



National Security Agency/Central Security Service



Information
Assurance
by NSA

Defending Against the Malicious Use of Admin Tools: PowerShell™

Contents

1	Background	1
2	Methodology.....	2
3	Guidance	2
3.1	Upgrade to Windows 10	2
3.2	Configure Execution Policy.....	3
3.3	Enforce Script Signing	4
3.4	Protect Execution Policy Settings.....	5
3.5	Deploy Application Whitelisting	6
3.6	Enable Constrained Language Mode (CLM).....	6
3.7	Enable Constrained Language Mode Integration with AppLocker®	7
3.8	Utilize the Antimalware Scan Interface (AMSI)	7
3.9	Restrict Alternate Execution Paths	7
3.10	Use Constrained Endpoints for Remote PowerShell	8
3.11	Leverage Analytics	8
4	Resources	9

Disclaimer

This Work is provided "as is." Any express or implied warranties, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the United States Government be liable for any direct, indirect, incidental, special, exemplary or consequential damages (including, but not limited to, procurement of substitute goods or services, loss of use, data or profits, or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this Work, even if advised of the possibility of such damage.

The User of this Work agrees to hold harmless and indemnify the United States Government, its agents and employees from every claim or liability (whether in tort or in contract), including attorneys' fees, court costs, and expenses, arising in direct consequence of Recipient's use of the item, including, but not limited to, claims or liabilities made for injury to or death of personnel of User or third parties, damage to or destruction of property of User or third parties, and infringement or other violations of intellectual property or technical data rights.

Nothing in this Work is intended to constitute an endorsement, explicit or implied, by the U.S. Government of any particular manufacturer's product or service.

Trademark Information

This publication has not been authorized, sponsored, or otherwise approved by Microsoft.

Microsoft®, Windows®, and Active Directory® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

1 Background

Malicious actors are using our own tools against us. Why reinvent the wheel or drop something new, something distinguishable, when the tools used on every network every day will provide you all you need? Especially when many of those tools are built into every endpoint with minimal hardening or monitoring. Why not hide in the noise of legitimate administrator traffic? These aren't just rhetorical questions but are real strategies that have already been adopted by attackers around the world. What attackers have realized is that tools such as PowerShell™¹, Remote Desktop Protocol (RDP), PsExec, Windows®² Management Instrumentation (WMI), and Secure Shell (SSH) coupled with elevated credentials offer a full cadre of capabilities that allow for persistence, enumeration, exfiltration, system modification, security evasion, access to the kernel and memory as well as other critical resources.[1] These tools are at times rarely monitored and typical network traffic pertaining to administrative activities (local and remote) are identified as legitimate "administrator (admin) traffic." Unfortunately, differentiating between legitimate and malicious use of administrative tools becomes difficult as both usages may appear to be very similar.

This paper provides a strategy for hardening, defending, and detecting anomalous, and malicious, use of administrator toolsets. In particular, this paper will focus on Microsoft's®³ PowerShell™ and will provide a methodology for hardening and defending it from adversarial use.

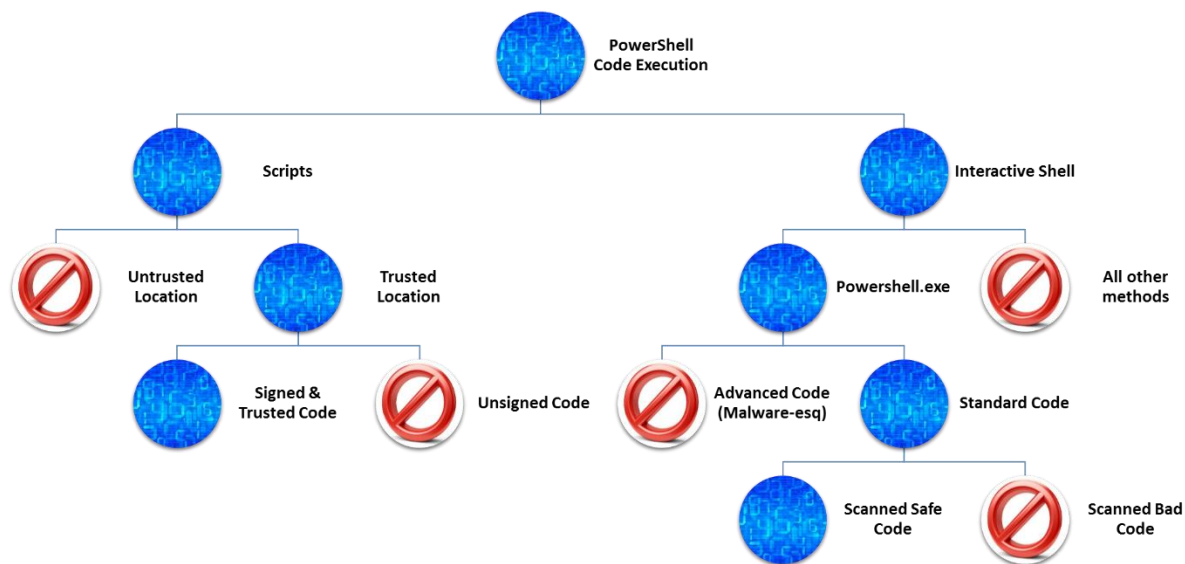
¹ PowerShell is a trademark of Microsoft Corp.

² Windows is a registered trademark of Microsoft Corp.

³ Microsoft is a registered trademark of Microsoft Corp.

2 Methodology

The intent behind this PowerShell™ research is to restrict unfettered access to the full complement of PowerShell™ capabilities where possible, and to force the usage of PowerShell™ commands into predictable lanes for execution while maintaining an appropriate level of availability to enterprises and applications. The expectation of limiting avenues of execution for adversaries utilizing PowerShell™ would be to expose them to monitoring, detection, and potential prevention. The following image is a generalized representation of this methodology:



Emphasis was placed on leveraging the security mechanisms built into PowerShell™ to achieve the desired result. These mechanisms with their accompanying mitigations, detection, and monitoring strategies are described below. Keep in mind that there are no silver bullets and that all recommendations made hereafter are intended to impede and deter attackers from maliciously using PowerShell™ post exploitation. These recommendations will be amended as needed to address new preventive and detection capabilities, improved capabilities, or removal of these capabilities.

3 Guidance

3.1 Upgrade to Windows 10

NSA recommends that organizations upgrade to the latest version of Windows® (currently Windows® 10 Anniversary Edition) [2]. Only by upgrading to Windows 10 can users take advantage of new security features, including those in PowerShell 5. This includes Deep Script Block Logging, transcription of interactive sessions, the Anti-Malware Scan Interface (AMSI), and automated Constrained Language Mode (CLM) where AppLocker® is deployed. See *PowerShell:*

Security Risks and Defenses [3] for an overview of PowerShell security features introduced with each version.

Upgrading systems to Windows 10 additionally offers further mitigations designed to impede an adversary's capability to infect, maneuver, and establish persistence on the endpoint. Features such as Credential Guard, Device Guard, Application Guard offer a more robust capability to defend the enterprise against malicious actors.

3.2 Configure Execution Policy

The Execution Policy is a PowerShell™ setting that governs the execution of PowerShell™ scripts and loading of configuration files.⁵ By default this setting is configured to “Unrestricted” which allows for open execution of all PowerShell™ scripting. PowerShell™ can be configured to restrict script execution such that they are not allowed to run or may be allowed if signed by a trusted authority.

This feature provided by Microsoft® was not introduced as a robust security feature, but as a mechanism to allow admins to set basic rules and prevent users from violating them unintentionally (“safety-belt” feature).⁴ Microsoft's® intent with this capability was not to protect against adversarial use of the product but rather to prevent admins, or users, from accidentally launching a script at an inopportune time. While this is not a true mitigation or security boundary, setting the execution policy can be used as another layer of protection that drives up the cost to adversaries who would otherwise be able to download scripts and run them at will. This feature does **not** prevent attempts to execute code from a script interactively (i.e., given to an interactive PowerShell™ session line-by-line, read and piped in from another source, or copied). Coupling this setting configuration with a mechanism to ensure this configuration is not modified will introduce a more formidable mitigation.

The Execution Policy has four possible settings:

1. Unrestricted – allows the free execution of all PowerShell™ scripts
2. Restricted – no scripts can be run, PowerShell™ can only be run through interactive mode
3. RemoteSigned – any script created outside of the enterprise must be digitally signed in order to run
4. AllSigned – all scripts must be digitally signed in order to run [4]

Where conditions permit, organizations should consider setting the ExecutionPolicy to “AllSigned” via Group Policy (GP). Where this strategy is not feasible it is recommended that this same setting be configured to “RemoteSigned”. In order to do so, the enterprise will need a protected Certificate Authority (CA) in place with the proper code signing certs imported into the certificate trust stores [5] of the endpoints. This recommendation assumes that the

⁴ PowerShell Documentation “about_Execution_Policies” – <https://technet.microsoft.com/en-us/library/hh847748.aspx>

organization is following security best practices for CA infrastructure and certificate management. Making this setting via GP sets the policy for the machine which overrides modifications at the user or instance level.

Group Policy: “**Computer Configuration | Policies | Administrative Templates | Windows Components | Windows PowerShell™**”

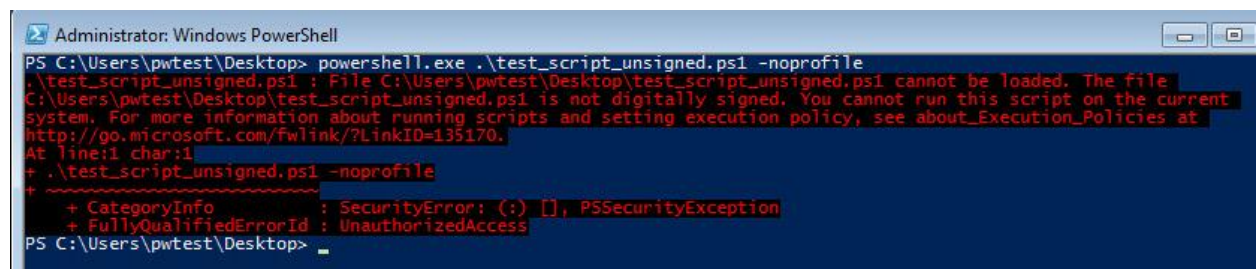
3.3 Enforce Script Signing

Once the ExecutionPolicy is set and the infrastructure is in place, signing a script is a single line command from PowerShell™ using the **Set-AuthenticodeSignature** command:

Set-AuthenticodeSignature <path to script> <path to cert> -TimestampServer <path to server>

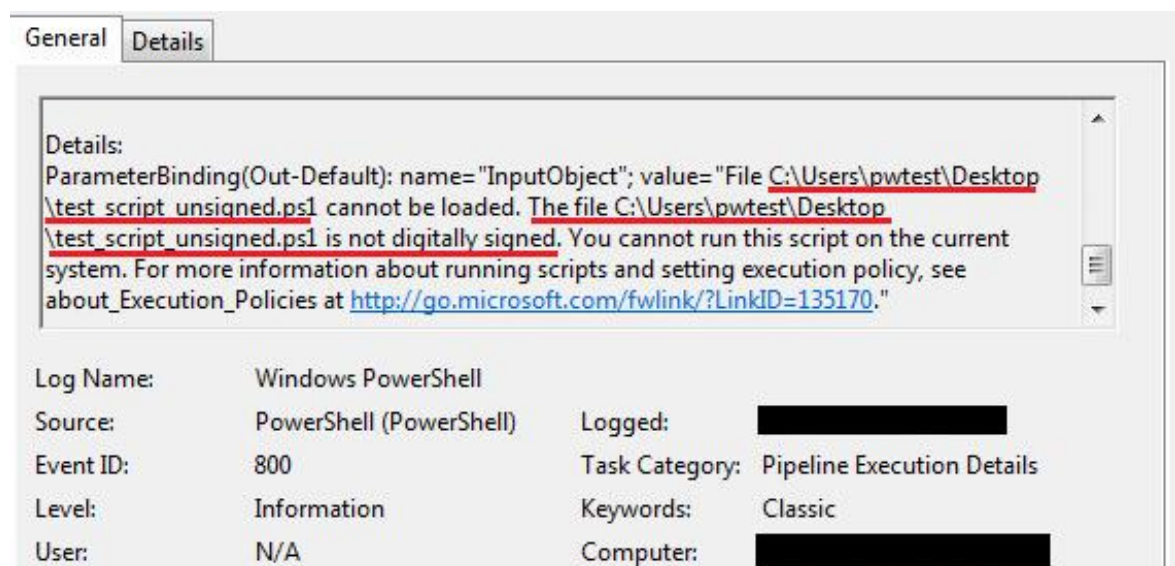
Utilizing the aforementioned guidelines, execution attempts for the following circumstances will yield the pictured results:

User tries to execute an unsigned script:



```
Administrator: Windows PowerShell
PS C:\Users\pwtest\Desktop> powershell.exe .\test_script_unsigned.ps1 -nopprofile
.\test_script_unsigned.ps1 : File C:\Users\pwtest\Desktop\test_script_unsigned.ps1 cannot be loaded. The file
C:\Users\pwtest\Desktop\test_script_unsigned.ps1 is not digitally signed. You cannot run this script on the current
system. For more information about running scripts and setting execution policy, see about_Execution_Policies at
http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\test_script_unsigned.ps1 -nopprofile
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\pwtest\Desktop>
```

Corresponding log entry (PowerShell™ V3 or above):



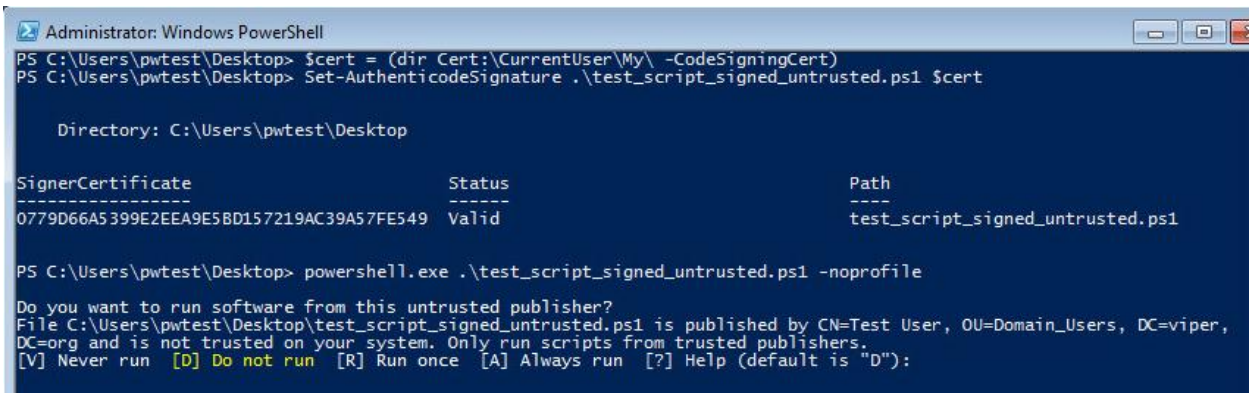
General Details

Details:

ParameterBinding(Out-Default): name="InputObject"; value="File C:\Users\pwtest\Desktop\test_script_unsigned.ps1 cannot be loaded. The file C:\Users\pwtest\Desktop\test_script_unsigned.ps1 is not digitally signed. You cannot run this script on the current system. For more information about running scripts and setting execution policy, see about_Execution_Policies at <http://go.microsoft.com/fwlink/?LinkID=135170>."

Log Name:	Windows PowerShell		
Source:	PowerShell (PowerShell)	Logged:	
Event ID:	800	Task Category:	Pipeline Execution Details
Level:	Information	Keywords:	Classic
User:	N/A	Computer:	

User tries to execute a script that is signed by an enterprise resource but is not trusted:



```
Administrator: Windows PowerShell
PS C:\Users\pwtest\Desktop> $cert = (dir Cert:\CurrentUser\My\ -CodeSigningCert)
PS C:\Users\pwtest\Desktop> Set-AuthenticodeSignature .\test_script_signed_untrusted.ps1 $cert

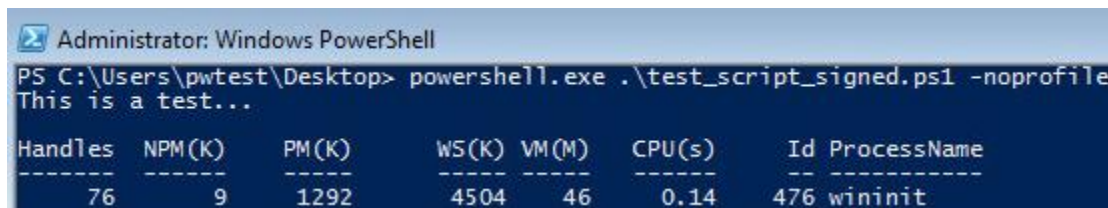
Directory: C:\Users\pwtest\Desktop

SignerCertificate          Status          Path
-----
0779D66A5399E2EEA9E5BD157219AC39A57FE549 Valid          test_script_signed_untrusted.ps1

PS C:\Users\pwtest\Desktop> powershell.exe .\test_script_signed_untrusted.ps1 -noprofile

Do you want to run software from this untrusted publisher?
File C:\Users\pwtest\Desktop\test_script_signed_untrusted.ps1 is published by CN=Test User, OU=Domain_Users, DC=viper, DC=org and is not trusted on your system. Only run scripts from trusted publishers.
[V] Never run [D] Do not run [R] Run once [A] Always run [?] Help (default is "D"):
```

User tries to execute a script that is enterprise signed and trusted:



```
Administrator: Windows PowerShell
PS C:\Users\pwtest\Desktop> powershell.exe .\test_script_signed.ps1 -noprofile
This is a test...

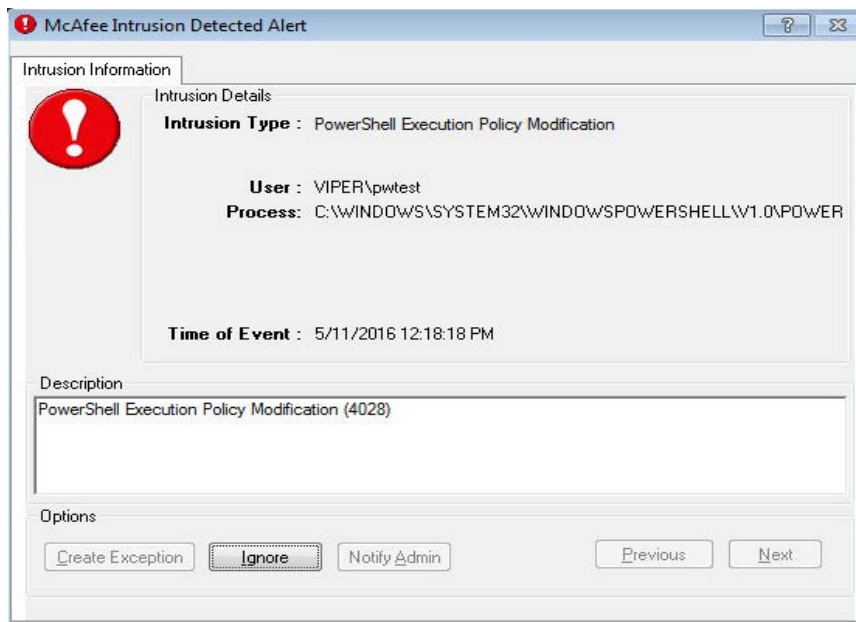
Handles  NPM(K)  PM(K)  WS(K)  VM(M)  CPU(s)  Id ProcessName
-----
76      9      1292   4504   46     0.14    476 wininit
```

3.4 Protect Execution Policy Settings

ExecutionPolicy is backed by the Windows® Registry and configuration settings are stored there. Accounts with administrative privileges can modify registry keys for ExecutionPolicy. Multiple registry locations dictate the script execution capabilities at various scope levels. Listed below are the registry keys with their corresponding scope:

- **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\Shell1ds\ScriptedDiagnostics\ExecutionPolicy** (scope = LocalMachine for Microsoft “Troubleshoot” scripts) [6]
- **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\Shell1ds\Microsoft.PowerShell\ExecutionPolicy** (scope = LocalMachine)
- **HKEY_Current_User\SOFTWARE\Microsoft\PowerShell\1\Shell1ds\Microsoft.PowerShell\ExecutionPolicy** (scope = CurrentUser)
- **HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ExecutionPolicy** (scope = MachinePolicy i.e. Group Policy setting)

In order to effectively restrict adversaries from modifying this policy, it is imperative that these registry keys are protected from modification. Where possible use a protection mechanism that can restrict even admin level modifications. In our testing we utilized the Intel® Security (McAfee®6) HIPS product. [7]



3.5 Deploy Application Whitelisting

NSA recommends that enterprises deploy Application Whitelisting (AWL) to their environments and ensure that their chosen tool can monitor PowerShell™ scripting. [2][8] By default Microsoft® AppLocker® monitors different native scripting languages (e.g. PowerShell/.ps1), other products vary in their support of script monitoring and may require manually adding the extension into the monitor base. [9]

PowerShell™ (.ps1) files should be treated like any other executable in the enterprise's AWL approach. At a minimum, .ps1 files should be restricted in their ability to execute from common malware locations (e.g. "C:\windows\temp\", "C:\user\appdata\", download folders, etc.). More appropriately, a strategy and common location should be defined by the enterprise for script execution. All approved locations should also be access controlled to ensure that only approved admins can populate these locations with executable material.

3.6 Enable Constrained Language Mode (CLM)

CLM allows Windows® command-lets (cmdlets) and PowerShell™ commands but restricts the usage of advanced features common in most PowerShell™ malware. Specifically, CLM restricts object creation as well as usage of the .Net framework. CLM also restricts loading arbitrary C# code.

CLM functionality is available in PowerShell™ instances V3.0 and above. [10] Enabling this feature is currently only available by setting an environment variable or by using Windows® 10 in conjunction with AppLocker®.

Enabling CLM via the environment variable is not an officially supported method by Microsoft® and is subject to change.

CLM could be enabled across the enterprise via Group Policy after identifying and potentially updating existing mission critical PowerShell™ scripts relying on extended language functions like interaction with COM objects and invocation of Win32 APIs. Failure to evaluate and potentially update all mission critical PowerShell™ scripts may result in operational impact.

Enabling CLM across the enterprise adds a registry entry on the local machine that enforces the setting at startup of each session. It is further recommended that the corresponding registry entry be monitored by an enterprise protection mechanism.

Ensuring least privilege guidance and other Domain Administrator level mitigations are in place for Users running PowerShell™ in Full Language Mode is a critical part of this strategy.

Environment Variable Setting:

"[ENVIRONMENT]::SetEnvironmentVariable('__PSLockdownPolicy'; '4'; 'Machine')

Group Policy: **"Computer Configuration | Preferences | Windows Settings | Environment"**

3.7 Enable Constrained Language Mode Integration with AppLocker®

Beginning with Windows® 10, CLM is enforced automatically once AppLocker® is enabled and in **Enforced** mode. This feature only pertains to enterprises using AppLocker® as their AWL enforcement product. When AppLocker® is configured to run in any other mode (i.e. Audit or not configured) besides Enforced, the interactive PowerShell™ session operates in **FullLanguage** mode (i.e. allow full usage of the PowerShell™ language) otherwise PowerShell™ sessions will operate in CLM. Constrained Language Mode integration with AppLocker® enhances security by enforcing where and if PowerShell™ scripts can run as well as restricting exposed PowerShell™ capabilities. [11] This feature can only be used when AppLocker® and AppLocker's® Script policy is properly configured.

3.8 Utilize the Antimalware Scan Interface (AMSI)

The Antimalware Scan Interface was introduced in Windows 10. This new capability allows the installed and registered anti-virus software to be leveraged to scan PowerShell™ commands prior to being executed by the PowerShell™ interpreter. Because it gets de-obfuscated and then sent to the AMSI, even PowerShell™ code that has been packed, encoded, or encrypted can now be scanned by the system's virus scanner before it is executed.

AMSI is a new feature that requires AV vendor support. While only a small set of security vendors currently support the AMSI functionality (e.g. Windows® Defender and AVG), broader adoption and support is expected.

3.9 Restrict Alternate Execution Paths

Part of the power of PowerShell™ resides in the integration of the capability into the operating system. Looking forward, Microsoft® is clearly demonstrating a desire to only make that

integration more comprehensive. While this is great from a usability perspective it also presents challenges to organizations who seek to restrict PowerShell™ usage within their environments from adversarial use.

As an interpreted language, PowerShell™ commands are sent to the interpreter for execution. The interpreter is separate from powershell.exe, thus restricting powershell.exe only inhibits one method for executing PowerShell™ code. Powershell.exe is simply a front end for the interpreter. Other front ends include powershell_ise.exe and PowerShell™ code integrated into C#. Executing PowerShell™ code via either of these mechanisms would allow the code to run even when powershell.exe is blocked altogether.

At least one commonality with all front ends to the interpreter is their reliance on System.Management.Automation.dll to create a runspace for the code. Specifically, any code could call the System.Management.Automation.Runspaces.RunspaceFactory.CreateRunspace() method to spawn a new instance of the runspace to interact with the interpreter. With this in mind, a reasonable methodology for restricting PowerShell™ code to run only within powershell.exe, where it can be better monitored, would be to utilize a capability such as HIPS or AWL to monitor the System.Management.Automation.dll for any reads and then allow only powershell.exe to execute that action. Alternatively, Microsoft's® Sysmon and AWL DLL enforcement could be utilized to log when this DLL is loaded into a process. This approach has proven to be feasible for standard workstations and most servers. This approach may not be feasible on development workstations or servers that rely heavily on PowerShell™ for core functionality (e.g. Exchange).

3.10 Use Constrained Endpoints for Remote PowerShell

Constrained Endpoints (“endpoints”) is a capability available within PowerShell™ that permits configuration of remote PowerShell™ sessions. Endpoints provide an ability to control how administrators and users perform specific PowerShell™ commands and operate within a specified PowerShell™ Language Mode for their respective role function. A proper evaluation of a role and identification of PowerShell™ components are required to enable proper, secure, and functional use of Constrained Endpoints. Beginning with PowerShell™ v5, endpoints can be configured to utilize Transcriptions to record all activities within a remote PowerShell™ session. When remote PowerShell™ is used in environments with PowerShell™ version 5 on clients and servers, client-side and server-side PowerShell™ activities can be both logged. This feature is encouraged for deployment in environments where remote PowerShell™ activities is used.

3.11 Leverage Analytics

A passive analytic approach is appropriate as part of the layered defense strategy to detect abnormal activity when PowerShell™ is leveraged as an attack platform. During the attack timeline, the analytic will help identify the adversary's activities by identifying abnormal or suspicious use of PowerShell™ to achieve their objectives.

A key component to passive analysis is properly characterizing behavioral aspects of network activities, and establishing a Pattern-of-Life (POL), for legitimate and abnormal components, services, commands, or configurations. Continuous monitoring of network assets or connections is key to understanding when and how PowerShell™ activities are performing normal or suspect malicious routines. The analytic will first identify multiple system configurations and then classify broad network topologies to support real-time network monitoring. The analytic will then assist in baselining PowerShell™ activities occurring in healthy networks for identification of abnormal behavior.

Over time, using PowerShell™ analytics, analysts may develop patterns or predictive indicators, which might allow for mitigation before execution.

4 Resources

- [1] Investigating PowerShell Attacks – Black Hat USA 2014
<https://www.blackhat.com/docs/us-14/materials/us-14-Kazanciyan-Investigating-Powershell-Attacks-WP.pdf>
- [2] IAD's Top 10 Information Assurance Mitigation Strategies –
<https://www.iad.gov/iad/library/ia-guidance/iads-top-10-information-assurance-mitigation-strategies.cfm>
- [3] PowerShell: Security Risks and Defenses – <https://www.iad.gov>
- [4] Microsoft® TechNet (Using the Set-ExecutionPolicy Cmdlet) -
<https://technet.microsoft.com/en-us/library/ee176961.aspx>
- [5] Microsoft® TechNet (Deploy Certificates by Using Group Policy) –
[https://technet.microsoft.com/en-us/library/cc770315\(v=ws10\).aspx](https://technet.microsoft.com/en-us/library/cc770315(v=ws10).aspx)
- [6] Microsoft® Developer (PowerShell™ Execution Policy and Windows® 7) –
<https://blogs.msdn.microsoft.com/vivekkum/2009/02/04/powershell-executionpolicy-and-win7>
- [7] Intel® Security (McAfee®) HIPS custom rule (Note tuning would be required to allow GP to update the setting):
Rule {
Tag "Registry Protection"
Class Registry
ID 4xxx
Level 4
values { Include
"\\REGISTRY\\MACHINE\\Software\\Microsoft\\Powershell\\1\\ShellIds\\ScriptedDiagnostics\\ExecutionPolicy"
"\\REGISTRY\\MACHINE\\Software\\Microsoft\\Powershell\\1\\ShellIds\\Microsoft.Powershell\\ExecutionPolicy"
"\\REGISTRY\\CURRENT_USER\\Software\\Microsoft\\Powershell\\1\\ShellIds\\Microsoft.Powershell\\ExecutionPolicy"
"\\REGISTRY\\MACHINE\\Software\\Policies\\Microsoft\\Windows\\Powershell\\ExecutionPolicy"
}
Directives registry:modify registry:permissions registry:create registry:delete
}

- [8] Application Whitelisting – www.iad.gov/iad/library/ia-guidance/security-tips/application-whitelisting.cfm
- [9] Microsoft® TechNet (Use AppLocker and Software Restriction Policies in the same domain) – [https://technet.microsoft.com/en-us/library/hh994614\(v=ws.11\).aspx](https://technet.microsoft.com/en-us/library/hh994614(v=ws.11).aspx)
- [10] Microsoft® TechNet (About Language Modes) – <https://technet.microsoft.com/en-us/library/dn433292.aspx>
- [11] AdSecurity.org (Detecting Offensive PowerShell™ Attack Tools) - <https://adsecurity.org/?p=2604>