BOHOPS

A blog about cybersecurity research, education, and news

WRITTEN BY BOHOPS

MAY 12, 2020

WS-MANAGEMENT COM: ANOTHER APPROACH FOR WINRM LATERAL MOVEMENT

QUICK LINKS

- Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence (Part 2)
- Abusing .NET Core CLR Diagnostic Features (+ CVE-2023-33127)
- Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence

INTRODUCTION

Lateral movement techniques in the wonderful world of enterprise Windows are quite finite. There are only so many techniques and variations of those techniques that attackers use to execute remote commands and payloads. With the rise of PowerShell well over a decade ago, most ethical hackers may agree that Windows Remote

- Abusing the COM Registry Structure (Part 2): Hijacking & Loading Techniques
- Abusing the COM Registry Structure: CLSID, LocalServer32, & InprocServer32
- Analyzing and Detecting a VMTools Persistence Technique
- Investigating .NET CLR Usage Log Tampering Techniques For EDR Evasion (Part 2)
- WS-Management COM: Another Approach for WinRM Lateral Movement
- Executing Commands and Bypassing AppLocker with PowerShell Diagnostic Scripts
- Abusing Exported Functions and Exposed DCOM Interfaces for Pass-Thru Command Execution and Lateral Movement

Management (WinRM) became a major of part of their "lateral movement toolkit" when the right (privileged) credential or identity was captured. With "remote" cmdlets like *Invoke-Command*, **-PSSession*, and **-CimSession*, ethical hackers rode the WinRM wave because PowerShell made it that much easier to do so.

PowerShell certainly still has its uses cases in today's climate. However, the rise of better detection optics and enhanced visibility in version 5+ have made PowerShell less appealing for post-exploitation. Furthermore, modern tooling has shifted away from PowerShell to managed .NET and (back to) unmanaged C/C++. Although the offensive trends are shifting, WinRM can still a viable option (at least, in my opinion).

In this post, we will make a valiant effort to decouple WinRM from PowerShell and take a look at a few other tools that leverage WinRM for remote command execution and lateral movement. Additionally, we will showcase how we can leverage *WSMAN.Automation*, a very interesting COM object, to run remote commands over WinRM transport. To accomplish this, we will walk through the process of building a simple proof-of-concept .NET C# tool.

Let's get started…

A BRIEF OVERVIEW OF WINRM

Windows Remote Management (WinRM) "is the Microsoft implementation of WS-Management Protocol (Web Services for

Management aka WSMan), a standard Simple Object Access Protocol (SOAP)-based, firewall-friendly protocol that allows hardware and operating systems, from different vendors, to interoperate" (Microsoft Docs).

As an alternative to DCOM and WMI for remote management, WinRM is used to establish sessions with remote computers over WSMan, which leverages HTTP/S as transport mechanism to deliver XML formatted messages. In modern Windows systems, WinRM HTTP communication occurs over TCP port 5985 and HTTPS (TLS) communication occurs over TCP port 5986. WinRM supports NTLM and Kerberos (domain) authentication natively. After initial authentication, the WinRM sessions are protected with AES encryption (Microsoft Docs).

Note: The WinRM service must be configured and running in order to accept remote connections. This can be setup quickly with the *winrm.cmd quickconfig* command or through Group Policy. A few more steps may be required for WinRM to accept connections. Please see this Pentest Lab article for more info.

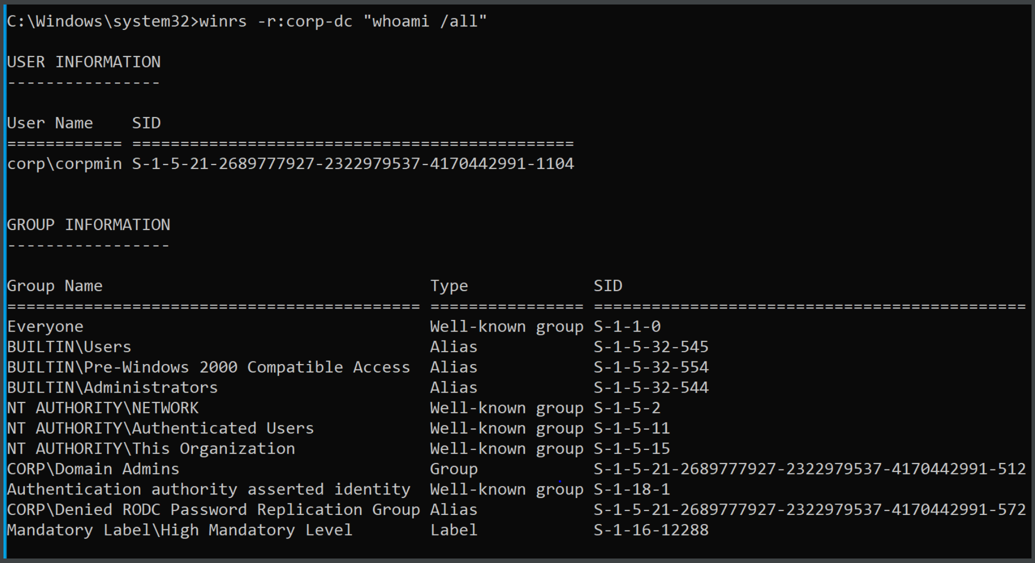
WINRM TOOLS & CAPABILITIES

Let's take a quick look at a few WinRM capabilities (outside of PowerShell):

WINRS.EXE

Winrs.exe is a built-in command line tool that allows for the execution of remote commands over WinRm with a properly credentialed user. The command supports a variety of switches as well as the ability to use alternate credentials for authentication. Example command usage is as follows:

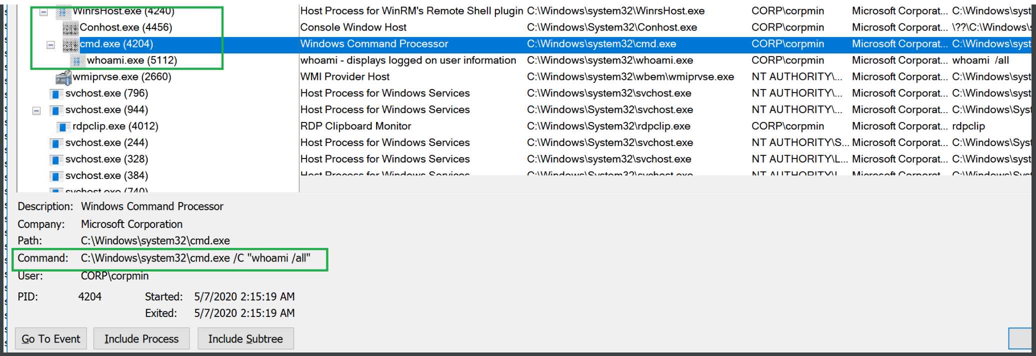
```
winrs -r:corp-dc "whoami /all"
```



```
C:\Windows\system32>winrs -r:corp-dc "whoami /all"

USER INFORMATION
----------------

User Name    SID
============ ==============================================
corp\corpmin S-1-5-21-2689777927-2322979537-4170442991-1104


GROUP INFORMATION
-----------------

Group Name                                 Type             SID
========================================== ================ ==============================================
Everyone                                   Well-known group S-1-1-0
BUILTIN\Users                              Alias            S-1-5-32-545
BUILTIN\Pre-Windows 2000 Compatible Access Alias            S-1-5-32-554
BUILTIN\Administrators                     Alias            S-1-5-32-544
NT AUTHORITY\NETWORK                       Well-known group S-1-5-2
NT AUTHORITY\Authenticated Users           Well-known group S-1-5-11
NT AUTHORITY\This Organization             Well-known group S-1-5-15
CORP\Domain Admins                         Group            S-1-5-21-2689777927-2322979537-4170442991-512
Authentication authority asserted identity Well-known group S-1-18-1
CORP\Denied RODC Password Replication Group Alias           S-1-5-21-2689777927-2322979537-4170442991-572
Mandatory Label\High Mandatory Level       Label            S-1-16-12288
```

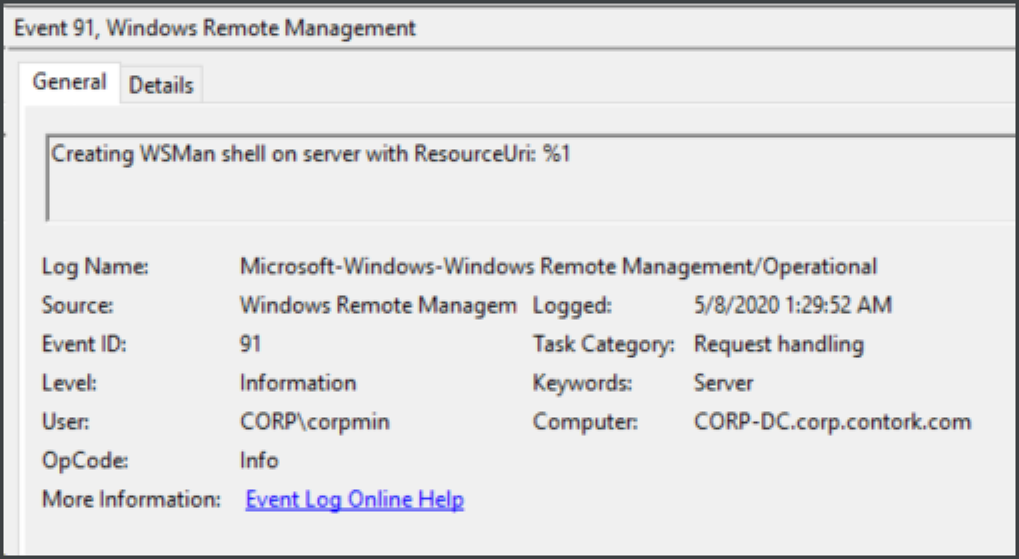
Although the tool offers an easy way to invoke remote commands, detection opportunities are relatively trivial. As Matt Graeber (@mattifestation) points out in this Tweet, the remote process chain for successful command execution is as follows:

```
svchost.exe (DcomLaunch)-> winrshost.exe -> cmd.exe [/c remote command] -> [remote command/binary]
```



```
WinrsHost.exe (4240)      Host Process for WinRM's Remote Shell plugin  C:\Windows\system32\WinrsHost.exe          CORP\corpmin      Microsoft Corporat... C:\Windows\syst
  Conhost.exe (4456)      Console Window Host                           C:\Windows\System32\Conhost.exe            CORP\corpmin      Microsoft Corporat... \??\C:\Windows\
  cmd.exe (4204)          Windows Command Processor                     C:\Windows\system32\cmd.exe                CORP\corpmin      Microsoft Corporat... C:\Windows\syst
    whoami.exe (5112)     whoami - displays logged on user information  C:\Windows\system32\whoami.exe             CORP\corpmin      Microsoft Corporat... whoami  /all
wmiprvse.exe (2660)       WMI Provider Host                             C:\Windows\system32\wbem\wmiprvse.exe      NT AUTHORITY\...  Microsoft Corporat... C:\Windows\syst
svchost.exe (796)         Host Process for Windows Services             C:\Windows\system32\svchost.exe            NT AUTHORITY\...  Microsoft Corporat... C:\Windows\syst
svchost.exe (944)         Host Process for Windows Services             C:\Windows\System32\svchost.exe            NT AUTHORITY\...  Microsoft Corporat... C:\Windows\Syst
  rdpclip.exe (4012)      RDP Clipboard Monitor                         C:\Windows\System32\rdpclip.exe            CORP\corpmin      Microsoft Corporat... rdpclip
svchost.exe (244)         Host Process for Windows Services             C:\Windows\System32\svchost.exe            NT AUTHORITY\S... Microsoft Corporat... C:\Windows\Syst
svchost.exe (328)         Host Process for Windows Services             C:\Windows\system32\svchost.exe            NT AUTHORITY\L... Microsoft Corporat... C:\Windows\syst
svchost.exe (384)         Host Process for Windows Services             C:\Windows\System32\svchost.exe            NT AUTHORITY\...  Microsoft Corporat... C:\Windows\Syst
svchost.exe (740)

Description:  Windows Command Processor
Company:      Microsoft Corporation
Path:         C:\Windows\system32\cmd.exe
Command:      C:\Windows\system32\cmd.exe /C "whoami /all"
User:         CORP\corpmin
PID:          4204        Started:  5/7/2020 2:15:19 AM
                          Exited:   5/7/2020 2:15:19 AM
```

Additionally, Winrs events are logged on the remote host as *Microsoft-Windows-WinRM/Operational (Event ID 91)*.



```
Creating WSMan shell on server with ResourceUri: %1

Log Name:          Microsoft-Windows-Windows Remote Management/Operational
Source:            Windows Remote Managem   Logged:          5/8/2020 1:29:52 AM
Event ID:          91                       Task Category:   Request handling
Level:             Information              Keywords:        Server
User:              CORP\corpmin             Computer:        CORP-DC.corp.contork.com
OpCode:            Info
More Information:  Event Log Online Help
```

WINRM.VBS (AS CALLED THROUGH WINRM.CMD)

Winrm.vbs is a Visual Basic Script that allows administrators "to configure WinRM and to get data or manage resources" (Microsoft Docs). As discussed in Matt Graeber's "Abusing Windows Management Instrumentation (WMI) to Build a Persistent, Asyncronous, and Fileless Backdoor" whitepaper, *WinRM(.vbs)* allows for remote interaction of WMI objects over WinRM transport. This is interesting because several WMI classes can be leveraged to perform remote command execution. For instance, a very well-known WMI class, *Win32_Process*, can be used to spawn a (remote) process by leveraging the *Create* method. In the following *Winrm(.vbs)* example, the *invoke* command spawns a remote *notepad.exe* process on a host named *corp-dc*:

```
winrm invoke Create wmicimv2/Win32_Process
@{CommandLine="notepad"} -r:corp-dc
```

The *Process Identifier* (PID) and *ReturnValue* is returned in event XML metadata to confirm successful remote execution:

On the remote host, the process execution chain appears as follows:

```
svchost.exe (DcomLaunch) -> wmiprvse.exe -> [remote
command/binary]
```

THIRD-PARTY WINRM TOOLS

It is also worth mentioning that other 3rd party WinRM capabilities exist outside of Windows including:

- Metasploit – Contains modules in *auxiliary/scanner/winrm/** and *exploit/windows/winrm/**
- PyWinRM – A Python client to execute remote commands
- CrackMapExec– Contains a Python WinRM command module
- Evil-WinRM – Is a fully featured WinRM shell implemented in Ruby

WS-MANAGEMENT COMPONENT OBJECT MODEL (COM)

At the time when I was investigating this topic, I noticed that there was not really much offered in the way of Windows tooling outside of PowerShell that leveraged WinRM for remote command execution/lateral movement. I've decided to take a closer look and discovered some information about the WinRM Scripting API on Microsoft Doc/Windows Dev Center and on a few interesting StackOverflow/message board posts. Much of the core capabilities are driven by an underappreciated COM class that works very hard behind the scenes – **WSMan.Automation**.

Exposed methods and properties of the *WSMan.Automation* COM object revealed several interesting methods and properties:

A quick peek at the source code of *WinRM.vbs* showed that *WsMan.Automation* was leveraged to establish the WinRM (remote) sessions for management, querying, and invocation:

Furthermore, we can leverage OleView.Net by James Forshaw (@tiraniddo) to take a look at the COM class in greater detail. These key takeaways are noted:

- The CLSID is *BCED617B-EC03-420B-8508-977DC7A686BD*
- The In-Process server is named *WsmAuto.dll*
- The Type Library is called *Microsoft WSMAN Automation V1.0 Library*, and it is implemented in the WsmAuto.dll library
- There are two primary supported interfaces of interest – *IWSMan* and *IWSManEx*

We now have some background on a vector that we could (potentially) use to develop our own POC WinRM tools. Let's dive in…

BUILDING A POC WINRM REMOTE COMMAND

EXECUTION TOOL IN .NET C#

In this section, we will walk through the process of creating a POC tool – **SharpWSManWinRm.exe**

A BRIEF NOTE ON .NET COM INTEROP

For .NET, Visual Studio makes integrating (many) COM components (objects) quite seamless. A COM reference is added to an assembly project by selecting a predefined component in the *Reference Manager* or by selecting another component file from disk such as an unmanaged library (DLL) or Type Library (TLB) file. Visual Studio parses the Type Library and maps COM interfaces and classes to a namespace structure within an auto-generated managed interop library (DLL) that is included within the project.

At runtime, the .NET Common Language Runtime (CLR) creates "Runtime Callable Wrappers" (RCW) to hold interface pointers and marshal calls between .NET and created COM objects (instances). From a .NET perspective, this makes COM objects appear as .NET objects and "simplifies" the managed code necessary to work with those respective COM objects.

Note: For more information about RCW, please refer to the COM Interop pages on Microsoft Docs. In some instances, it may be necessary to work with external COM objects that requires a manual approach for creating the interop definitions. For information about manually creating wrappers with Interface Definition

Language (IDL) or Type Libraries, refer to this Microsoft Doc write-up.

SETTING UP A WSMAN/WINRM PROJECT WITH VISUAL STUDIO

After we create a new .NET Framework (4) console application project, add the COM reference by right clicking the *References* menu in *Solutions Explorer* and selecting *Add Reference*:

In the *Reference Manager*, select *Browse* and import the *WsmAuto.dll* file from \Windows\System32\\:

Alternatively, *Microsoft WSMAN Automation V1.0* Type Library can be selected from the COM menu in *Reference Manager* to achieve the same result:

After the Reference is added, Visual Studio kindly generates the *Interop.WSManAutomation.dll* interop assembly and namespace for interacting with the *WSMan Automation* COM components as noted in the *Reference* properties:

With the added *Reference*, we can now leverage the *WSManAutomation* namespace in a *using* statement to access the targeted interfaces, methods, and properties:

AUTO-GENERATED INTEROP ASSEMBLY

Before moving on, let's take a quick peek into the *Interop.WSManAutomation.dll* that was auto-generated. Using dnSpy, we can 'decompile' the assembly to view the source code:

We can see how the DLL assembly wraps up the interfaces and methods/properties/etc. in a convenient way to call in our C# console project. When the project code is compiled, this DLL is merged with the output assembly.

CODING THE WINRM POC

With the *WSman.Automation* object, we can leverage the same WMI classes for remote command execution that was showed earlier with *WinRm.vbs*. This is implemented in our POC C# code (below) as follows:

1. *wsman* is our initialized a .NET/C# object, which implements the *iWSManEx* interface that extends the methods and properties of the *iWSMan* interface.
2. *options* is a NET/C# object that calls the *CreateConnectionOptions()* method for setting *IWSManConnectionOptions*, which specifies the username and password (if any are set) for the session.
3. *session* is a NET/C# object that calls the *CreatSession()* method for establishing the connection with the specified target (*sessionUrl*). If a username and password are set in *CreateConnectionOptions*(), the *SessionFlagCredUsernamePassword***()** method sets the authentication flag for the specified target.
   Note: Other flags can be set to specify authentication methods such as *Negotiate* or *Kerberos*.
4. The *Invoke()* method is called to invoke a WMI class action. In this case, the WMI *Create* method (action) is called to specify

the creation of a remote process (from *Win32_Process*). *resource* is the retrieved WMI class identifier, and *parameters* contains the XML input, which includes the process/command to be executed.

RUNNING THE WINRM POC

After compiling the project assembly, we can run it against a target machine and retrieve the Process ID and return information:

Command 1-

```
SharpWSManWinRM.exe corp-dc notepad
```

Command 2 (with credentials) –

```
SharpWSManWinRM.exe corp-dc notepad corp\corpmin CorpM@ster
```

On the remote host, the process execution chain appears as follows:

```
svchost.exe (DcomLaunch) -> wmiprvse.exe -> [remote
command/binary]
```

"WSMAN-WINRM" PROJECT

Project source code is accessible in the *WSMan-WinRM* GitHub repository, which includes the CSharp version as well as several other POC implementations that leverage the WMI *Win32_Process* class execution method. The following are included in the project:

- SharpWSManWinRM.cs (CSharp)
- CppWSManWinRM.cpp (C++)
- WSManWinRM.vbs (Visual Basic Script)
- WSManWinRM.js (JScript)
- WSManWinRM.ps1 (PowerShell – similar to the Invoke-WSManAction cmdlet)

TRADECRAFT

Calling the Win32_Process class is not the only way to leverage WMI classes for remote command execution. Philip Tsukerman (@PhilipTsukerman) released a lot of great information about WMI Lateral Movement tradecraft in the "No Win32_Process Needed – Expanding The WMI Lateral Movement Arsenal" blog post as well this talk.

DEFENSIVE CONSIDERATIONS

There are certainly approachable opportunities for detecting and restricting WinRM remote command execution/lateral movement. Consider the following:

- Monitor the remote process execution chains stemming from *wmiprvse.exe* and *winrshost.exe*
- Monitor the *Microsoft-Windows-WinRM/Operational* Event Log for suspicious entries. Note: During testing for the WMI class interactions (*WinRM.vbs*/*SharpWSManWinRM*/etc), it did not appear that successful connected sessions were logged here. However, unsuccessful errors were logged with the name of the WMI resource.
- Consider whitelisting *trusted hosts* to allow only certain machines to connect to WinRM servers. More information can be found in this Red Canary blog post. Note: WinRM *trusted hosts* control what client can connect *to*. As such, this setting can be advantageous to control this in a centralized manner, but a better approach is to leverage jump hosts and (host) firewall rules to control which machines that should be allowed to connect to WinRM hosts.
- Administrators are not the only users that can leverage WinRM for remote management. Members of the *Remote Management Users* local/domain group can connect to WMI resources over WinRM. Ensure that group membership is only allowed for authorized personnel only.

CONCLUSION

Thank you for taking the time to read this blog post. I believe there are more interesting research opportunities in this area (maybe a CSharp "PSSession" capability?). I look forward to seeing others take this to the next level. Lastly, I'd like to give a shout out to Leo Loobeek (@leoloobeek) – thanks you for the fantastic insight in our discussions about COM and coding. It certainly helped shape this post!

~ @bohops

PREVIOUS POST

NEXT POST