

Managing PowerShell in a modern corporate environment

Prepared by:
Dean Hardcastle, Senior Security Consultant

Table of contents

1. Introduction	3
2. Current defence methods	5
3. Modern countermeasures	6
4. Conclusion	14
5. References	16

1. Introduction

Since its incarnation in 2006, PowerShell has grown to be a powerful and extensible management tool, allowing for large-scale automation of system administration and maintenance tasks with ease.

However, it has also gained notoriety as the preferred post-exploitation tool for attackers. This is largely as a result of having a relatively low barrier to entry due to the ease in which PowerShell scripts can be written and the extensive Microsoft documentation that is readily available. It is also included by default on all Windows hosts since Vista.

The use of PowerShell for nefarious purposes has grown in popularity. It is, however, often incorrectly assumed that PowerShell is a weakness which makes the exploitation of Windows hosts possible. While some published exploits have been ported to PowerShell, the tool itself is merely a management framework. To date it has very few reported vulnerabilities in its actual implementation and is also primarily used as a secondary means of attack after an initial breach.

Analysis by Carbon Black of several breach reports identified that nearly 38 per cent of recent cyberattacks made use of PowerShell in some form [1]. In addition, a large number of affected organisations did not detect the use of PowerShell until it was identified during post-incident response activities. The data gathered by Carbon Black also revealed that social engineering was the favoured delivery technique for malicious PowerShell payloads.

“More than 85 per cent of the attacks leveraging PowerShell were ... commodity attacks such as clickfraud, ransomware, fake antivirus and other opportunistic threats. Many of these attacks appeared focused on stealing customer and financial data, and intellectual property, or on disrupting services.”[2]

The prevalence of PowerShell-based post-exploitation frameworks has likely increased public awareness, but also significantly reduced the effort required to perform more widespread attacks following successful exploitation. Such frameworks include Nishang, PowerUp, PowerSploit and Empire. Additionally, a number of tools offer similar functionality to PowerShell without relying on access to the underlying PowerShell binaries (powershell.exe and powershell_ise.exe). These include p0wnedShell, nps and PowerPick.

The use of such frameworks makes detection significantly more difficult than typical malware. This is primarily due to the so-called “fileless” nature, in which scripts or libraries may be downloaded and loaded into memory without writing to disk. This renders detection by common Host-based Intrusion Detection System (HIDS) and signature-based anti-virus (AV) solutions ineffective.

Kaspersky Labs reported that several banking and financial institutions had been infected in this manner, in which attackers leveraged PowerShell to inject a malicious payload (that would otherwise be detected by AV signatures) directly into memory [3]. Several other examples in which malware had been delivered using PowerShell were also identified by Trend Micro [4]. These included Fareit, VawTrak and PowerWare, all of which rely on social engineering as a means of initial exploitation. In these instances, attackers were able to execute PowerShell via embedded Office macros or through the use of malicious JavaScript within crafted PDF documents.

The tightly-coupled relationship between PowerShell and the underlying operating system (OS) makes administration tasks significantly easier and more efficient. In addition, access to the underlying .Net libraries allows access to functionality which would otherwise be inaccessible to other scripting languages; allowing for execution of C# within PowerShell scripts and vice versa. Conversely, this relationship makes it nearly impossible to disable support for the execution of PowerShell scripts without negatively affecting the day-to-day management of Windows hosts. This therefore provides a reliable and extensive framework which may be leveraged by attackers.

In some situations, such as when managing Core and Nano installations of Windows Server OS, the only effective means of access are through PowerShell remoting. This presents a unique situation in which the tools leveraged by an attacker are the only means of performing legitimate system administration. Additionally, recent Windows components such as Microsoft Exchange, Hyper-V and SQL Server are now primarily managed through PowerShell. At the same time Azure PowerShell provides a set of cmdlets that use the Azure Resource Manager model for managing Azure (cloud) resources [5]. This places greater reliance on the ability to detect and react to the misuse of PowerShell rather than simply preventing its execution.

2. Current defence methods

A common approach to preventing the exploitation of PowerShell is through the use of the built-in execution policy. This can be enforced via group policy. Using this feature allows script execution to be limited to:

- Signed scripts (AllSigned)
- Signed for execution from a remote location (RemoteSigned)
- Allowing execution of all scripts (Unrestricted)
- Disallowing execution of all scripts (Restricted)

This approach aims to prevent users from executing untrusted PowerShell scripts, such as those downloaded from public repositories or those accessible from network shares. However, in reality, this method offers little to no protection beyond accidental script execution. This is due to the presence of a native argument (-executionPolicy bypass) which can be used to temporarily “ignore” the configured policy state.

However, the execution policy is not a security boundary and should not be relied upon as a sole means of defence. This has also been stated by Microsoft themselves [6] and in addition to this there are several publications that detail numerous ways in which the execution policy may be bypassed with relative ease [7].

Alternative approaches include the complete removal or restriction of the PowerShell console and ISE binaries through application restriction policies. However, this process often overlooks the presence of both x86 and x64 binaries on 64-bit platforms.

While this may prevent a casual attacker, this protection may also be circumvented through a number of means. These include direct access to, or reflective injection of the System.Management.Automation assembly, inclusion of PowerShell within compiled C and C# binaries or leveraging weaknesses in default application restriction policies.

A number of the frameworks described in this document can automate this task, allowing a user with little to no technical expertise to circumvent protections that rely solely on the execution policy or by preventing access to the PowerShell binaries.

3. Modern countermeasures

More recent versions of PowerShell and the Windows OS have introduced a vast array of new features that aid potential breach detection and post-incident analysis. These features, if correctly configured, could drastically reduce the time required to identify malicious activity and allow incident response efforts to be more accurately targeted.

Furthermore, a number of the features offered by these revisions may drastically reduce the potential for abuse of PowerShell. This is done by restricting access to the functionality it exposes to users.

Below a number of these features are outlined along with their advantages as well as any potential pitfalls and shortcomings.

3.1 Constrained Language Mode (CLM)

CLM provides a means to disable access to a number of underlying PowerShell functions, particularly those referenced by potentially malicious tools. This is done by preventing access to unsigned assemblies and a number of native Windows components.

CLM forms a small part of a larger “language mode” feature that was initially introduced in PowerShell version 3. The language mode feature allows varying degrees of control in terms of the features which may be utilised by users. For example, using a language mode of “restricted” prevents the use of script blocks but allows execution of any function. Comparatively, use of the “constrained” language directive further limits access to only native Windows cmdlets, disabling access to direct .Net scripting, use of the Add-Type cmdlet to access Win32 APIs and interaction with COM objects.

On the surface, enabling this feature seems like an ideal method to reduce the potential for nefarious PowerShell usage. However, CLM can only be applied system-wide and therefore affects users and administrators alike, including remote sessions via PowerShell Remoting and WinRM. Such configuration may negatively impact the use of legitimate tools for system automation or administration and as a result, it should be used with caution. Thorough testing of this feature is recommended before large-scale deployment to any production environments.

3.2 Anti-Malware Scripting Interface (AMSI)

The AMSI aims to provide a layer between AV solutions and the script runtime engine used by automation languages such as VBS, JScript and PowerShell. AMSI-aware AV products are able to detect and prevent attempts to access and execute malicious content. Examples of this functionality include preventing the download and subsequent execution of common password recovery tools such as Mimikatz, Meterpreter and Empire payloads and binaries for which signatures exist.

AMSI, like many other AV solutions, relies on signature-based detection and as such can be bypassed through use of obfuscation or signature masking. Additionally, freely available tools such as p0wnedShell [8] provide simple methods to bypass AMSI by loading a custom DLL at runtime, causing AMSI to be immediately unloaded.

Although AV is useful as a protective measure, it is not a perfect solution with the existence of several bypass techniques. AMSI is no exception, with one researcher publishing a bypass method which could fit within a single tweet (fewer than 140 characters) [9].

In addition to this method, a number of native commands exist which allow the disabling of AMSI for the current process. However, usage of this feature is subject to ScriptBlock logging. Also if PowerShell version 2 is enabled, it is possible to circumvent AMSI-aware AV products through simply executing commands within a PowerShell version 2 console. In doing so, an attacker would be able to download malicious content (that would ordinarily be blocked by AV) and execute it without detection. Such activities would, however, be recorded if transcripts or ScriptBlock logging were enabled and appropriately configured.

3.3 Module, ScriptBlock & transcript logging

Transcripts were originally introduced to accompany PowerShell version 2 providing a crude form of logging by writing all command input and output to a text file at a defined location. Earlier revisions of the transcript logging feature were configured on a per-user basis and required a change to their respective PowerShell profile in order to enable transcripts. As this setting was controlled by the user profile it was trivial to disable and as a result, it allowed malicious activities to go unnoticed.

PowerShell version 5 introduced a revision to transcripts. This enabled them to be configured at a system level and enforced via group policy, making it possible to perform basic post-incident analysis in which PowerShell had been leveraged. However, such logging was not applied to any other processes. In addition, any use of Write-Host, code obfuscation or pipeline processing was not captured and instead commands were logged verbatim, leaving deobfuscation efforts for post-analysis activities.

Additions to transcript logging include the ability to record commands executed by “interface-less” instances of PowerShell, such as those executed within compiled binaries. This also extends to the ISE, which was not previously captured in transcripts.

Code obfuscation techniques often rely on the use of unclear variable names. For example, the use of StringFormat commands to reconstruct disparate code sections, custom aliases or character encoding. In order to execute code that utilises these techniques, the obfuscated content is parsed through a processing pipeline. The output of the pipeline processing may still contain short variable names and custom aliases, however, code sections that were broken into unintelligible segments are combined into human readable data in the log output.

The version 5 engine also introduced a variation on transcript logging that was known as ScriptBlock logging. This feature introduced pipeline parsing, making it possible to identify the precise command executed by a user, irrespective of any obfuscation applied to it. Analysis of the logs produced by ScriptBlock logging may be combined with a string or pattern to discern legitimate from malicious PowerShell usage.

Tools such as Invoke-Obfuscation [10] can perform multiple methods of transformation in order to obfuscate code. The code snippet below makes use of Invoke-Obfuscation to transform the "Write-Output 'foo'" command using multiple iterations of string and token obfuscation

```
((("{31}{21}{16}{33}{7}{56}{68}{42}{23}{0}{51}{43}{54}{4}{8}{48}{57}{34}{66}{65}{2}{49}{24}{35}{6}{67}{58}{29}{19}{11}{62}{64}{30}{44}{59}{60}{45}{13}{14}{52}{12}{9}{32}{40}{20}{26}{22}{18}{61}{36}{63}{27}{1}{15}{10}{50}{47}{46}{17}{53}{38}{28}{3}{41}{55}{39}{37}{5}{25}" -f '{,("{0}{2}{1}" -f ChAr, ' ', ]34)'),("{0}{1}" -f 'sD,','ks'),'B',("{3}{1}{0}{2}" -f 'C','h,8','ks','tput8C'),("{0}{1}" -f 'IC[5,']'),("{1}{0}" -f 'D.(', 'ks'),')6','D',("{1}{0}" -f '9'),']3'),'D',("{0}{1}" -f 'k','sD,')',',r','st',("{1}{0}" -f 'iNG','R'),',u', '{0}',',v','oks','C','C','1}{3}',("{1}{0}" -f 'sDxn','k'),'1',("{0}{1}" -f '8','Chk'),("{1}{0}{2}" -f 'sD','+ksDxk,')'),'E',("{0}{1}" -f 't','RiNG][('),('{1}{0}' -f '3]+E','1'),("{2}{0}{1}" -f 'D,ks','De8','ks'),'xn',', ((6iA{6}{2}{','re','{5}{7}{4}{8}',('{1}{0}{3}{2}' -f 'k','sD','hwriks','sD'),'sD','',',',',',',',b','PIA',("{0}{1}" -f 'UENv:', 'P'),("{0}{2}{1}{3}" -f 'Dt8','D','Ch,ks','',ksD'),'}{0}xno-f8Ch-', 'o',("{0}{1}" -f 'ks','D,['), 'UEN','EB',',',', 'D',("{0}{1}" -f '4&(', '),'2',("{1}{0}" -f '[ChA,']'),("{2}{0}{1}" -f 'PuBl','iC,':), 'ou','u',("{1}{0}" -f 'k','iA-f'),'ksD{k','no',("{1}{0}" -f 'e','ksD').r'),("{0}{1}{3}{2}" -f 'PIACE(', 'ks','8Ch','D'),'D',("{1}{0}" -f 'Dxnof','ks'),'[s','oo',("{2}{0}{1}" -f 'Dh'),'k','s'),'D,k','x','s')).("{1}{0}" -f 'Place','re').Invoke('uD4','l')."R`ePLACe"(([cHAR]54+[cHAR]105+[cHAR]65),[StRinG][cHAR]34).("{2}{1}{0}" -f 'e','ePlac','r').Invoke('EBU','$')."rE`pLa`CE"(([cHAR]107+[cHAR]115+[cHAR]68),[StRinG][cHAR]39) |.($ {ENv` :CO`M`SPeC}[4,26,25]-jolin")
```

While this command may be largely unreadable to a human, the eventual pipeline processing that is required for this to be executed by the PowerShell runtime engine, will return a version of the command which will be legible when using ScriptBlock logging. It is also a major advantage over simple transcripts, as they only record raw command input and output.

The event log messages below demonstrate the pipeline processing from the initial obfuscated command to the final executed command:

Creating Scriptblock text (1 of 1):

```
((["{31}x{21}x{16}x{33}x{7}x{56}x{68}x{42}x{23}x{0}x{51}x{43}x{54}x{4}x{8}x{48}x{57}x{34}x{66}x{65}x{2}x{49}x{24}x{35}x{6}x{67}x{58}x{29}x{19}x{11}x{62}x{64}x{30}x{44}x{59}x{60}x{45}x{13}x{14}x{52}x{12}x{9}x{32}x{40}x{20}x{26}x{22}x{18}x{61}x{36}x{63}x{27}x{1}x{15}x{10}x{50}x{47}x{46}x{17}x{53}x{38}x{28}x{3}x{41}x{55}x{39}x{37}x{5}x{25}"]-f'{'('{"0x21"}'-f'ChAr','',j34)'),("{"0x1"}'-f'sD','ks'),'B','{"31}x{0}x{2}'}-f'C','h,8','ks','tput8C'),("{"0x1"}'-f'IC[5','J']'),('{"1x0"}'-f'D.(','ks'),'j6','D','{"1x0"}'-f'9'),'j3'),'D','{"0x1"}'-f'k','sD'),'r','st','{"1x0"}'-f'iNG','R'),'u','{"0}','v','oks','C','C','1x3}','{"1x0"}'-f'sDxn','k'),'1'),'{"0x1"}'-f'8','Chk'),("{"1x0x2"}'-f'sD','+ksDxk',''),'E','{"0x1"}'-f't','RiNG[''),('{"1x0"}'-f'3'+E','1'),("2x0x1"}'-f'D,ks','De8','ks'),'xn','((6iA{6x2}','.re','{5x7x4x8','{"1x0x3x2"}'-f,k','sD','hwriks','sD'),'sD','','l','l','b','PIA','{"0}x{1}'}-f'UENv:','P'),("0x2x1x3"}'-f'Dt8','D','Ch,ks','ksD'),'xno-f8Ch-','o','{"0x1"}'-f'ks','D,[''],'UEN','EB','','D','{"0x1"}'-f'4&(','','),2','{"1x0"}'-f'[ChA','J']'),('2x0x1"}'-f'PuBl','iC',''),'ou','u','{"1x0"}'-f'k','iA-f'),'ksD[k','no','{"1x0"}'-f'e','ksD))..r'),("0x1x3x2"}'-f'PIACE(','ks','8Ch','D'),'D','{"1x0"}'-f'Dxnof','ks'),'[s','oo','{"2x0x1"}'-f'Dh'),'k','s'),'D,k','x','s'))('{"1x0"}'-f'Place','re').Invoke('uD4','l')."R'ePLAcE"((([cHAR]54+[cHAR]105+[cHAR]65),[StRinG][cHAR]34).("2x1x0"}'-f'e','ePlac','r').Invoke('EBU','$')."rE'pLa'CE"((([cHAR]107+[cHAR]115+[cHAR]68),[StRinG][cHAR]39)|.( ${ENv}:CO'M'SPeC)[4,26,25]-join"))
```

ScriptBlock ID: 63fc9d7a-08f4-41c1-945c-4fb20bdeea49

Initial obfuscated command

```
CommandInvocation(Write-Output): "Write-Output"  
ParameterBinding(Write-Output): name="InputObject"; value="foo"
```

Context:

```
Severity = Informational
Host Name = ConsoleHost
Host Version = 5.1.14393.1066
Host ID = c467d2b3-01ef-4001-8842-e8c91b1beb83
Host Application = C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Engine Version = 5.1.14393.1066
Runspace ID = f66618f9-fdd6-4210-93d2-1331e3c74368
Pipeline ID = 6
Command Name = Write-Output
Command Type = Cmdlet
```

Post-pipeline processing command output

3.4 Cryptographic Message Syntax (CMS)

Enabling features such as ScriptBlock logging may greatly increase overall visibility, but may also allow sensitive data to be inadvertently recorded in user-readable locations. Any data stored in event log messages may also be read by a non-privileged user, thus allowing logged sensitive data, such as user credentials, to be retrieved. This could potentially lead to horizontal and/or vertical privilege escalation.

To counter this, PowerShell version 5 introduced CMS [11], which uses a certificate to encrypt data before writing to the event log. This data may later be decrypted and analysed, but only when combined with the corresponding private key. This greatly reduces the risk of inadvertent disclosure of sensitive data through verbose logging, as any logged data cannot be read until decrypted.

Keys can be configured and deployed using group policy, allowing unique keys to be deployed per device. The “Unprotect-CmsMessage” cmdlet supports the presence of multiple private keys in the local certificate store. This makes it a seamless process to manage the use of multiple private keys without the need to manually specify the corresponding private key for a given encrypted message.

Once “protected”, event log messages are written with a Base64 encoded version of the encrypted data and are delimited by “—BEGIN CMS—” and “—END CMS—” to denote the start and end of the message. This format is similar to that seen by public/private key implementations, such as PGP and GPG. Furthermore, CMS is interoperable with other IETF compliant tools, such as OpenSSL, although some manipulation of the message is required for this.

3.5 Secure code generation

With the increase in administrative activities that are performed via PowerShell, there is a need for user-friendly tools which can consume and gracefully handle untrusted user input. Accepting the use of user-supplied input brings with it the need for proper input validation and sanitisation. Failure to do so can have negative consequences, which in the case of PowerShell is likely to lead to command injection.

Secure code generation is akin to protecting against cross-site scripting or SQL injection, as would be typically seen in a web application, though applied to PowerShell scripts. This feature allows user-supplied input to be escaped using a variety of functions enabling it to be dynamically included in commands or script blocks without the risk of negative or unforeseen states of execution.

This functionality is offered via the *CodeGeneration* [12] class, allowing it to also be leveraged by native .Net applications.

This class exposes the following four functions:

Name	Description
EscapeBlockCommentContent	Escapes content so that it is safe for inclusion in a block comment.
EscapeFormatStringContent	Escapes content so that it is safe for inclusion in a string that will later be used as a format string. If this is to be embedded inside of a single-quoted string, be sure to also call <code>EscapeSingleQuotedStringContent</code> .
EscapeSingleQuotedStringContent	Escapes content so that it is safe for inclusion in a single-quoted string.
EscapeVariableName	Escapes content so that it is safe for inclusion in a string that will later be used in a variable name reference. This is only valid when used within PowerShell's curly brace naming syntax. For example: <code>'\${' + EscapeVariableName('value') + '}'</code>

Use of these functions is strongly advised if any user-supplied input is to be parsed by PowerShell scripts, in particular those which operate in a privileged user context.

3.6 Application restrictions

Application restrictions, such as those imposed via AppLocker, can often prove to be an effective means of controlling access to applications and, in some instances, a means of enforcing the use of specific application versions. Policies can be configured in black and whitelist modes, however, the latter requires considerable effort in order to develop and maintain a suitable policy without negatively impacting legitimate business activities.

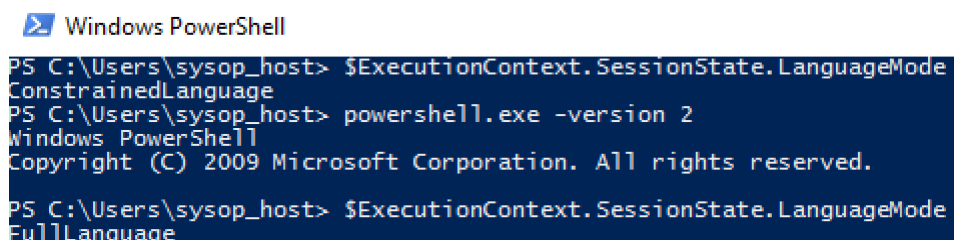
In addition, while it is successful in defeating a less sophisticated attacker, multiple documented instances exist which may make it possible to circumvent AppLocker policies. In particular those which make use of default baselines, such as whitelisting the entirety of the Windows directory.

A commonly overlooked default rule permits execution of binaries in `C:\Windows\Tasks`. Combining this default configuration with a compiled C# binary makes it possible to bypass multiple system protection features with relative ease. These include CLM, script execution policies, AMSI AV integration and application restrictions. However, while it may be possible to bypass these protections, ScriptBlock logging may be used to audit and alert on the execution of any PowerShell command, (which depending on configuration) may include details of the specific commands that are executed.

3.7 PowerShell version 2 engine

The PowerShell version 2 scripting engine, while present by default alongside later versions, is not subject to the same restrictions that apply to later versions. As a result of this, features including CLM and AMSI AV integration are not supported. In an environment where the PowerShell version 2 engine has not been removed, circumventing CLM and AMSI-aware AV can be trivially achieved by launching PowerShell in version 2 mode. This is done by simply supplying the “-version 2” parameter at runtime.

The screenshot below demonstrates the use of CLM, firstly in a PowerShell version 5 and secondly in a version 2 shell. Note that PowerShell version 2 does not execute in CLM, despite this being enabled system-wide. CLM is also ineffective when accessing the System.Management.Automation assembly via compiled C and C# binaries, irrespective of system configuration.



```
Windows PowerShell
PS C:\Users\sysop_host> $ExecutionContext.SessionState.LanguageMode
ConstrainedLanguage
PS C:\Users\sysop_host> powershell.exe -version 2
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\sysop_host> $ExecutionContext.SessionState.LanguageMode
FullLanguage
```

Language modes across PowerShell versions 2 and 5

The version 2 scripting engine can be removed to enforce the use of more recent PowerShell versions through Add/Remove programs. Thorough testing is recommended before performing this action as it may come at the cost of legacy function support.

3.8 Just Enough Admin (JEA)

In an environment where large numbers of services exist, all of which requiring distinct administrative users, it is easy to see how privileged accounts are often compromised. This is due to their widespread usage as part of normal business activities.

While good practice dictates that administrative accounts are only used when necessary and a lower privileged account is used for normal operational activities, it is almost impossible to properly enforce this across a large estate. Even in instances where this can be enforced, there comes a time where administrative users will need to log into a host, and, as such, potentially expose their accounts to compromise.

JEA [13] aims to solve this problem by providing a means to create accounts that can be limited to performing a precise task and nothing further. At present this feature is only supported through PowerShell remoting sessions.

Using JEA it is possible to create an account with sufficient privileges in order to perform a discrete task, such as updating DNS records or restarting a specific service. So called “role capabilities” can

also be defined, which can be open or sufficiently fine grained to limit the cmdlets that may be used as well as their associated parameter values to a precise set. Users are then assigned access to specific hosts and associated with role profiles through the use of session configuration files.

The role capability seen below creates a function in which the associated users are only able to execute the Restart-Service command and may only supply either “Dns” or “Spooler” as arguments.

```
VisibleCmdlets = @{ Name = 'Restart-Service'; Parameters = @{ Name = 'Name';  
ValidateSet = 'Dns', 'Spooler' }}
```

Policies are not limited to restricting the use of PowerShell cmdlets. They also provide support to whitelist external binaries or scripts by supplying their absolute path. It is important to note that allowing a user to execute external PowerShell scripts and binaries, also comes with the inability to restrict the parameters that may be supplied by the user. Careful review of any whitelisted binaries or scripts should be performed before assigning permissions to execute them within a policy. Where possible, native cmdlets should be used in place of third-party scripts as the input parameters. Respective values can also be contained using JEA.

When configuring JEA, it is recommended to make use of virtual accounts. This is because these are one-time ephemeral accounts which exist only for the duration of the session and as soon as a session is terminated the account is destroyed. In addition, the credentials for the virtual account need not be known by the connecting user, nor can the account be used to establish graphical remote desktop sessions or access an unrestricted PowerShell endpoint.

It is important to note that JEA does not prevent administrative users from accessing hosts. Instead it attempts to provide a means in which non-administrative users can perform a restricted set of administrative tasks without the need for a fully-fledged administrator account.

“One of the core principles of JEA is that it allows non-admins to perform *some* admin tasks. JEA does not protect against those who already have administrator privileges. Users who belong to "domain admins," "local admins," or other highly privileged groups in your environment will still be able to get around JEA's protections by signing into the machine via another means.” [14]

4. Conclusion

There are a number of options available to help reduce or identify the misuse of PowerShell within an organisation. However, there is no single solution that can effectively deny all access to the underlying functionality that it offers.

Application whitelisting can help to prevent unauthorised use of applications, including PowerShell and associated scripts. However, it comes at a cost of considerable effort being required to develop and maintain policies that effectively restrict application usage, without negatively affecting users or business processes. An alternative blacklisting approach may be taken, although this would only serve to prevent execution of known applications and any newly written or modified applications would not be subject to the blacklist.

Windows 10 provides native support for PowerShell version 5, which includes enhanced logging and detection features. This may also be extended by the use of Anti-Malware Scripting Interface AMSI capable AV such as Windows Defender.

When using module and ScriptBlock logging, it is vital that logs are centrally recorded and correlated. Using data from a variety of sources may provide insight into the activities of an attacker or malicious user and aid in identifying them easier in the attack chain.

Logging should be enabled and (where possible) a SIEM solution should be used to perform automated correlation, analysis and alerting based on clearly defined rules. Logging functionality offered by PowerShell version 5 may provide useful insight, not only into day-to-day business operations but also in identifying any potential conflicts caused by other protective mechanisms such as CLM or application restriction policies.

Log correlation may include keyword matching, however, this should not simply be restricted to searching for known tools by name. As previously mentioned, an attacker may be able to obfuscate or entirely remove references to a particular well-known name or function to prevent signature-based detection. Instead, searches should be performed for immutable constants, without which certain functionality would not be accessible. These may include references to specific Win32 API calls, assemblies or system libraries [15].

When utilising verbose logging, it is important to consider the nature of the data that may be logged as well as any downstream systems that may ingest this data. Given the verbose levels of logging that are offered by later versions of PowerShell, consideration should also be given when implementing protected event logging. This allows for the capture of potentially sensitive data without inadvertent disclosure through access to log messages, as log messages are encrypted and may only be decrypted by the corresponding private key. Encrypted log messages may be decrypted and parsed in order to remove or obfuscate sensitive data in a controlled manner before these are ingested by other services.

Not only does this aid in securing access to sensitive data, but it also masks logging activities from an attacker or malicious user. As a result, it provides an advantage to security and incident response analysts.

While Windows 10 offers a number of enhancements it is also bundled with PowerShell version 2, which is not subject to a number of native projects. As a result, it may undermine hardening attempts and (where possible), all hosts should be updated to the latest supported version of PowerShell. Not only does this bring a large number of functional improvements, it also provides access to the array of security enhancements detailed herein.

Use of these enhanced security controls should also be combined with a system-wide removal of the PowerShell version 2 engine as it is not subject to the same protection offered to later PowerShell versions. Leaving this component in place may allow an attacker or malicious user to circumvent controls, such as CLM and AMSI AV components with relative ease by simply launching PowerShell and supplying the “-version 2” argument.

It should be noted that commands executed using the version 2 engine are still subject to ScriptBlock logging. This would, however, limit the ability to identify and react to potential threats that leverage PowerShell.

When accepting and parsing user input in scripts, the functions exposed by the CodeGeneration class should be leveraged in order to sanitise all input before inclusion in any expressions.

In addition to the above measures, the use of JEA (where possible) is highly recommended. This reduces the need for large numbers of administrative users and provides a controlled environment in which non-administrative users may perform administrative activities without the need for a fully-fledged administrator account. Not only does the use of JEA aid in controlling management duties, it also provides greater insight into the day-to-day management of an environment. When combined with ScriptBlock logging, it may be possible to further refine JEA policies to more tightly restrict the precise commands that any given user may execute.

The combination of the technologies outlined in this document should be used as part of a layered defence approach. Effective log collection and analysis may prove to be vital, particularly when responding to an incident. However, this alone will not prevent malicious use of PowerShell-based tools.

Though these are by no means a perfect solution, careful implementation of features such as application restriction policies and use of CLM may help to reduce the potential impact that an attacker or malicious user may have. Additionally, the use of application restriction policies may also yield a substantial management overhead in order to ensure that policies are maintained without negatively impacting normal business operations.

5. References

- [1] <https://www.carbonblack.com/company/news/press-releases/carbon-black-united-threat-research-report-reveals-how-cyber-attackers-exploit-microsoft-powershell-to-launch-attacks/>
- [2] <http://www.darkreading.com/endpoint/powershell-increasingly-being-used-to-hide-malicious-activity/d/d-id/1325157>
- [3] <http://securityaffairs.co/wordpress/56110/malware/fileless-malware-campaign.html>
- [4] <http://securityaffairs.co/wordpress/46717/cyber-crime/fareit-data-stealer-powershell.html>
- [5] <https://docs.microsoft.com/en-us/powershell/azure/overview?view=azurermps-4.4.0&viewFallbackFrom=azurermps-4.0.0>
- [6] <https://blogs.msdn.microsoft.com/powershell/2008/09/30/powershells-security-guiding-principles/>
- [7] <https://blog.netspi.com/15-ways-to-bypass-the-powershell-execution-policy/>
- [8] <https://github.com/Cn33liz/p0wnedShell/tree/master/p0wnedShell>
- [9] <https://twitter.com/mattifestation/status/735261176745988096?lang=en>
- [10] <https://github.com/danielbohannon/Invoke-Obfuscation>
- [11] <https://blogs.technet.microsoft.com/srd/2015/06/10/advances-in-scripting-security-and-protection-in-windows-10-and-powershell-v5/>
- [12] [https://msdn.microsoft.com/en-us/library/system.management.automation.language.codegeneration\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/system.management.automation.language.codegeneration(v=vs.85).aspx)
- [13] <https://msdn.microsoft.com/powershell/jea/overview>
- [14] <https://msdn.microsoft.com/powershell/jea/security-considerations>
- [15] <https://adsecurity.org/wp-content/uploads/2015/01/BSidesDC2016-PowerShellSecurity-Defending-the-Enterprise-from-the-Latest-Attack-Platform-Presented.pdf>