

# PowerShell Monitoring - Regain Control

Michael Schneider

Offense Department, scip AG

[misc@scip.ch](mailto:misc@scip.ch)

<https://www.scip.ch>

Marc Ruef (Editor)

Research Department, scip AG

[maru@scip.ch](mailto:maru@scip.ch)

<https://www.scip.ch>

Keywords: Active Directory, Block, Blue Team, Detect, Exploit, False-Positive, Framework, Logging, Master, Microsoft

## 1. Preface

This paper was written in 2016 as part of a research project at scip AG, Switzerland. It was initially published online at <https://www.scip.ch/en/?labs.20160407> and is available in English and German. Providing our clients with innovative research for the information technology of the future is an essential part of our company culture.

## 2. Introduction

PowerShell-based attacks have long been a nightmare for IT security divisions, because they barely leave a trace and can gain access to an impressive range of system functions. We're at a point where IT security divisions really need a tool to gain control of the situation, or even stay one step ahead.

## 3. Starting point

In the article *PowerShell ♥ the Blue Team* [1] on the *Windows PowerShell Blog*, Microsoft presents the new PowerShell Version 5.0 security functions. With the *Constrained PowerShell* function, control is *improved with AppLocker* [2]. When AppLocker is used with the standard rules in *Allow* mode, PowerShell recognizes this and sets the language mode to *Constrained*. This reduces the range of functions so that only basic functions are then permitted. As a result, enhanced functions, including .NET scripting and the interaction with COM objects, are no longer executed. This restriction serves to protect against attack tools that apply these functions so that they no longer execute correctly. On the other hand, scripts that are explicitly permitted by an AppLocker policy because they're signed or located in a trustworthy directory continue to run in *Full Mode*. This function makes it much easier to *block PowerShell* [3].

However, blocking PowerShell is difficult and can presumably only be consistently carried out in specific environments, such as a terminal server. PowerShell can still be executed on all other systems, or must be explicitly permitted. As a consequence of this and in keeping with the motto *In God we trust, all others we monitor*, a monitoring solution for logging and analyzing PowerShell activities should be implemented.

## 4. Logging PowerShell activity

With version 3.0 of the *Windows Management Framework* [4], Microsoft introduced the group policy setting *Turn On Module Logging*. This can be found at *Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell*. When activated, PowerShell commands and modules are logged in the *Microsoft-Windows-PowerShell/Operational* event log. This means the execution of PowerShell can be analyzed and monitored by one system.

The group policy must define which modules are to be logged. If the wildcard setting *\** is used to log all modules, this can lead to a large volume of entries. The following example provides a vivid illustration of this. The one-time execution of the script *Invoke-Mimikatz* [5], a PowerShell adaptation by *Joe Bialek* [6] of the *Mimikatz tool* [7] by *Benjamin Delpy* [8] for extracting Windows credentials, leads to around 29,000 entries in the event log, which thus reaches a size of 56 MB. A search for the character string *Mimikatz* then results in 22,691 hits:

```
PS C:\> Get-WinEvent -LogName "Microsoft-Windows-PowerShell/Operational" | Where { $_.Message -like "**Mimikatz*" } | Group-Object Eventid | Format-Table Count,Name
```

```
Count Name
-----
22691 800
```

Consequently, it's important to establish a compromise between data volume and the modules to be logged. The wildcard setting must be used to allow detection of all attacks, because attackers may be using their own, user-defined modules which aren't included in the filter list. The data volume should at least be considered when implementing the monitoring solution. In order to reduce the accrued data volume, it may be worth carrying out a preliminary filter of the results on the client without sending all entries from the client to the monitoring solution.

*Windows Management Framework 5.0* [9] introduces an additional logging function called *PowerShell Script Block Logging*. A *script block* forms the basis for executable code and occurs in interactive console commands, in command calls using the *-Command \$command* parameter, as well as in functions and scripts. Activating the setting means that all script blocks processed through PowerShell are logged. The

advantage of this setting over *module logging* is that when dynamic code is used in a script block, the generated and actually executed variants are also logged alongside this code. This allows logging of code from applications that specifically utilize dynamic code generation expressly for concealment purposes, including the use of coding and encryption methods. This is a common method for bypassing security solutions, because the malicious code can only be detected during run time and can't be picked up beforehand by an anti-virus scan.

One popular technique is the use of Base64-coded payloads. A Base64 payload can be directly executed using the `-EncodedCommand $commandBase64` parameter, as the following example illustrates. Here a coded command is executed through the console. The call does not allow us to determine what the command is doing.

```
PS C:\> powershell.exe -Enc
"VwByAGkAdAB1AC0ATwB1AHQAcAB1AHQAIAAnAFIAdQBuaG4Aa
QBuaGcA1ABXAGkAbgBkAG8AdwBzAF8ATABvAGMAYQBsAF8ARQB
4AHAAbABvAGkAdAAuAC4ALgAnAA=="
```

At first the event log only logs the actual call. Another log entry then logs the coded form of the command. From this we can reconstruct what was actually executed. Of course simple analysis is also possible with PowerShell, as the following code example shows. Using the command `Get-WinEvent`, the last twenty entries are selected and saved to the `$eventLog` object as an array. The entry `$eventLog[10]` contains the log entry of the script block in the form in which it was executed in the console, with the Base64 payload. The entry `$eventLog[6]` then logs the decoded, actually executed form.

```
PS C:\> $eventLog = Get-WinEvent -LogName
"Microsoft-Windows-PowerShell/Operational" -
MaxEvents 20

PS C:\> $eventLog[10].Message
Creating Scriptblock text (1 of 1):
powershell.exe -Enc
"VwByAGkAdAB1AC0ATwB1AHQAcAB1AHQAIAAnAFIAdQBuaG4Aa
QBuaGcA1ABXAGkAbgBkAG8AdwBzAF8ATABvAGMAYQBsAF8ARQB
4AHAAbABvAGkAdAAuAC4ALgAnAA=="

ScriptBlock ID: 6310fcc4-c2e5-4790-98d7-
69ca0133abce
Path:

PS C:\> $eventLog[6].Message
Creating Scriptblock text (1 of 1):
Write-Output 'Running Windows_Local_Exploit...'

ScriptBlock ID: 152104d8-dc65-41a7-8c2e-
67a87828a9a4
Path:
```

To return to the example of the *Invoke-Mimikatz* script: with the *Turn On PowerShell Script Block Logging* setting, only 413 entries are recorded in the event log. Compared with the approximately 29,000 entries, this is a manageable volume of data and can be used directly for further analyses.

## 5. Collecting event logs

But logging PowerShell activities is only the first step in a monitoring solution. It is important to ensure that log entries can't be altered or deleted. In December 2013, the National Security Agency (NSA) and the Central Security Service (CSS) issued a white paper entitled *Spotting the Adversary with Windows Event Log Monitoring* [10]. The document describes how event logs should be used as well

as necessary protective measures. It is important here to customize the maximum size of the event log and prevent entries from being overwritten when the maximum size is reached. It is also important to ensure the integrity of the log. This includes not granting write permission in log files to administrators. It is advisable to set up a dedicated server for collecting event logs. The white paper describes the configuring of event subscriptions and the event-forwarding policy. Naturally this can also be achieved with other means or products from third-party providers.

## 6. Analyzing PowerShell activity

Logging PowerShell activities in a system or multiple systems results in a large volume of data. This data then has to be analyzed to allow detection of malicious processes. A simple signature-based approach, like searching for keywords such as *Mimikatz*, *Invoke-Mimikatz* or *Invoke-Shellcode*, is insufficient and easy to work around.

The expedient approach is a search for certain keywords of PowerShell objects, methods or commandlets that attack tools require to achieve their ends. *Sean Metcalf* [11], CTO of DAn Solutions and security researcher, gave a talk entitled *Red vs. Blue: Modern Active Directory Attacks & Defense* [12] at DerbyCon 2015, in which he gathered a list of all these keywords. Any such list of keywords should be regularly checked and updated.

- General keywords
  - (New-Object Net.WebClient).DownloadString
  - Invoke-WebRequest
  - Invoke-Expression / IEX
  - EncodedCommand / -enc
  - Bypass
- Invoke-Mimikatz
  - System.Reflection.AssemblyName
  - System.Reflection.Emit.AssemblyBuilderAc
  - System.Runtime.InteropServices.MarshalAs/
  - TOKEN\_PRIVILEGES
  - SE\_PRIVILEGE\_ENABLED
- Invoke-TokenManipulation
  - TOKEN\_IMPERSONATE
  - TOKEN\_DUPLICATE
  - TOKEN\_ADJUST\_PRIVILEGES
- Invoke-CredentialInjection
  - TOKEN\_PRIVILEGES
  - GetDelegateForFunctionPointer
- Invoke-DLLInjection
  - System.Reflection.AssemblyName
  - System.Reflection.Emit.AssemblyBuilderAc
- Invoke-Shellcode
  - System.Reflection.AssemblyName
  - System.Reflection.Emit.AssemblyBuilderAc
  - System.MulticastDelegate
  - System.Reflection.CallingConventions
- Get-GPPPassword
  - System.Security.Cryptography.AesCryptoSer
  - 0x4e,0x99,0x06,0xe8,0xfc,0xb6,0x6c,0xc9,(
  - Groups.User.Properties.cpassword
  - ScheduledTasks.Task.Properties.cpassword

- Out-MiniDump
  - System.Management.Automation.WindowsErrorReporting
  - MiniDumpWriteDump

These keywords allow you to carry out an initial filtering of the data volume. In this case, too, you can call on PowerShell functions. The field Message is searched for hits and all relevant entries are listed.

```
PS C:\> $keywordsMimikatz =
"System.Reflection.AssemblyName|System.Reflection.
Emit.AssemblyBuilderAccess|System.Runtime.InteropServices.MarshalAsAttribute|TOKEN_PRIVILEGES|SE_PRIVILEGE_ENABLED"

PS C:\> Get-WinEvent -ErrorAction SilentlyContinue
-FilterHashtable @{ProviderName="Microsoft-Windows-PowerShell"} | Where { $_.Message -imatch
$keywordsMimikatz } | Format-Table -AutoSize
TimeCreated,Id,RecordId,MachineName,Message
-----
-----
```

These keywords are only an initial indication that an attack may have taken place. Because these keywords may also be used by legitimate PowerShell applications, you may need to carry out another – possibly manual – analysis of the results. The complete output of the script block should be checked, as this will contain the entire command. Every event log entry also contains additional information including a time stamp and the computer and user name which can be used for additional research.

Instead of outputting all found results, these can also be loaded in an object as an array, and the individual entries can then be accessed through the shell:

```
PS C:\> $logs = Get-WinEvent -ErrorAction
SilentlyContinue -FilterHashtable
@{ProviderName="Microsoft-Windows-PowerShell"} |
Where { $_.Message -imatch $keywordsMimikatz }

PS C:\> $logs[123].Message
Creating Scriptblock text (1 of 131):
function Invoke-Mimikatz
{
<#
.SYNOPSIS

This script leverages Mimikatz 2.0 and Invoke-
ReflectivePEInjection to reflectively load
Mimikatz completely in memory. This allows you to
do things such as dump credentials without ever
writing
the mimikatz binary to disk. The script has a
ComputerName parameter which allows it to be
executed against multiple computers...
<redacted by scip AG>
```

In the log entry, the script block contains the definition of the function Invoke-Mimikatz. From this you can assume that the logged events represent an attack.

## 7. Summary

As with any other monitoring solution, the analysis of PowerShell activities requires the configuration and compilation of filters to be regularly checked and optimized for them to work and be practically executed. It's not enough to simply implement the collection of event logs or use a one-off collated list of characteristics or signatures for analysis. Any such list should be expanded to reflect new developments.

In the initialization phase of monitoring, you need to develop a basic understanding for PowerShell activities within your own infrastructure. From this you can determine which tools and scripts are regularly executed and necessary for operation. With this knowledge you can fine-tune the monitoring solution to both prevent false-positive alarms and improve detection of anomalies in the infrastructure.

When these anomalies are recognized and an alarm triggered, it has to be carefully investigated. Here you can use a log of all activities in a script block. This analysis should make it possible to recognize threats to the infrastructure. And IT security divisions that establish and run this kind of PowerShell monitoring solution should sleep more soundly in the future.

## 8. External Links

- [1] <https://blogs.msdn.microsoft.com/powershell/2015/06/09/powershell-the-blue-team/>
- [2] <https://www.scip.ch/en/?labs.20121018>
- [3] <https://www.scip.ch/en/?labs.20150507>
- [4] <https://www.microsoft.com/en-us/download/details.aspx?id=34595>
- [5] <https://raw.githubusercontent.com/PowerShellMafia/PowerSploit/master/Exfiltration/Invoke-Mimikatz.ps1>
- [6] <https://twitter.com/JosephBialek>
- [7] <http://blog.gentilkiwi.com>
- [8] <https://twitter.com/gentilkiwi>
- [9] <https://www.microsoft.com/en-us/download/details.aspx?id=50395>
- [10] [http://www.nsa.gov/ia/\\_files/app/Spotting\\_the\\_Adversary\\_with\\_Windows\\_Event\\_Log\\_Monitoring.pdf](http://www.nsa.gov/ia/_files/app/Spotting_the_Adversary_with_Windows_Event_Log_Monitoring.pdf)
- [11] <https://twitter.com/Pyrotek3>
- [12] <https://adsecurity.org/?p=1632>