PowerShell> Reducing the Attack Surface

October 2017
Version 1.0
Jason Brown
This document is distributed in the hope that it will be useful, but without any warranty. It is provided "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for any specific purpose.

The entire risk as to the quality and performance of the document is with you. In no event, unless required by applicable law, will the author be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the information contained within this document, (including, but not limited to, loss of data, data being rendered inaccurate, losses sustained by you or third parties, or a failure of a policy to operate with any other policies), even if the author has been advised of the possibility of such damages.

This document is maintained at https://community.mcafee.com/docs/DOC-10292.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Figures</td>
<td>4</td>
</tr>
<tr>
<td>Background</td>
<td>5</td>
</tr>
<tr>
<td>Introduction</td>
<td>5</td>
</tr>
<tr>
<td>Combatting PowerShell-based Malware</td>
<td>7</td>
</tr>
<tr>
<td>A Word of Caution</td>
<td>8</td>
</tr>
<tr>
<td>Reporting PowerShell Usage</td>
<td>8</td>
</tr>
<tr>
<td>Restricting PowerShell Usage</td>
<td>10</td>
</tr>
<tr>
<td>Reporting PowerShell Parameter Usage</td>
<td>12</td>
</tr>
<tr>
<td>Restricting PowerShell Parameter Usage</td>
<td>14</td>
</tr>
<tr>
<td>Reporting PowerShell Usage by Process</td>
<td>15</td>
</tr>
<tr>
<td>Restricting PowerShell Usage by Process</td>
<td>17</td>
</tr>
<tr>
<td>Reporting PowerShell Usage by User</td>
<td>18</td>
</tr>
<tr>
<td>Restricting PowerShell Usage by User</td>
<td>19</td>
</tr>
<tr>
<td>Reporting PowerShell IP Connectivity</td>
<td>21</td>
</tr>
<tr>
<td>Restricting PowerShell IP Connectivity</td>
<td>23</td>
</tr>
<tr>
<td>Reporting PowerShell Internal Connectivity</td>
<td>25</td>
</tr>
<tr>
<td>Restricting PowerShell Internal Connectivity</td>
<td>28</td>
</tr>
<tr>
<td>Reporting PowerShell Usage by Time</td>
<td>29</td>
</tr>
<tr>
<td>Restricting PowerShell Usage by Time</td>
<td>31</td>
</tr>
<tr>
<td>Conclusions</td>
<td>34</td>
</tr>
<tr>
<td>About the Author</td>
<td>35</td>
</tr>
<tr>
<td>About McAfee</td>
<td>35</td>
</tr>
</tbody>
</table>
Table of Figures

Figure 1. Illustrating PowerShell's default execution policy ................................................................. 5
Figure 2. Bypassing PowerShell's default execution policy ................................................................. 5
Figure 3. Testing PowerShell operation ............................................................................................. 6
Figure 4. Obfuscation of common PowerShell parameters ............................................................... 6
Figure 5. Obfuscating PowerShell commands .................................................................................. 7
Figure 6. PowerShell Execution Characteristics ............................................................................ 8
Figure 7. An Access Protection rule to allow PowerShell usage to be monitored............................... 9
Figure 8. Testing the Access Protection Monitoring Rule ................................................................. 9
Figure 9. A query to allow PowerShell usage to be reported ........................................................... 9
Figure 10. A sample query showing PowerShell usage .................................................................... 10
Figure 11. An Access Protection rule to block PowerShell usage .................................................... 11
Figure 12. Testing the Access Protection Blocking Rule ................................................................. 11
Figure 13. A query to allow PowerShell blocking to be reported ..................................................... 11
Figure 14. PowerShell parameter rules available within Exploit Prevention .................................... 12
Figure 15. An Exploit Prevention policy to monitor PowerShell parameter usage .............................. 12
Figure 16. Testing PowerShell parameter usage ............................................................................. 13
Figure 17. A query to report PowerShell parameter usage ............................................................... 13
Figure 18. A sample query showing PowerShell parameter usage .................................................... 14
Figure 19. An Access Protection rule to block specific PowerShell parameter usage ....................... 14
Figure 20. A query to report PowerShell parameter blocking ........................................................ 15
Figure 21. A query to report on PowerShell launch by parent process ............................................ 15
Figure 22. Testing Parent Process Monitoring .................................................................................. 16
Figure 23. A sample query showing PowerShell launched by Parent Process ................................. 17
Figure 24. An Access Protection rule to prevent WinWord.exe launching PowerShell .................... 17
Figure 25. Testing WinWord.exe execution of PowerShell execution is blocked ............................... 18
Figure 26. Additional executables to be considered for process launch control .............................. 18
Figure 27. A query to report on PowerShell usage by user name ..................................................... 19
Figure 28. A sample query showing PowerShell usage by User Name ............................................. 19
Figure 29. An Access Protection rule to allow only the account named sccm to launch PowerShell ... 20
Figure 30. User account format by account type used in Access Protection Rules ............................. 20
Figure 31. Testing PowerShell execution has been controlled by user name .................................... 21
Figure 32. A Firewall rule to monitor PowerShell IP connectivity .................................................... 22
Figure 33. Testing PowerShell IP connectivity .................................................................................. 22
Figure 34. A query to report on PowerShell IP Connectivity .......................................................... 23
Figure 35. A sample query showing PowerShell Network Connectivity ......................................... 23
Figure 36. Firewall Rule to allow PowerShell access only to specified IP locations ............................ 24
Figure 37. Second Firewall Rule to block PowerShell access to any other locations ....................... 24
Figure 38. Testing PowerShell connectivity is blocked .................................................................... 25
Figure 39. An Access Protection rule to monitor PowerShell internal connectivity ........................ 25
Figure 40. Testing the Access Protection rule to monitor PowerShell internal connectivity ............. 26
Figure 41. Create a query to report on PowerShell internal connectivity ......................................... 26
Figure 42. A sample query showing PowerShell Internal Connectivity .......................................... 27
Figure 43. Subrule refinement process ............................................................................................ 27
Figure 44. Access Protection rule to control PowerShell internal connectivity ................................ 29
Figure 45. Testing the Access Protection rule to restrict PowerShell internal connectivity ............. 29
Figure 46. Create a query to report on PowerShell usage by time .................................................... 30
Figure 47. A sample query showing PowerShell usage by time ....................................................... 31
Figure 48. Components required to restricting PowerShell usage by time ....................................... 32
Figure 49. The workflow process illustrated ................................................................................... 33
Figure 50. Testing the workflow process components ..................................................................... 34
Background
PowerShell® is a task automation and configuration management framework from Microsoft. It comprises of a command-line shell and associated scripting language built on the .NET Framework, and supports COM, WMI, WS-Management and CIM. PowerShell enables administrative tasks on both local and remote Windows systems via a series of cmdlets.

Development started in 2002, (as Monad), and it was subsequently renamed to PowerShell and released in November 2006 for Windows XP SP2, Windows Server 2003 SP1 and Windows Vista. It is now a component of every new version of Windows.

Whilst PowerShell offers almost limitless management capabilities, it also offers malware authors a new platform on which to build attacks. PowerShell-based fileless scripts are an example of a new approach to malware delivery where an attack can be launched by loading a malicious text stream directly into memory without any need to write to disk. Since a file never gets written, it is easy to bypass any endpoint security solution that depends on file I/O to detect threats.

Introduction
This document will examine how PowerShell can be misused to compromise endpoints and how Endpoint Security 10.x can be used to reduce the attack surface of endpoints and therefore provide mitigation against these techniques.

Note that the examples will typically focus on ePolicy Orchestrator policies when discussing configuration, but equally most examples described can be delivered using identical configuration through the Endpoint Security 10 console.

Execution Policy
By default, PowerShell is configured to prevent the execution of scripts on Windows systems. Below, if a script is attempted to be executed, Windows will prevent this and highlight the fact that execution is disabled.

However, whilst this provides a level of protection, there are many ways to overcome this restriction, and this information can easily be located with a simple Internet search. If the same script is launched using the `-ExecutionPolicy` switch, then the script executes successfully. This is one example of bypassing protection, but there are many.

---

**Figure 1. Illustrating PowerShell's default execution policy.**

**Figure 2. Bypassing PowerShell's default execution policy.**
Scott Sutherland highlights "15 Ways to Bypass the PowerShell Execution Policy" and undoubtedly there are additional techniques documented or otherwise.

**Testing PowerShell execution**

This can be tested by following the steps below.

<table>
<thead>
<tr>
<th>Test Steps and Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.</strong> Create a text file containing the text below, and save as C:\Temp\Hello.ps1.</td>
</tr>
<tr>
<td>Write-Host</td>
</tr>
<tr>
<td>Write-Host 'Hello World!'</td>
</tr>
<tr>
<td>Write-Host &quot;Good-bye World! `n&quot;</td>
</tr>
<tr>
<td><strong>2.</strong> Run the following command from a command prompt running as administrator or equivalent:</td>
</tr>
<tr>
<td>PowerShell -ExecutionPolicy Bypass C:\Temp\Hello.ps1</td>
</tr>
<tr>
<td><strong>Expected output:</strong></td>
</tr>
<tr>
<td>Hello World!</td>
</tr>
<tr>
<td>Good-bye World!</td>
</tr>
</tbody>
</table>

*Figure 3. Testing PowerShell operation.*

**Parameters Commonly used by Malware**

The following parameters change the way that PowerShell operates, and can be beneficial to malware authors.

- **-Windowstyle hidden** - prevent Windows being displayed when executing code
- **-ExecutionPolicy Bypass** - bypass the default PowerShell script execution policy, (restricted)
- **-Encodedcommand** – Interpret text as base64 encoded string and bypass execution policy
- **-Noninteractive** – Do not create interactive prompt when executing code
- **-NoProfile** – prevent profile scripts launching when running PowerShell

**Script Obfuscation**

PowerShell can be obfuscated easily. The figure below gives examples of how the parameters above can be substituted for stripped-down but equivalent commands.

<table>
<thead>
<tr>
<th>Original</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Windowstyle hidden</td>
<td>-W Hidden</td>
</tr>
<tr>
<td>-ExecutionPolicy Bypass</td>
<td>-exec bypass</td>
</tr>
<tr>
<td>-Encodedcommand</td>
<td>-eNCo</td>
</tr>
<tr>
<td>-Noninteractive</td>
<td>-noni</td>
</tr>
<tr>
<td>-NoProfile</td>
<td>-nop</td>
</tr>
</tbody>
</table>

*Figure 4. Obfuscation of common PowerShell parameters.*

The command: `powershell -executionpolicy bypass -command write-host "Hi"`

...can be obfuscated as: `powershell -eNCo "`V`w`B`y`A`GkAdAB1AC0ASABvA`HMAdAAgA`EgAsQA="`
This simple obfuscation can be dramatically enhanced using Invoke-Obfuscation by Daniel Bohannon.

The command...
Write-Host 'Hello World!
...can be transformed into...

```powershell
([runtIMe.iNteRopSerVIces.MarSHaL].GetMeMBErs()[2].nAME).iNvoke([rUNTiME.interOpSERVICeS.marshAl]:SeCUresTrINGTOG10BaLALLoCunICOde({"76492d1116743f0423413b16050a5345MgB8AEMUAQBFAIUQA5ADIAYwBPADIATwBDAEMAAb5AHQASgBSAhcAbQBjAgcApQA9AHwANgA4ADANwA5ADMAZgBkADEAzgA1ADcAQQA0ADQANQBlADMAZQhAGQAMQA2A
DAA2AAwADAAYyAyADkAYzaAGEAyQBkADQAZgA2ADAAywAxAgQAMQA2AGIANwAwADcAOAAwADAANgBiADEAZAA1ADAAMgBjADYAYQAyAGMYwAzADUANQAIAGUANgBmAAGMAQAA4ADIANwBjADIMQAzADEAYQB1AGMANQAxA
DEAZQyADIA0A4AGIA2QA5ADYAYwBjADcAMAAyADMANABkADkANQB1AGUAMgA4ADUAYwBmADYAYgBmAyYAMQQA3ADIANAA0ADkAYwA2ADAAANAA3AA==" | ConveRTto-SECUrESTrIng -K 163,218,25,12,204,203,87,144,241,219,
23,163,227,217,254,40,182,221,83,222,119,192,252,169) }}|IEx
```

Figure 5. Obfuscating PowerShell commands.

Use of PowerShell in attacks
Malicious PowerShell scripts are predominantly used as downloaders, such as within Microsoft Office document macros, during the incursion phase. This document will focus on steps that can be taken to reduce the attack surface used during the incursion phase. The second most common use is during the lateral movement phase, allowing a threat to execute code on a remote computer when spreading inside the network. Whilst this document does not specifically examine this situation, the rules generated to hamper incursion, may also disrupt lateral movement.

Combatting PowerShell-based Malware
Given the immense power of PowerShell, and the numerous techniques that can be used to circumvent protection against its misuse, how can McAfee Endpoint Security 10 be used to provide protection?

The traditional approach to defending against malware of all types is to provide a signature that matches the malware, and then check that against any file accessed to ensure it is not malicious. However, there are factors that render that approach problematic, e.g. scripts can easily be changed, (since they are text-based), scripts lend themselves to obfuscation, (so heuristics approaches are difficult), and scripts may be loaded as a text stream, (and so there will be no file to scan).

The approach discussed here focusses around reducing the attack surface. For this whitepaper, the attack surface being referred to is that exposed due to the operating system supporting PowerShell.

PowerShell Execution Characteristics
To understand the attack surface, and attack surface reduction it is useful to consider the operation of PowerShell and its execution characteristics. These characteristics of execution can be readily identified, without needing to have any view, understanding or access to any script being executed. This provides a key benefit over any content-driven approach, because any attack surface reduction is effective irrespective of content obfuscation, variation or modification.
Execution characteristics include:

- The command line parameters associated with the launch of PowerShell.
- The process used to launch PowerShell, and potentially any processes launched by PowerShell.
- The user, or more correctly, the Windows security principal within which PowerShell is executing.
- The LAN or Internet location PowerShell is accessing, either to load a script or for other purposes.
- The time at which PowerShell is executed.

Each of the above can be explored to determine if a restriction can be applied in some way to make an effective attack via PowerShell difficult or impossible. Whilst the author acknowledges that it is unlikely that any large organization will be able to utilize all the approaches examined, it is essential to understand that using only a single approach may dramatically reduce an organization's exposure to this type of malware.

A Word of Caution

All the following information involves configuration changes being made to McAfee Endpoint Security, typically via ePolicy Orchestrator. Before making any changes to a live production environment, it is recommended that any proposed changes are tested in a test or staging environment to ensure that they can be implemented successfully without having a negative impact on the operation of the endpoints being managed.

Once this has completed, it is recommended that a pilot group of production systems is chosen to test the new configuration. This is essential since live and test systems often behave differently, and so the impact of the configuration changes may be very different in a live environment.

In addition, the volume of events generated in a production environment may be an order of magnitude greater than that generated in a test environment. This fact must be accommodated when introducing new configuration operating in a monitoring mode, where large volumes of events are typical anyway.

Reporting PowerShell Usage

Rationale

The first step in controlling PowerShell is to understand where it is being used, and how often. If it is never typically used, blocking it may be a trivial task. If it is used extensively across the environment, restricting it may involve significant investigation and understanding before this can be anticipated. Finally, in some parts of the organization it may be used extensively, whilst in others never at all, so different approaches may be possible for different groups of systems.
PowerShell> Reducing the Attack Surface

PowerShell Usage Access Protection Rule
To determine where PowerShell is being used, create an access protection rule as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Within the Policy Catalog access the Endpoint Security Threat Prevention product.</td>
</tr>
<tr>
<td>2. Set the Category to Access Protection.</td>
</tr>
<tr>
<td>3. Select the appropriate policy and edit it to make the following changes.</td>
</tr>
<tr>
<td>4. Add a rule...</td>
</tr>
<tr>
<td>5. Name: Monitor PowerShell Usage</td>
</tr>
<tr>
<td>6. Action: Report</td>
</tr>
<tr>
<td>7. Add a Subrule...</td>
</tr>
<tr>
<td>8. Name: PowerShell</td>
</tr>
<tr>
<td>9. Operations: Execute</td>
</tr>
<tr>
<td>10. Add a Target...</td>
</tr>
<tr>
<td>11. Inclusion status: Include</td>
</tr>
<tr>
<td>12. Select File Path from dropdown</td>
</tr>
<tr>
<td>13. File, folder name, or file path: PowerShell.exe.</td>
</tr>
</tbody>
</table>

Figure 7. An Access Protection rule to allow PowerShell usage to be monitored.

Testing the PowerShell Usage Rule

This can be tested by following the steps below, and confirming an event is generated on the endpoint.

<table>
<thead>
<tr>
<th>Test Steps and Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the following command from a command prompt running as administrator or equivalent:</td>
</tr>
<tr>
<td>PowerShell -ExecutionPolicy Bypass C:\Temp\Hello.ps1</td>
</tr>
<tr>
<td>Expected output:</td>
</tr>
<tr>
<td>Hello World!</td>
</tr>
<tr>
<td>Good-bye World!</td>
</tr>
</tbody>
</table>

Figure 8. Testing the Access Protection Monitoring Rule.

Query to Monitor PowerShell Usage

To report PowerShell usage using ePolicy Orchestrator reporting, create a query as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a new query.</td>
</tr>
<tr>
<td>2. Result type: Threat Events.</td>
</tr>
<tr>
<td>3. Chart type: Bar Chart.</td>
</tr>
<tr>
<td>4. Bar Labels: System Name (this may need to be modified based on the number of endpoints returning data).</td>
</tr>
<tr>
<td>5. Bar Values: Number of Threat Events.</td>
</tr>
<tr>
<td>6. Selected Columns: Event Generated Time, Threat Target Host Name, Threat Source Process Name.</td>
</tr>
<tr>
<td>7. Filtering:</td>
</tr>
<tr>
<td>Threat Name Equals Monitor PowerShell Usage.</td>
</tr>
<tr>
<td>8. Query Name: Monitor PowerShell Usage.</td>
</tr>
</tbody>
</table>

Figure 9. A query to allow PowerShell usage to be reported.
Sample Reporting of PowerShell Usage

The figure below shows the output from the above query. In a large environment, it may be worth using the output from this to tag endpoints with a PowerShell tag to indicate that they have been observed to use Powershell.

![Sample Reporting of PowerShell Usage](image)

Figure 10. A sample query showing PowerShell usage.

This monitoring and tagging process could be automated by changing the chart type to be table; adding an Event Generated Time filter to the query, (such as weekly); adding an ePolicy Orchestrator server scheduled task to run periodically, (such as weekly to match the filtering interval); and setting a resultant action to apply a tag.

Restricting PowerShell Usage

Rationale

Whilst it may not be practical to prevent all or maybe any endpoints from executing PowerShell, it remains the most powerful technique that can be used against Powershell-based malware.

Even if this is the case, some organizations will create a set of lockdown policies for use when they are actively under attack and fighting a malware outbreak, or expecting this to happen imminently. When PowerShell cannot be blocked under normal circumstances, it should be considered for inclusion in this type of policy.

A single Access Protection rule, (very like that used for monitoring), can be created to prevent execution, as below. Whilst the Monitor PowerShell Usage rule above could simply be reused, (configured to block in addition to report), the use of a separate rule simplifies reporting if monitoring of some systems, and blocking of others, is occurring concurrently.
PowerShell Block Access Protection Rule

The following rule will prevent PowerShell being launched on an endpoint.

**Configuration Steps**

1. Within the Policy Catalog access the Endpoint Security Threat Prevention product.
2. Set the Category to Access Protection.
3. Select the appropriate policy and edit it to make the following changes.
4. Add a rule…
   - Name: Block PowerShell Usage
   - Action: Block & Report
   - Add a Subrule…
     1. Name: PowerShell
     2. Operations: Execute
     3. Add a Target…
        1. Inclusion status: Include
        2. Select File Path from dropdown
        3. File, folder name, or file path: PowerShell.exe.

*Figure 11. An Access Protection rule to block PowerShell usage.*

Testing the PowerShell Usage Rule

The above rule can be tested using the command below.

**Test Steps and Expected Output**

1. Run the following command from a command prompt running as administrator or equivalent:
   ```
   PowerShell -ExecutionPolicy Bypass C:\Temp\Hello.ps1
   ```

   Expected output:
   ```
   Access is denied
   ```

*Figure 12. Testing the Access Protection Blocking Rule.*

Query to Monitor PowerShell Usage

To report on blocking of PowerShell through ePolicy Orchestrator reporting, create a query as below.

**Configuration Steps**

1. Create a new query.
2. Result type: Threat Events.
3. Chart type: Bar Chart.
4. Bar Labels: System Name (this may need to be modified based on the number of endpoints returning data).
5. Bar Values: Number of Threat Events.
6. Selected Columns: Event Generated Time, Threat Target Host Name, Threat Source Process Name.
7. Filtering:
   Threat Name Equals Block PowerShell Usage.
8. Query Name: Block PowerShell Usage.

*Figure 13. A query to allow PowerShell blocking to be reported.*
So ultimately, one simply rule can remove any possibility that PowerShell can be used as an attack vector executing on an endpoint. Note however, this will not protect endpoints from PowerShell scripts executing on other, (unprotected), endpoints and connecting remotely to a protected endpoint.

**Reporting PowerShell Parameter Usage**

**Rationale**

As discussed previously, there are several parameters, (or switches), that can be used to change the default behaviour of PowerShell once launched. Whilst there may be legitimate reasons for their use, some of these parameters are used extensively by malware. If specific parameters can be isolated as never being used within an organization, then banning their use can be an effective way of preventing malware that utilizes these from being effective.

The Exploit Prevention capability within the Threat Prevention module provides individual rules for monitoring the use of commonly abused parameters.

**PowerShell rules within Exploit Prevention**

At time of writing the following Exploit Prevention rules were available within the Endpoint Security Threat Prevention module.

<table>
<thead>
<tr>
<th>Signature Id.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6070</td>
<td>Hidden Powershell Detected.</td>
</tr>
<tr>
<td>6073</td>
<td>ExecutionPolicy Bypass in Powershell.</td>
</tr>
<tr>
<td>6081</td>
<td>Powershell Command Restriction – NoProfile.</td>
</tr>
<tr>
<td>6082</td>
<td>Powershell Command Restriction – ExecutionPolicy Unrestricted.</td>
</tr>
<tr>
<td>6083</td>
<td>Powershell Command Restriction - NonInteractive.</td>
</tr>
<tr>
<td>6084</td>
<td>Powershell Command Restriction – NoLogo.</td>
</tr>
<tr>
<td>6085</td>
<td>Powershell Command Restriction – File.</td>
</tr>
<tr>
<td>6086</td>
<td>Powershell Command Restriction – Command.</td>
</tr>
<tr>
<td>6087</td>
<td>Powershell Command Restriction – EncodedCommand.</td>
</tr>
<tr>
<td>6096</td>
<td>Powershell Command Restriction – InvokeExpression.</td>
</tr>
</tbody>
</table>

*Figure 14. PowerShell parameter rules available within Exploit Prevention.*

**Monitor PowerShell Parameter Usage Access Protection Rule**

To determine which PowerShell parameters are being used, create an exploit prevention rule as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Within the Policy Catalog access the Endpoint Security Threat Prevention product.</td>
</tr>
<tr>
<td>2. Set the Category to Exploit Prevention.</td>
</tr>
<tr>
<td>3. Select the appropriate policy and edit it to make the following changes.</td>
</tr>
<tr>
<td>4. Locate the signatures listed above. Note that these signatures have a severity of Disabled or</td>
</tr>
<tr>
<td>Informational.</td>
</tr>
<tr>
<td>5. Set the signatures Action: Report.</td>
</tr>
</tbody>
</table>

*Figure 15. An Exploit Prevention policy to monitor PowerShell parameter usage.*
Testing the PowerShell Parameters Usage Rule

This can be tested by following the steps below, and confirming events are generated on the endpoint.

**Test Steps and Expected Output**

1. Run the following command from a command prompt running as administrator or equivalent:
   ```powershell
   PowerShell -ExecutionPolicy Bypass -nop -noni -command write-host "Hi"
   ```

   **Expected output:**
   ```
   Hi
   ```

2. Run the following command from a command prompt running as administrator or equivalent:
   ```powershell
   $encodedCommand = [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes('Write-Host Hi'));
   Write-Host "Encoded" $encodedCommand; powershell.exe -EncodedCommand $encodedCommand
   ```

   **Expected output:**
   ```
   Encoded VwByAGkAdABlAC0ASABvAHMAaAqAqAqQA=
   Hi
   ```

*Figure 16. Testing PowerShell parameter usage.*

Test 1 above should trigger 4 signatures and create 4 events, one for each of the monitored commands, (ExecutionPolicy Bypass, Powershell Command Restriction – NoProfile, Powershell Command Restriction - NonInteractive, and Powershell Command Restriction – Command). Test 2 should trigger 1 signature - Powershell Command Restriction - EncodedCommand.

**Query to report on PowerShell Parameter Usage**

To report on PowerShell parameter usage through ePolicy Orchestrator reporting, create a query as below.

**Configuration Steps**

1. Create a new query.
2. Result type: Threat Events.
3. Chart type: Bar Chart.
4. Bar Labels: Analyzer Rule Id.
5. Bar Values: Number of Threat Events.
6. Selected Columns: Event Generated Time, Threat Target Host Name, Analyzer Rule Id, Threat Parent Process Name.
7. Filtering:
   - Threat Target Process Name Equals Powershell.exe and Event Id equals 18054 and Action Taken Equals Would block.
8. Query Name: Monitor PowerShell Parameter Usage.

*Figure 17. A query to report PowerShell parameter usage.*
Sample Reporting of PowerShell Parameter Usage

The query above can produce the report shown below. Note that the Analyzer Rule ID matches the Exploit Prevention Signature Id.

![Sample Reporting of PowerShell Parameter Usage](image)

Figure 18. A sample query showing PowerShell parameter usage.

Restricting PowerShell Parameter Usage

Rationale

Whilst it may not be practical or desirable to block all PowerShell parameter usage, it may be entirely possible to block some commands, e.g. signature id 6087, “Powershell Command Restriction – EncodedCommand”. This is often associated with malware, but rarely used for legitimate management purposes.

PowerShell Block Parameter Access Protection Rule

To prevent the use of any specific parameters, simply switch the signature action from report to block & report, as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Within the Policy Catalog access the Endpoint Security Threat Prevention product.</td>
</tr>
<tr>
<td>2. Set the Category to Exploit Prevention.</td>
</tr>
<tr>
<td>3. Select the appropriate policy and edit it to make the following changes.</td>
</tr>
<tr>
<td>4. Locate the signatures required to be blocked. Note that these signatures have a severity of Disabled or Informational.</td>
</tr>
<tr>
<td>5. Set the signatures Action: Block and Report.</td>
</tr>
</tbody>
</table>

![PowerShell Block Parameter Access Protection Rule](image)

Figure 19. An Access Protection rule to block specific PowerShell parameter usage.

Testing the PowerShell Block Parameter Rule

To test this, simply repeat the test procedure listed above, substituting appropriate parameters. Note that if a single PowerShell command is blocked, only the first signature match on a parameter will trigger. This is different behaviour to that seen when simply monitoring for parameter usage, since when blocking the
command will be blocked from attempting any further parameter usage. As a result, only a single event will be captured for each blocked execution.

**Query to monitor PowerShell Parameter Restrictions**

To report PowerShell Parameters blocked using ePolicy Orchestrator reporting, create a query as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a new query.</td>
</tr>
<tr>
<td>2. Result type: Threat Events.</td>
</tr>
<tr>
<td>3. Chart type: Bar Chart.</td>
</tr>
<tr>
<td>4. Bar Labels: Analyzer Rule Id.</td>
</tr>
<tr>
<td>5. Bar Values: Number of Threat Events.</td>
</tr>
<tr>
<td>6. Selected Columns: Event Generated Time, Threat Target Host Name, Analyzer Rule Id, Threat Parent Process Name.</td>
</tr>
<tr>
<td>7. Filtering: Threat Target Process Name Equals Powershell.exe and Event Id equals 18054 and Action Taken equals blocked.</td>
</tr>
<tr>
<td>8. Query Name: Block PowerShell Parameter Usage.</td>
</tr>
</tbody>
</table>

*Figure 20. A query to report PowerShell parameter blocking.*

**Reporting Powershell Usage by Process**

**Rationale**

Within some organizations it may be clearly understood where and how Powershell is being used. However, it may be that in others, especially larger organizations, this is not so clear. One method of gaining some clarity is to report on PowerShell usage by process, i.e. reporting on any process that was used to launch PowerShell – i.e. the parent process. Not only may this provide an insight into how PowerShell is being used legitimately, it may uncover illegitimate or at least unwanted activity.

**PowerShell Launch Process Access Protection Rule**

This can be achieved very easily, by reusing the data from the original Monitor PowerShell Usage rule and displaying it differently within a query.

**Query to Monitor Processes used to Launch Powershell**

To report on process usage of PowerShell through ePolicy Orchestrator reporting, create a query as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a new query.</td>
</tr>
<tr>
<td>2. Result type: Threat Events.</td>
</tr>
<tr>
<td>3. Chart type: Bar Chart.</td>
</tr>
<tr>
<td>5. Bar Values: Number of Threat Events.</td>
</tr>
<tr>
<td>6. Selected Columns: Event Generated Time, Threat Target Host Name, Threat Source Process Name.</td>
</tr>
<tr>
<td>7. Threat Name Equals Monitor PowerShell Usage.</td>
</tr>
</tbody>
</table>

*Figure 21. A query to report on PowerShell launch by parent process.*
Testing PowerShell Parent Launch Process Monitoring

Test 1 can be used to show PowerShell being launched with cmd.exe as the parent process, whilst test 2 shows PowerShell being launched by WinWord.exe.

<table>
<thead>
<tr>
<th>Test Steps and Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the following command from a command prompt running as administrator or equivalent:</td>
</tr>
<tr>
<td>PowerShell -ExecutionPolicy Bypass C:\Temp\Hello.ps1</td>
</tr>
<tr>
<td>Expected output:</td>
</tr>
<tr>
<td>Hello World!</td>
</tr>
<tr>
<td>Good-bye World!</td>
</tr>
<tr>
<td>2. Create a Macro within Microsoft Word using the text below, (View</td>
</tr>
<tr>
<td>Sub Simple()</td>
</tr>
<tr>
<td>' Simple Macro</td>
</tr>
<tr>
<td>Dim objShell, PScommand</td>
</tr>
<tr>
<td>Set objShell = CreateObject(&quot;Wscript.shell&quot;)</td>
</tr>
<tr>
<td>PScommand = &quot;powershell -ExecutionPolicy Bypass -file c:\temp\hello.ps1&quot;</td>
</tr>
<tr>
<td>objShell.Run PScommand, 0</td>
</tr>
<tr>
<td>'</td>
</tr>
<tr>
<td>End Sub</td>
</tr>
<tr>
<td>3. The rule can be tested by running the Macro, (View</td>
</tr>
<tr>
<td>Expected output:</td>
</tr>
<tr>
<td>No output will be shown but an event will be generated</td>
</tr>
</tbody>
</table>

Figure 22. Testing Parent Process Monitoring.

Sample Report of PowerShell Parent Launch Process

The figure below shows the output from the above query. Results will differ based on the actual usage of PowerShell within the specific environment.
PowerShell> Reducing the Attack Surface

Figure 23. A sample query showing PowerShell launched by Parent Process.

Restricting Powershell Usage by Process

Rationale
From the results above it may be determined that some processes should not be launching PowerShell. To enforce this, duplicate the Block PowerShell Usage access protection rule and modify it, as shown in italics below, where WinWord.exe is being prevented from launching PowerShell. Substitute WinWord.exe for all process names to be blocked.

Block Process used to launch Powershell Access Protection Rule
To block a specific process from launching PowerShell, create an access protection rule as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Within the Policy Catalog access the Endpoint Security Threat Prevention product.</td>
</tr>
<tr>
<td>2. Set the Category to Access Protection.</td>
</tr>
<tr>
<td>3. Select the appropriate policy and edit it to make the following changes.</td>
</tr>
<tr>
<td>4. Add a rule...</td>
</tr>
<tr>
<td>Name: Block PowerShell launch by WinWord.exe</td>
</tr>
<tr>
<td>Action: Block &amp; Report</td>
</tr>
<tr>
<td>Executable:</td>
</tr>
<tr>
<td>1. Name: WinWord</td>
</tr>
<tr>
<td>2. Inclusion Status: Include</td>
</tr>
<tr>
<td>3. File Name or path: WinWord.exe</td>
</tr>
<tr>
<td>Add a Subrule...</td>
</tr>
<tr>
<td>4. Name: PowerShell</td>
</tr>
<tr>
<td>5. Operations: Execute</td>
</tr>
<tr>
<td>6. Add a Target...</td>
</tr>
<tr>
<td>1. Inclusion status: Include</td>
</tr>
<tr>
<td>2. Select File Path from dropdown</td>
</tr>
<tr>
<td>3. File, folder name, or file path: PowerShell.exe.</td>
</tr>
</tbody>
</table>

Figure 24. An Access Protection rule to prevent WinWord.exe launching PowerShell.
Testing PowerShell Block Parent Launch Process

The rule can be tested using the information below.

### Test Steps and Expected Output

1. The rule can be tested by running the Macro created earlier, (View | Macros | View Macros | Select the Macro | Run).

<table>
<thead>
<tr>
<th>Expected output:</th>
<th>Run-time error ‘70’:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permission denied</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 25. Testing WinWord.exe execution of PowerShell execution is blocked.*

This approach specifically prohibits (blacklists) WinWord.exe from launching PowerShell. With more effort, a whitelisting approach could be adopted, whereby rather than blocking named processes, only named processes would be allowed to launch PowerShell. To achieve this, the executable inclusion status should be changed to Exclude, and authorized processes substituted for WinWord.exe.

Extending Process Launch Control

If adopting the approach as above, there are several other processes that should be considered for control.

The figure below makes some recommendations as to both the executable, (the parent process attempting to launch a process), and the subrule target, (the process to be launched).

<table>
<thead>
<tr>
<th>Access Protection component</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executables</td>
<td>WinWord.exe Excel.exe PowerPnt.exe Wscript.exe Cscript.exe</td>
</tr>
<tr>
<td>Subrule targets</td>
<td>PowerShell.exe Cmd.exe Wscript.exe Cscript.exe</td>
</tr>
</tbody>
</table>

*Figure 26. Additional executables to be considered for process launch control.*

Reporting PowerShell Usage by User

Rationale

Depending on how PowerShell is used within the environment, it may be that different Windows accounts, i.e. different Windows security principals, are used to execute PowerShell, and that typical user accounts are not being used. For example, if System Center Configuration Manager (SCCM) is used to launch PowerShell, and the SCCM service (CcmExec.exe) is running under a different Windows security principal to the user, then it may be that the only account that requires the rights to launch PowerShell is that used by the SCCM service.

One method of gaining some clarity is to report on PowerShell usage by user, i.e. reporting on the user that was used to launch PowerShell. Not only may this provide an insight into how PowerShell is being used legitimately, but again it may uncover illegitimate or unwanted activity.
PowerShell User Accounts Usage Access Protection Rule
This can be achieved very easily, by reusing the data from the original Monitor PowerShell Usage rule and displaying it differently within a query.

Query to Monitor User Accounts used to launch PowerShell
To report on process usage of PowerShell through ePolicy Orchestrator reporting, create a query as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a new query.</td>
</tr>
<tr>
<td>2. Result type: Threat Events.</td>
</tr>
<tr>
<td>3. Chart type: Bar Chart.</td>
</tr>
<tr>
<td>5. Bar Values: Number of Threat Events.</td>
</tr>
<tr>
<td>6. Selected Columns: Event Generated Time, Threat Target Host Name, Threat Source Process Name.</td>
</tr>
<tr>
<td>7. Threat Name Equals Monitor PowerShell Usage.</td>
</tr>
<tr>
<td>8. Query Name: Monitor PowerShell Usage by User Name.</td>
</tr>
</tbody>
</table>

*Figure 27. A query to report on PowerShell usage by user name.*

Sample Report of PowerShell User Account Usage
The figure below shows the output from this query.

*Figure 28. A sample query showing PowerShell usage by User Name.*

Restricting PowerShell Usage by User

Rationale
From the results above it may be determined that some user accounts should not be launching PowerShell.

Restrict Users Accounts for PowerShell Access Protection Rule
To enforce this, duplicate the Block PowerShell Usage access protection rule and modify it, as highlighted below, where in this example only the local account ‘sccm’ is authorized to access PowerShell. Substitute ‘sccm’ with all authorized user names.
**Configuration Steps**

1. Within the Policy Catalog access the Endpoint Security Threat Prevention product.
2. Set the Category to Access Protection.
3. Select the appropriate policy and edit it to make the following changes.
4. Add a rule...
   - Name: Allow only SCCM user to run PowerShell
     - Action: Block & Report
   - **User Names:**
     1. **Name:** .\sccm
     2. **Inclusion Status:** Exclude
   - Add a Subrule...
     3. **Name:** PowerShell
     4. **Operations:** Execute
     5. Add a Target...
       1. Inclusion status: Include
       2. Select File Path from dropdown
       3. File, folder name, or file path: PowerShell.exe

*Figure 29. An Access Protection rule to allow only the account named sccm to launch PowerShell.*

**Local and Domain Names**

The above example uses a local account, but domain accounts can also be used, as below.

<table>
<thead>
<tr>
<th>Type of user account</th>
<th>Format required in Access Protection rule and notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain user</td>
<td><strong>Domain\username</strong></td>
</tr>
<tr>
<td></td>
<td>Note that as part of the rule creation process the local machine will attempt to resolve any usernames to corresponding Windows Security Identifiers (SIDs). Therefore, if a domain account is specified, a domain controller must be available to service this request, in order for the rule to be applied successfully.</td>
</tr>
<tr>
<td>Local user</td>
<td><strong>.\username</strong></td>
</tr>
<tr>
<td></td>
<td>Note that whilst the format ‘machine name\local user name’ can equally be used to represent a local account, the format above is especially useful if the policy needs to be applied to multiple different endpoints.</td>
</tr>
</tbody>
</table>

*Figure 30. User account format by account type used in Access Protection Rules.*

**Testing User Account Restrictions**

The rule can be tested using the information below.
Test Steps and Expected Output

1. Log in to Windows with a user name specified within the Access Protection rule.
2. Run the following command from a command prompt running as administrator or equivalent:
   ```powershell
   PowerShell -ExecutionPolicy Bypass C:\Temp\Hello.ps1
   ```

   Expected output:
   ```
   Hello World!
   Good-bye World!
   ```

3. Log in to Windows with a user name not specified within the Access Protection rule.
4. Run the following command from a command prompt running as administrator or equivalent:
   ```powershell
   PowerShell -ExecutionPolicy Bypass C:\Temp\Hello.ps1
   ```

   Expected output:
   ```
   Access is denied.
   ```

Figure 31. Testing PowerShell execution has been controlled by user name.

Reporting Powershell IP Connectivity

Rationale

Whilst Powershell may be used extensively within some organizations, it is expected that it will not need to connect to many locations either inside or outside of the organization, and that these locations will be well known and trusted.

However, if the exact connectivity requirements are not known, this information can be collected through the Endpoint Security Firewall module. Later, once all locations are known, the Endpoint Security Firewall module can then be used to enforce a policy that only allows connectivity to these locations.

In doing so, this reduces the possibility of PowerShell being used in a fileless attack, where a PowerShell script is downloaded as a text stream from any Internet-accessible location.

PowerShell IP Connectivity Firewall Monitoring Rule

To determine to where PowerShell is making connections, modify Firewall policies as below.

Note that this control requires that the Endpoint Security Firewall is enabled, and has rules configured to allow normal network communications to occur. If the Firewall module is not installed or installed and not enabled, then it should be configured and enabled prior to following the steps below. Exactly how to achieve this requirement is out of the scope of this document.
PowerShell> Reducing the Attack Surface

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Within the Policy Catalog access the Endpoint Security Firewall product.</td>
</tr>
<tr>
<td>2. Set the Category to Options.</td>
</tr>
<tr>
<td>3. Select the appropriate policy and edit it to make the following changes.</td>
</tr>
<tr>
<td>4. Ensure that the checkbox Enable Firewall intrusion alerts is checked, and save this policy.</td>
</tr>
<tr>
<td>5. Within the Policy Catalog set the Category to Rules.</td>
</tr>
<tr>
<td>6. Select the appropriate policy and edit it to make the following changes.</td>
</tr>
<tr>
<td>7. Create a new rule with the following configuration...</td>
</tr>
<tr>
<td>• Name: Allow PowerShell Traffic</td>
</tr>
<tr>
<td>• Status: Enabled</td>
</tr>
<tr>
<td>• Actions: Allow &amp; Treat match as intrusion</td>
</tr>
<tr>
<td>• Network protocol: IP protocol; IPv4 protocol &amp; IPv6 protocol</td>
</tr>
<tr>
<td>• Connection types, Wired, Wireless and Virtual</td>
</tr>
<tr>
<td>• Transport: All Protocols</td>
</tr>
<tr>
<td>• Applications...</td>
</tr>
<tr>
<td>o Name: PowerShell</td>
</tr>
<tr>
<td>o Executable...</td>
</tr>
<tr>
<td>▪ Name: PowerShell.exe</td>
</tr>
<tr>
<td>▪ File Name or Path: PowerShell.exe</td>
</tr>
<tr>
<td>8. Move this rule above any other rules that would allow PowerShell to gain Internet access, such as the default Web/FTP rule. The rule can be placed directly below McAfee core networking if required.</td>
</tr>
</tbody>
</table>

*Figure 32. A Firewall rule to monitor PowerShell IP connectivity.*

**Testing the IP Connectivity Firewall Monitoring Rule**

This can be tested by following the steps below, and confirming an event is generated on the endpoint. Prior to running this test, best practice recommends that hello.jpg, (a powershell script), is downloaded and checked to ensure it is harmless.

<table>
<thead>
<tr>
<th>Test Steps and Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the following command from a command prompt running as administrator or equivalent:</td>
</tr>
<tr>
<td>Expected output:</td>
</tr>
<tr>
<td>Hello World!</td>
</tr>
<tr>
<td>Good-bye World!</td>
</tr>
</tbody>
</table>

*Figure 33. Testing PowerShell IP connectivity.*

**Query to Monitor PowerShell IP Connectivity**

To report on PowerShell Internet connectivity through ePolicy Orchestrator reporting, create a query as below.
### Configuration Steps

1. Create a new query.
2. Result type: Endpoint Security Threat Events.
3. Chart type: Groupe Bar Chart.
6. Bar Values: Number of Endpoint Security Threat Events.
7. Selected Columns: Event Generated Time, System Name, Threat Target IPv4, Threat Target Port Number, Action Taken, Source Parent Process Name.
8. Detecting Product Name Equals McAfee Endpoint Security and Event ID equals 35001.

*Figure 34. A query to report on PowerShell IP Connectivity.*

#### Sample Report of PowerShell IP Connectivity

The figure below shows the output possible from this query.

*Figure 35. A sample query showing PowerShell Network Connectivity.*

### Restricting Powershell IP Connectivity

#### Rational

Once required network connectivity has been identified, this can be enforced by modifying the original PowerShell Firewall rule, and adding a second rule to block all other PowerShell access.

#### PowerShell IP Connectivity Restriction Firewall Rule

To restrict where PowerShell is able to make connections, modify Firewall policies as below. The first rule ensures that PowerShell can access the required IP locations, whilst the second blocks all remaining PowerShell connection attempts.
Configuration Steps

1. Within the Policy Catalog access the Endpoint Security Firewall product.
2. Set the Category to Options.
3. Select the appropriate policy and edit it to make the following changes.
4. Ensure that the checkbox Enable Firewall intrusion alerts is checked, and save this policy.
5. Set the Category to Rules.
6. Select the appropriate policy and edit it to make the following changes.
7. Create a new rule with the following configuration...
   - Name: Restrict PowerShell Traffic
   - Status: Enabled
   - Actions: Allow
   - Network protocol: IP protocol; IPv4 protocol & IPv6 protocol
   - Connection types, Wired, Wireless and Virtual
   - Under Network Name, add all required IP addresses, IP address ranges, or fully qualified domain names.
   - Transport: All Protocols
   - Applications...
     - Name: PowerShell
     - Executable...
       - Name: PowerShell.exe
       - File Name or Path: PowerShell.exe.
8. Ensure this rule is above any other rules that would allow PowerShell to gain Internet access, such as the default Web/FTP rule.

Figure 36. Firewall Rule to allow PowerShell access only to specified IP locations.

Configuration Steps

1. Within the Policy Catalog access the Endpoint Security Firewall product.
2. Set the Category to Rules.
3. Select the appropriate policy and edit it to make the following changes.
4. Create a new rule with the following configuration...
   - Name: Block PowerShell Traffic
   - Status: Enabled
   - Actions: Block & Treat match as intrusion
   - Network protocol: IP protocol; IPv4 protocol & IPv6 protocol
   - Connection types, Wired, Wireless and Virtual
   - Transport: All Protocols
   - Applications...
     - Name: PowerShell
     - Executable...
       - Name: PowerShell.exe
       - File Name or Path: PowerShell.exe.
5. Ensure this rule is directly below the Restrict PowerShell Traffic rule.

Figure 37. Second Firewall Rule to block PowerShell access to any other locations.

Testing PowerShell IP Restrictions

This can be tested by following the steps below, and confirming an event is generated on the endpoint. The above test assumes that jaybeetest.000webhostapp.com is not included within the list of allowed network locations.
PowerShell> Reducing the Attack Surface

Test Steps and Expected Output

1. Run the following command from a command prompt running as administrator or equivalent:


   **Expected output:**

   ```
   Exception calling "DownloadString" with "1" argument(s): "Unable to connect to the remote server"
   At line:1 char:47
   + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
   + FullyQualifiedErrorId : DotNetMethodException
   ```

Figure 38. Testing PowerShell connectivity is blocked.

Reporting PowerShell Internal Connectivity

Rationale

Another method of locking down PowerShell is to limit the internal locations from which PowerShell may launch scripts. Since it's not typically possible to determine where malicious scripts may be launched, this approach is that of whitelisting - specifying only the allowed internal locations, and blocking all other locations. These allowed locations may be on the local disk or on the local area network.

The first step of this process is to understand what locations are required to be accessed by PowerShell. Unlike the rule designed to determine general network connectivity, via a Firewall rule, this approach once again uses access protection to report on, (and ultimately limit), PowerShell's ability to access internal locations.

Monitor PowerShell Internal Connectivity Access Protection Rule

To determine which internal locations are being accessed by PowerShell, create an access protection rule as below.

Configuration Steps

1. Within the Policy Catalog access the Endpoint Security Threat Prevention product.
2. Set the Category to Access Protection.
3. Select the appropriate policy and edit it to make the following changes.
4. Add a rule...
   - Name: Monitor PowerShell Internal Connectivity
   - Action: Report
   - **Executable:**
     1. Name: PowerShell
     2. Inclusion Status: Include
     3. File Name or path: PowerShell.exe
   - Add a Subrule...
     1. Name: PowerShell file locations
     2. Operations: Read
     3. Add a Target...
        1. Inclusion status: Include
        2. Select File Path from dropdown
        3. File, folder name, or file path: *

Figure 39. An Access Protection rule to monitor PowerShell internal connectivity.
Testing the Monitor PowerShell Internal Connectivity Rule

This rule can be tested by following the steps below, and confirming an event is generated on the endpoint. In fact, due to the huge number of interactions that PowerShell has with files within the operating system, a huge number of events are likely to be generated. For this reason, it makes sense to limit the number of endpoints this rule is initially developed on, until the policy is more accurately defined.

<table>
<thead>
<tr>
<th>Test Steps and Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Run the following command from a command prompt running as administrator or equivalent:</td>
</tr>
<tr>
<td><code>PowerShell -ExecutionPolicy Bypass C:\Temp\Hello.ps1</code></td>
</tr>
<tr>
<td>Expected output:</td>
</tr>
<tr>
<td>Hello World!</td>
</tr>
<tr>
<td>Good-bye World!</td>
</tr>
</tbody>
</table>

*Figure 40. Testing the Access Protection rule to monitor PowerShell internal connectivity.*

Query to Monitor PowerShell Internal Connectivity

To report on PowerShell internal connectivity through ePolicy Orchestrator reporting, create a query as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a new query.</td>
</tr>
<tr>
<td>2. Result type: Threat Events.</td>
</tr>
<tr>
<td>3. Chart type: Table.</td>
</tr>
<tr>
<td>4. Reporting: Threat Target File Path.</td>
</tr>
<tr>
<td>5. Selected Columns: Event Generated Time, Threat Target File Path.</td>
</tr>
<tr>
<td>7. Query Name: Monitor PowerShell Internal Connectivity.</td>
</tr>
</tbody>
</table>

*Figure 41. Create a query to report on PowerShell internal connectivity.*

Sample Report of PowerShell Internal Connectivity

The figure below shows the output from this query.
PowerShell> Reducing the Attack Surface

Figure 42. A sample query showing PowerShell Internal Connectivity.

Rule Refinement Process.

The rule is initially configured to capture all file read access made by PowerShell, no matter the location, (hence the Include * entry in the subrule).

The objective of the refinement process is to modify this subrule to exclude all locations that PowerShell needs to access to run, so that when it does run, it does not trigger on these locations, and, (once the rule is being enforced in a blocking mode), PowerShell is not blocked from executing.

For this reason, testing and rule refinement may be better carried on the endpoint directly, (to develop and test the exclusions), and then recreated within an ePolicy Orchestrator policy once complete.
Authors Subrule Refinements
During the refinement process within a simple test environment, the following subrule entries were created. Note that a number of these entries were designed to capture multiple exclusions, rather than creating an individual exclusion for every file read by PowerShell.

<table>
<thead>
<tr>
<th>Subrule entry type</th>
<th>Entry</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include</td>
<td>*</td>
<td>The initial entry to capture all file access performed by PowerShell during operation.</td>
</tr>
<tr>
<td>Exclude</td>
<td>C:\WINDOWS*.*.CLB</td>
<td>Entries that represent PowerShell file read access during normal operations, and so location that should be excluded from the rule.</td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.CONFIG</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.CPL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.DAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.DLL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.HIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.MUI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.NL?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.PF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS\SYSTEM32*.*.PS1XML</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*\POWERSHELL.EXE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DESKTOP.INI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS*.*.EXE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\PROGRAM FILES*.*.EXE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\PROGRAM FILES\MCAFEE\ENDPOINT\SECURITY*.DLL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\USERS*APPDATA\ROAMING\MICROSOFT\WINDOWS\RECENT\CUSTOMDESTINATIONS*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS\APP\PATCH\SYSMAIN.SDB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS\SYSTEM32*.*.DLL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\PIPE\SRVSVC\</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\USERS*.*.DB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C:\WINDOWS\WINNOW*MANIFEST</td>
<td></td>
</tr>
<tr>
<td>Exclude</td>
<td>IMPOWER2017-EPO\SCRIPTS$\GOOD*.PS1</td>
<td>Additional locations required to be accessed by PowerShell on the sample network, due to administrative requirements, (and so should be excluded from the rule).</td>
</tr>
<tr>
<td></td>
<td>C:\SCRIPTS*.PS1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>These will need to be customized to suit individual customer network requirements.</td>
</tr>
</tbody>
</table>

Restricting PowerShell Internal Connectivity
Once the subrule has been fully defined and tested, it can be enforced by switching from a report mode to a block and report mode, as below.
**Configuration Steps**

1. Within the Policy Catalog access the Endpoint Security Threat Prevention product.
2. Set the Category to Access Protection.
3. Select the appropriate policy and edit it to make the following changes.
4. Configure the rule...
   - Name: Enforce PowerShell Internal Connectivity
   - Action: Block & Report
   - **Executable:**
     1. Name: PowerShell
     2. Inclusion Status: Include
     3. **File Name or path:** PowerShell.exe
   - Add a Subrule...
     4. Name: PowerShell file locations
     5. Operations: Read
     6. Target should include the initial include * rule, combined with all the exclude rules developed in the refinement process.

*Figure 44. Access Protection rule to control PowerShell internal connectivity.*

**Testing PowerShell Internal Connectivity Restrictions**

The rule can be tested using the format below. In the example two shared paths are used to illustrate the rule in action. These will need to be customized to suit individual customer network requirements.

**Test Steps and Expected Output**

1. Run the following command from a command prompt running as administrator or equivalent:

   ```powershell
   PowerShell -ExecutionPolicy Bypass \MPOWER2017-EPO\SCRIPTS$\GOOD\Hello.ps1
   ```

   Expected output:

   ```
   Hello World!
   Good-bye World!
   ```

2. Run the following command from a command prompt running as administrator or equivalent:

   ```powershell
   PowerShell -ExecutionPolicy Bypass \MPOWER2017-EPO\SCRIPTS$\BAD\Hello.ps1
   ```

   Expected output:

   ```
   Access to the path '\mpower2017-epo\scripts$\bad\hello.ps1' is denied.
   + CategoryInfo          : ObjectNotFound: (:String) [], ParentContainsError
   + RecordException       : CommandNotFoundException
   + FullyQualifiedErrorId : CommandNotFoundException
   ```

*Figure 45. Testing the Access Protection rule to restrict PowerShell internal connectivity.*

**Reporting PowerShell Usage by Time**

**Rationale**

When considering smaller groups of endpoints, (those that are always on and always connected such as servers), it may be possible to chart their use of PowerShell by time of day, (e.g. maintenance may always occur
between 2am and 3am, or day of week, (for example application upgrades may only ever occur at the weekend).

If this is possible then another way to restrict the attack surface of these systems is to only allow PowerShell to execute at specific times. Whilst there is no time-based policy available within ePolicy Orchestrator, this can be emulated using the automation capabilities present within ePolicy Orchestrator.

**PowerShell Usage by Time Access Protection Rule**

This can be achieved very easily, by reusing the data from the original Monitor PowerShell Usage rule and displaying it differently within a query.

**Query to Monitor PowerShell Usage over Time**

To report on PowerShell usage over time using ePolicy Orchestrator reporting, create a query as below.

<table>
<thead>
<tr>
<th>Configuration Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a new query.</td>
</tr>
<tr>
<td>2. Result type: Threat Events.</td>
</tr>
<tr>
<td>3. Chart type: Bar Chart.</td>
</tr>
<tr>
<td>5. Bar time units: Hour.</td>
</tr>
<tr>
<td>7. Selected Columns: Event Generated Time, Threat Target Host Name.</td>
</tr>
<tr>
<td>8. Threat Name Equals Monitor PowerShell Usage.</td>
</tr>
</tbody>
</table>

*Figure 46. Create a query to report on PowerShell usage by time.*

**Sample Report of PowerShell Usage over Time**

The figure below shows the output from this query. Note that additional filtering may be required within this report to limit the data shown to a subgroup of all systems, and to a specified period of time.
Restricting PowerShell Usage by Time

Rationale

From the results above it may be determined that PowerShell should only be allowed to launch at specific times. To enforce this an ePolicy Orchestrator workflow must be created.
Workflow Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Example Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.  An Access Protection policy configured to block and report on PowerShell access.</td>
<td>AP – Deny PS.</td>
</tr>
<tr>
<td>b.  An Access Protection policy configured to allow PowerShell access.</td>
<td>AP – Allow PS.</td>
</tr>
<tr>
<td>c.  A tag that will be applied to an endpoint at times when access to PowerShell is to be allowed.</td>
<td>PowerShell Allowed.</td>
</tr>
<tr>
<td>d.  A table-based query designed to find all endpoints that will have their access to PowerShell controlled by time.</td>
<td>Time-based access to PowerShell.</td>
</tr>
<tr>
<td>e.  A policy assignment rule configured to apply the ‘AP – Allow PS’ policy to any endpoints that have the ‘PowerShell Allowed’ tag applied.</td>
<td>Time-based PowerShell access.</td>
</tr>
<tr>
<td>f.  A method of applying the ‘AP – Deny PS’ policy to the machines identified by the query ‘Time-based access to PowerShell’.</td>
<td>Not applicable.</td>
</tr>
</tbody>
</table>

The easiest option here is to place the specific endpoints into one or more system tree groups. If this method is easy to achieve, then unless all endpoints in scope are already in such a group(s), creating a subgroup of their existing group(s), (rather than moving the endpoints to different groups), will ensure than any other product policies assigned to these systems will be inherited down into the subgroup(s).

An alternative to this is to create a policy assignment rule configured to apply the ‘AP – Deny PS’ policy to these endpoints. If this approach is taken, this policy assignment rule must be placed lower than the ‘Time-based PowerShell access’ policy assignment rule, so that it is only enforced if the ‘Time-based PowerShell access’ policy assignment rule has not been applied, (i.e. due to the ‘PowerShell Allowed’ tag not being present).

| g.  A server task configured as below: Actions: Run Query: Time-based access to PowerShell Sub-action 1: Apply Tag: PowerShell Allowed Sub-action 2: Wake Up Agents: Agent Wake-up call; Randomization 0 Schedule: To run a short time prior to when PowerShell access is required. | PowerShell – enable. |
| h.  A server task configured as below: Actions: Run Query: Time-based access to PowerShell Sub-action 1: Clear Tag: PowerShell Allowed Sub-action 2: Wake Up Agents: Agent Wake-up call; Randomization 0 Schedule: To run at a time sufficiently after the above server task has completed, to allow all PowerShell tasks to complete, and so only run when access to PowerShell is no longer required. | PowerShell – disable. |

Figure 48. Components required to restricting PowerShell usage by time.
Workflow Process
The workflow process is described below.

<table>
<thead>
<tr>
<th>PowerShell Access State</th>
<th>Time</th>
<th>Description of situation or activity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denied</td>
<td>Any time prior to access being required, e.g. 1am.</td>
<td>The AP – Deny PS policy is being enforced on the endpoints by whatever method has been selected, (see component f above).</td>
</tr>
<tr>
<td>Denied</td>
<td>A short time prior to access being required, e.g. 1:50am.</td>
<td>The PowerShell – enable server task scheduled run time is met, and the tasks starts to run, (g above).</td>
</tr>
<tr>
<td>Denied</td>
<td>Immediately following the previous activity, e.g. 1:50am.</td>
<td>The task runs the query Time-based access to PowerShell, and the results from this are actioned by firstly applying the PowerShell Allowed tag, and secondly sending a wake-up call to those endpoints.</td>
</tr>
<tr>
<td>Denied</td>
<td>A short time after the previous activity has completed, e.g. 1:52am.</td>
<td>The endpoints will talk in to the ePolicy Orchestrator or Agent Handler, and check for policy changes. They will recognize the PowerShell Allowed tag has been applied, and because of the policy assignment rule Time-based PowerShell access, retrieve the AP – Allow PS policy.</td>
</tr>
<tr>
<td>Allowed</td>
<td>Typically, a minute or so after the endpoint has retrieved the new policy, 1:53am.</td>
<td>The endpoints will apply the AP – Allow PS policy.</td>
</tr>
<tr>
<td>Allowed</td>
<td>The period in which PowerShell access is required, e.g. 2am to 4am.</td>
<td>The endpoints use PowerShell to make any configuration changes.</td>
</tr>
<tr>
<td>Allowed</td>
<td>At a time after which PowerShell activity is no longer required on endpoints, e.g. 4:05am.</td>
<td>The PowerShell – disable server task scheduled run time is met, and the tasks starts to run, (h above).</td>
</tr>
<tr>
<td>Allowed</td>
<td>Immediately following the previous activity, 4:05am.</td>
<td>The task runs the query Time-based access to PowerShell, and the results from this are actioned by firstly clearing the PowerShell Allowed tag, and secondly sending a wake-up call to those endpoints.</td>
</tr>
<tr>
<td>Allowed</td>
<td>A short time after the previous activity has completed, e.g. 4:08.</td>
<td>The endpoints will talk in to the ePolicy Orchestrator or Agent Handler, and check for policy changes. They will recognize the PowerShell Allowed tag has been removed, and the policy assignment rule Time-based PowerShell access is no longer applied, and so will revert to their default policy of AP – Deny PS.</td>
</tr>
<tr>
<td>Denied</td>
<td>Typically, a minute or so after the endpoint has retrieved the new policy, e.g. 4:10am.</td>
<td>The endpoints will apply the AP – Deny PS policy.</td>
</tr>
</tbody>
</table>

Figure 49. The workflow process illustrated.
Testing the workflow

The workflow should be tested methodically, to ensure each operation works as expected on the target systems, and does not impact systems that are out of scope.

<table>
<thead>
<tr>
<th>Test Step</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Run the PowerShell - enable Server Task and ensure the PowerShell Allowed tag is applied.</td>
</tr>
<tr>
<td>2</td>
<td>Run the PowerShell - disable Server Task and ensure the PowerShell Allowed tag is removed.</td>
</tr>
<tr>
<td>3</td>
<td>Check the AP – Allow PS policy is applied once the tag has been applied. Perform a wake-up call and confirm the endpoint retrieves and applies a new policy. Confirm the endpoint can run a basic PowerShell script.</td>
</tr>
<tr>
<td>4</td>
<td>Check the AP – Deny PS policy is applied once the tag is removed. Perform a wake-up call and confirm the endpoint retrieves and applies a new policy. Confirm the endpoint cannot run a basic PowerShell script.</td>
</tr>
<tr>
<td>5</td>
<td>Schedule the PowerShell - enable task to run at a point when PowerShell access is required. Confirm the PowerShell Allowed tag is applied and the AP – Allow PS policy is applied. Confirm the endpoint can run a basic PowerShell script. Determine how long this process takes to complete.</td>
</tr>
<tr>
<td>6</td>
<td>Schedule the PowerShell - disable task to run at a point when PowerShell access is not required. Confirm the PowerShell Allowed tag is removed and the AP – Deny PS policy is applied. Confirm the endpoint cannot run a basic PowerShell script. Determine how long this process takes to complete.</td>
</tr>
<tr>
<td>7</td>
<td>Apply this to a pilot server and confirm the complete workflow operates correctly.</td>
</tr>
</tbody>
</table>

Figure 50. Testing the workflow process components.

Once success is achieved, this policy can be applied to production servers. Ensure that the workflow is monitored initially to ensure it works in production as expected.

Monitoring of events that should not occur, (for example, a PowerShell block event occurring at a time when the access should be enabled), can be extended by using an ePolicy Orchestrator automatic response. This configuration is beyond the scope of this document.

Conclusions

Whilst PowerShell-based malware is a new and innovative method of delivering threats to endpoints, multiple different traditional security capabilities can be leveraged to detect and protect against these types of threats.

Detection of PowerShell-based malware can be challenging using traditional anti-virus technology, but a combination of access protection, firewall and exploit detection techniques can effectively and efficiently secure an endpoint from this new threat vector.

The author hopes this paper has given administrators several options to help limit any impact of PowerShell-based malware, and would be happy to receive any feedback, suggestions, corrections or recommendations: Jason_Brown@McAfee.com.
About the Author

Jason Brown is an Enterprise Technology Specialist focusing in Endpoint and working within Europe, the Middle East and Africa, (EMEA). He is responsible for representing the McAfee Endpoint Business Unit strategy to customers, working on the largest and most strategic deals with the region, and empowering Sales Engineering & Channel Partners to utilize McAfee’s endpoint technology in the most effective way. He is currently responsible for driving adoption of McAfee's Dynamic Endpoint solution across the region. He has authored many white papers and best practice guides, and presented at McAfee’s prestigious Focus security event in Las Vegas. He has created and executed partner and customer training, delivered professional services and assisted pre-sales activity across EMEA, having worked within the security industry for the last 20 years. Prior to joining McAfee, Jason worked at a defense contractor after gaining a Master’s degrees in Operational Research and Finance. He is a Certified Information Systems Security Professional (CISSP).

About McAfee

McAfee is one of the world's leading independent cybersecurity companies. Inspired by the power of working together, McAfee creates business and consumer solutions that make the world a safer place. For businesses, McAfee helps orchestrate holistic cybersecurity environments that work smarter, not harder. For consumers, McAfee helps by securing their digital lifestyle at home and away.

www.mcafee.com

---

i https://en.wikipedia.org/wiki/PowerShell
iv https://github.com/danielbohannon/Invoke-Obfuscation