer.

Processes are represented in WMI by instances of the `Win32_Process` class. This is derived from `CIM_Process`, which in turn is derived from `CIM_LogicalElement`, which we have met several times already. The primary key of a `Win32_Process` object is its `Handle` property; this value represents a numeric identifier that the operating system assigns to a process when it is created. Every process running on a computer at any given time has a different handle; the operating system uses this number to uniquely identify each process. Hence, you can easily retrieve a WMI object representing a process whose ID you know using code such as this:

```
Set refExistingProcess = GetObject( _
                            "winmgmts:Win32_Process.Handle='1000'")
```

Superb! From a practical point of view, however, there is one slight problem: A system administrator is likely to want to get a reference to a `Win32_Process` object representing an instance of a specified application, but how do we find out its `ProcessID` in order to retrieve it? We have a boot-strapping problem! In common with all the subclasses of `CIM_LogicalElement` and, in fact, its superclass `CIM_ManagedSystemElement`, the `Win32_Process` class has a `Name` property. Unlike the filesystem objects discussed in the preceding chapter, the `Name` property of a `Win32_Process` is not unique and cannot be used in a moniker string. After all, there could be several instances of a single program running on a computer! The solution, of course, is to use a query to retrieve all instances of `Win32_Process`

Trend Micro, Inc.
EX1017-3

with a specified name. This is illustrated in the following fragment, which retrieves a collection of processes representing all running copies of Internet Explorer:

```
Set colProcesses = refWMI.ExecQuery( _
          "Select * from Win32_Process Where Name='IEXPLORE.exe'")
```

If you want to hone the collection further, you can draw on one of the many other properties of `Win32_Process`. For example, `CreationDate`, as the name implies, specifies the date a process first came into being. You can retrieve a collection of `Win32_Process` objects representing running copies of Internet Explorer that have been launched within the last 30 minutes with a code fragment such as this:

```
Set colProcesses = refWMI.ExecQuery( _
  "SELECT * FROM Win32_Process WHERE Name='iexplore.exe' AND " & _
  "CreationDate < '" & GetWMIDate(DateAdd("n",-30,Now),"+000") & "'"
```

Note that this fragment uses the `GetWMIDate()` function introduced in the preceding chapter.

Having retrieved a collection of `Win32_Process` objects, you can manipulate them within a `For Each...Next` loop in the usual way.

Manipulating Processes Remotely

The past few months have seen widespread press coverage (at least in the UK) of the playing of Quake on corporate networks. According to our newspapers, millions of dollars are wasted as employees spend time, money, and resources on Quake. Many companies apparently struggle to control its use. Thus, it seems appropriate to begin our discussion of process manipulation with a script that can terminate Quake sessions on a network!

The key to this is the `Win32_Process Terminate()` method, which, as the name implies, terminates the process on which it is called. This method, just like many of the others we have already encountered, returns an error code of 0 if successful or another number if the call fails. This method takes a single, optional parameter—an `Exit` code with which the process should terminate. This parameter is not used for most system administration purposes. The following short script example illustrates the use of the `Terminate()` method by killing all instances of Quake running on a local machine:

```
'killquake.vbs - terminate quake process on local machine
Option Explicit
On Error Resume Next
Dim refWMI
Dim strQuery
Dim colProcesses
Dim refItem

Set refWMI = GetObject("winMgmts:")
If Err <> 0 Then
    WScript.Echo "Could not connect to WMI"
    WScript.Quit
End If
StrQuery = "SELECT * FROM Win32_Process WHERE Name='quake.exe'"
Set colProcesses = refWMI.ExecQuery(strQuery)
For Each refItem In colProcesses
    If Err = 0 Then
        RefItem.Terminate()
    End If
Next
Set colProcesses = Nothing
Set refWMI = Nothing
```

To make this script truly useful, we need one more feature: the ability to run the script on whole groups of machines remotely without having to rewrite a line of the script for every machine! Next, we explore several ways in which this can be achieved.

Machine Names as Command-Line Arguments

Perhaps the easiest way to provide runtime information to a script is to give it command-line arguments. For example, if you could adapt the preceding script to read, say, a process name and a list of machines from the command line, you could create a much more useful tool. Because WSH has a built-in facility for handling command-line arguments, this is easy to accomplish.

Command-line arguments passed to WSH at runtime are available in the `Arguments` property of `WScript`. This property is in fact a `wshArguments` collection, each of whose elements is a whitespace-separated argument. For example, an invocation such as the following would start a script whose `WScript.Arguments` contains four items: `goose.exe`, `Erasmus`, `albatross`, and `romeo`:

```
c:>argtest.vbs goose.exe erasmus albatross romeo
```

Within this script, we can loop through the arguments with a `For Each...Next` loop, as in this code fragment:

```
For Each strItem in Wscript.Arguments
    Wscript.echo "Argument: " & strItem
Next
```

In common with all collections, elements of a `wshArguments` collection can be accessed individually via the `Item` property. This property is somewhat different from the ones you have met so far in that it behaves more like a method because it takes a parameter—an imply index that uniquely identifies the item you want to retrieve. This index is usually either a number or a text string. We have not used the `Item` property to retrieve individual members of collections so far because when WMI returns a collection, you do not necessarily know the index of the item you want to retrieve. In any event, iterating through a `For Each...Next` loop ensures that you never miss an object that you were not expecting! In the case of `wshArgument S`, however, the items are all indexed numerically: The first command-line

argument is stored in `WScript.Arguments.Item(0)`, the second in `WScript.Arguments.Item(1)`, and so on. Thus, if you know how many arguments there are, it is easy to address each directly without wasting time writing loops.

A second property of all collections, including `wshArguments`, is `Count`, which reports the number of items in the collection. Given that the first command-line argument passed to a script has an index of 0, the final argument has an index of `WScript.Arguments.Count - 1`. The following code fragment loops through each command-line argument again, this time using a `For` ... loop in conjunction with the `Count` property:

```
For i = 0 to WScript.Arguments.Count - 1
    WScript.Echo "Argument " & i & ":" & _
                 WScript.Arguments.Item(i)
Next
```

This fragment outputs lines such as `Argument 0: goose.exe`.

`Item` is the default property of a collection. This means that a line such as this:

```
WScript.Arguments.Item(i)
```

can be made slightly more concise:

```
WScript.Arguments(i)
```

We will use this second syntax from now on.

When you loop through all the command-line arguments, a combination of `Count` and `Item(index)` is no more convenient and a lot less concise than a `For Each...Next` loop. The direct index method becomes extremely useful,

however, when you want to treat arguments individually. For example, a sensible use of command-line arguments for the Quake-killing script would be to make the first argument the name of a process to kill and any number of remaining arguments a set of computers on which to kill it. This can be accomplished with the following code fragment:

```
strProcessName = WScript.Arguments(0)
For i = 1 to WScript.Arguments.Count - 1
    strMachine = WScript.Arguments(i)
    'Main body of script goes here
Next
```

This code generates an error if an attempt is made to execute it without supplying an argument. This, of course, can be handled using the `Err` object, as described earlier in this chapter. Alternatively, a more graceful approach would be to check the value of `WScript.Arguments.Count`; if it is 0, no arguments have been provided!

The following code shows a script that puts all of this together. It takes as its parameters a process name followed by any number of machine names. Its job is to terminate any instance of the named process found on any of the machines. To terminate all Quake sessions on a group of networked machines, you might consider using the script with a command line such as this:

```
c:\>terminate.vbs quake.exe finance1 finance2 finance3
```

This script works by attempting to connect to each named machine (provided as command-line arguments after the process name) and then to get a reference to any `Win32_Process` objects representing processes whose names match the first command-line argument (`quake.exe`). For each reference it finds, it calls the `terminate()` method to kill the represented process. Note that in addition to providing the upper bounds for the loop, `WScript.Arguments.Count` is used to perform some basic error-avoidance checks, as well as using the `Err` object where it is required.

```
'killit.vbs - terminate a named process on a set of computers
Option Explicit
On Error Resume Next
Dim refWMI
Dim colProcesses
Dim refProcess
Dim strMoniker
Dim strQuery
Dim i

'check to see that we have supplied at least two arguments
'if not, then print a usage message and quit
If WScript.Arguments.Count < 2 Then
    WScript.Echo "Usage: killit.vbs <process name> [<machine name>]"
    WScript.Quit
End If

'build the query string using the first command line argument
'this query will be used on all machines
strQuery = "select * from Win32_Process where Name='" __
           & WScript.Arguments(0) & "'"

'loop through each of the remaining command-line arguments
'these are assumed to be machine names (or, of course, IP addresses)
For i = 1 to WScript.Arguments.Count - 1
    'reset the Err object to ensure that any non-zero Err.Number comes
    'from this part of the code
    Err.Clear()
    'build a moniker string for the machine and try to get
    'an object reference
    strMoniker = "winmgmts:\\" & WScript.Arguments(i)
    set refWMI = GetObject(strMoniker)
    'did we successfully connect to the machine?
    If Err.Number <> 0 Then
        WScript.Echo "Failed to connect to " & WScript.Arguments(i)
    Else
        'good - we did. Get a collection of all relevant process objects
        Set colProcesses = refWMI.ExecQuery(strQuery)
        If colProcesses.Count=0 Then
            'none was found - we perform this check even though it is
            'not strictly necessary (For Each ... Next would not error out
```

```
            'on Count=0) because it allows us to print this friendly message
            WScript.Echo WScript.Arguments(0) & " - Not found on " __
            & WScript.Arguments(i)
        Else
            'we have some process objects, so loop through and terminate the
            For Each refProcess in colProcesses
                If refProcess.Terminate()=0 Then
                    WScript.Echo "(PID " & refProcess.Handle & ") " __
                                & refProcess.Name & " Terminated on - " __
                                & WScript.Arguments(i)
                Else
                    WScript.Echo "Unable to terminate - " __
                                & refProcess.Name & " on " __
                                & WScript.Arguments(i)
                End If
            Next
        End If
    End If
    Set colProcesses = Nothing
    Set refWMI = Nothing
Next
```

So, now you can terminate any process running on one or more machines on your network. Fantastic!

Although this technique is ideal for running a script that performs actions on a small, named group of machines, it is highly impractical for carrying out bulk activities on a large network. Using our script to stop Quake on several hundred machines would take an excessively long command line and would be extraordinarily unwieldy.

Multiple Machines as an IP Address Range

As you will recall from the start of this chapter, a moniker string will happily take an IP number in place of the machine name. Therefore, one possible approach to running a script on a large number of computers would be to loop through an entire IP address range, attempting to connect to each address in turn. You could do this by simply creating moniker strings that include an IP number instead of a machine name. The following script demonstrates how this can be done. The core is identical to the

one we just saw, but this time, the command-line parameters consist of the name of the process we want terminated, followed by two IP addresses, representing each end of an IP address range. The script will be invoked with a command line such as this:

```
c:>terminate_all.vbs quake.exe 192.168.18.1 192.168.20.254
```

This constructs a moniker string to connect to WMI on every IP address within the given range. Error handling here is, of course, essential, because it is highly likely that several of the IP numbers will not correspond to Windows 2000 boxes—or, in fact, anything at all!

The core of the script works in exactly the same way as the previous one did: It takes a given range and iterates its way through a loop until it has checked every machine in the desired range. There is one slight complication, however: An IP address is not a number as such, but a string representation of 4 bytes. An attempt to write code such as this:

```
For i = 192.168.18.1 to 192.168.20.254
    'do stuff
Next
```

is obviously nonsense and will result in an error—a type mismatch, to be precise! The solution to this problem is to convert the two IP addresses into true numeric values and loop through these values instead. Within the loop, each value can be converted back to an IP address string before being passed to WMI as part of a moniker. We accomplish this translation with two functions, `IPToNum()` and `NumToIP()`.

`IPToNum()` takes a string representation of an IP address and converts it into the 32-bit integer represented by that string. For example, `IPToNum(192.168.16.10)` and `IPToNum(192.168.16.20)` evaluate to 3232239626 and 3232239636, respectively. As you can see, these numbers are 10 apart, as are the addresses from which they were generated. In this form, how-

ever, we can easily loop through the entire range with reckless disregard for sub-net boundaries. If passed an illegal IP address, the function returns a nonzero error code that we can trap in the main script. The second function, `NumToIP()`, reverses this transformation and converts the IP number back to its dotted notation. Thus, as we cycle our way through the `For...Next` loop, we can use this function, along with the loops counter variable, to help generate a sane moniker string.

Here, then, is the full code listing. The main body of the script appears first, followed by the definitions of `IPToNum()` and `NumToIP()`:

```
'killitIP.vbs - terminate a named process on computers in a given
'IP address range
Option Explicit
On Error Resume Next

Dim colProcesses
Dim refProcess
Dim refWMI
Dim strMoniker
Dim strQuery
Dim numLow
Dim numHigh
Dim i

'check that we have the right number of command-line args
'if not, then quit with a usage message
If WScript.Arguments.Count <> 3 Then
    WScript.Echo "Usage: killitIP.vbs <process> <lower IP> <upper IP>"
    WScript.Quit
End If

'Convert the second and third command-line argument into numbers
'using IPToNum() and store their results
numLow = IPToNUm(WScript.Arguments(1))
numHigh = IPToNUm(WScript.Arguments(2))
'check that numLow and numHigh are valid (IPToNum() returns -1 if not)
If numLow < 0 Or numHigh < 0 Then
    WScript.Echo "Incorrect IP address range"
    WScript.Quit
End If
```

```
'check that numLow is smaller than numHigh!
If numLow > numHigh Then
    WScript.Echo "The second IP address must be higher than the first."
    WScript.Quit
End If
'Build query string based on process name provided as first cmd-line arg
strQuery = "SELECT * FROM Win32_Process WHERE " & _
           "Name='" & WScript.Arguments(0) & "'"
'loop through address range
For i = numLow To numHigh
    Err.Clear()
    'try to connect to machine with given address
    'NumToIP() converts the counter value back to a textual address
    strMoniker = "winmgmts:\\" & NumToIP(i)
    Set refWMI = GetObject(strMoniker)
    If Err.Number <> 0 Then
        WScript.Echo "Failed to connect to " && NumToIP(i)
    Else
        'get a collection of Win32_Process objects
        Set colProcesses = refWMI.ExecQuery(strQuery)
        If colProcesses.Count=0 Then
            WScript.Echo WScript.Arguments(0) & _
                         " - Not found on " & NumToIP(i)
        Else
            'there are some matching process, so loop through and kill
            For Each refProcess in colProcesses
                If refProcess.Terminate()=0 Then
                   WScript.Echo "(PID " & refProcess.Handle & ") " & _
                                refProcess.Name & _
                                " Terminated on - "& NumToIP(i)
                Else
                   WScript.Echo "Unable to terminate - " & _
                                refProcess.Name & " on " & NumToIP(i)
                End If
            Next
        End If
    End If
    Set colProcesses = Nothing
    Set refWMI = Nothing
Next


'IPToNum() function - turns a textual IP address into a 32-bit number
```

```
Function IPToNum(strIP)
    Dim numOctetsArray
    Dim i
    numOctetsArray = Split(strIP,".")

    'sanity checks
    If UBound(numOctetsArray) <> 3 Then
        'oops = wrong number of octets
        IPToNum = -1
        Exit Function
    End If
    For i = 0 to 3
        If Not IsNumeric(numOctetsArray(i)) Then
            'oops - not an IP address
            IPToNum = -2
            Exit Function
        End If
        If numOctetsArray(i) > 254 Then
            'oops - octet out of range
            IPToNum = -3
            Exit Function
        End If
    Next

    'now compile a number
    IPToNum = numOctetsArray(0) * (2^24)
    IPToNum = IPToNum + numOctetsArray(1) * (2^16)
    IPToNum = IPToNum + numOctetsArray(2) * (2^8)
    IPToNum = IPToNum + numOctetsArray(3)
End Function

'NumToIP() function - turns a 32-bit number into an IP address
Function NumToIP(numNum)
    Dim numOctetsArray(4)
    Dim param 'simulate passbyval
    Dim i
    param = numNum
    'don't bother with sanity checks here because
    'we are always passing something created by IPToNum
    numOctetsArray(0) = Int(param / (2^24))
    param = param - (numOctetsArray(0) * (2^24))
    numOctetsArray(1) = Int(param / (2^16))
    param = param - (numOctetsArray(1) * (2^16))
```

```
    numOctetsArray(2) = Int(param / (2^8))
    numOctetsArray(3) = param - (numOctetsArray(2) * (2^8))
    NumToIP = numOctetsArray(0) & "." & numOctetsArray(1) & _
              "." & numOctetsArray(2) & "." & numOctetsArray(3)
End Function
```

We hope you will agree that this technique for connecting to WMI on whole ranges of machines is really rather powerful!

Note: IPToNUM and NumToIP

`IPToNum()` starts by performing some basic sanity checks: it uses VBScript's builtin `Split()` function to turn the IP string into an array of segments, delimited by .. It then uses the built-in `UBound()` function to check the size of the array and `IsNumeric()` to check that each array item really is a number. Having carried out these checks, it converts the four numbers representing each octet of the address into a single 32-bit integer. This is done, effectively, by considering the array as a four-digit number in base 256: the "column" on the right represents units, the second from the right "256es", the third from the right "256-times-256es," and the left column "256-times-256-times-256es". Each array element is multiplied by the requisite number of 256es before the total is added together to produce a single number.

`NumToIP()` performs the reverse process. It does not carry out any error checking, because in the context of the script, it knows for sure that any number passed to it is a valid IP address generated by `IPToNum()`. It first splits the single number into an array of four "digits" in base 256, using a combination of division and subtraction. It then concatenates the array into a dot-separated string.

Specifying Machines in a File

There are, of course, many ways to get a job done, and finding ways to pass machine names to scripts is no exception. Before leaving this topic, however, we will demonstrate one final approach—reading a list of machine names (or IP addresses) from a text file.

VBScript includes support for reading and writing files not through a built-in function but via the `FileSystemObject`. A `FileSystemObject` object can be created within a script using the built-in `CreateObject()` function. This function is similar to the `GetObject()` that you have seen several times before, but with one important difference: Whereas `GetObject()` is used to retrieve a reference to an existing object (or one that is transparently created on-demand by a DCOM server), `CreateObject()` asks an application to explicitly create an object on the client's behalf. The syntax for `CreateObject()` is as follows:

```
Set refObject = CreateObject("<application>.<classname>.<version>")
```

If you either do not know or do not care about the specific version, a shorter syntax uses the default version on a system:

```
Set refObject = CreateObject("<application>.<classname>")
```

`<application>` is the registered name of the COM server providing the object (the registered name of the executable file or library containing the code that defines and implements the class in question). `<classname>` is the name of the class of object to create. The registered name of the COM server providing the `FileSystemObject` class is `Scripting`, so an object of this type is created with a code fragment such as this:

```
Set refFSO = CreateObject("Scripting.FileSystemObject")
```

Only one instance of this object can be created at any one time from within any given script. This makes sense, of course, because the `FileSystemObject` represents the filesystem within a script, and there is always only one filesystem!

In order to read the contents of a text file, we first use the `OpenTextFile()` method of a `FileSystemObject` to retrieve a reference to a `TextStream`. The

`ReadLine()` method of the `TextStream` can be used to read the file one line at a time.

`OpenTextFile()` requires a single parameter—the name of the file to open. This can be either a fully-qualified path and filename or the name of a file in the working directory. In addition, there are three optional parameters. The first specifies whether the file should be opened for reading, writing, or appending (values 1, 2, and 8, respectively); the second is a flag indicating whether a nonexistent file should be created; the third specifies whether the file should be opened in Unicode or ASCII mode. Given that we are interested only in reading here, we can ignore all but the first optional parameter. Thus, the code fragment required to open a text file for reading given a `FileSystemObject` reference is

```
Set refInputFile = refFSO.OpenTextFile("input.txt", 1)
```

Having retrieved a reference to a `TextStream`, lines can be read in turn with a call to `ReadLine()`. This method, which takes no parameters, returns a string corresponding to the first unread line in a file; each time it is called, it returns the next line. Thus, reading each line of a file in turn into a string variable simply requires multiple calls to the method.

An attempt to call `ReadLine()` when there are no more lines to read results, unsurprisingly, in an error! There is never a need to handle the error, though, because a `TextStream` has an `AtEndOfStream` property that can be read prior to a call to `ReadLine()` to check whether there are any lines left to read:

```
If Not refStream.AtEndOfStream Then strLine = refStream.ReadLine()
```

In order to loop through a file, reading each line in turn, we can place the call to `ReadLine()` within a `While...Wend` loop whose condition is `Not refStream.AtEndOfStream`. This is all illustrated in the following script, which

opens a command-line-specified file, reads computer names from it line by line, and creates a WMI moniker string from each:

```
'killitfile.vbs - terminate a named process on all machines whose names
'are listed in a given file
Option Explicit
On Error Resume Next
Dim refFSO
Dim refInputFile
Dim strMoniker

'Check that an argument has been provided
If WScript.Arguments.Count < 1 Then
    WScript.Echo "You must provide a filename!"
    WScript.Quit
End If
'create a FileSystemObject object
Set refFSO = CreateObject("Scripting.FileSystemObject")
Set refInputFile = refFSO.OpenTextFile(WScript.Arguments(0),1)
If Err<>0 then
    WScript.Echo "Could not find the file " & WScript.Arguments(0)
Else
    While Not refInputFile.AtEndOfStream
        StrMoniker = "winMgmts:\\" & refInputFile.ReadLine()
        Wscript.echo strMoniker
        'do something useful in here
    Wend
    refInputFile.Close
    Set refInputFile = Nothing
End If
Set refFSO = Nothing
```

We leave it to your imagination to work out how a script structure like this can be used in conjunction with some of the examples you have already seen to perform remote administration on a set of computers!

Note that before setting `refInputFile` to `Nothing`, we sneaked in a new method called `Close()`, which tells the underlying operating system that

we no longer need a handle to the file, so it can sever our links with it and allocate any used resources to someone else.

Creating Processes

Having demonstrated at length techniques that can be used to terminate processes running on remote machines, it seemed appropriate to discuss briefly the converse activity—namely, creating new processes on remote machines. As you might expect by now, the procedure for doing this is not at all complex, but it is conceptually different from anything you have encountered so far in this book.

All the methods we have applied thus far in the book work on instances of a given class. For example, the `Terminate()` method of `Win32_Process` destroys the instance upon which it is called. To create a new process, however, you have to use a static method—one that is called on the class itself. The reason is obvious: Before a process is created, there is no `Win32_Process` object that represents it! The method in question is `Create()`, and in order to call it, we need a reference to an object representing the `Win32_Process` class itself.

Retrieving such a reference is simple: It merely involves a call to `GetObject()` or the `Get()` method of a `SWbemServices` object, specifying a class object path rather than an instance object path (see **Chapter 1**, "Introduction to WMI"). A class object path looks exactly the same as an instance object path, except that it does not include the `Key=value` part that uniquely identifies an instance of a class. An object representing the `Win32_Process` class, then, can be retrieved with the following code fragment:

```
Set refProcessClass = GetObject("winMgmts:Win32_Process")
```

Once retrieved, `Create()` can be called just like any other method. It takes four parameters: the executable name of the program to launch, the startup directory, a reference to a `STARTUPINFO` structure, and a numeric variable whose value is unimportant.

The meaning of the first two parameters is self-explanatory. The first is simply the name of the program (including the fully-qualified path), and the second is the fully-qualified path of a directory that should be used as the program's working directory. This second parameter can safely be set to `null`, which means "no specified value." In this case, the new process will have the same working directory as the caller (our script).

Currently, the third parameter of the `Create()` method is not described by the Microsoft MSDN documentation. In fact, its existence is not mentioned at all! In the absence of any indication to the contrary, we assume that, in common with the Win32 API `CreateProcess()` function, it is used to specify the size and shape of the main window of the new process. It can safely be set to `null`, which is what we advise until the documentation is fixed.

The final parameter deserves further explanation. It is rather different from anything we have met so far, because it is an "out" parameter. Rather than being used to provide information to the method, it is used by the method to provide information to the caller. Specifically, if the method executes successfully and starts a new process, this variable contains the process ID (handle) of the newly created process. This information is invaluable, because it allows us to retrieve a reference to a `Win32_Process` object representing the newly created process. Why doesn't `Create()` simply return the process ID? Because, in common with many other WMI methods that we have already encountered, it uses the return value to specify an error code. It returns 0 for success and any other number for failure.

The following script illustrates the creation of a new process and the subsequent retrieval of the `Win32_Process` object that represents it. For the purposes of conciseness, it does not loop through a list of machines, creating series of processes based on complex criteria; instead, it merely launches a copy of Internet Explorer on the local machine. We hope it is now clear how this script could be combined with the techniques presented earlier to create processes on remote machines.

```
'createproc.vbs - launch IExplore.exe
Option Explicit
On Error Resume Next
Dim refProcClass
Dim numProcID
Dim refIEProcess

Set refProcClass = GetObject("winMgmts:Win32_Process")
If Err <> 0 Then
    WScript.echo "Could not connect to WMI"
    WScript.quit
End If
If refProcClass.Create( _
"C:\Program Files\Internet Explorer\iexplore.exe", _
null,null,numProcID) = 0 Then
    set refIEProcess = GetObject("winMgmts:Win32_Process.Handle='" _
                        & numProcID & "'")
Else
    WScript.Echo "Could not launch IE"
End If
Set refProcClass = Nothing
'now we can test that refIEProcess is a valid reference and use it
set refIEProcess = Nothing
```

For the record, we should point out that creating instances of Internet Explorer on a local machine is not a suitable task for WMI; this sort of task is better done through Windows Explorer. Creating instances of Internet Explorer on a remote machine is probably even more stupid in most situations, and certainly rather confusing for a user sitting at the remote machine: they will suddenly find an Internet Explorer window appearing on the desktop for no apparent reason! In fact, such a situation has far more serious ramifications than mere confusion at a user's desktop; it can also pose a rather nasty security risk. Thanks to DCOM impersonation (see **Chapter 5** - "WMI Security") a remote application will run with the identity and permissions of the person who launched it (for example, you); this means that the Internet Explorer window that randomly pops up on the remote user's screen allows him to open files and run

code with your credentials. If you do not think that this sounds particularly serious, consider the implications if instead of Internet Explorer you launch a command-shell on a remote machine!

Note: GUI-Based Remote Processes Persist Across Logins

If you launch a GUI-based application remotely, it will remain on the remote user's desktop until explicitly killed. Unlike programs that are launched locally, it will persist even after the user has logged out; in other words, it will still be there next time the user logs in and, more alarmingly, will appear on the desktop of any other user who logs on to the remote machine. Microsoft are aware of this problem; it is due to be fixed in Service Pack 3.

Despite the dangers of running GUI-based applications on remotely, there are plenty of situations in which the ability to launch processes remotely is invaluable. For example, you might want to launch a network analyzer program on a remote machine, or start up a server application such as Apache; if the application runs in the background and is not GUI-based, it does not pose the security risk outlined above. There are even times when launching a GUI-based application remotely can be useful; you may actually want to allow a user to run a specific application with your credentials. We cannot say for sure under what circumstances you will need to launch remote processes on your particular network, but we can say with utter certainty that the ability to do so is an extremely valuable tool in a system administrator's armory.

Win32_Process Associators

`Win32_Process` objects are involved in associations with two other classes of object— `CIM_DataFile` and `Win32_ComputerSystem`. The associators that define these relationships are `CIM_ProcessExecutable` and `Win32_SystemProcesses`, respectively.

`Win32_SystemProcesses` is a subclass of `CIM_SystemComponent` and ultimately of `CIM_Component`. As the name implies, it associates a process (PartComponent) with the machine upon which it is running

(GroupComponent). If you already have a reference to a `Win32_Process` object, the `Win32_SystemProcesses` associator is unlikely to be very useful, because it tells you nothing about a process that you do not already know or could easily find out through other means! It could, however, be used as a mechanism for finding a list of all |processes running on a given computer. Whether a query such as

```
ASSOCIATORS OF {Win32_ComputerSystem} WHERE
         AssocClass=Win32_SystemProcesses
```

is actually better or more convenient than

```
SELECT * FROM Win32_Process
```

is a moot point, but as we have said many times, with WMI there is always more than one way!

`CIM_ProcessExecutable` connects a process with `CIM_DataFile` objects representing the program itself and any libraries with which it links. `CIM_ProcessExecutable` is a subclass of `CIM_Dependency`. It attaches a `Win32_Process` (referenced in a `Dependent` property) to a `CIM_DataFile` (referenced in an `Antecedent` property). We will end this chapter with a small script that illustrates its use. This script takes two command-line options —a machine name and the name of a process. It prints a list of all executable files upon which the process depends.

```
'whatuses.vbs - list the program files and libraries
'referenced by a given process on a given machine
Option Explicit
On Error Resume Next
Dim refWMI
Dim colProcesses
Dim refProcess
Dim colFiles
Dim refFile
```

```
If WScript.Arguments.Count <> 2 Then
    WScript.Echo "Usage: whatuses.vbs <machine> <process name>"
    WScript.Quit
End If

set refWMI = GetObject("winMgmts:\\" & WScript.Arguments(0))
If Err <> 0 Then
    WScript.Echo "Could not connect to WMI on " & WScript.Arguments(0)
    WScript.Quit
End If

set colProcesses = refWMI.ExecQuery( _
                "SELECT * FROM Win32_Process WHERE Name='" & _
                                    wscript.arguments(1) & "'")

For Each refProcess in colProcesses
    WScript.Echo refProcess.Name & "(" & refProcess.Handle & ") depends on:"
    set colFiles = refWMI.ExecQuery(_
                  "ASSOCIATORS OF {Win32_Process='" & _
                  refProcess.handle & "'} WHERE " & _
                  "AssocClass=CIM_ProcessExecutable ")
    For Each refFile in colFiles
        WScript.Echo refFile.Name
    Next
Next

set colFiles = Nothing
set colProcesses = Nothing
set refWMI = Nothing
```

Open a copy of Internet Explorer on your machine, and run the script with your machine name and `iexplore.exe` as command-line arguments. You might be surprised at just how many shared libraries it takes to get IE off the ground!