The Impacket Arsenal: A Deep Dive into Impacket Remote Code Execution Tools

May 22, 2025 • 11 minute read

In today's evolving threat landscape, we continually see new threat actors emerge and novel attack techniques surface. To keep pace, defenders must monitor the tactics, techniques, and procedures (TTPs) leveraged by these threat actors. A critical part of this understanding comes from analyzing the tools attackers use to achieve their objectives. Among the most widely adopted and powerful tools in adversarial arsenals is Impacket[S0357], a versatile, Python-based collection of modules and scripts that has functionality to interact with and manipulate low-level network protocols, particularly those used in Windows environments. Impacket enables users to craft custom packets and perform operations at the protocol level, which makes it incredibly useful for tasks like remote command

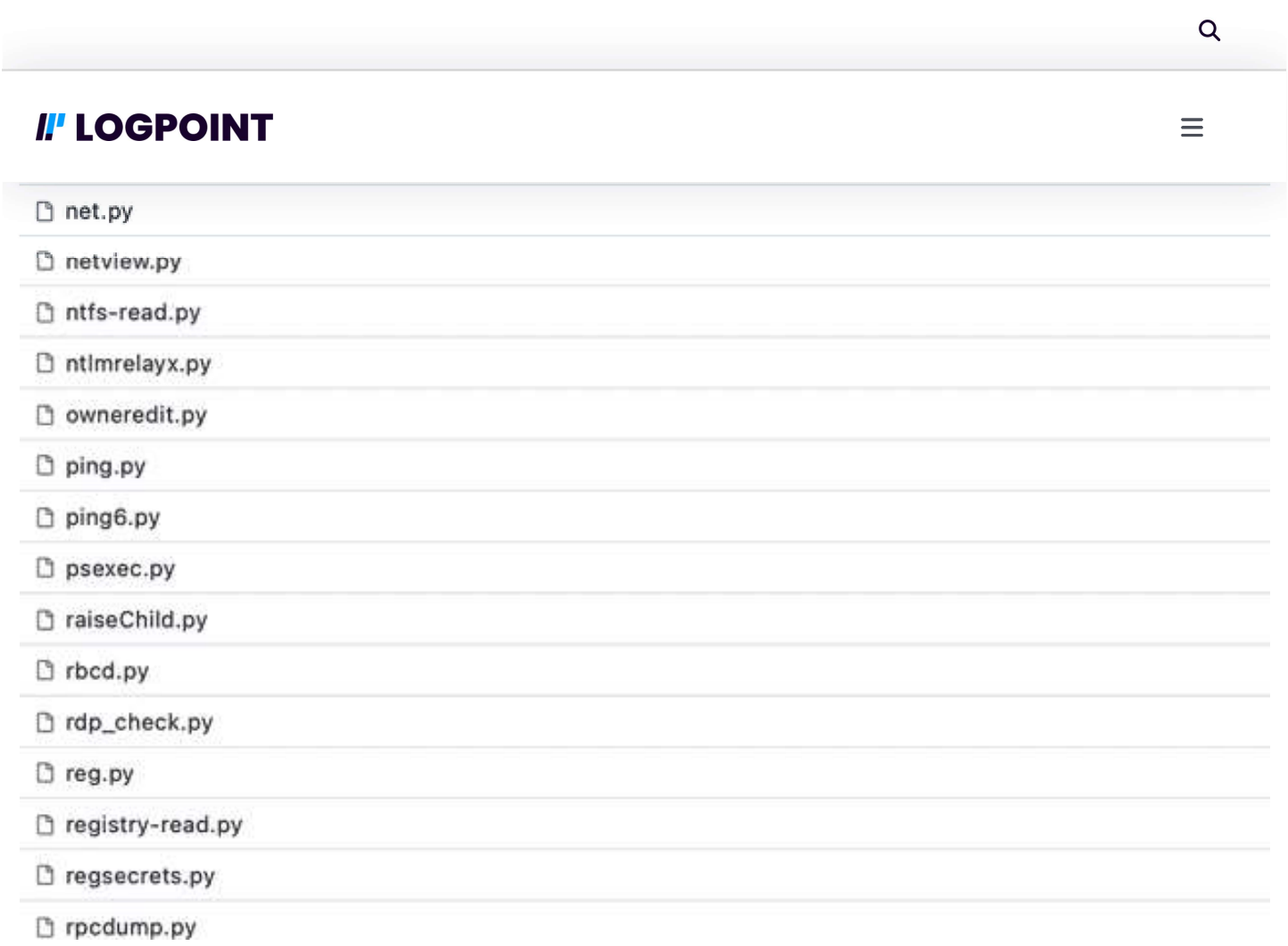
IMPACKET OVERVIEW

Impacket was originally developed as a toolkit for penetration testing, providing security professionals with powerful capabilities to simulate real-world attacks. However, like many tools used by both red teamers and threat actors, Impacket has seen widespread abuse, with threat actor groups including Advanced Persistent Threats (APTs) like APT28, APT29, and Mustang Panda, as well as ransomware groups such as ALPHV and Rhysida actively incorporating it into their arsenals.

According to Red Canary's Threat Detection Report 2024, Impacket was identified as the second most frequently observed threat.

Red Canary Top 10 Threats

When it comes to Impacket, it serves as a comprehensive collection of tools designed to help achieve a wide range of offensive objectives. However, in this blog, we'll focus specifically on the most commonly used and impactful Impacket tools that are frequently leveraged by adversaries for Remote Command Execution.



Impacket Toolkit

Remote Command Execution with WmiExec, SmbExec, and PsExec

In this blog, we'll explore three of the most widely used Impacket tools that facilitate Remote Command Execution and enable lateral movement.

WmiExec

WmiExec.py (WmiExec) is one of the Impacket widely used tool among red teams and threat actors. It is commonly leveraged for remote command execution due to its ability to blend in with legitimate system activity. WmiExec achieves this by relying on Windows Management Instrumentation (WMI).

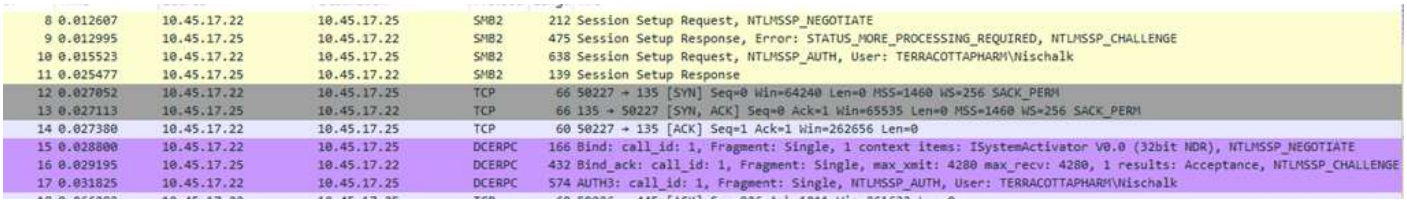
configurations, execute processes, and manage services both locally and remotely.

`WmiExec` works by remotely executing commands on a target system through Windows Management Instrumentation. It requires valid user credentials, provided through a username and password, NTLM hash, or Kerberos for authentication. Additionally, Administrative privileges on the target system are also necessary to execute commands and interact with system-level components.

From the screenshot below, we can see `WmiExec` initiates by prompting for credentials without requiring any additional flags or configurations. Once authenticated, it seamlessly provides a semi-interactive shell on the remote system. Since, `WmiExec` does not create new services, drop binaries to disk, it is more stealth, making it a popular choice in the arsenals of many threat actors.

```
C:\Users\NischalK\Downloads\impacket-master\impacket-master\examples>python3 wmiexec.py "TERRACOTTAPHARM/Nischalk@10.45.
17.25"
Impacket v0.13.0.dev0 - Copyright Fortra, LLC and its affiliated companies

Password:
[*] SMBv3.0 dialect used
[!] Launching semi-interactive shell - Careful what you execute
[!] Press help for extra shell commands
C:\>whoami
terracottapharm\nischalk
```

When WmiExec is executed, the first step it takes is to initiate a connection on the target system over the SMB (Server Message Block) protocol. This connection is also, used to retrieve the output of any commands executed on the remote system. After establishing the SMB session, WmiExec uses the Distributed Component Object Model (DCOM) to initiate remote execution. This interaction takes place over port 135 using the DCE/RPC (Distributed Computing Environment / Remote Procedure Call) protocol to communicate with the target's ISystemActivator interface.



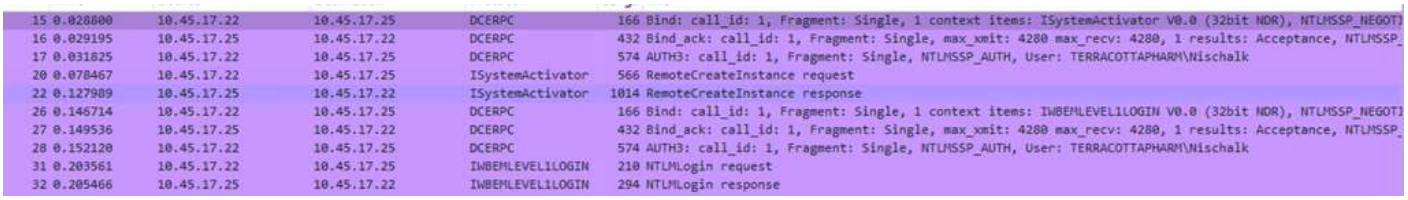
8	0.012607	10.45.17.22	10.45.17.25	SMB2	212 Session Setup Request, NTLMSSP_NEGOTIATE
9	0.012995	10.45.17.25	10.45.17.22	SMB2	475 Session Setup Response, Error: STATUS_MORE_PROCESSING_REQUIRED, NTLMSSP_CHALLENGE
10	0.015523	10.45.17.22	10.45.17.25	SMB2	638 Session Setup Request, NTLMSSP_AUTH, User: TERRACOTTAPHARM\Nischalk
11	0.025477	10.45.17.25	10.45.17.22	SMB2	139 Session Setup Response
12	0.027052	10.45.17.22	10.45.17.25	TCP	66 50227 → 135 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
13	0.027113	10.45.17.25	10.45.17.22	TCP	66 135 → 50227 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
14	0.027380	10.45.17.22	10.45.17.25	TCP	60 50227 → 135 [ACK] Seq=1 Ack=1 Win=262656 Len=0
15	0.028800	10.45.17.22	10.45.17.25	DCERPC	166 Bind: call_id: 1, Fragment: Single, 1 context items: ISystemActivator V0.0 (32bit NDR), NTLMSSP_NEGOTIATE
16	0.029195	10.45.17.25	10.45.17.22	DCERPC	432 Bind_ack: call_id: 1, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance, NTLMSSP_CHALLENGE
17	0.031825	10.45.17.22	10.45.17.25	DCERPC	574 AUTH3: call_id: 1, Fragment: Single, NTLMSSP_AUTH, User: TERRACOTTAPHARM\Nischalk

PCAP showing WmiExec initiating SMB session followed by DCERPC communication

code or commands on remote systems by managing how components request services from one another across a network.

In this context, DCOM serves as the underlying mechanism that allows Windows Management Instrumentation (WMI) to initiate and execute commands remotely.

From the screenshot below, We can see `WmiExec` initiates a Bind request to the `ISystemActivator` interface using DCOM over port 135.



15	0.028800	10.45.17.22	10.45.17.25	DCERPC	166 Bind: call_id: 1, Fragment: Single, 1 context items: ISystemActivator V0.0 (32bit NDR), NTLMSSP_NEGOTI
16	0.029195	10.45.17.25	10.45.17.22	DCERPC	432 Bind_ack: call_id: 1, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance, NTLMSSP_
17	0.031825	10.45.17.22	10.45.17.25	DCERPC	574 AUTH3: call_id: 1, Fragment: Single, NTLMSSP_AUTH, User: TERRACOTTAPHARM\Nischalk
20	0.078467	10.45.17.22	10.45.17.25	ISystemActivator	566 RemoteCreateInstance request
22	0.127989	10.45.17.25	10.45.17.22	ISystemActivator	1014 RemoteCreateInstance response
26	0.146714	10.45.17.22	10.45.17.25	DCERPC	166 Bind: call_id: 1, Fragment: Single, 1 context items: IWBEMLEVEL1LOGIN V0.0 (32bit NDR), NTLMSSP_NEGOTI
27	0.149536	10.45.17.25	10.45.17.22	DCERPC	432 Bind_ack: call_id: 1, Fragment: Single, max_xmit: 4280 max_recv: 4280, 1 results: Acceptance, NTLMSSP_
28	0.152120	10.45.17.22	10.45.17.25	DCERPC	574 AUTH3: call_id: 1, Fragment: Single, NTLMSSP_AUTH, User: TERRACOTTAPHARM\Nischalk
31	0.203561	10.45.17.22	10.45.17.25	IWBEMLEVEL1LOGIN	210 NTLMLogin request
32	0.205466	10.45.17.25	10.45.17.22	IWBEMLEVEL1LOGIN	294 NTLMLogin response

PCAP showing DCERPC bind to the ISystemActivator

From the below code snippet, we can observe, Once authenticated, `WmiExec` sets up a DCOM connection using the `DCOMConnection` class, which internally initiates a call to `ISystemActivator` , the COM interface responsible for creating remote COM objects. Through this, it instantiates the `IWbemLevel1Login` interface, which is a key part of WMI's remote access layer. Using this interface, the tool performs a login to the `root\cimv2` namespace, the default location for most system-level WMI operations. After gaining access to WMI, it loads the `Win32_Process` class and uses it to spawn commands on the target system.

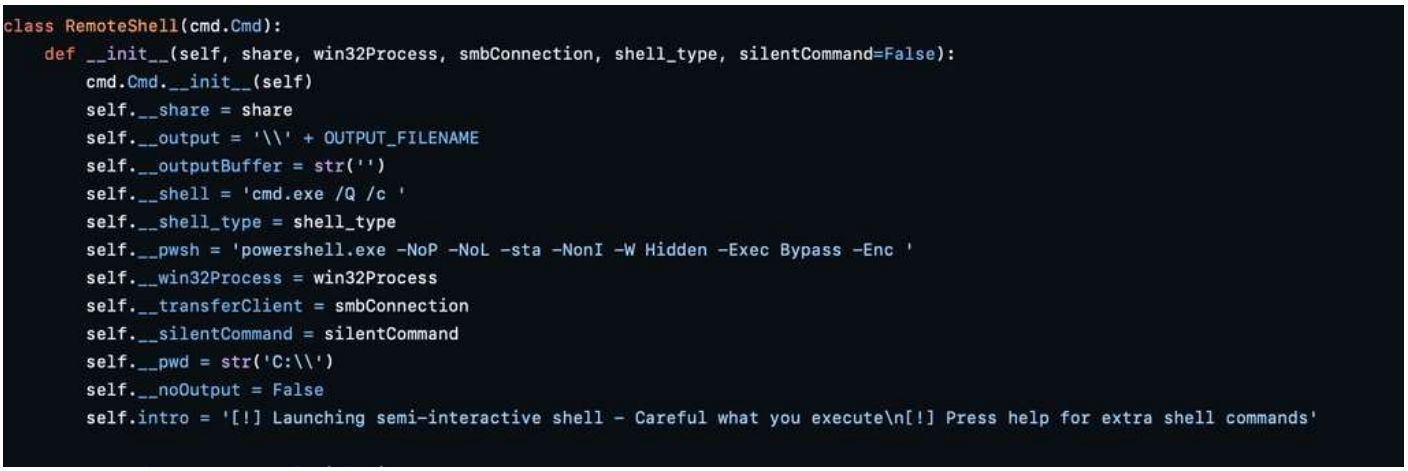
```
dcom = DCOMConnection(addr, self.__username, self.__password, self.__domain, self.__lmhash, self.__nthash,
                      self.__aesKey, oxidResolver=True, doKerberos=self.__doKerberos, kdcHost=self.__kdcHost, remoteHost=self.__remoteHost)
try:
    iInterface = dcom.CoCreateInstanceEx(wmi.CLSID_WbemLevel1Login, wmi.IID_IWbemLevel1Login)
    iWbemLevel1Login = wmi.IWbemLevel1Login(iInterface)
    iWbemServices = iWbemLevel1Login.NTLMLogin('//./root/cimv2', NULL, NULL)
    iWbemLevel1Login.RemRelease()

    win32Process, _ = iWbemServices.GetObject('Win32_Process')

    self.shell = RemoteShell(self.__share, win32Process, smbConnection, self.__shell_type, silentCommand)
    if self.__command != ' ':
        self.shell.onecmd(self.__command)
    else:
        self.shell.cmdloop()
```

Since WMI does not natively return command output, **WmiExec** wraps the command using structure like shown in the example below:

Now let's break down the WmiExec command execution process step by step.

- On the target system, WmiExec spawns **cmd.exe** with the flags **/Q /c**. Here, the flag **/Q** suppresses command echo, while **/c** tells the shell to execute the operator's command and then terminate.
- It then accesses the target system's ADMIN$ share using the loopback IP **\\127.0.0.1\ADMIN$**.
- Then, The command output both standard output and error is redirected into a temporary file, **__1746603268.41452** using **1>** and **2>&1**.

> `2>&1` **(Standard Error to Standard Output)**
> `2` refers to **standard error (stderr)** , the channel that handles error messages. `2>&1` tells the system to redirect stderr to wherever stdout is going.



```
class RemoteShell(cmd.Cmd):
    def __init__(self, share, win32Process, smbConnection, shell_type, silentCommand=False):
        cmd.Cmd.__init__(self)
        self.__share = share
        self.__output = '\\' + OUTPUT_FILENAME
        self.__outputBuffer = str('')
        self.__shell = 'cmd.exe /Q /c '
        self.__shell_type = shell_type
        self.__pwsh = 'powershell.exe -NoP -NoL -sta -NonI -W Hidden -Exec Bypass -Enc '
        self.__win32Process = win32Process
        self.__transferClient = smbConnection
        self.__silentCommand = silentCommand
        self.__pwd = str('C:\\')
        self.__noOutput = False
        self.intro = '[!] Launching semi-interactive shell - Careful what you execute\n[!] Press help for extra shell commands'
```

After executing the command and redirecting it's output to a temporary file on the remote system's ADMIN$ share, WmiExec reads the file using existing SMB connection and displays the output to the user.

Immediately after retrieving the output, WmiExec ensures stealth by deleting the file using:

This function call deletes the same output file from the `ADMIN$` share using SMB.

SmbExec

SmbExec.py (SmbExec) is another post-exploitation utility from the Impacket toolkit that enables attackers to remotely execute command on a target system, making it a common tool for lateral movement within a network. Like WmiExec, SmbExec does not initiate a full interactive login session. As shown in the screenshot below, SmbExec requires valid credentials for the target system, which can be provided either via domain authentication or by using obtained password hashes. Additionally, it requires administrative privileges to execute commands remotely and interact with system-level services.

```
C:\Users\NischalK\Downloads\impacket-master\impacket-master\examples>python3 smbexec.py "TERRACOTTAPHARM/NischalK@10.45.17.25"
Impacket v0.13.0.dev0 - Copyright Fortra, LLC and its affiliated companies

Password:
[!] Launching semi-interactive shell - Careful what you execute
C:\Windows\system32>hostname
WS-SAL-01
```

Once the user is authenticated over SMB, the first active task `SmbExec` performs is creating a service on the remote system. SmbExec randomizes the service name by generating an 8-character string of ASCII letters.

```
        if serviceName is None:
            self.__serviceName = ''.join([random.choice(string.ascii_letters) for i in range(8)])
        else:
            self.__serviceName = serviceName
```

Windows services can be remotely created and controlled via Remote Procedure Call. In the case of `SmbExec`, these services are instantiated using the `hRCreateServiceW` function to create service on the remote system.

```
        resp = scmr.hRCreateServiceW(self.__scmr, self.__scHandle, self.__serviceName, self.__serviceName,
                                     lpBinaryPathName=command, dwStartType=scmr.SERVICE_DEMAND_START)
        service = resp['lpServiceHandle']
```

When the service is created, the actual command path used to launch it looks like this:

%COMSPEC%	Environment variable that points to `cmd.exe`
/Q	Quiet mode, suppresses command echoing
/c	Execute command and terminate
echo	Writes the executed command (e.g., `whoami`) into a `.bat` file
&	Chained commands
del ...	Deletes the batch file to clean up

This setup writes the command to a `.bat` file located in `C:\Windows`, and then runs it via `cmd.exe`, then deletes bat file leaving only the output file for retrieval. The bat filename is generated using 8 random ASCII letters.

```
batchFile = '%SYSTEMROOT%\\' + ''.join([random.choice(string.ascii_letters) for _ in range(8)]) + '.bat'

command = self.__shell + 'echo ' + data + ' ^> ' + self.__output + ' 2^>^&1 > ' + batchFile + ' & ' + \
          self.__shell + batchFile
```

After the service executes the bat file containing the command, the next important task is to capture the output. Like `WmiExec`, `SmbExec` captures command output by redirecting it to a temporary file on the target system.

```
self.__output = '\\\\%COMPUTERNAME%\\' + self.__share + '\\' + OUTPUT_FILENAME
```

For example, in this case, I have executed `powershell` as the command via `SmbExec`, and the result can be viewed from the temporary output file created by `SmbExec`.

Next, the tool retrieves the output file using the existing SMB session and displays the results to the user.

```
def get_output(self):
    def output_callback(data):
        self.__outputBuffer += data

    if self.__mode == 'SHARE':
        self.transferClient.getFile(self.__share, OUTPUT_FILENAME, output_callback)
        self.transferClient.deleteFile(self.__share, OUTPUT_FILENAME)
    else:
        fd = open(SMBSERVER_DIR + '/' + OUTPUT_FILENAME,'rb')
        output_callback(fd.read())
        fd.close()
        os.unlink(SMBSERVER_DIR + '/' + OUTPUT_FILENAME)
```

PsExec

PsExec.py(**PsExec**) is another widely used post-exploitation tool for remote command execution. It is modeled after Microsoft Sysinternals original **PsExec** tool, which is widely used for remote administration. While functionally effective, **PsExec** is considered less stealthy than tools like **SmbExec** and **WmiExec**, because it explicitly writes and executes a binary service on the target disk which leaves many traces for detection. Despite its detectability, **PsExec** remains highly popular among threat actors, red teamers, and penetration testers.

Like **WmiExec**, and **Smbexe**, From the screenshot, we can see that PsExec requires valid credentials with administrative privileges on the target system to execute commands remotely.

```
C:\Users\NischalK\Downloads\impacket-master\impacket-master\examples>python3 psexec.py "TERRACOTTAPHARM/NischalK@10.45.17.25"
Impacket v0.13.0.dev0 - Copyright Fortra, LLC and its affiliated companies

Password:
[*] Requesting shares on 10.45.17.25.....
[*] Found writable share ADMIN$
[*] Uploading file OAqxcPAK.exe
[*] Opening SVCManager on 10.45.17.25.....
[*] Creating service cqot on 10.45.17.25.....
[*] Starting service cqot.....
```

The execution flow of Impacket `PsExec` begins with authentication via SMB like every other Impacket tools. Once authenticated, it connects to the target's `IPC$` share and establishes a `DCE/RPC` connection to the `Service Control Manager (SCM)` via the `svcctl` named pipe.

```
if hasattr(rpctransport, 'set_credentials'):
    # This method exists only for selected protocol sequences.
    rpctransport.set_credentials(self.__username, self.__password, self.__domain, self.__lmhash,
                                 self.__nthash, self.__aesKey)
rpctransport.set_kerberos(self.__doKerberos, self.__kdcHost)
self.doStuff(rpctransport)
```

Then, PsExec uploads its payload to a writable administrative share `ADMIN$` , which maps to `C:\Windows` using the low-level putFile() operation.

To execute the uploaded binary, PsExec creates a new Windows service on the target machine. The service name can be user-defined or randomly generated by the script. Once the service is started, it proceeds to execute the specified command on the target system

```
s.setTimeout(100000)
if self.__exeFile is None:
    installService = serviceinstall.ServiceInstall(rpctransport.get_smb_connection(), remcomsvc.RemComSvc(), self.__serviceName, self.__remoteBinaryName)
else:
    try:
        f = open(self.__exeFile, 'rb')
    except Exception as e:
        logging.critical(str(e))
        sys.exit(1)
    installService = serviceinstall.ServiceInstall(rpctransport.get_smb_connection(), f, self.__serviceName, self.__remoteBinaryName)

if installService.install() is False:
    return

if self.__exeFile is not None:
    f.close()
```

Unlike **WmiExec** and **SmbExec**, which use temporary file redirection for capturing output, PsExec establishes a custom communication channel using named pipes:

- **RemCom_stdin** for input,
- **RemCom_stdout** for standard output, and
- **RemCom_stderr** for error output.These pipes allow full duplex communication between attacker and target, with command input and output streamed in real-time. After execution, PsExec attempts to clean up by uninstalling the created service and optionally deleting the uploaded binary.

Hunting for Impacket Remote Code Execution Tools using Logpoint SIEM

In this blog we have explored **WmiExec**, **SmbExec**, and **PsExec**, we've covered three of the most commonly used remote code execution tools of Impacket frequently leveraged by threat actors during lateral movement and post-exploitation. We have broken down how each of these tools

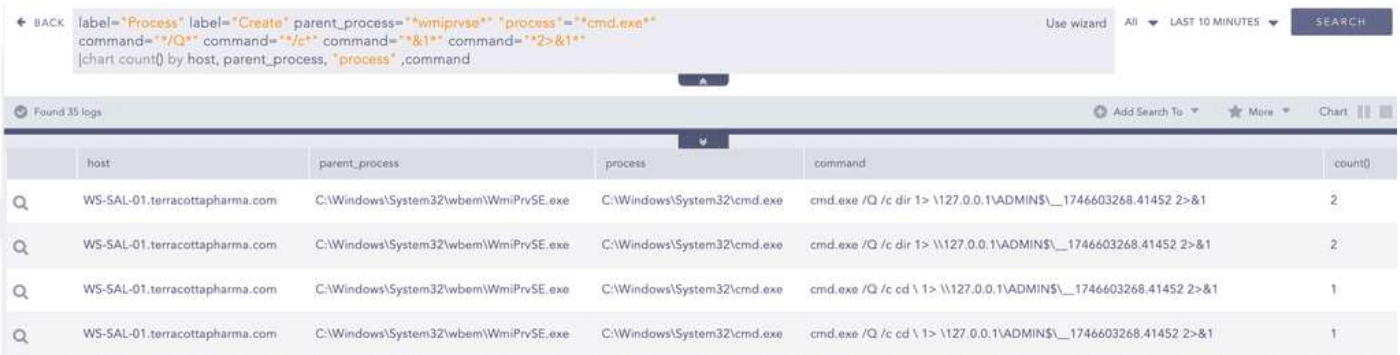
Required Log Source

1. Windows
    a. Process Creation with Command Line Auditing should be enabled
    b. Detailed File Sharing Auditing should be enabled
2. Windows Sysmon
    a. Our Sysmon baseline configuration

Hunting For WmiExec

In the case of **WmiExec**, one of the initial behavioral indicators is the **WmiPrvSE.exe** process spawning a child process such as **cmd.exe**. Based on this behavior, we can use the following query to hunt for **WmiExec** execution filtering it's command line pattern.

```
label="Process" label=Create parent_process="*wmiprvse.exe*" "process"="*cmd.exe*"
command="*/Q*" command="*/c*" command="*&1*" command="*2>&1*"
```



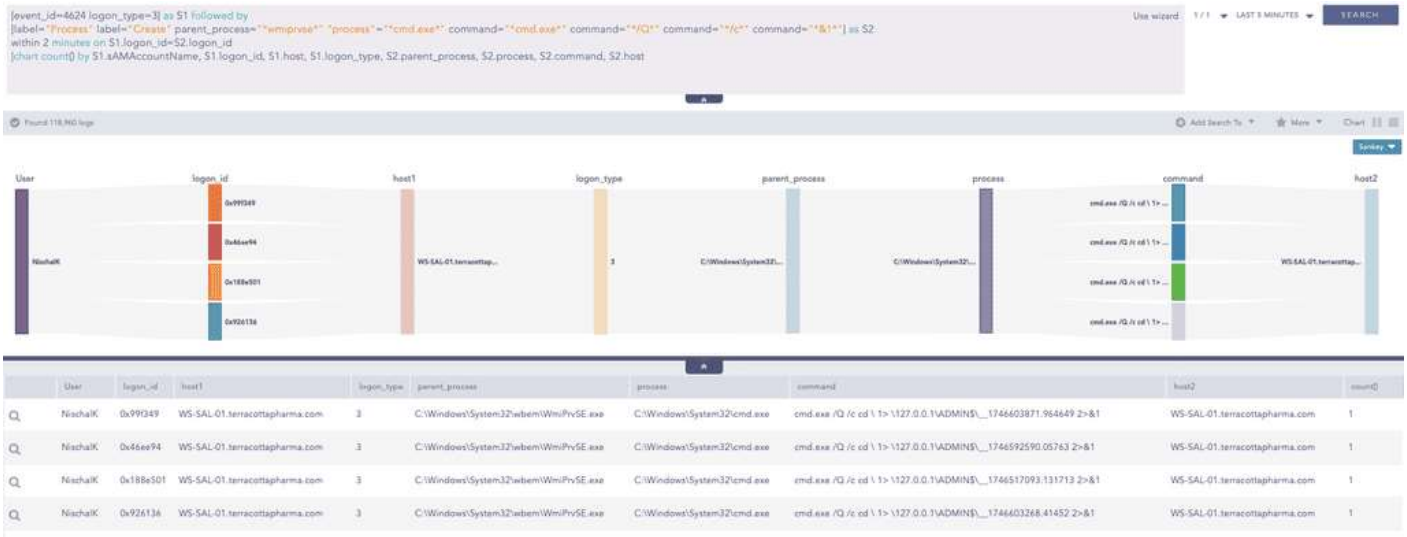
Furthermore, we can use the below query to hunt for lateral movement leveraging `WmiExec` correlating with `logon id` and the query of `WmiExec` which we have used above. This query correlates two events, A Logon Event `4624` , Logon Type `3` and process creation event. This is important because when one host accesses another host over the network, `Event ID 4624` with `Logon Type 3` is generated. Along with it, a unique logon ID is assigned to that session, which can also be observed in subsequent process creation events, allowing us to correlate remote logins with actions performed on the target system.

events, process creation, and resource access.

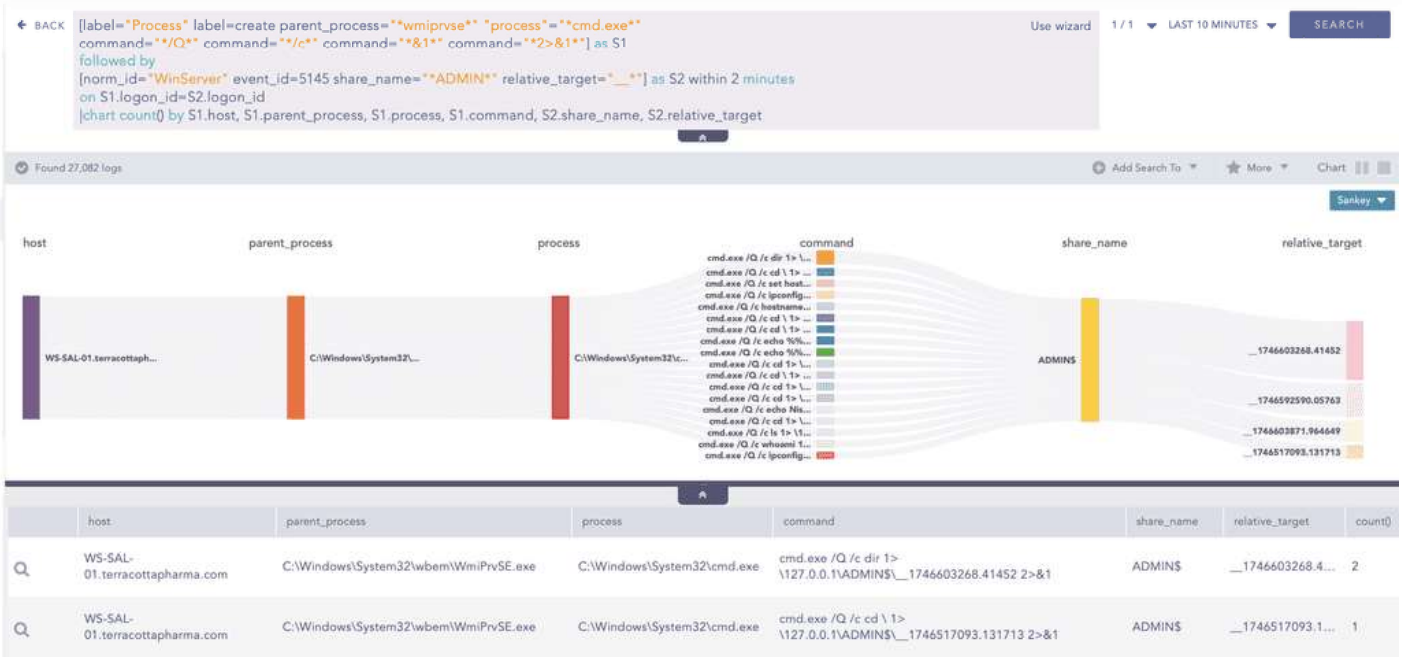
Logon Type 3 represents a network logon, which happens when a user accesses a system remotely for example, through SMB, WMI, or PsExec.

Furthermore, This query also helps us identify the direction of lateral movement by revealing where the attacker moved from and where they moved to.

```
[norm_id="WinServer" event_id=4624 logon_type=3] as S1
followed by
[label="Process" label=create parent_process="*\wmiprvse*" "process"="*\cmd.exe*"
```



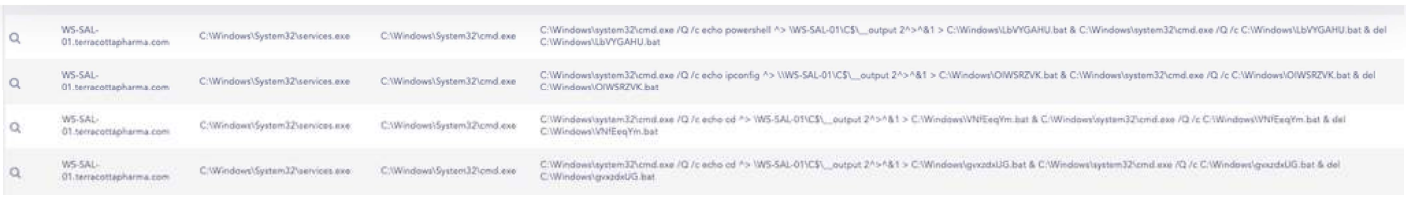
We can further hunt for indicators of `WmiExec` by focusing on how it retrieves command output from the temporary files it creates on the target system. `WmiExec` reads these files over the same SMB session used to execute the command, which means the logon ID remains consistent across related events, Event ID 1 (process creation) and Event ID 5145 (file access over SMB). By correlating these two events and filtering for filenames that begin with double underscores, such as `__1746603268.41452` , we can identify `WmiExec` pattern. The following query can be used to hunt for this activity.



Hunting for SmbExec

When hunting for **SmbExec** activity, one reliable pattern is it's use of temporary Windows services to execute commands. These services typically invoke **cmd.exe** to run batch files containing the attacker's command. As seen in the screenshot, we can consistently observe **services.exe** as the parent process and **cmd.exe** as the child, along with command lines that include output redirection, execution of a **.bat** file, and its subsequent deletion (**.bat & del**). This behavior is characteristic of Impacket's **SmbExec** tool. Therefore, we can use the below query to hunt for this activity by filtering for **services.exe** spawning **cmd.exe** with these specific command line indicators.

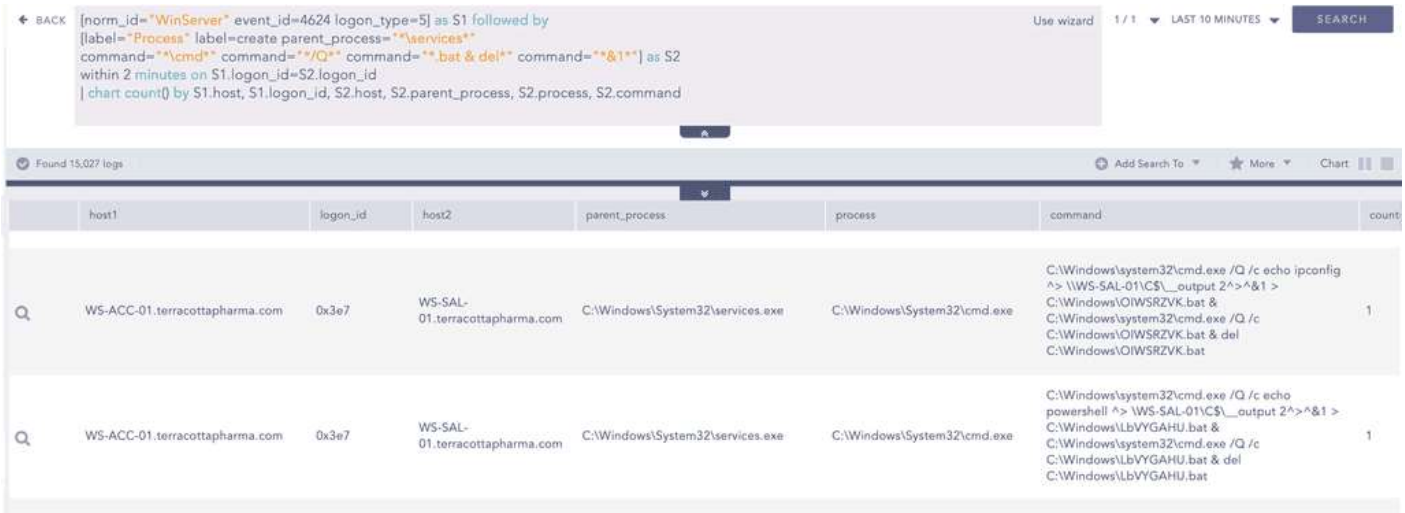
```
label="Process" label=create parent_process="*services.exe*"
command="*cmd*" command="*/Q*" command="*.bat & del*" command="*&1*"
```



We can further enhance the above detection by correlating service logon events with process creation events using the shared `logon id` field. When `SmbExec` creates a temporary Windows service to run a command, Windows spawns that service under the `NT AUTHORITY\SYSTEM` account, triggering a Logon Event ID 4624 with Logon Type 5, which is a service logon.

This session is then used to execute the bat file which contains attackers specified commands. Since the service logon and the resulting command execution happen within the same session, the logon ID remains the same across both events. Therefore we can use the below query to correlates logon ID with logon type 5 with the subsequent process creation activity.

```
[norm_id="WinServer" event_id=4624 logon_type=5] as S1
followed by
[label="Process" label=create parent_process="*\services*"
```



Lastly, we can use the below query to hunt for `Smbexec` activity by monitoring for batch file creation and Windows service installation on the target system. Since `SmbExec` writes the attacker's command into a temporary `.bat` file and sets up a service to execute it silently, this behavior is traceable in the Windows System log, it trigger Event ID `7045`, indicating a new service was installed. Furthermore, we can look for ImagePath of the service that includes

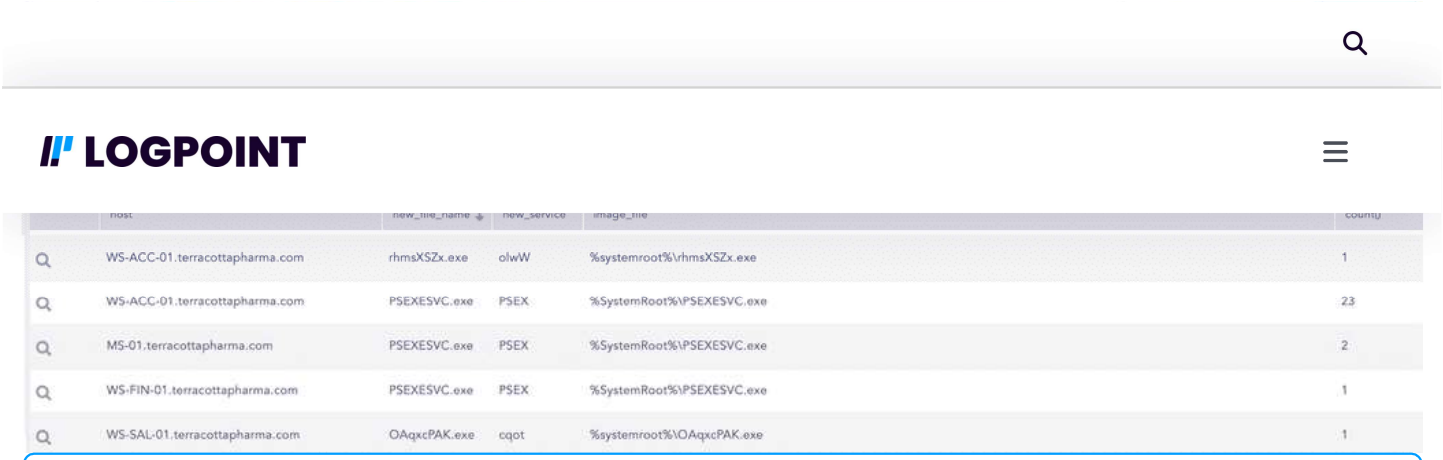
```
norm_id="WinServer" label="service" label="install" label="successful"
path="*COMSPEC* /Q /c*" path="*.bat*del*" path="*__output 2^*"
```

Hunting for PsExec

To hunt for Impacket PsExec, we can hunt for an Event ID 7045, which logs the creation of new Windows services. Impacket's **PsExec**, similar to Sysinternals **PsExec**, creates a temporary service on the target system to execute its payload. In Impacket's implementation, the service name is exactly 4 characters long, composed of random uppercase and lowercase ASCII letters. The executable dropped to the target system is exactly 8 characters long, also randomly generated using mixed-case letter.

We can use below query to hunt for **PsExec** Service creations, This query uses regex to match the binary file name that are exactly 8 characters long and service names that are 4 characters in length.

```
norm_id="WinServer" label="service" label="install" label="successful"
| process regex("(?P<new_file_name>[A-Za-z]{8}\.exe)", file)
| process regex("(?P<new_service>[A-Za-z]{4})", service)
```



Note: The above query may produce false positives if legitimate services or executables happen to match the same naming pattern.

Lastly, we can filter for Event IDs 17 and 18, which log named pipe creation and connection activity. PsExec uses a named pipe called `RemCom_communication` for command exchange. Therefore, we can use the below query to hunt for suspicious named pipe.

```
norm_id=WindowsSysmon event_id IN [17, 18]
pipe IN ["*RemCom*", "*-stdin", "*-stderr", "*-stdout"]
```

Comprehensive Detection of Impacket Remote Code Execution Tools

Lastly, We have observed that many Impacket tools share a similar command-line pattern, particularly in how they execute commands using **cmd.exe** with silent flags and output redirection. Based on these consistencies, we can use the following query to build a comprehensive detection for Impacket-based activity.

```
label="process" label=create
   command="*cmd.exe*" command="*/c*" command="*&1*"
```

WS-SAL-01.terracottapharma.com	C:\Windows\System32\wbem\WmiPrvSE.exe	C:\Windows\System32\cmd.exe	cmd.exe /Q /c dir 1> \\127.0.0.1\ADMIN$__1746603268.41452 2>&1
WS-SAL-01.terracottapharma.com	C:\Windows\System32\services.exe	C:\Windows\System32\cmd.exe	C:\Windows\system32\cmd.exe /Q /c echo whoami ^> \\WS-SAL-01\C$__output 2^>^&1 > C:\Windows\IkuhMQnX.bat & C:\Windows\system32\cmd.exe /Q /c C:\Windows\IkuhMQnX.bat & del C:\Windows\IkuhMQnX.bat
WS-SAL-01.terracottapharma.com	C:\Windows\System32\services.exe	C:\Windows\System32\cmd.exe	C:\Windows\system32\cmd.exe /Q /c echo cd ^> \WS-SAL-01\C$__output 2^>^&1 > C:\Windows\VNfEeqYm.bat & C:\Windows\system32\cmd.exe /Q /c C:\Windows\VNfEeqYm.bat & del C:\Windows\VNfEeqYm.bat
MS-01.terracottapharma.com	C:\Windows\System32\wbem\WmiPrvSE.exe	C:\Windows\System32\cmd.exe	cmd.exe /Q /c cd \ 1> \\127.0.0.1\ADMIN$__1746697042.3212285 2>&1
WS-SAL-01.terracottapharma.com	C:\Windows\System32\services.exe	C:\Windows\System32\cmd.exe	C:\Windows\system32\cmd.exe /Q /c echo cd ^> \WS-SAL-01\C$__output 2^>^&1 > C:\Windows\gvxzdxUG.bat & C:\Windows\system32\cmd.exe /Q /c C:\Windows\gvxzdxUG.bat & del C:\Windows\gvxzdxUG.bat
WS-SAL-01.terracottapharma.com	C:\Windows\System32\services.exe	C:\Windows\System32\cmd.exe	C:\Windows\system32\cmd.exe /Q /c echo cd ^> \WS-SAL-01\C$__output 2^>^&1 > C:\Windows\HauSEqoe.bat & C:\Windows\system32\cmd.exe /Q /c C:\Windows\HauSEqoe.bat & del C:\Windows\HauSEqoe.bat

Recommendation

To effectively defend against remote code execution and lateral movement leveraged by tools like Impacket, organizations should implement the following key security measures:

- Implement network segmentation to isolate critical assets and systems from the broader environment, effectively controlling traffic flows and access between subnetworks to limit lateral movement by threat actors. Enhance this with micro-segmentation by grouping similar systems and enforcing granular access controls and policies, supporting Zero Trust principles across both the network perimeter and internal infrastructure.
- Adopt a Defense-in-Depth strategy by layering multiple security controls such as EDR (Endpoint Detection & Response), SIEM, identity and access management, and web/email filtering. This multi-layered defense approach ensures that threats are detected and mitigated at different stages of the attack lifecycle, reducing the risk of a single point of failure.
- Enable comprehensive logging and visibility across endpoints, network infrastructure, and cloud environments. Continuous monitoring helps detect suspicious behaviors such as lateral movement or unauthorized execution early in the attack lifecycle, helping to minimize overall impact.
- Establish a formal incident response plan and regularly conduct tabletop exercises and live simulations. This helps ensure rapid containment, investigation, and recovery during real-world attacks, while also identifying and remediating response gaps.

Related Posts

After React2Shell: Following the Attacker From Access to Impact

December 08, 2025

Arming Loki with jArvIs: How AI Is Powering Real-World Intrusions

December 05, 2025

How to detect lateral movement before it spreads

November 03, 2025

Cyber Defense Platform

SIEM

NDR

Automation

Pricing

Sizing Calculator

Why Logpoint?

Customer Cases

Newsletter

Company

About us

Careers at Logpoint

Media Room

Blog & Webinars

Support

Service Desk

Documentation

Community

Contact

info@logpoint.com

+45 7060 6100