PowerShell | Red Canary Threat Detection Report

Archived: 2026-04-05 20:25:18 UTC

PowerShell's versatility is on display in many of the phishing campaigns we see. Adversaries commonly send their victims email messages that include malicious attachments containing embedded code intended to launch a payload. In many cases, this payload executes encoded or obfuscated PowerShell commands that download and execute additional code or a malicious binary from a remote resource.

Based on our analysis of commonalities across threats leveraging PowerShell, we frequently observe adversaries abusing PowerShell in the following ways:

- as a component of an offensive security or attack toolkit like Mimikatz, Empire, PoShC2, and Cobalt Strike—to encode or otherwise obfuscate malicious activity, using Base64 and variations of the encoded command switch as well as various breach and attack simulation tools to encode or otherwise obfuscate malicious activity, using Base64 and variations of the encoded command switch
- to perform **ingress tool transfer** by downloading payloads from the internet using cmdlets, abbreviated cmdlets, or argument names, and calling .NET methods, among other PowerShell features
- to load and malicious .NET assemblies in memory
- to facilitate process injection

Adversaries also occasionally leverage PowerShell to disable Windows security tools and to decrypt encrypted or obfuscated payloads.

Increasingly, adversaries utilize popular PowerShell modules like AzureAD, Azure, Microsoft.Graph, and AADInternals to perform attacks against identity, cloud, and SaaS environments upon compromising an Entra ID identity. These tools are not as likely to be used for malicious purposes on compromised endpoints but are used remotely to conduct attacks on cloud and identity infrastructure. In the case of Entra ID abuse, detection should focus on collection and analysis of sign-in and audit logs. There is also a growing list of PowerShell-based tools that are designed to abuse Entra ID and Azure cloud environments including:

- GraphRunner
- PowerZure
- MicroBurst
- AzureHound

Associated threats

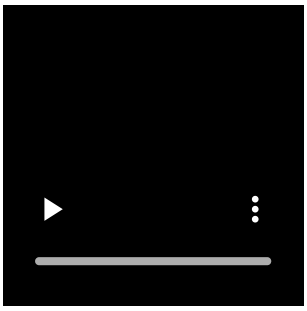
- LummaC2
- Scarlet Goldfinch
- PowerGhost
- NetSupport Manager
- KongTuke

- Amber Albatross
- HijackLoader

Take action

For those using Microsoft Defender products in their enterprise, it is crucial to enable Tamper Protection. Microsoft has made substantial investments in identifying and mitigating against a large class of tampering opportunities. In the case of PowerShell tradecraft, with Tamper Protection enabled, the Set-MpPreference cmdlet cannot be used to disable or create rule exceptions.

The most effective protection against PowerShell tradecraft is through the implementation and enforcement of a strong Windows App Control policy, which places PowerShell into Constrained Language mode, mitigating a wide array of PowerShell tradecraft.



Visibility

Note: The visibility sections in this report are mapped to MITRE ATT&CK data components.

Defenders have been able to detect malicious use of PowerShell since the tool's inception—and the array of relevant telemetry sources has expanded in near lockstep with adversary abuse over the years. The following data or telemetry sources are available to enterprise defenders or security vendors alike on a case-to-case basis. Some of this telemetry can be collected from commercial EDR or other security products, via native operating system logs, or both.

Process creation and metadata

Process execution and lineage are among the most common sources of telemetry that we leverage at Red Canary to detect all varieties of malicious activity. PowerShell is no exception. We frequently focus our detection analytics primarily on process starts, stops, and parent/child relationships while using other sources of data, like

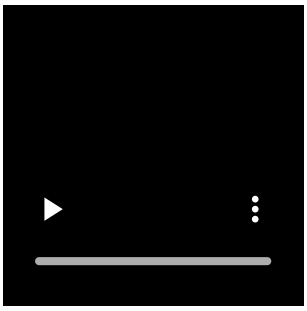
command-line parameters or network connections to enrich our detection logic. Many security tools collect process telemetry, and Red Canary leverages a variety of EDR tools to gather this information.

Command execution

Command-line parameters are also an effective telemetry source for detecting potentially malicious PowerShell behavior. They're particularly useful when used in conjunction with other telemetry that's widely available from security software, like process monitoring and network connection telemetry.

Network communication and connection creation

Since adversaries often use PowerShell to conduct Ingress Tool Transfer or to otherwise make external network connections, security teams should consider including network telemetry in their detection analytics. In fact, one of the most common ways we detect PowerShell abuse is via a detection analytic that looks for invoke expressions that download content via HTTP.



Collection

Note: The collection sections of this report showcase specific log sources from Windows events, Sysmon, and elsewhere that you can use to collect relevant security information.

Windows Security Event ID 1101: Antimalware-Scan-Interface (AMSI)

Designed to be consumed by security vendors, AMSI telemetry offers visibility into on-disk and in-memory execution of PowerShell and other scripting languages like VBScript, JScript, and Windows Management Instrumentation (WMI). AMSI events are not surfaced via Windows event logging, but they may be accessed through Event Tracing for Windows (ETW). We covered AMSI in depth in the Better Know a Data Source blog series.

Red Canary observes a large amount of AMSI bypasses within PowerShell payloads. A successful AMSI bypass can allow an adversary to disable logging of in-memory PowerShell execution. Fortunately, AMSI bypasses often entail a chicken-and-egg problem for adversaries, as AMSI logs the AMSI bypass attempt. It is for this reason that PowerShell detection strategies should account for robust detection of AMSI bypass attempts.

Windows Security Event ID 4104: Scriptblock logging

There are two levels of scriptblock logging: global and automatic. In Microsoft parlance, these operate at the "verbose" and "warning" levels, meaning that global scriptblock logging collects all PowerShell activity, whereas automatic scriptblock logging narrows the aperture slightly and focuses on script code that's more likely to include malicious content. Automatic scriptblock logging is enabled by default, and it logs PowerShell script code containing suspicious terms. Microsoft's list of suspicious terms includes the majority of the most commonly abused cmdlets and .NET APIs, so this level of logging is pretty reliable. Global scriptblock logging, on the other hand, must be enabled. It collects the same script content as automatic logging but isn't limited to what Microsoft deems "suspicious." It logs everything. Both methods write logs to Event ID 4104.

Note: Scriptblock logging is only compatible with PowerShell versions 5 and above.

Windows Security Event ID 400: PowerShell command-line logging

While the most effective PowerShell logging and telemetry are available in PowerShell versions 5 and above, there are some event sources that defenders can fall back on in cases where an adversary is leveraging an older version of PowerShell. Since PowerShell version 2, Event ID 400 in the "Windows PowerShell" classic event log has always provided context about PowerShell host process starts, including command-line logging.

Windows Security Event IDs 800 and 4103: Module loading and Add-Type logging

Module logging logs all loaded modules to Event ID 800 in the "Windows PowerShell" event log. This feature must be explicitly enabled. What isn't well documented though is that 800 events also log the contents of source code supplied to the Add-Type cmdlet. Adversaries often use Add-Type in order to compile and interact with C# code. Starting in PowerShell version 5, Add-Type context is also supplied in Event ID 4103 in the Microsoft-Windows-PowerShell/Operational log.

Sysmon Event IDs 1 and 7: Process creation and Image loaded

Since monitoring for suspicious PowerShell will involve the execution of the PowerShell process, Sysmon Event ID 1 warrants mention. However, Sysmon Event ID 7 is likely an even better source for gaining visibility into PowerShell since it will record any process that runs PowerShell by focusing on the modload for `System.Management.Automation.dll` and `System.Management.Automation.ni.dll`. Olaf Hartong's Sysmon Modular includes a PowerShell-specific configuration looking for these and other important modules, and it's a great place to start if you're interested in using Sysmon to track PowerShell activity.

Endpoint detection and response (EDR) tools

A good EDR product will provide detailed visibility into all of the data sources referenced above and offer great value to security teams seeking to detect adversaries abusing PowerShell.

Detection opportunities

Red Canary has 368 detection analytics designed to catch suspicious PowerShell activity, 120 of which raised events that converted to confirmed threat detections in 2025. Security teams seeking to detect malicious and suspicious PowerShell will want to look for process chains or combinations of process starts and command lines that suggest malicious activity. The following detection opportunities are adapted from the Red Canary behavioral analytics that find the most—or most impactful—threats.

Note: These detection analytics may require tuning.

PowerShell `-encodedcommand` switch

This detection analytic looks for the execution of `powershell.exe` with command lines that include variations of the `-encodedcommand` argument; PowerShell will recognize and accept anything from `-e` onward, and it will show up outside of the encoded bits.

```
 process == powershell.exe
&&
command_includes ('-e' || '-en' || '-enc' || '-enco'|| [any variation of the encoded command switch])
```

The following is an example encoded command that will print “tweet, tweet” to the console:

```
powershell.exe -encod VwByAGkAdABlAC0ASABvAHMAdAAgACIAdAB3AGUAZQB0ACwAIAB0AHcAZQBlAHQAIQAiAA==
```

PowerShell Base64 encoding

This analytic looks for the execution of a process that seems to be `powershell.exe` along with a corresponding command line containing the term `base64`. Base64 encoding isn’t inherently suspicious, but it’s worth looking out for in a lot of environments, and the following pseudo-detection logic can help detect a wide variety of malicious activity:

```
 process == powershell.exe
&&
command_includes ('base64')
```

Note: Beyond alerting on PowerShell that leverages Base64 encoding, consider leveraging a tool—like CyberChef, for example—that is capable of decoding encoded commands.

The following example highlights the execution of Base64-encoded PowerShell content:

```
Invoke-Expression -Command ([Text.Encoding]::Unicode.GetString([Convert]::FromBase64String('VwByAGkAdABlAC0ASAI
```

Obfuscation and escape characters

Obfuscation can disrupt detection logic by splitting commands or parameters or inserting extra characters (that are ignored by PowerShell). Monitor for the execution of PowerShell with unusually high counts of characters like `^`, `+`, `$`, and `%`.

```
 process == powershell.exe
&&
command_includes [high counts of] ('^' || '+' || '$' || '%')
```

There are myriad ways to obfuscate code in PowerShell. Here is one example of an obfuscated Write-Host command:

```
& ([ScriptBlock]::Create("Write-Host '$("{0}{1}{2}{2}{0}" -f 't','w','e'), $([Char] 116)$([Char] 119)$("$(([Cha
```

Suspicious PowerShell cmdlets

Many of our PowerShell detection analytics look for cmdlets, methods, and switches that may indicate malicious activity. The following analytic is by no means exhaustive but offers a few valuable examples of suspicious cmdlets and other oft-abused features to look out for:

```
 process == powershell.exe
&&
command_includes ('-nop' || '-noni' || 'invoke-expression' || 'iex' || '.downloadstring' || 'downloadfile')
```

The following example combines many of these suspicious cmdlets together to print “tweet, tweet” to the console:

```
powershell.exe -nop iex "\"Write-Host `\"$((New-Object Net.WebClient).DownloadString('https://gist.githubuserc
```

Suspicious script directories

Adversaries may attempt to evade process command-line monitoring for suspicious strings by using PowerShell to execute scripts instead. Script execution from unusual directories like public folders may indicate malicious activity. The following analytic provides an example of looking for PowerShell executing scripts from an unusual location:

```
 process == powershell.exe
&&
command_includes ('c:\users\public\' && '.ps1')
```

The following example shows a simple execution of a PowerShell script from the public directory:

```
 powershell.exe -nop C:\Users\Public\tweet.ps1
```

PowerShell creating remote TCP connection

Adversaries frequently abuse built-in PowerShell functionality to create a TCP reverse shell that enables remote control of a compromised device. The following pseudo detector should serve as a good starting point for developing detection coverage for this behavior.

```
 process == ('powershell')
&&
command_line_includes ['tcpclient']
&&
command_line_includes ['diagnostics.process' || 'redirectstandardinput' || 'iex' || 'invoke-expression' || '.inv
```

Weeding out false positives

Monitoring for encoded commands may seem like an easy win, and it is certainly a place to start. However, you will quickly find that many platforms and administrators leverage PowerShell and use encoded commands as a part of normal workflows. As such, flagging activity simply based on variations of the `-encodedcommand` switch may generate a significant amount of noise. Start with queries against offline or static data to get a feel for volume.

Once you have a better understanding of your overall volume, identify patterns within the decoded data. Leverage your knowledge of what is normal for your environment in order to identify what is potentially malicious. You may find that behaviors associated with certain permitted enterprise applications are triggering your detection analytics, and therefore you may need to create exclusions or tune detection logic accordingly.

An example of a common false positive is when users install Chocolatey per their installation instructions:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System
```

Testing

Start testing your defenses against PowerShell using Atomic Red Team—an open source testing framework of small, highly portable detection tests mapped to MITRE ATT&CK.

Getting started

View atomic tests for T1059.001: PowerShell. In most environments, these should be sufficient to generate a useful signal for defenders.

Run this test on a Windows system using PowerShell or Command Prompt:

```
powershell.exe -e JgAgACgAZwBjAG0AIAAoACcAaQBlAHsAMAB9ACcAIAAtAGYAIAAnAHgAJwApACkAIAAoACIAVwByACIAKwAiAGkAdAAi
```

What to expect

Running this test should print “Hello, from PowerShell!” to the terminal. Decoded, the command runs the following obfuscated PowerShell code:

```
& (gcm ('ie{0}' -f 'x')) ("Wr"+"it"+"e-H"+"ost 'H"+"el"+"lo, fr"+"om P"+"ow"+"erS"+"h"+"ell!'")
```

Deobfuscated, it is equivalent to the following:

```
Invoke-Expression "Write-Host 'Hello, from PowerShell!'"
```

Useful telemetry

Visibility	Telemetry	Collection
Visibility: **Process monitoring**	Telemetry: A `powershell.exe` process will start.	Collection: EDR software, Sysmon, and Windows Security Event ID 4688 will log relevant process telemetry.
Visibility:	Telemetry:	Collection:

Visibility	Telemetry	Collection
Command monitoring	Command-line logging will capture the encoded PowerShell content.	Windows Security Event IDs 1101, 4104, 400, 800, and 4103 will log relevant command telemetry.

If you'd like to conduct more advanced PowerShell testing, consider leveraging the Atomic Test Harness for PowerShell.

Review and repeat

Now that you have executed one or several common tests and checked for the expected results, it's useful to answer some immediate questions:

- Were any of your actions detected?
- Were any of your actions blocked or prevented?
- Were your actions visible in logs or other defensive telemetry?

Repeat this process, performing additional tests related to this technique. You can also create and contribute tests of your own.
