4

Detection – Auditing and Monitoring

Although organizations already try to harden their environments, only a few take into account that auditing and monitoring are two of the most important things when it comes to securing your environment.

For many years while working at Microsoft, I have preached the *protect*, *detect*, and *respond* approach. Most companies try to just *protect* their devices, but that's where they stop. To *detect* and *respond*, there needs to be not only a working **Security Operations Center** (**SOC**) in place but also infrastructure and resources.

Those people and resources require money – a budget that many companies don't want to spend in the first place, unless they have been breached.

When working with customers, I saw only a few environments with a working SOC in place, as well as the infrastructure to host a **Security Information and Event Management** (**SIEM**) system. I was really happy that when I left those customers, most of them started rethinking their approach and improved their security practices, as well as their monitoring and detection.

However, I also had customers that were already breached when I was introduced to them for the first time. Customers that never had the budget nor employees for detections suddenly had the budget to improve immediately, as soon as they were breached.

And over the years, I learned that it's not a question of *whether* an organization will be hacked – it is rather *when* they will be hacked, and *how long* the attacker stays in the environment unnoticed. That's if they are detected at all.

Therefore, I recommend to every IT decision-maker that I meet to *assume a breach* and protect what is important.

Over the years, I saw more and more organizations that actually had operating SOCs in place, which made me really happy. But unfortunately – especially when looking at small and medium-sized enterprises – most organizations have either no monitoring in place or are just starting their journey.

PowerShell has been covered in the media several times when it comes to attacks. Ransomware malware was distributed, sending malicious emails that launched PowerShell in the background to execute a payload, a file-less attack in which the malware does not need to be downloaded on the client but runs in the memory instead, and even legitimate system tools that have been abused by adversaries to execute their attacks (also known as **Living Off the Land** or **LOLbins**).

And yes, attackers like to leverage what they already find on a system. However, if organizations had not only the appropriate mitigations in place but also the right detection, it would make it way harder for adversaries to launch a successful attack and stay unnoticed.

Many tools that adversaries use in their attacks provide little to no transparency, so it can be really hard for defenders (a.k.a. the **blue team**) to detect and analyze such an attack.

PowerShell, in contrast, provides such amazing logging opportunities that it is quite easy to analyze and detect an attack that was launched using it. Therefore, if you are a blue teamer and you notice that you were targeted with a PowerShell-based attack, you are in luck (as much as you can be in luck if your infrastructure was attacked)! This makes it much easier for you to find out what happened.

Having an extensive (not exclusively restricted to) PowerShell logging infrastructure in place helps your SOC team to identify attackers and get insights into what commands and code adversaries executed. It also helps to improve your detection and security controls.

In this chapter, you will learn the basics of security monitoring with PowerShell, which will help you to get started with your detections or improve them. In this chapter, you will get a deeper understanding of the following topics:

- Configuring PowerShell Event Logging
- PowerShell Module Logging
- PowerShell Script Block Logging
- Protected Event Logging
- PowerShell transcripts
- Analyzing event logs
- Getting started with logging
- The most important PowerShell related event logs and IDs

Technical requirements

To get the most out of this chapter, ensure that you have the following:

- PowerShell 7.3 and above.
- Access to the GitHub repository for `Chapter04`:

https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter04

Configuring PowerShell Event Logging

Implementing robust auditing mechanisms for PowerShell to help you monitor, detect and prevent potential threats is an essential step to ensure effective security practices for PowerShell. By leveraging PowerShell logging, you can capture detailed information about PowerShell activities on your systems, which is essential for detecting and investigating security incidents. PowerShell logging can help you identify suspicious activities, such as the execution of malicious commands or the modification of critical system settings.

In this section, we will discuss the different types of PowerShell logging that you can enable, including PowerShell Module Logging, PowerShell Script Block Logging, Protected Event Logging, and PowerShell transcripts. We will also look into how to configure these logging features to meet your organization's specific security requirements.

PowerShell Module Logging

PowerShell Module Logging was added with **PowerShell 3.0**. This feature provides extensive logging of all PowerShell commands that are executed on the system. If Module Logging is enabled, pipeline execution events are generated and written to the `Microsoft-Windows-Powershell/Operational` event log in the context of event ID `4103`.

How to configure Module Logging

You can either enable Module Logging for the execution of a module in the current session, or you can configure it to be turned on permanently.

Enabling it only within a single session only makes sense if you want to troubleshoot the behavior of a certain module. If you want to detect the commands that adversaries run in your infrastructure, it makes sense to turn on Module Logging permanently.

To enable Module Logging within the current session, only for a certain module, you need to import the module first. In this example, we will use the `EventList` module:

```
> Import-Module EventList
> (Get-Module EventList).LogPipelineExecutionDetails = $true
> (Get-Module EventList).LogPipelineExecutionDetails
True
```

Of course, you can replace the module name, `EventList`, with any other module name that you want to log pipeline execution details for:

```
Import-Module <Module-Name>
(Get-Module <Module-Name>).LogPipelineExecutionDetails = $true
```

If you want to monitor a managed environment, you don't want to enable PowerShell Module Logging manually on every host. In this case, you can use Group Policy to enable Module Logging.

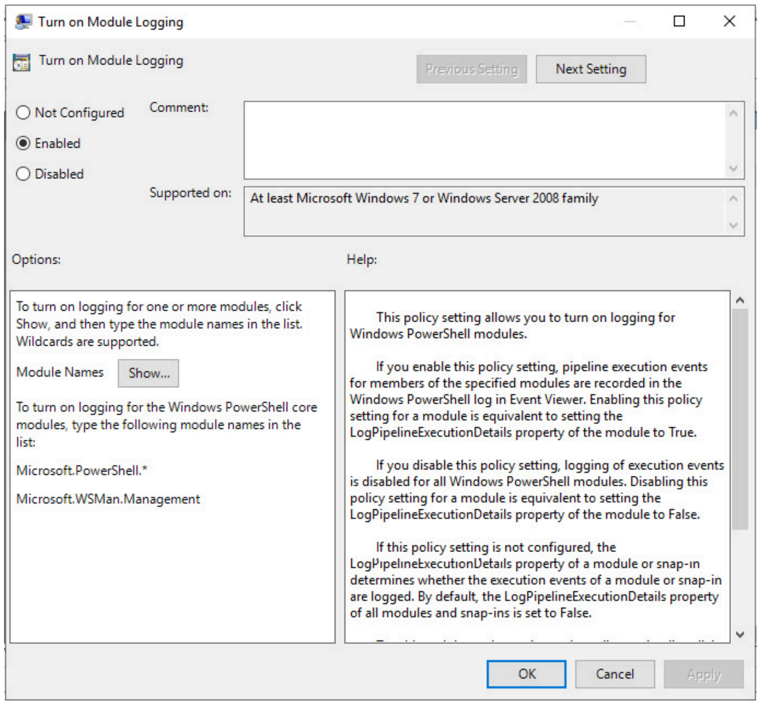
Create a new **Group Policy Object** (**GPO**). As Windows PowerShell and PowerShell Core were designed to co-exist and can be configured individually, it depends on what PowerShell version you want to configure:

- To configure Windows PowerShell, navigate to **Computer Configuration** | **Policies** | **Administrative Templates** | **Windows Components** | **Windows PowerShell**
- To configure PowerShell Core, navigate to **Computer Configuration** | **Administrative Templates** | **PowerShell Core**

WHERE ARE MY POWERSHELL CORE .ADMX TEMPLATES?

If you haven't imported the `.admx` *templates into your Group Policies yet to configure PowerShell Core, please refer to* ***Chapter 1****, Getting Started with PowerShell.*

Select and edit the **Turn on Module Logging** policy. A window opens to configure Module Logging:



Not Configured [✓ Enabled] Disabled

Comment:

Supported on: At least Microsoft Windows 7 or Windows Server 2008 family

Options:

To turn on logging for one or more modules, click Show, and then type the module names in the list. Wildcards are supported.

Module Names

To turn on logging for the Windows PowerShell core modules, type the following module names in the list:

Microsoft.PowerShell.*

Microsoft.WSMan.Management

Help:

This policy setting allows you to turn on logging for Windows PowerShell modules.

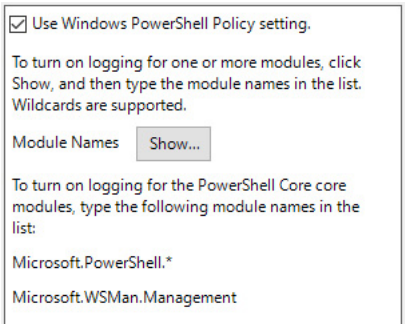
If you enable this policy setting, pipeline execution events for members of the specified modules are recorded in the Windows PowerShell log in Event Viewer. Enabling this policy setting for a module is equivalent to setting the LogPipelineExecutionDetails property of the module to True.

If you disable this policy setting, logging of execution events is disabled for all Windows PowerShell modules. Disabling this policy setting for a module is equivalent to setting the LogPipelineExecutionDetails property of the module to False.

If this policy setting is not configured, the LogPipelineExecutionDetails property of a module or snap-in determines whether the execution events of a module or snap-in are logged. By default, the LogPipelineExecutionDetails property of all modules and snap-ins is set to False.

Figure 4.1 – Configuring Module Logging for Windows PowerShell via Group Policy

For PowerShell Core, the configuration Window looks almost the same, except for the **Use Windows PowerShell Policy setting.** option. If this option is selected, PowerShell Core relies on the existing Windows PowerShell configuration.



☑ Use Windows PowerShell Policy setting.

To turn on logging for one or more modules, click Show, and then type the module names in the list. Wildcards are supported.

Module Names Show...

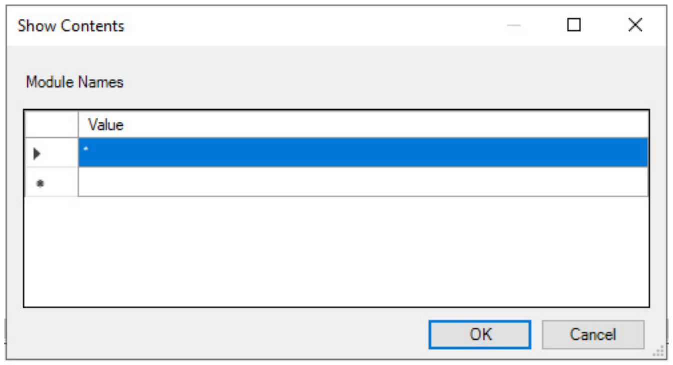
To turn on logging for the PowerShell Core core modules, type the following module names in the list:

Microsoft.PowerShell.*

Microsoft.WSMan.Management

Figure 4.2 – Configure Module Logging for PowerShell Core via Group Policy

Enable **Use Windows PowerShell Policy setting** if you want to only use one GPO for your Module Logging configuration. Next, depending on your configuration, either in the Windows PowerShell or PowerShell Core Module Logging GPO, go to **Module Names**, and click on the **Show...** button to configure the modules for which Module Logging should be turned on. A new window opens.



Show Contents

Module Names

	Value
▶	*
*	

Figure 4.3 – Configuring a wildcard (*) to log all modules

Now, you can configure single modules for which Module Logging should be turned on, but for security monitoring, it makes sense to monitor all Module Logging events – no matter which module was executed.

You can achieve this by configuring a wildcard (*****) as a module name. Confirm twice with **OK** and exit the GPO editor to make your changes active.

Of course, you can also add Module Logging for a single instance instead of monitoring all of them by specifying the module name as a value. However, I recommend logging all PowerShell activity (*****), which is especially useful if adversaries import custom PowerShell modules.

All events generated by this configuration can be found in the Microsoft Windows PowerShell Operational event log (`Microsoft-Windows-Powershell/Operational`).

PowerShell Script Block Logging

A **script block** is a collection of expressions and commands that is grouped together and executed as one unit. Of course, a single command can be also executed as a script block.

Many commands support the **`-ScriptBlock`** parameter, such as the **`Invoke-Command`** command. which you can use to run entire script blocks, locally or remotely:

```
> Invoke-Command -ComputerName PSSec-PC01 -ScriptBlock {Restart-Service -Name Spooler -Verbose}
VERBOSE: Performing the operation "Restart-Service" on target "Print Spooler (Spooler)".
```

It is important to note that all actions performed in PowerShell are considered script blocks and will be logged if *Script Block Logging* is enabled – regardless of whether or not they use the **`-ScriptBlock`** parameter.

Most of the time, companies and organizations do not care about logging and event log analysis unless a security incident occurs. However, by that point, it is already too late to enable logging retroactively. Therefore, the PowerShell team made the decision that security-relevant script blocks should be logged by default.

Starting with PowerShell 5, a *basic version of Script Block Logging* is enabled by default – only scripting techniques that are commonly used in malicious attacks are written to the **`Microsoft-Windows-Powershell/Operational`** event log.

This basic version of Script Block Logging does not replace full Script Block Logging; it should only be considered as a last resort, if logging was not in place when an attack happened.

If you want to protect your environment and detect malicious activities, you still should consider turning on *full Script Block Logging*.

Additionally, there's an even more verbose option when configuring Script Block Logging – *Script Block Invocation Logging*.

By default, only script blocks are logged the first time they are used. Configuring Script Block Invocation Logging also generates events every time script blocks are invoked and when scripts start or stop.

Enabling Script Block Invocation Logging can generate a high volume of events, which may flood the log and roll out useful security data from other events. Be careful with enabling Script Block Invocation Logging, as a high volume of events will be generated – usually, you don't need it for incident analysis.

How to configure Script Block Logging

There are several ways to configure Script Block Logging – manually as well as centrally managed. Let's have a look at what needs to be configured to log all the code executed in your environment.

To manually enable Script Block Logging, you can edit the registry. The settings that you want to change are within the following registry path:

```
HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging
```

Using the **`EnableScriptBlockLogging`** (**`REG_DWORD`**) registry key, you can configure to enable Script Block Logging:

- **Enabled**: Set the value to **`1`** to enable it
- **Disabled**: Set the value to **`0`** to disable it

If Script Block Logging is enabled, you will find all the executed code under event ID **`4104`**.

Using the **`EnableScriptBlockInvocationLogging`** (**`REG_DWORD`**) registry key, you can configure it to enable Script Block Invocation Logging (event IDs **`4105`** and **`4106`**):

- **Enabled**: Set the value to **`1`** to enable it
- **Disabled**: Set the value to **`0`** to disable it

If Script Block Logging, as well as Script Block Invocation Logging, is enabled, event IDs **`4105`** and **`4106`** will be generated.

If Script Block Invocation Logging is enabled, a lot of noise is generated and the log file size increases. Therefore, the maximum size should be reconfigured (see the *Increasing log size* section). For general security monitoring, you won't need to configure verbose Script Block Logging.

You can configure Script Block Logging manually by running the following commands in an elevated PowerShell console:

```
New-Item -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging" -Force
Set-ItemProperty -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging" -
```

The first command creates all the registry keys if they don't exist yet, and the second one enables Script Block Logging.

When enabling **`ScriptBlockLogging`** using the described commands, **`ScriptBlockLogging`** will be enabled for both 32-bit and 64-bit applications. You can verify that both settings were configured under the following:

- **`HKLM:\HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging`**
- **`HKLM:\HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Policies\Microsoft\Windows\PowerShell\ScriptBlockLog`**

In managed environments, it makes sense to manage your machines centrally. Of course, this can be done via PowerShell and/or **Desired State Configuration** (**DSC**), but it can be also done using Group Policy.

Create a new GPO. Depending on which PowerShell version you want to configure, navigate to either of the following:

- **Computer Configuration** | **Policies** | **Administrative Templates** | **Windows Components** | **Windows PowerShell** for Windows PowerShell
- **Computer Configuration** | **Administrative Templates** | **PowerShell Core** for PowerShell Core

Select and edit the **Turn on PowerShell Script Block Logging** policy. A window will open to configure Module Logging.

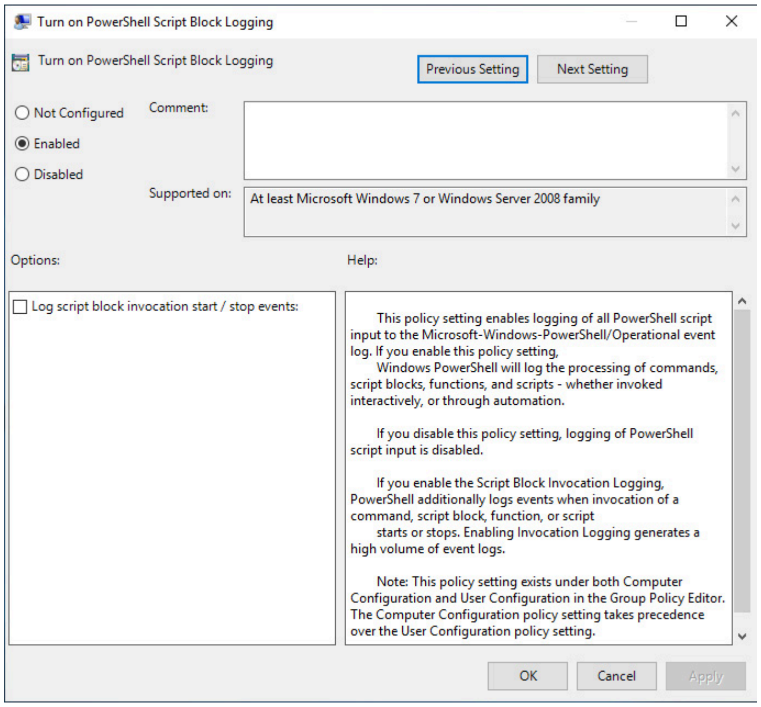
If you decide to configure the **Log script block invocation start** / **stop events** option, a lot more events will be generated, and a lot of noise will be generated. Depending on your use case, this option might be interesting nevertheless, but if you have just started doing security monitoring, I advise to not turn on this option.

INCREASING THE LOG SIZE FOR SCRIPT BLOCK INVOCATION LOGGING

If Script Block Invocation Logging is enabled, using the **Log script block invocation start** / **stop events** *option, the log file size increases, and the maximum size should be reconfigured.*

Event ID `4105` and `4106` will only be generated if the **Log script block invocation start** / **stop events** option is enabled.

In our example, we will *not* configure **Log script block invocation start** / **stop events** to avoid noise; therefore, we'll leave the checkbox unchecked:



Turn on PowerShell Script Block Logging

Not Configured [✓ Enabled] Disabled

Comment:

Supported on: At least Microsoft Windows 7 or Windows Server 2008 family

Options:

☐ Log script block invocation start / stop events:

Help:

This policy setting enables logging of all PowerShell script input to the Microsoft-Windows-PowerShell/Operational event log. If you enable this policy setting,
Windows PowerShell will log the processing of commands, script blocks, functions, and scripts - whether invoked interactively, or through automation.

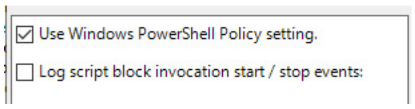
If you disable this policy setting, logging of PowerShell script input is disabled.

If you enable the Script Block Invocation Logging, PowerShell additionally logs events when invocation of a command, script block, function, or script
starts or stops. Enabling Invocation Logging generates a high volume of event logs.

Note: This policy setting exists under both Computer Configuration and User Configuration in the Group Policy Editor. The Computer Configuration policy setting takes precedence over the User Configuration policy setting.

Figure 4.4 – Turning on PowerShell Script Block Logging for Windows PowerShell

In the PowerShell Core policy, you will – as with the PowerShell Module Logging policy and some other policies – find the option to use the current Windows PowerShell Policy setting as well for PowerShell Core.



☑ Use Windows PowerShell Policy setting.

☐ Log script block invocation start / stop events:

Figure 4.5 – Turning on PowerShell Script Block Logging for PowerShell Core

All events generated by this configuration can be found in the Microsoft Windows PowerShell Operational event log (`Microsoft-Windows-Powershell/Operational`), or for PowerShell Core, in the PowerShell Core event log (`PowerShellCore/Operational`).

Protected Event Logging

Event logging is a sensitive topic. Often, sensitive information such as passwords is exposed and written to the event log.

Sensitive information is pure gold in the hand of an adversary who has access to such a system, so to counter this, beginning with Windows 10 and PowerShell version 5, Microsoft introduced Protected Event Logging.

Protected Event Logging encrypts data using the **Internet Engineering Task Force (IETF) Cryptographic Message Syntax** (**CMS**) standard, which relies on public key cryptography. This means that a public key is deployed on all systems that should support Protected Event Logging. Then, the public key is used to encrypt event log data before it is forwarded to a central log collection server.

On this machine, the highly sensitive private key is used to decrypt the data, before the data is inserted into the SIEM. This machine is sensitive and, therefore, needs special protection.

Protected Event Logging is not enabled by default and can currently only be used with PowerShell event logs.

Enabling Protected Event Logging

To enable Protected Event Logging, you can deploy a *base64-encoded X.509* certificate or another option (for example, deploying a certificate through **Public Key Infrastructure (PKI)** and providing a thumbprint, or providing a path to a local or file share-hosted certificate). In our example, we'll use a *base64-encoded X.509* certificate.

Here are the certificate requirements:

- The certificate must also have the *"Document Encryption"* **Enhanced Key Usage** (**EKU**) with the OID number (`1.3.6.1.4.1.311.80.1`) included
- The certificate properties must include either the "*Data Encipherment*" or "*Key Encipherment*" key usage

There's a great SANS blog post where you can see how to check your certificate's properties: **https://www.sans.org/blog/powershell-protect-cmsmessage-example-code/**.

Protected Event Logging leverages **IETF CMS** to secure the event log content. Therefore, you can also refer to the documentation pages for the

`Protect-CMSMessage` and **`Unprotect-CMSMessage`** cmdlets for more information on encrypting and decrypting using CMS:

- **`Protect-CMSMessage`**: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/protect-cmsmessage
- **`Unprotect-CMSMessage`**: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/unprotect-cmsmessage

Be careful that the certificate file that you plan to deploy **does not** contain the private key. Once you have obtained the certificate, you can either enable it manually or by using Group Policy.

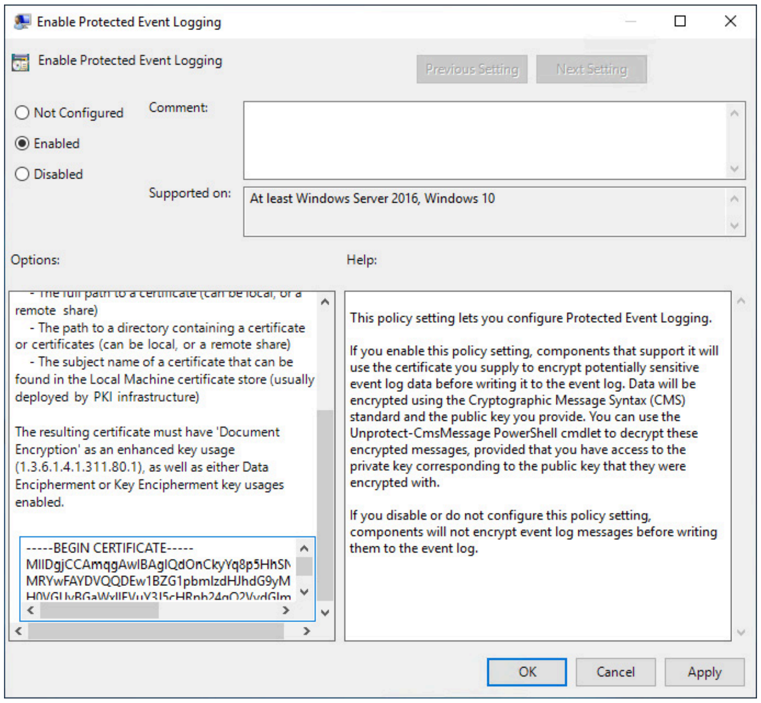
In the blog post *PowerShell ♥ the blue team*, the PowerShell team provides you with the **`Enable-ProtectedEventLogging`** function, which you can use to enable Protected Event Logging using PowerShell: https://devblogs.microsoft.com/powershell/powershell-the-blue-team/#protected-event-logging.

To leverage this script, save your certificate in the **`$cert`** variable, which you will use in the second command to pass the public key certificate to the **`Enable-ProtectedEventLogging`** function, enabling Protected Event Logging on the local system:

```
> $cert = Get-Content C:\tmp\PEL_certificate.cer -Raw
> Enable-ProtectedEventLogging -Certificate $cert
```

You can also enable Protected Event Logging using Group Policy. Create a new GPO or reuse an existing GPO, and then navigate to **Computer Configuration** | **Policies** | **Administrative Templates** | **Windows Components** | **Event Logging**.

Open the **Enable Protected Event Logging** policy.


Enable Protected Event Logging

Enable Protected Event Logging

Not Configured [✓ Enabled] Disabled

Comment:

Supported on: At least Windows Server 2016, Windows 10

Options:

- The full path to a certificate (can be local, or a remote share)
- The path to a directory containing a certificate or certificates (can be local, or a remote share)
- The subject name of a certificate that can be found in the Local Machine certificate store (usually deployed by PKI infrastructure)

The resulting certificate must have 'Document Encryption' as an enhanced key usage (1.3.6.1.4.1.311.80.1), as well as either Data Encipherment or Key Encipherment key usages enabled.

```
-----BEGIN CERTIFICATE-----
MIIDgjCCAmqgAwIBAgIQdOnCkyYq8p5HhSN
MRYwFAYDVQQDEw1BZG1pbmlzdHJhdG9yM
H0VGUvBGaWxlIFVuY3I5cHRpb24gO2VvdGlm
```

Help:

This policy setting lets you configure Protected Event Logging.

If you enable this policy setting, components that support it will use the certificate you supply to encrypt potentially sensitive event log data before writing it to the event log. Data will be encrypted using the Cryptographic Message Syntax (CMS) standard and the public key you provide. You can use the Unprotect-CmsMessage PowerShell cmdlet to decrypt these encrypted messages, provided that you have access to the private key corresponding to the public key that they were encrypted with.

If you disable or do not configure this policy setting, components will not encrypt event log messages before writing them to the event log.

Figure 4.6 – Enabling Protected Event Logging

Set **Enable Protected Event Logging** to **Enabled**, provide your certificate, and confirm with **OK**.

Use the **`Unprotect-CmsMessage`** cmdlet on a secure and protected system to decrypt the data before storing it in your SIEM, provided that an appropriate decryption certificate (that is, the one that has the private key) is installed on the machine.

To decrypt the data before storing it in your SIEM, make use of the **`Unprotect-CmsMessage`** cmdlet on a secure and protected system, where an appropriate decryption certificate containing the private key is installed:

```
> Get-WinEvent Microsoft-Windows-PowerShell/Operational | Where-Object Id -eq 4104 | Unprotect-Cms
```

In this example, all events from the Operational PowerShell log with the event ID **`4104`** will be decrypted, assuming the private key is present.

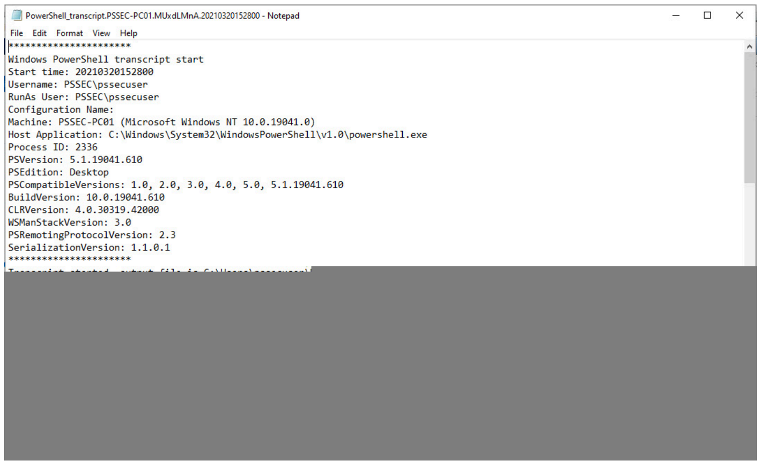
There is also an option to document what exactly was run in a session and what output was shown. This option is called a transcript – let's have a closer look in our next section.

PowerShell transcripts

PowerShell transcripts have been available in PowerShell since PowerShell version 1.0 as part of the **`Microsoft.PowerShell.Host`** module. Transcripts are a great way to monitor what happens in a PowerShell session.

If a PowerShell transcript is started, all executed PowerShell commands and their output are recorded and saved into the folder that was specified. If not specified otherwise, the default output folder is the **`My Documents`** folder (**`%userprofile%\Documents`**) of the current user.

The following screenshot is an example of how such a transcript could look.


PowerShell_transcript.PSSEC-PC01.MUxdLMnA.20210320152800 - Notepad

```
**********************
Windows PowerShell transcript start
Start time: 20210320152800
Username: PSSEC\pssecuser
RunAs User: PSSEC\pssecuser
Configuration Name:
Machine: PSSEC-PC01 (Microsoft Windows NT 10.0.19041.0)
Host Application: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Process ID: 2336
PSVersion: 5.1.19041.610
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.19041.610
BuildVersion: 10.0.19041.610
CLRVersion: 4.0.30319.42000
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
**********************
```

Figure 4.7 – A screenshot of a PowerShell transcript

The name of the **`.txt`** file starts with **`PowerShell_transcript`**, followed by **`computername`**, a random string, and a time stamp.

This is a typical example of a PowerShell transcript filename that was started on *PSSec-PC01* – **`PowerShell_transcript.PSSEC-PC01.MUxdLMnA.20210320152800.txt`**.

How to start transcripts

There are several options for enabling transcripts. However, the simplest method to record PowerShell transcripts is by simply typing the **`Start-Transcript`** command in the current session and hitting *Enter*. In this case, only commands that are run in this local session will be captured.

When running the **`Start-Transcript`** cmdlet directly, the most interesting parameters are **`-OutputDirectory`**, **`-Append`**, **`-NoClobber`**, and **`-IncludeInvocationHeader`**:

- **`-Append`**: The new transcript will be added to an existing file.
- **`-IncludeInvocationHeader`**: Time stamps when commands are run are added to the transcript, along with a delimiter between commands to make the transcripts easier to parse through automation.
- **`-NoClobber`**: This transcript will not overwrite an existing file. Normally, if a transcript already exists in the defined location (for example, if the defined file has the same name as an already existing file, or the filename was configured using the **`-Path`** or **`-LiteralPath`** parameter), **`Start-Transcript`** overwrites this file without warning.
- **`-OutputDirectory`**: Using this parameter, you can configure the path where your transcripts can be stored.
- **`-UseMinimalHeader`**: This parameter was added in **PowerShell version 6.2** and ensures that only a short header is prepended instead of the detailed header.

Read more about the full list of parameters in the `Start-Transcript` help files or in the official PowerShell documentation: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.host/start-transcript?view=powershell-7#parameters.

SECURING YOUR TRANSCRIPTS

As with any security logging you collect, it's important to ensure that your transcripts are securely stored to prevent attackers from tampering with them. Make sure to configure a secure path that is difficult for attackers to access, taking into consideration the possibility of stolen corporate identities. Once an attacker gains access to transcripts, they can modify them and render your detection efforts useless.

Transcripts that were initialized with `Start-Transcript` are only recorded as long as the session is active or until `Stop-Transcript` is executed, which stops the recording of executed PowerShell commands.

Enabling transcripts by default

To enable transcripts *by default* on a system, you can either configure transcripts via a **registry** or by using **Group Policy** to configure transcripts for multiple systems.

Enabling transcripts by registry or script

When PowerShell transcripts are configured, the following registry hive is used:

```
HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription
```

For example, to enable transcription, using invocation headers and the `C:\tmp` output folder, you need to configure the following values to the registry keys:

- `[REG_DWORD]EnableTranscripting = 1`
- `[REG_DWORD]EnableInvocationHeader = 1`
- `[REG_SZ]OutputDirectory = C:\tmp`

To manage multiple machines, it's more comfortable to use GPO, but in some cases, some machines are not part of the Active Directory domain; hence, they cannot be managed. For this example, I have added the `Enable-PSTranscription` function to the GitHub repository for this book: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter04/Enable-PSTranscription.ps1.

Load the `Enable-PSTranscription` function into the current session and specify the folder where your transcripts should be saved, such as the following:

```
> Enable-PSTranscription -OutputDirectory "C:\PSLogs"
```

If no **`-OutputDirectory`** is specified, the script will write transcripts into **`C:\ProgramData\WindowsPowerShell\Transcripts`** as the default option.

This function just configures all defined values and overwrites your existing registry keys. Feel free to adjust the function to your needs and to reuse it.

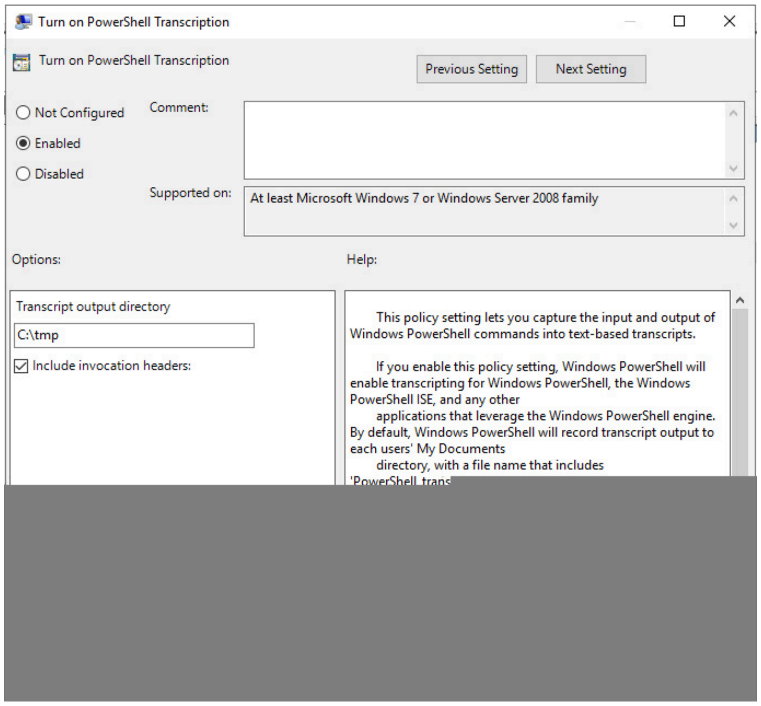
As soon as a new session is started, transcripts will be written to the configured folder.

Enabling transcripts using Group Policy

In Active Directory-managed environments, the easiest way to configure transcripts is by using Group Policy.

Create a new GPO or reuse an existing one. Then, navigate to **Computer Configuration** | **Policies** | **Administrative Templates** | **Windows Components** | **Windows PowerShell**.

Double-click and open the **Turn on PowerShell Transcription** policy to configure PowerShell transcription:


Turn on PowerShell Transcription
Turn on PowerShell Transcription
Previous Setting
Next Setting
Comment:
Options:
Help:

○ Not Configured
● Enabled
○ Disabled

Supported on: At least Microsoft Windows 7 or Windows Server 2008 family

Transcript output directory: C:\tmp

☑ Include invocation headers:

This policy setting lets you capture the input and output of Windows PowerShell commands into text-based transcripts.

If you enable this policy setting, Windows PowerShell will enable transcripting for Windows PowerShell, the Windows PowerShell ISE, and any other applications that leverage the Windows PowerShell engine. By default, Windows PowerShell will record transcript output to each users' My Documents directory, with a file name that includes 'PowerShell_trans

Figure 4.8 – Turning on PowerShell transcription

Set the policy to **Enabled**, and select whether a transcript output directory and invocation headers should be included. If the output directory is not specified, transcriptions are saved to the **`My Documents`** folder of the current user (**`%userprofile%\Documents`**).

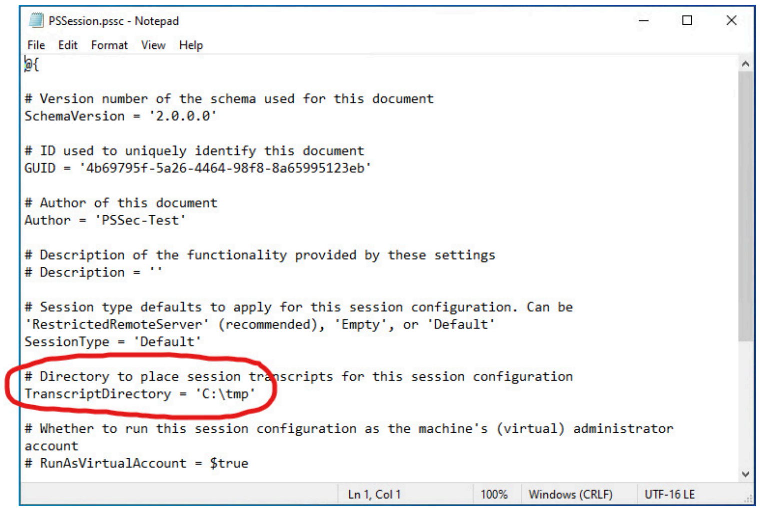
Enabling transcripts for PowerShell Remoting sessions

Custom endpoints are an excellent way to apply default settings to PowerShell Remoting sessions. If transcriptions were configured, they will be enabled by default for local sessions, but configuring them additionally in **Just Enough Administration** allows you to group and collect logs specific to that endpoint when used for remote sessions. By configuring transcription and other settings on a custom endpoint, you can enforce these settings for all remote sessions connected to that endpoint, making it easier to ensure consistency and compliance across your environment.

To get started, create a session configuration file, using the **New-PSSessionConfigurationFile** cmdlet with the **-TranscriptDirectory** parameter to specify where transcripts should be written to:

```
> New-PSSessionConfigurationFile -Path "$env:userprofile\Documents\PSSession.pssc" -TranscriptDire
```

This command creates a new session configuration file, enforcing transcription, and stores it in **%userprofile%\Documents\PSSession.pssc**, the path that was defined within the **-Path** parameter.



```
@{

# Version number of the schema used for this document
SchemaVersion = '2.0.0.0'

# ID used to uniquely identify this document
GUID = '4b69795f-5a26-4464-98f8-8a65995123eb'

# Author of this document
Author = 'PSSec-Test'

# Description of the functionality provided by these settings
# Description = ''

# Session type defaults to apply for this session configuration. Can be
'RestrictedRemoteServer' (recommended), 'Empty', or 'Default'
SessionType = 'Default'

# Directory to place session transcripts for this session configuration
TranscriptDirectory = 'C:\tmp'

# Whether to run this session configuration as the machine's (virtual) administrator
account
# RunAsVirtualAccount = $true
```

Figure 4.9 – The newly created session configuration

We introduced custom endpoints in ***Chapter 3***, *Exploring PowerShell Remote Management Technologies and PowerShell Remoting*, and we will dive deeper into Just Enough Administration in ***Chapter 10***, *Language Modes and Just Enough Administration (JEA)*. To learn more about the concept of custom endpoints and Just Enough Administration, please make sure to review both chapters.

Best practices for PowerShell transcripts

As a security best practice, *use session transcripts for every user*. This does not mean that your administrators are doing nasty stuff on your machines and they need to be monitored. In no way do I encourage mistrust in your own staff. However, credential theft is a real threat, and if your administrator's identity is stolen and misused, you will be happy to understand what was done by the adversary.

If you use transcripts, make sure that they cannot be modified. If they can be altered by an attacker, they are of almost no use at all.

So, make sure to provide a path to a preconfigured folder, and specify it either via a GPO, manual configuration, or in the session configuration file. Prevent all users from modifying or deleting any data in this folder.

The local system account requires read and write access, so make sure to configure the access permissions accordingly.

And last but not least, it makes sense to forward all the transcript files to a central logging server or your SIEM to analyze them regularly.

One effective approach to centralizing the transcript files is to configure their destination as a **Uniform Naming Convention (UNC)** path with a dynamic filename. For example, you can set the transcript directory to a network share with write-only permission, using the PowerShell profile to log all activity to a file with a unique name, such as the following:

```
\\server\share$\env:computername-$($env:userdomain)-$($env:username)-$(Get-Date Format YYYYMMddhhm
```

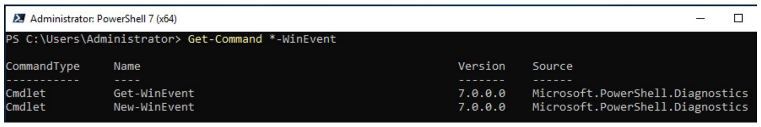
Also, ensure that this share is not readable by normal users. By using this approach, you can easily collect and analyze the logs from all machines in a centralized location, allowing you to better detect and respond to security incidents without the need to set up an entire logging infrastructure.

In addition to collecting logs, analyzing them is equally important. In the next section, we will explore the techniques and tools used for log analysis.

Analyzing event logs

There are several ways to work with Windows event logs using PowerShell. Of course, you can always forward your event logs to the SIEM of your choice, but sometimes, it happens that you want to directly analyze the event logs on a certain machine. For this use case, it makes sense to look at the available options that come with PowerShell.

The easiest option if you just want to analyze events or create new events is the **`*-WinEvent`** cmdlets, which are still available in PowerShell Core 7. You can use **`Get-Command`** to find all available cmdlets:

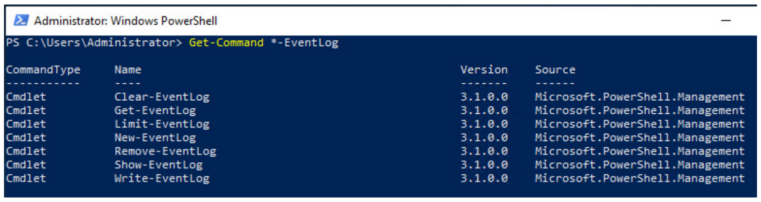


```
Administrator: PowerShell 7 (x64)
PS C:\Users\Administrator> Get-Command *-WinEvent

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Get-WinEvent                                       7.0.0.0    Microsoft.PowerShell.Diagnostics
Cmdlet          New-WinEvent                                       7.0.0.0    Microsoft.PowerShell.Diagnostics
```

Figure 4.10 – The available *-WinEvent cmdlets

In PowerShell 5.1, there was also the possibility of using the **`*-EventLog`** cmdlets, but they were removed in PowerShell Core 6 and above. Since PowerShell 5.1 is installed by default on all Windows 10 operating systems, I refer to **`*-EventLog`** here. Again, use **`Get-Command`** to find all available cmdlets:



```
Administrator: Windows PowerShell
PS C:\Users\Administrator> Get-Command *-EventLog

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Clear-EventLog                                     3.1.0.0    Microsoft.PowerShell.Management
Cmdlet          Get-EventLog                                       3.1.0.0    Microsoft.PowerShell.Management
Cmdlet          Limit-EventLog                                     3.1.0.0    Microsoft.PowerShell.Management
Cmdlet          New-EventLog                                       3.1.0.0    Microsoft.PowerShell.Management
Cmdlet          Remove-EventLog                                    3.1.0.0    Microsoft.PowerShell.Management
Cmdlet          Show-EventLog                                      3.1.0.0    Microsoft.PowerShell.Management
Cmdlet          Write-EventLog                                     3.1.0.0    Microsoft.PowerShell.Management
```

Figure 4.11 – The available *-EventLog cmdlets

The third option is to use **`wevtutil`**. This command-line executable is not very intuitive to understand, but it can be used to operate and analyze event logs. Using the **`/?`** parameter, you can get more details on the usage.

Figure 4.12 – wevtutil.exe usage

For example, clearing the **`Security`** event log can be achieved with the following command:

```
> wevtutil.exe cl Security
```

Refer to the official documentation to get more details on **`wevtutil`**: https://docs.microsoft.com/de-de/windows-server/administration/windows-commands/wevtutil.

Finding out which logs exist on a system

If you want to find out which event logs exist on a system, you can leverage the **`-ListLog`** parameter followed by a wildcard (**`*`**) – **`Get-WinEvent -ListLog *`**:

Figure 4.13 – Listing all event logs

You might want to pipe the output to **`Sort-Object`** to sort by record count, maximum log size, log mode, or log name.

Querying events in general

To get started, let's have a look how we can analyze some of the most common scenarios for PowerShell auditing.

Using the **`Get-WinEvent`** command, you can get all the event IDs from the event log that you specified – **`Get-WinEvent Microsoft-Windows-PowerShell/Operational`**:

Figure 4.14 – Querying the Microsoft Windows PowerShell Operational log

In this example, you would see all event IDs that were generated in the PowerShell Operational log.

If you only want to query the last *x* events, the **`-MaxEvents`** parameter will help you to achieve this task. For example to query the last 15 events of the *security* event log use **`Get-WinEvent Security -MaxEvents 15`**:

Figure 4.15 – Querying the last 15 events from the Security event log

This is especially helpful if you want to analyze recent events without querying the entire event log.

Using the **`-Oldest`** parameter reverts the order so that you see the oldest events in this log – **`Get-WinEvent Security -MaxEvents 15 -Oldest`**:

Figure 4.16 – The 15 oldest events from the Security event log

To find all events in the Microsoft Windows PowerShell Operational log that contain code that was executed and logged by **`ScriptBlockLogging`**, filter for event id **`4104`**: **`Get-WinEvent Microsoft-Windows-PowerShell/Operational | Where-Object { $_.Id -eq 4104 } | fl`**:

Figure 4.17 – Finding all executed and logged code

You can also filter for certain keywords in the message part. For example, to find all events that contain the **`"logon"`** string in the message, use the **`-match`** comparison operator – **`Get-WinEvent Security | Where-Object { $_.Message -match "logon" }`**:

Figure 4.18 – Finding all events that contain "logon" in their message

You can also filter using XPath-based queries, using the **`-FilterXPath`** parameter:

```
Get-WinEvent -LogName "Microsoft-Windows-PowerShell/Operational" -FilterXPath "*[System[(EventID=4
```

The output is shown in the following screenshot:

Figure 4.19 – Filtering using an XPath query

It is also possible to filter by a specified **hash table**, using the **-FilterHashtable** parameter:

```
> $eventLog = @{ ProviderName="Microsoft-Windows-PowerShell"; Id = 4104 }
> Get-WinEvent -FilterHashtable $eventLog
```

Using hash tables can reduce your usage of **Where-Object** filter clauses significantly.

If you want to query complex event structures, you can use the **-FilterXml** parameter and provide an **XML** string. I have prepared such an example and uploaded it to this book's GitHub repository: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter04/Get-AllPowerShellEvents.ps1:

Figure 4.20 – Using the Get-AllPowerShellEvents.ps1 script

This example queries the **Microsoft-Windows-PowerShell/Operational**, **PowerShellCore/Operational**, and **Windows PowerShell** event logs and retrieves all the events that I will describe in the *Basic PowerShell event logs* section in this chapter.

Now that you know how to work with event logs and query events, let's look at how to detect and analyze which code was run on a system.

Which code was run on a system?

Filtering and scrolling through all events that contain executed code can be a tedious task, if you decide to perform this task manually. But, thankfully, PowerShell allows you to automate this task and quickly find what you are searching for.

In general, all events that contain logged code can be found either in the Microsoft Windows PowerShell or the PowerShell Core Operational log, indicated by event ID **4104**:

```
> Get-WinEvent Microsoft-Windows-PowerShell/Operational | Where-Object Id -eq 4104
> Get-WinEvent PowerShellCore/Operational | Where-Object Id -eq 4104
```

To better find and filter what code was executed, I have written the **`Get-ExecutedCode`** function, which you can find in the GitHub repository for this book: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter04/Get-ExecutedCode.ps1**.

Downgrade attack

As newer versions such as 5.1 and upward introduced a lot of new security features, older PowerShell versions such as version 2.0 became more attractive to attackers. Therefore, a common way to leverage older versions is a so-called **downgrade attack**.

A downgrade attack can be executed by specifying the version number when running **`powershell.exe`**:

```
> powershell.exe -version 2 –command <command>
```

If the specified version is installed, the command runs, using the deprecated binary, which implies that only security features that existed when that version was written are applied.

All machines that run Windows 7 and above have at least PowerShell version 2.0 installed. Although Windows 7 is not supported and does not receive any security updates anymore, it is still widespread.

Additionally, PowerShell version 2.0 still relies on **.NET Framework 2.0**, which does not include advanced security features and provides no advanced logging. Therefore, that's perfect for attackers that do not want anybody to know what they did on your system.

.NET Framework 2.0 is not included by default on Windows 10, but it can be installed manually – for example, by an attacker or an administrator. On operating systems prior to Windows 10, .NET Framework 2.0 is installed by default.

On Windows 8, PowerShell version 2.0 can be disabled by running the following command in an elevated console:

```
Disable-WindowsOptionalFeature -Online -FeatureName MicrosoftWindowsPowerShellV2Root
```

.NET Framework 2.0, which is required to run PowerShell version 2.0, is by default not installed on newer systems such as Windows 10.

So, if you try to run **`powershell.exe -version 2`**, you get an error message, stating that version 2 of .NET Framework is missing:

```
> powershell.exe -version 2
Version v2.0.50727 of the .NET Framework is not installed and it is required to run version 2 of W
```

As .NET Framework 2.0 can be installed manually – either by system administrators or attackers – make sure to check for PowerShell version 2.0 and disable it.

Run the following command to check whether PowerShell version 2.0 is enabled or disabled:

```
> Get-WindowsOptionalFeature -Online | Where-Object {$_.FeatureName -match "PowerShellv2"}
FeatureName : MicrosoftWindowsPowerShellV2Root
State       : Enabled
FeatureName : MicrosoftWindowsPowerShellV2
State       : Enabled
```

So, it seems like PowerShell version 2.0 is still enabled on this machine. Therefore, if the missing .NET Framework 2.0 is installed, this system will be vulnerable to a downgrade attack.

Therefore, let's disable PowerShell version 2.0 to harden your system by running the following command:

```
Get-WindowsOptionalFeature -Online | Where-Object {$_.FeatureName -match "PowerShellv2"} | ForEach
```

You will see in the output that a restart is needed, so after you restart your PC, the changes are applied and PowerShell version 2.0 is disabled:

```
> Get-WindowsOptionalFeature -Online | Where-Object {$_.FeatureName -match "PowerShellv2"} | ForEa
Path          :
Online        : True
RestartNeeded : False
Path          :
Online        : True
RestartNeeded : False
```

So, if you verify once again, you will see that the state is set to **`Disabled`**:

```
> Get-WindowsOptionalFeature -Online | Where-Object {$_.FeatureName -match "PowerShellv2"}
FeatureName : MicrosoftWindowsPowerShellV2Root
State       : Disabled
FeatureName : MicrosoftWindowsPowerShellV2
State       : Disabled
```

However, on Windows 7, PowerShell version 2.0 cannot be disabled. The only way to disallow PowerShell version 2.0 usage is to leverage **Application Control** or **AppLocker**, which we will discuss in ***Chapter 11***, *AppLocker, Application Control, and Code Signing*.

For adversaries, there is also another way to run a downgrade attack – if, for example, a compiled application leverages an older PowerShell version, and links against the compiled PowerShell v2 binaries, a downgrade attack can be launched by exploiting the application. So, whenever this application runs, PowerShell v2 is also active, and it can be used by the attacker if they manage to exploit the application.

In this case, disabling PowerShell version 2.0 can help to protect against this type of attack by blocking the deprecated binaries in the **Global Assembly Cache** (**GAC**) or removing the PowerShell component altogether. Nevertheless, it's important to note that other applications that rely on these binaries will be blocked as well, as they usually don't ship with all of the PowerShell binaries.

In general, a downgrade attack is a highly critical issue, and therefore, you should monitor for it. You can do so by monitoring the event with the event id **`400`** in the Windows PowerShell event log – if the specified version is lower than **`[Version] "5"`**, you should definitely investigate further.

Lee Holmes, who was part of the Windows PowerShell team at Microsoft, provides a great example of how to monitor for potential downgrade attacks by looking for event ID **`400`** in the PowerShell event log in his blog article *Detecting and Preventing PowerShell Downgrade Attacks*: **https://www.leeholmes.com/detecting-and-preventing-powershell-downgrade-attacks/**.

Use this example to find lower versions of the PowerShell engine being loaded:

```
Get-WinEvent -LogName "Windows PowerShell" | Where-Object Id -eq 400 | Foreach-Object {
        $version = [Version] ($_.Message -replace '(?s).*EngineVersion=([\d\.]+)*.*','$1')
        if($version -lt ([Version] "5.0")) { $_ }
}
```

EventList

During my time as a Premier Field Engineer at Microsoft, I worked with a lot of customers that were just building their SOCs from scratch. Most of those customers not only wanted to set up log event forwarding but also asked me for best practices to harden their Windows environment.

When talking about hardening Windows environments, you can't ignore the Microsoft **Security and Compliance Toolkit** (**SCT**): **https://www.microsoft.com/en-us/download/details.aspx?id=55319**.

I will talk more about some parts of this toolkit later in ***Chapter 6***, *Active Directory – Attacks and Mitigation* as well as in ***Chapter 13***, *What Else? – Further Mitigations and Resources*. In general, this toolkit contains several tools for comparing and verifying your configuration, as well as the so-called **baselines**.

These baselines are meant to provide hardening guidance – a lot of settings that are important for your security posture, as well as *monitoring configuration*.

Needless to say, you should not just enforce those baselines without having a structured plan and knowing the impact of the settings that you are configuring.

If a baseline is configured for a certain computer, thanks to the monitoring configuration piece, new events are generated in the **`Security`** event log.

When I worked with customers, I always recommended applying the Microsoft Security baselines after a well-structured plan.

On one occasion, I was at a customer's site and just recommended that they should apply Microsoft Security baselines to see more event IDs. After recommending applying those baselines, my customer asked me whether there was an overview to see what additional event IDs were being generated if they enabled a particular baseline, like the *Windows 2016 Domain Controller baseline*.

I only knew of a documentation document that they could use to find it out themselves, the *Windows 10 and Windows Server 2016 security auditing and monitoring reference*: **https://www.microsoft.com/en-us/download/details.aspx?id=52630**.

Although this document provided amazingly detailed information on all **Advanced Audit Policy Configuration** items, with its 754 pages, it was quite extensive.

So, the customer was not happy studying this big document and asked me to write down what events would be generated if they applied this baseline. I was not happy about such stupefying work, but I started to write down all events for this one baseline.

While I was doing this, the customer approached me and realized that they had not one but multiple kinds of baselines that they wanted to apply in their environment. Also, these were not only Domain Controller baselines but also baselines for member servers and client computers of all kinds of operating systems. So, they asked me to write down the event IDs for *ALL* existing baselines.

As you can imagine, I was not super-excited about this new task. This seemed like a very dull and exhausting task that would take years to complete.

Therefore, I considered the need to automate matching baselines to event IDs, and that's how my open source tool **EventList** was born.

Although it all started as an Excel document with Visual Basic macros, it became a huge project in the meantime, with a huge database behind the code.

Figure 4.21 – The EventList logo

And whenever I need to work with event IDs, my EventList database became my source of truth, and it is still growing constantly.

Working with EventList

To get started, EventList can be easily installed from the PowerShell Gallery:

```
> Install-Module EventList
```

EventList is built in PowerShell; therefore, even if you want to work solely with the user interface, you need to run at least one PowerShell command. Open the PowerShell console as an administrator and type in the following:

```
> Open-EventListGUI
```

Confirm by hitting *Enter*. After a few seconds, the EventList UI appears.

Figure 4.22 – The EventList UI

At the top left, you can select an existing baseline and see the **MITRE ATT&CK** techniques and areas that are being populated in the UI. So, you can see directly what MITRE ATT&CK techniques are covered if a certain baseline is applied.

You have also the possibility to import your own baselines or exported GPOs and delete existing ones.

Once you have selected a baseline and the MITRE ATT&CK checkboxes are filled, choose **Generate Event List**.

Figure 4.23 – EventList – showing the baseline events

A pop-up window opens, and you can choose whether you want to generate an EventList for baseline events only or all MITRE ATT&CK events.

To see which event IDs would be generated if you applied a certain baseline, select **Baseline Events only**. Confirm with **OK** to see the EventList for the baseline/GPO that you selected.

Figure 4.24 – A generated EventList

An EventList is generated, in which you see each event ID that will be generated if this baseline is applied, as well as (if available) a link to the documentation and a recommendation on whether this event should be monitored or not.

If **Export as CSV** is checked, you can select where the output should be saved, and a `.csv` file is generated.

As Microsoft Security baselines mostly rely on the **Advanced Audit Logs**, by using the **Baseline only** function, EventList helps a lot to understand and demystify the Advanced Audit Logs.

You can achieve the same thing by using the following commands on the CLI:

```
> Get-BaselineEventList -BaselineName "MSFT Windows Server 2019 - Domain Controller"
```

The baseline needs to be imported into the EventList database, so make sure that the baseline name is shown when verifying with the **`Get-BaselineNameFromDB`** function.

Of course, you can also select different MITRE ATT&CK techniques and areas and generate an EventList to see which event IDs cover a certain MITRE ATT&CK area. Generate an EventList, select **All MITRE ATT&CK Events**, and confirm with **OK**.

A popup will open, and you can see all event IDs that were correlated to the selected MITRE ATT&CK techniques.

Figure 4.25 – A MITRE ATT&CK EventList

Again, this can be achieved by passing either a baseline or MITRE ATT&CK technique numbers to the **`Get-MitreEventList`** function, using the **`-Identity`** parameter:

```
> Get-MitreEventList -Identity "T1039"
```

The following screenshot shows the output of the command.

Figure 4.26 – The Get-MitreEventList function can also be run via the command line

Of course, EventList provides many more functions. It also provides possibilities to generate forwarder agent snippets of all event IDs that should be forwarded for your use case. You can also generate your own GPOs and hunting queries that support your very own use case.

However, there are too many functions to describe everything in detail in this book. If you are interested in learning more about EventList, make sure to read the EventList documentation in its GitHub repository, that is mentioned at the end of this section. Some experts also find it useful to query the database behind EventList manually.

I wrote EventList to help SOCs worldwide understand what to monitor and simplify their event ID forwarding.

I am constantly improving EventList, so if you want to learn more, you are more than welcome to download and test it. It can be either downloaded and installed from my GitHub repository (**https://github.com/miriamxyra/EventList**) or installed from the PowerShell Gallery:

```
> Install-Module EventList -Force
```

To understand the functionalities of EventList more comprehensively, I recommend reading the documentation and help files and watching some of the recordings of the talks that I have given on it:

- **Hack.lu 2019: (version 1.1.0)**: **https://www.youtube.com/watch?v=nkMDsw4MA48**
- **Black Hat 2020 (version 2.0.0)**: **https://www.youtube.com/watch?v=3x5-nZ2bfbo**

If you have any ideas on what is missing in EventList, I would love to hear more, and I'm looking forward to your pull request on GitHub or your message on Twitter or via email.

Getting started with logging

To improve your detection, it makes sense to set up a SIEM system for event collection so that you have all event logs in one place, allowing you to hunt and even build automated alerting.

There are many options if you want to choose a SIEM system – for every budget and scenario. Over the years, I have seen many different SIEM systems – and each one just fitted perfectly for each organization.

The most popular SIEM systems that I have seen out in the wild were **Splunk**, **Azure Sentinel**, **ArcSight**, **qRadar**, and the **"ELK stack" (Elastic, LogStash, and Kibana)**, just to mention a few. I also saw and used **Windows Event Forwarding** (**WEF**) to realize event log monitoring.

Of course, it is also possible to analyze events on a local machine, but it is not practical – depending on the configuration, if the maximum log size is reached, old events are deleted, and you cannot easily correlate them with logs from another system.

In this chapter, we will also analyze events directly on the machine (or remotely if you like), but for an actual production environment, I recommend having an SIEM system in place – just make sure that it fits your use case before you start.

An overview of important PowerShell-related log files

Before we get started, you might want to configure all the logs that you want to forward to your SIEM or a central log server.

In this section, you will find an overview of all the logs that I consider important when it comes to PowerShell logging.

Basic PowerShell event logs

When working with PowerShell, there are three event logs that are of interest – the **Windows PowerShell log**, the **Microsoft Windows PowerShell Operational log**, and the **PowerShellCore Operational log**. Let's discuss each of them in the following subsections.

The Windows PowerShell Log

Windows PowerShell has always had a strong focus on security and logging, even in its earliest versions. In fact, compared to other shell or scripting languages, PowerShell's early versions already had significantly better security logging capabilities. However, over the years, the language evolved, and its logging capabilities expanded enormously, providing us with even better logging nowadays.

Although early versions did not provide us with the security logging that you know from today's PowerShell versions, Windows PowerShell has written events to the **Windows PowerShell event log** since version 1 when important engine events occurred. Back then, PowerShell provided only basic logging functionalities, which are still available in current operating systems, as shown here:

- **Full name**: Windows PowerShell
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Windows PowerShell.evtx`
- **Path in the UI**: **Applications and Services** | **Windows PowerShell**

The *most interesting event IDs* in these event logs are the following:

- **Event ID 200** (a warning): **Command health.**

Look for `Host Application` to get more details on the executed command.

- **Event ID 400**: **The engine state is changed from none to available.**

This event might be *the most interesting event* in this event log, as it indicates when the engine was started and which version was used. This event is optimal for identifying and terminating outdated PowerShell versions (monitoring for `HostVersion` less than 5.0) – and is used for downgrade attacks (see the *Detecting a downgrade attack* section for more information).

- **Event ID 800**: **The pipeline execution details for the command line** – <*command-line command*>**.**

Although event ID `800` provides details on the execution of command lines that contain cmdlets, it doesn't include information about other executables such as `wmic`. It may be more useful to monitor the event IDs `4103` and `4104` from the *Microsoft Windows PowerShell Operational log* for additional details.

The Microsoft Windows PowerShell Operational log contains all relevant information when it comes to the usage of PowerShell – for example, **Module Logging** and also **Script Block Logging** events are written to this log.

The Microsoft Windows PowerShell Operational log

Starting with Windows Vista, Microsoft introduced a new type of logging system called **ETW**. As part of this change, the *Microsoft Windows PowerShell Operational log* was introduced, which included a range of event IDs such as `4100`, `4103` (although configuring them could be challenging), as well as `40961`, `40862`, and others related to PowerShell Remoting logs.

With *KB3000850*, Advanced Audit capabilities such as **Module Logging**, **Script Block Logging**, and **transcription** could be ported into PowerShell version 4 (Windows Server 2012 R2 and Windows 8.1). Later on, with PowerShell version 5 (Windows Server 2016 and Windows 10), these features were included by default.

With these new auditing capabilities, there were also new event types introduced, such as the event IDs `4104`, `4105`, and `4106`, which provide you with advanced logging capabilities:

- **Full name: `Microsoft-Windows-Powershell/Operational`**
- **Log path**: **`%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-PowerShell%4Operational.evtx`**
- **Path in the UI**: **Applications and Services** | **Microsoft** | **Windows** | **PowerShell** | **Operational**

The *most interesting event IDs* in this event logs are the following:

- **Event ID 4103**: **Executing pipeline/command invocation. An event is generated if PowerShell Module Logging is enabled.**
- **Event ID 4104**: **Creating Scriptblock text.**

An event is generated if **`ScriptBlockLogging`** is enabled. Common malicious activities such as loading a malicious module or executing a suspicious command are logged, regardless of whether **`ScriptBlockLogging`** is enabled or not.

- **Event ID 4105**: **ScriptBlock_Invoke_Start_Detail (message: started/completed an invocation of ScriptBlock).**

An event is generated if **`ScriptBlockLogging`** is enabled. This records start/stop events. It is very noisy and not necessarily needed for security monitoring.

- **Event ID 4106**: **ScriptBlock_Invoke_Complete_Detail (message: started/completed an invocation of ScriptBlock).**

An event is generated if **`ScriptBlockLogging`** is enabled. This records start/stop events. It is very noisy and not necessarily needed for security monitoring.

- **Event ID 40961**: **The PowerShell console is starting up.**

This event indicates that the PowerShell console was opened. Especially monitor for unusual user behavior using this event (for example, if the PowerShell console was executed by a user that should not log on to this system, or if it's a system account).

- **Event ID 40962**: **The PowerShell console is ready for user input.**

This event indicates that the PowerShell console was started and is now ready for user input. Especially monitor for unusual user behavior using this event (for example, if the PowerShell console was executed by a user that should not log on to this system or if it's a system account).

To filter for certain event IDs, you can pipe the output of **`Get-WinEvent`** to **`Where-Object`**:

```
> Get-WinEvent Microsoft-Windows-PowerShell/Operational | Where-Object Id -eq 4104
```

In this example, you will get all events with the event ID **`4104`**, which indicates that a script block was created.

The PowerShellCore Operational log

When PowerShell Core was introduced, so was the PowerShellCore Operational log. It provides Advanced Audit capabilities for PowerShell Core Event Logging:

- **Full name:** `PowerShellCore/Operational`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\PowerShellCore%4Operational.evtx`
- **Path in the UI**: **Applications and Services** | **PowerShellCore** | **Operational**

The event IDs that are logged within this log file are the same as the ones that are logged in the Microsoft Windows PowerShell Operational log. Please refer to the event IDs in the previous section.

The Windows Remote Management (WinRM) log

The **Microsoft Windows WinRM Operational log** records both inbound and outbound WinRM connections. Since PowerShell relies on WinRM for PowerShell remoting, you can also find PowerShell remote connections in this event log. Therefore, it is essential to also monitor and analyze event IDs from this log.

- **Full name**: `Microsoft-Windows-WinRM/Operational`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-WinRM%4Operational.evtx`
- **Path in the UI**: **Applications and Services** | **Microsoft** | **Windows** | **Windows Remote Management** | **Operational**

When working with PowerShell and WinRM, the following are *the most interesting events* to look for in the WinRM event log.

- **Event ID 6**: **Creating a WSMan session.**

This is recorded whenever a remote connection is established. It also contains the username, the destination address, and the PowerShell version that was used.

- **Event ID 81**: **Processing a client request for a CreateShell operation or processing a client request for a DeleteShell operation.**
- **Event ID 82**: **Entering the plugin for a CreateShell operation with a ResourceUri of <http://schemas.microsoft.com/powershell/Microsoft.PowerShell>**
- **Event ID 134**: **Sending a response for a CreateShell operation.**
- **Event ID 169**: **The** <*domain*>|<*user*> **user has authenticated successfully using NTLM authentication.**

You can query all events within the WinRM log using `Get-WinEvent Microsoft-Windows-WinRM/Operational`.

Security

The Security event log is not only PowerShell related but also helps to correlate events such as logon/logoff and authentication.

- **Full name**: `Security`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Security.evtx`
- **Path in the UI**: **Windows Logs** | **Security**

While not all event IDs in the Security log are generated by default, the most important ones are there to help identify security issues. If you want to implement extensive security logging, I recommend applying the Microsoft Security baselines from the Microsoft Security toolkit to your systems. However, it is important to note that the settings in the Security baseline should be commensurate with your organization's resources and capabilities. Therefore, it's advisable to evaluate which logging settings are appropriate for your organization's needs and capabilities before applying a baseline.

You can download the **Microsoft Security toolkit** here: https://www.microsoft.com/en-us/download/details.aspx?id=55319.

The event IDs in this event log are some of the most important to monitor for security purposes. While not all of them are specific to PowerShell, they are still critical to maintaining a secure environment. The following are the *most interesting event IDs* in this event log:

- **Event ID 4657**: **A registry value was modified**
- **Event ID 4688**: **A new process has been created. Look for processes with powershell.exe as the "New Process Name". You can use the Creator Process ID to link what process launched which other processes.**
- **Event ID 1100**: **The Event Logging service has shut down.**
- **Event ID 1102**: **The audit log was cleared.**
- **Event ID 1104**: **The security log is now full.**
- **Event ID 4624**: **An account was successfully logged on.**
- **Event ID 4625**: **An account failed to log on.**

The Security log is quite extensive and contains a lot of important event IDs. Covering just the Security log could fill an entire book; therefore, this list is not complete, and I only listed some of the most important ones when it comes to PowerShell.

Nevertheless, the question of *which security event IDs matter* has kept me awake many nights, and so I came up with an open source tool called **EventList**. If you want to find out which event IDs matter, have a look at the *Forwarding and analyzing event logs – EventList* section in this chapter.

System

In the system log, many system-relevant log IDs are generated:

- **Full name**: `System`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\System.evtx`
- **Path in the UI**: **Windows Logs** | **System**

The *most interesting event ID* in this event log for PowerShell security logging is as follows:

- **Event ID 104 – the** <*name*> **log was cleared.** This event indicates that the event log with the name <*name*> was cleared, which could indicate an adversary trying to hide traces. Especially use this event ID to monitor for the log names *"Windows PowerShell," "PowerShell Operational,"* or *"PowerShellCore"* to detect PowerShell-related event log clearing.

Depending on what you are monitoring for, there are many interesting events in this log – for example, details on every installation.

Windows Defender

The Windows Defender log has been enabled by default since Windows 10 and Windows Server 2016, and it provides a lot of helpful events. For example, it also contains events related to the **Antimalware Scan Interface** (**AMSI**), which is a part of Windows Defender:

- **Full name**: `Microsoft-Windows-Windows Defender/Operational`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-Windows Defender%4Operational.evtx`
- **Path in the UI**: **Applications and Services** | **Microsoft** | **Windows** | **Windows Defender** | **Operational**

The *most interesting event IDs* in this event log for PowerShell security logging are the following:

- **Event ID 1116**: **Microsoft Defender Antivirus has detected malware or other potentially unwanted software.**
- **Event ID 1117**: **Microsoft Defender Antivirus has taken action to protect this machine from malware or other potentially unwanted software.**

If Microsoft Defender is used on your machine, you will find many more interesting Defender-related log events in this event log. Use this reference to learn more about each Microsoft Defender-related event ID: https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/troubleshoot-microsoft-defender-antivirus.

We will take a closer look at AMSI in ***Chapter 12***, *Exploring the Antimalware Scan Interface (AMSI)*.

Windows Defender Application Control and AppLocker

Windows Defender Application Control (**WDAC**) and **AppLocker** can be used to allowlist applications to restrict which software is allowed to be used within an organization. Both solutions help you to protect against the unauthorized use of software.

We will take a closer look at WDAC and AppLocker in ***Chapter 11***, *AppLocker, Application Control, and Code Signing*.

When enabling allowlist solutions, auditing is the first major step; hence, analyzing WDAC and AppLocker-related event IDs is necessary for this process.

Windows Defender Application Control (WDAC)

WDAC is Microsoft's latest allowlisting solution, which was introduced with Windows 10 and was earlier known as *Device Guard*. In addition to allowlisting applications, WDAC can also be used to enforce code integrity policies on Windows machines.

WDAC has two main event logs – one event log named **MSI and Scripts** is shared with AppLocker, and another event log is used to log **Code Integrity**-related events.

Code Integrity

- **Full name**: `Microsoft-Windows-CodeIntegrity/Operational`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-CodeIntegrity%4Operational.evtx`
- **Path in the UI**: **Applications and Services Logs** | **Microsoft** | **Windows** | **CodeIntegrity** | **Operational**

The *most interesting event IDs* in this event logs for PowerShell security logging are the following:

- **Event ID 3001**: **An unsigned driver attempted to load on the system.**
- **Event ID 3023**: **The driver file under validation didn't meet the requirements to pass the Application Control policy.**
- **Event ID 3033**: **The file under validation didn't meet the requirements to pass the Application Control policy.**
- **Event ID 3034**: **The file under validation didn't meet the requirements to pass the Application Control policy if it was enforced. The file was allowed, since the policy is in audit mode.**
- **Event ID 3064**: **If the Application Control policy was enforced, a user mode DLL under validation didn't meet the requirements to pass the Application Control policy. The DLL was allowed, since the policy is in audit mode.**
- **Event ID 3065**: **If the Application Control policy was enforced, a user mode DLL under validation didn't meet the requirements to pass the Application Control policy.**
- **Event ID 3076**: **This event is the main Application Control block event for audit mode policies. It indicates that the file would have been blocked if the policy was enforced.**
- **Event ID 3077**: **This event is the main Application Control block event for enforced policies. It indicates that the file didn't pass your policy and was blocked.**

You can query all events within the WDAC log using **`Get-WinEvent Microsoft-Windows-CodeIntegrity/Operational`**. Monitoring and analyzing these events can help identify potential security breaches and improve the overall security posture of a system.

MSI and Script

All Microsoft Installer and script-related event IDs can be found in this event log:

- **Full name**: `Microsoft-Windows-AppLocker/MSI and Script`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-AppLocker%4MSI and Script.evtx`
- **Path in the UI**: **Applications and Services Logs** | **Microsoft** | **Windows** | **Applocker** | **MSI and Script**

The *most interesting event IDs* in the event logs for PowerShell security logging are the following:

- **Event ID 8028**: *** was allowed to run but would have been prevented if the Config CI policy was enforced.**
- **Event ID 8029**: *** was prevented from running due to the Config CI policy.**
- **Event ID 8036**: *** was prevented from running due to the Config CI policy.**
- **Event ID 8037**: *** passed the Config CI policy and was allowed to run.**

If you want to learn about more Application Control event IDs, have a look at the *AppLocker* section and the following documentation: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/event-id-explanations.

AppLocker

When it comes to AppLocker, there are four event log files that you might want to examine, depending on your use case – *EXE and DLL*, *MSI and Script*, *Packaged app-Deployment*, and *Packaged app-Execution*.

In the UI, you can find all four logs under the same path – simply replace <**Name of the log**> with the name of each event log, as shown here:

Path in the UI: **Applications and Services** | **Microsoft** | **Windows** | **AppLocker** | <**Name of the log**>

The following is the full name and the path of each AppLocker-related event log (please note that auditing must be enabled in order for any of these event logs to appear):

- **EXE and DLL**

All event IDs that are related to executing binaries (EXE) and DLLs can be found in this event log:

- **Full name**: `Microsoft-Windows-AppLocker/EXE and DLL`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-AppLocker%4EXE and DLL.evtx`

- **MSI and Script**

All Microsoft Installer and script-related event IDs can be found in this event log:

- **Full name**: `Microsoft-Windows-AppLocker/MSI and Script`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-AppLocker%4MSI and Script.evtx`

- **Packaged app-Deployment**

If a packaged app is deployed, you can find all related event IDs in this event log:

- **Full name**: `Microsoft-Windows-AppLocker/Packaged app-Deployment`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-AppLocker%4Packaged app-Deployment.evtx`

- **Packaged app-Execution**

All packaged app execution-related event IDs can be found in this event log.

- **Full name**: `Microsoft-Windows-AppLocker/Packaged app-Execution`
- **Log path**: `%SystemRoot%\System32\Winevt\Logs\Microsoft-Windows-AppLocker%4Packaged app-Execution.evtx`

The *most interesting event IDs* in these event logs for PowerShell security logging are the following:

- **Event ID 8000 (error)**: **The Application Identity Policy conversion failed. Status *<%1> This indicates that the policy was not applied correctly to the computer. The status message is provided for troubleshooting purposes.**
- **Event ID 8001 (information)**: **The AppLocker policy was applied successfully to this computer. This indicates that the AppLocker policy was successfully applied to the computer.**
- **Event ID 8002 (information)**: *<Filename>* **was allowed to run. This specifies that the .exe or .dll file is allowed by an AppLocker rule.**
- **Event ID 8003 (warning)**: *<Filename>* **was allowed to run but would have been prevented from running if the AppLocker policy were enforced. This is applied only when the Audit only enforcement mode is enabled. It specifies that the .exe or .dll file would be blocked if the Enforce rules enforcement mode were enabled.**
- **Event ID 8004 (error)**: *<Filename>* **was not allowed to run. Access to <filename> is restricted by the administrator. This is applied only when the Enforce rules enforcement mode is set either directly or indirectly through Group Policy inheritance. The .exe or .dll file cannot run.**
- **Event ID 8005 (information)**: *<Filename>* **was allowed to run. This specifies that the script or .msi file is allowed by an AppLocker rule.**

- **Event ID 8006 (warning)**: *<Filename>* **was allowed to run but would have been prevented from running if the AppLocker policy were enforced. This is applied only when the Audit only enforcement mode is enabled. It specifies that the script or .msi file would be blocked if the Enforce rules enforcement mode were enabled.**
- **Event ID 8007 (error)**: *<Filename>* **was not allowed to run. Access to <Filename> is restricted by the administrator. This is applied only when the Enforce rules enforcement mode is set either directly or indirectly through Group Policy inheritance. The script or .msi file cannot run.**
- **Event ID 8008 (error)**: **AppLocker is disabled on the SKU. This was added in Windows Server 2012 and Windows 8.**

If you are interested in learning about more AppLocker event IDs, please refer to the following link: https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/applocker/using-event-viewer-with-applocker.

There are, of course, many other interesting log files, such as **Firewall** and **DSC**. Mentioning and describing all of them would exceed the content of this book; therefore, I have only mentioned some of the most interesting log files when it comes to PowerShell Security.

Increasing log size

Every event that is generated lets a log file grow. As thousands of events can be written in a very short time, it is useful to increase the maximum log file size – especially if you also want to analyze events locally.

Of course, it is always recommended to forward your logs to a central log repository to make sure the logs will not be lost. However, if you want to analyze events locally, it is also helpful to increase the log file size.

The **`Limit-EventLog`** cmdlet can help you with this task in Windows PowerShell:

```
> Limit-EventLog -LogName "Windows PowerShell" -MaximumSize 4194240KB
```

This command sets the maximum size of the PowerShell log to *4 GB*. Please note that the “MB” and “GB” prefixes are also available in this cmdlet.

When setting the maximum size of the event log, it’s important to keep in mind that the size of an event log entry can vary, depending on the specific event log and the number of enabled events. Look how much space one event usually takes up in your environment on average per log. First, you need to get the log size of an event log. The following command returns the maximum size of the Windows PowerShell event log in *KB:*

```
> – Get-ItemProperty -Path  'HKLM:\SYSTEM\CurrentControlSet\Services\EventLog\Windows PowerShell\'
```

Then, divide it by the number of entries. Just like that you can calculate the estimated size of your event log and how many events it should hold before events will be rotated.

If you use PowerShell 7, the **`Limit-EventLog`** cmdlet is not available. Instead, you will need to alter the registry, using **`New-ItemProperty`**:

```
> New-ItemProperty -Path 'HKLM:\SYSTEM\CurrentControlSet\Services\EventLog\Windows PowerShell\' -N
```

Using the **`Limit-EventLog`** command, you can also specify the behavior when an event log is full: **https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/limit-eventlog**.

Summary

In this chapter, you learned how to get started with security logging for PowerShell. You now know which event logs are of interest and which event IDs you should look for. As security monitoring is a huge topic, you have learned just the basics on how to get started and continue.

You learned how to configure PowerShell Module Logging, Script Block Logging, and PowerShell transcripts – manually and centralized for Windows PowerShell, as well as for PowerShell Core.

Another important learning point is that log events can be tampered with, and you can implement some level of protection using Protected Event Logging.

Eventually, it is best to forward your log events to a centralized SIEM system, but if that's not possible, you also learned how to analyze events using PowerShell.

Now that you have been provided with some example scripts and code snippets, you are ready to investigate all PowerShell activity on your clients and servers.

Last but not least, if you want to dive deeper into security monitoring, EventList can help you to find out which events are important to monitor.

When we talk about auditing, detection, and monitoring; local systems are not far away. Let's dive deeper into the system and have a look at the Windows registry, the Windows API, COM, CIM/WMI, and how it is possible to run PowerShell without running **`powershell.exe`** in our next chapter.

Further reading

If you want to explore some of the topics that were mentioned in this chapter, follow these resources:

- **Auditing – further resources**:
  - Detecting Offensive PowerShell Attack Tools: **https://adsecurity.org/?p=2604**

    - Lee Holmes on downgrade attacks: **https://www.leeholmes.com/blog/2017/03/17/detecting-and-preventing-powershell-downgrade-attacks/**
    - Microsoft SCT: **https://www.microsoft.com/en-us/download/details.aspx?id=55319**
    - PowerShell ♥ the Blue Team: **https://devblogs.microsoft.com/powershell/powershell-the-blue-team/**
    - Windows 10 and Windows Server 2016 security auditing and monitoring reference: **https://www.microsoft.com/en-us/download/details.aspx?id=52630**
    - *PowerShell post-exploitation, the Empire has fallen, You CAN detect PowerShell exploitation* by Michael Gough: **https://de.slideshare.net/Hackerhurricane/you-can-detect-powershell-attacks**
- **EventList**:
    - GitHub: **https://github.com/miriamxyra/EventList**
    - Black Hat presentation 2020 (version 2.0.0): **https://www.youtube.com/watch?v=3x5-nZ2bfbo**
- **Helpful cmdlets and commands**:
    - `Limit-EventLog` documentation: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/limit-eventlog?view=powershell-5.1
    - `Start-Transcript` documentation: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.host/start-transcript?view=powershell-7#parameters
    - `wevtutil` documentation: https://docs.microsoft.com/de-de/windows-server/administration/windows-commands/wevtutil
    - `Unprotect-CmsMessage`: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/unprotect-cmsmessage
- **PowerShell Logging and event logs**:
    - RFC – CMS: https://www.rfc-editor.org/rfc/rfc5652
    - PowerShell Core Group Policy settings: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings?view=powershell-7.1
    - PowerShell logging on a non-Windows OS: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging_non-windows?view=powershell-7
    - About logging on a Windows OS: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging_windows?view=powershell-7.1
    - About event logs (v 5.1): https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_eventlogs

You can also find all links mentioned in this chapter in the GitHub repository for *Chapter 4* – there's no need to manually type in every link: https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter04/Links.md.