Global Information Assurance Certification Paper

You've Had the Power All Along: Process Forensics With Native Tools

GIAC (GCIH) Gold Certification

Author: Trevor McAfee, trevor.n.mcafee@gmail.com
Advisor: *Bryan Simon*

Accepted: *July 14th, 2020*

Abstract

Many organizations are interested in standing up threat response teams but are unable, or unwilling, to provide funding or approval for third-party tools. This lack of support requires threat response teams to utilize built-in, OS-specific tools, to investigate suspicious processes and files. These tools can provide a significant amount of useful information when scrutinizing a suspicious process or file. However, these tools and their output are often unwieldy. A lack of cohesiveness requires running multiple similar commands to gather all the data for an investigation, and then manually combining and correlating that data. This paper examines the data of interest during an incident response and the native Microsoft Windows tools used to obtain it. This paper also discusses how to use PowerShell to automate the collection and compilation of this important data.

1. Introduction

Malicious software needs to be actively running as a process to be a threat to the system. Due to this need, an analyst may notice and investigate it further. To determine the legitimacy of a process, an investigator will analyze the running process thoroughly and ascertain the details of the process in question. However, to adequately assess process legitimacy, one must be familiar with their environment and standard Windows processes and behaviors. Additional outside research into unfamiliar processes will likely be necessary for making an informed decision, however this information is outside the scope of this research. It is assumed that readers will conduct this external research on their own as necessary.

There are many tools, both free and paid, that can significantly aid in an investigation. The most common and indispensable set of tools is the Sysinternals suites from Microsoft, which can accomplish system administration tasks, monitor, and investigate what processes are doing and touching (Heddings, Sysinternals Pro: What Are the SysInternals Tools and How Do You Use Them?, 2019). However, some organizations are very strict about what software is allowed to be installed and executed on their networks. Sometimes the process to obtain additional software approval is very long and arduous, or the organization simply will not allow it. Unfortunately, even though Microsoft acquired Sysinternals in 2006 (Microsoft, 2006), the tools have never been included in a default Windows installation, thus triggering the additional software approval process mandated by many companies.

Fortunately, Microsoft Windows comes with almost everything an incident responder might need to investigate processes. However, the main drawback to these tools is that much of the output data is unfriendly and often does not provide all the information desired, requiring the use of multiple similar applications to gather all the data.

This paper describes the tools in question and the viability of using PowerShell to automate the collection and parsing of all the data into one standard output. The data of

interest includes: the Process ID (PID), Parent Process ID (PPID), process lineage, executable path, command-line arguments, the user account running the process, associated network activity, and the Dynamic Link Libraries (DLLs) loaded into the process. This paper also covers searching for persistence mechanisms potentially used by the process, including scheduled tasks, Windows services, autorun directories, and other locations in the registry known to launch applications automatically. Additionally, this paper discusses collecting data from the filesystem related to the process in question, including collecting hashes of the executable and loaded DLLs, the creation date and time of the executable, and searching the filesystem for other files created within a specified time window from the executable creation time. Finally, this paper also investigates the automation of submitting the collected hashes to VirusTotal to help discern if these files are known malicious.

The following Windows tools are used to gather the information mentioned above:

- Windows Management Instrumentation (WMI)
- Tasklist
- Netstat
- Reg
- Schtasks
- CertUtil
- PowerShell

PowerShell is used throughout this paper both to collect data and to parse, filter, and stitch together the output of the other Windows tools discussed to present the information clearly and coherently. This paper assumes that the reader has a basic knowledge of scripting and PowerShell.

2. Test Setup

A test lab was created with the following versions of Windows to test the automatic collection of data via native Windows tools using a PowerShell script:

Author Name, email@addressm

- Windows 7 Service Pack 1 Build 7601
- Windows 8.1 Build 9600
- Windows 10 Build 18363
- Windows Server 2008R2 Service Pack 1 Build 7601
- Windows Server 2012R2 Build 9600
- Windows Server 2016 Build 14393
- Windows Server 2019 Build 17763

Two malicious executables using three means of persistence were the subjects of this test. The first malicious executable is a Meterpreter Bind Shell configured to listen on port 8080. This executable was created using MSFVenom on Kali Linux version 2020.1 and saved as a Windows executable named meter.exe.

```
root@kali:~# msfvenom -p windows/x64/meterpreter/bind_tcp LPORT=8080 -f exe -o /tmp/meter
.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 499 bytes
Final size of exe file: 7168 bytes
Saved as: /tmp/meter.exe
root@kali:~#
```

Figure 1 Generating Meterpreter Bind Shell Executable

This executable was placed in C:\windows\ on each test machine. A Windows service named LocalProxy was created and configured to run on system startup. The service executes cmd.exe, which in turn runs c:\windows\meter.exe.

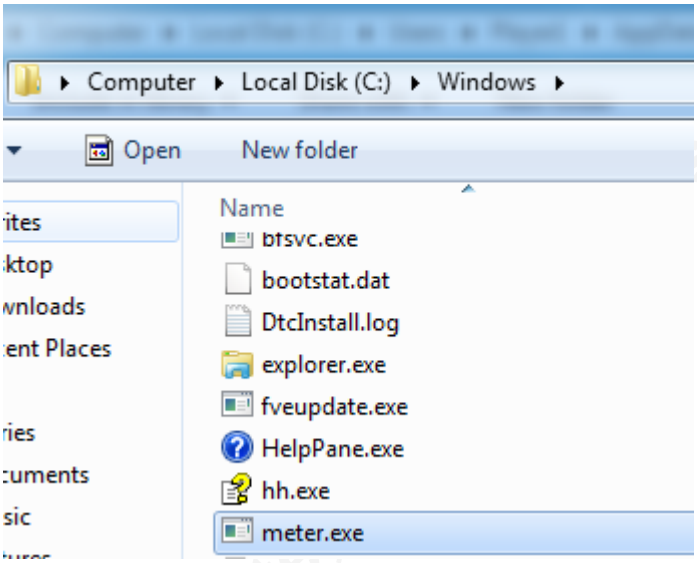


Figure 2 Meterpreter Bind Shell in place on victim

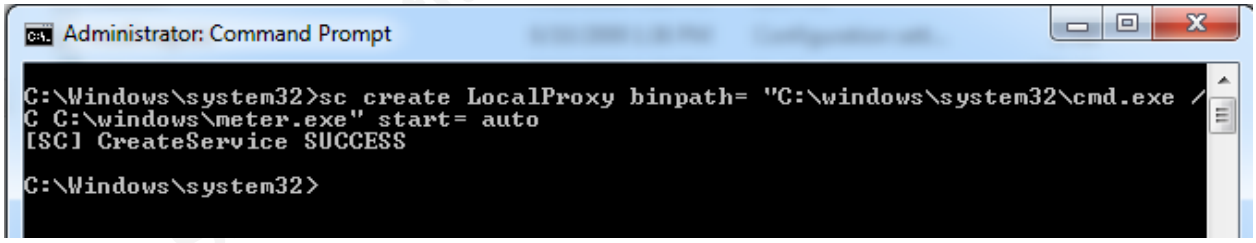


Figure 3 Creation of LocalProxy Service

The second executable is a copy of Netcat, nc.exe, taken from /usr/share/windows-resources/binaries/nc.exe on the same Kali Linux machine. Nc.exe was also placed in C:\windows\ on each test machine. A scheduled task was created named "Simple Web Server Starter" which was set to run every hour as the SYSTEM account with command-line arguments to set up a listener on port 80 that will execute cmd.exe upon connection.

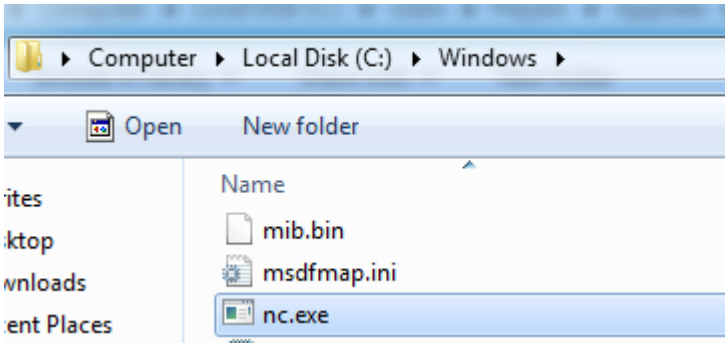


Figure 4 Netcat in place on victim

Author Name, email@addressm

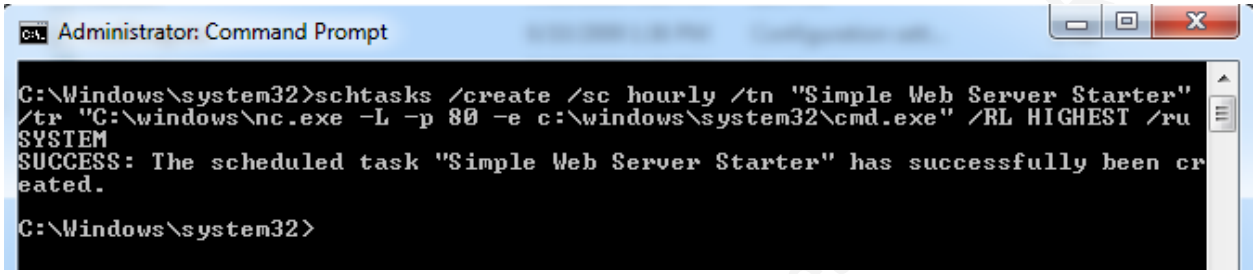


Figure 5 Creating Simple Web Server Starter scheduled task

Finally, a shortcut was created to nc.exe named "Management Interface," which was configured to start another Netcat listener on port 8888 that also executes cmd.exe upon connection. This shortcut was placed in the user's startup folder, "C:\Users\Player1\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\," which runs the shortcut every time the user logs on to the system.

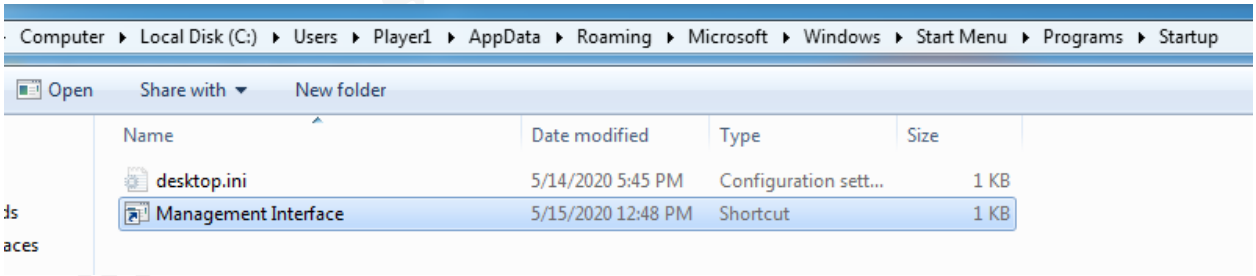


Figure 6 Management Interface shortcut placed in startup folder

Figure 7 shows the full target path: "C:\Windows\nc.exe -L -p 8888 -e C:\windows\system32\cmd.exe"

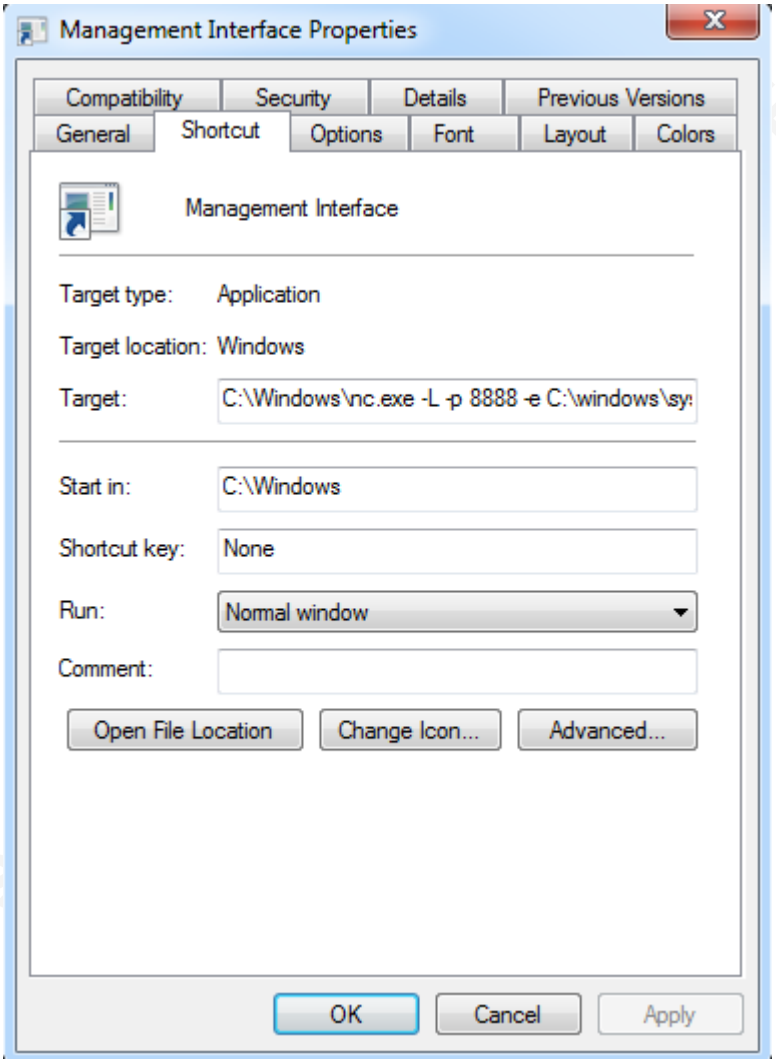


Figure 7 Management Interface shortcut configuration

After restarting the system, logging in, and waiting an hour, netstat was used to confirm that each means of persistence, for each of these malicious executables, worked successfully as processes were listening on port 80, 8080, and 8888. This is shown in figure 8.

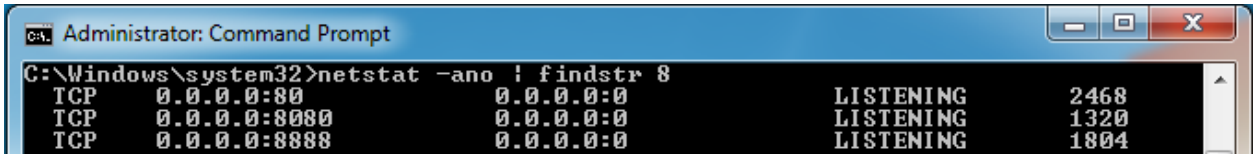
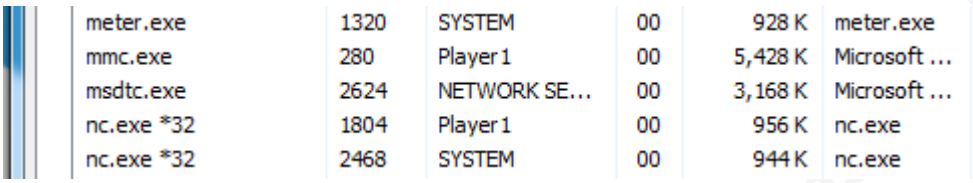


Figure 8 Netstat view of each malicious process

Task Manager, shown in figure 9, also confirmed that meter.exe and two instances of nc.exe were running, and the PIDs matched the listening ports shown in figure 8.

Author Name, email@addressm



meter.exe	1320	SYSTEM	00	928 K	meter.exe
mmc.exe	280	Player1	00	5,428 K	Microsoft ...
msdtc.exe	2624	NETWORK SE...	00	3,168 K	Microsoft ...
nc.exe *32	1804	Player1	00	956 K	nc.exe
nc.exe *32	2468	SYSTEM	00	944 K	nc.exe

Figure 9 Task Manager view of malicious processes

3. Gathering Data Using Windows Tools

The tools discussed in the following sections come pre-installed on every Windows version, at least as far back as Windows 7. Below is a brief description of each one, followed by how to gather the desired data with each tool.

3.1. Tools of the Trade

3.1.1. CertUtil

Certutil is a command-line application provided by Microsoft as part of Certificate Services (Microsoft, 2017). While it is primarily meant to dump, display, and configure certification authority configurations and verify certificates, it can also be used to hash files using the -hashfile option. Interestingly, while CertUtil has this functionality, the Microsoft documentation does not specify which hashing algorithms it supports. Upon testing, it is confirmed that as far back as Windows 7, CertUtil supports at least the MD5, SHA1, SHA256, and SHA512 hashing algorithms.

3.1.2. Netstat

Netstat is a Windows tool used to view all listening and active connections on a system. It will also show the corresponding PID of the process listening or initiating the connection (Microsoft, 2017). The PID enables cross-referencing of network activity with running processes to see if the process in question is active on the network.

3.1.3. PowerShell

PowerShell is an object-based command-line shell and scripting language developed by Microsoft and built on .NET (Microsoft, 2020). It has come pre-installed on every version of Windows since Windows 7 and Windows Server 2008R2 (Microsoft, 2010) with each new version of Windows coming with an updated version. Windows 7

and Windows Server 2008R2 come with PowerShell Version 2.0, while Windows 10 and Windows Server 2019 come with Version 5.1. Each release introduces new cmdlets and features. To ensure the widest compatibility with created scripts, an understanding of the environment is needed to determine the oldest version of PowerShell in use. All scripts created should be designed with that version in mind. PowerShell scripts are forward compatible, meaning scripts written for older versions will still work on newer versions, so there is no risk of breaking something by ensuring backward compatibility (Wilson, 2015). Additionally, PowerShell can be used to execute non-PowerShell executables on the system.

3.1.4. Reg

Reg is a command-line tool that performs operations on registry keys and values (Microsoft, 2017). The subcommand "reg query" is used to search the registry for any entry that matches a given string.

3.1.5. Schtasks

Schtasks is a Windows application that allows administrators to "create, delete, query, change, run, and end scheduled tasks on a local or remote computer" (Microsoft, 2018).

3.1.6. Tasklist

Tasklist is a Windows command-line application that displays a list of currently running processes (Microsoft, 2017). The main benefit of using Tasklist is that when executed with the verbose option, /v, it displays the user context of each process.

3.1.7. WMI

The WMI tool is "the infrastructure for management data and operations on Windows-based operating systems" (Microsoft, 2018). WMI was released in 1998 and was the core system management utility starting with Windows 2000 (Microsoft, 2006). Administrators use WMI to obtain detailed information about systems and to administer them, both locally and remotely.

3.2. Running Process Information Collection

In simple terms, a process is a program that is currently executing on the computer (Microsoft, 2018). Running processes are the most volatile sources of data discussed in this paper as they can stop at any time, eliminating much of the sought-after data along with it.

3.2.1. Executable Path, Command Line Arguments, and PPID

The executable path of a process provides essential insight into its legitimacy (Nolan, et al., 2005). For example, common locations for program execution include the Program Files and System32 folders, so processes running from these locations have a higher probability of being legitimate. However, other places, such as internet cache folders, user directories, and the root of the file system, are non-standard locations for executables, and processes running from these locations are less likely to be legitimate (Yonts, 2014). Many standard process viewing applications, such as Task Manager and Tasklist, show only the name of the executable and not the path, so a malicious executable can attempt to hide in plain sight by having the same name of a legitimate program but running from a different location. For example, svchost.exe running from c:\windows\system32\ is likely the valid Windows application, but svchost.exe running from anywhere else, such as C:\Windows\, is suspect.

Many applications have either standard, expected command-line arguments or don't use command-line arguments at all. Understanding what arguments are typical for an application will help determine if the process is legitimate or if a malicious program is masquerading as the legitimate one. For example, svchost.exe will almost always be invoked with the -k option followed by an argument, so instances of svchost.exe running without a -k option are worthy of investigation.

A process's parent process is of interest to an investigator because many applications have an expected lineage, and any deviation from this baseline is cause for concern. For example, Microsoft Word should not be starting cmd.exe, wscript.exe, or powershell.exe (Weyne, 2016), and only Wininit.exe should spawn lsass.exe (Lee &

Pilkington, 2018). Any deviation from this behavior should be considered suspicious and investigated further.

WMI retains the executable path, command line arguments, PPID, and more for each process. PowerShell's Get-WMIObject cmdlet, using the win32_process WMI class, will display this data.

All of the PIDs used in the following examples are the PIDs of the test malware processes created in the Test Setup section.

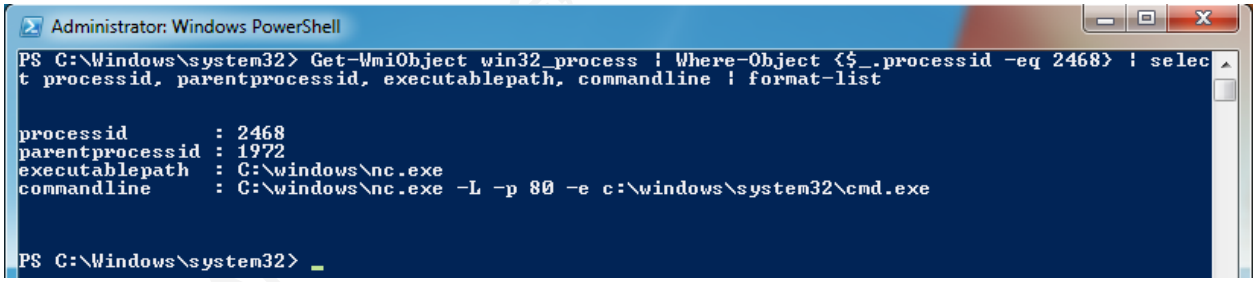


Figure 10 WMI process information for PID 2468

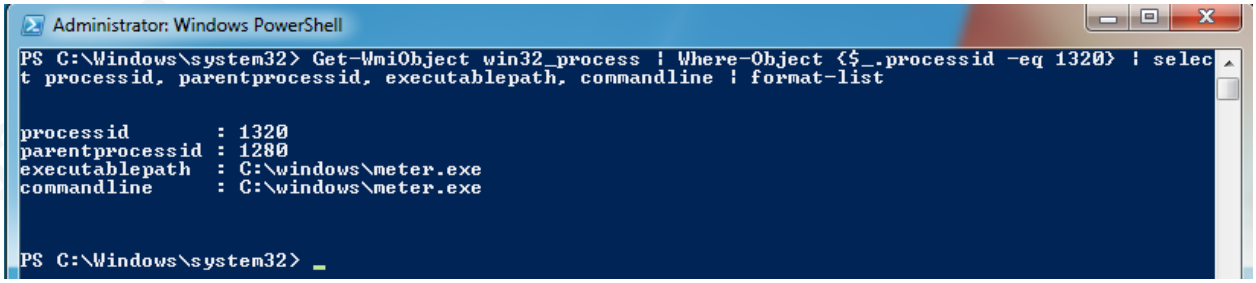


Figure 11 WMI process information for PID 1320

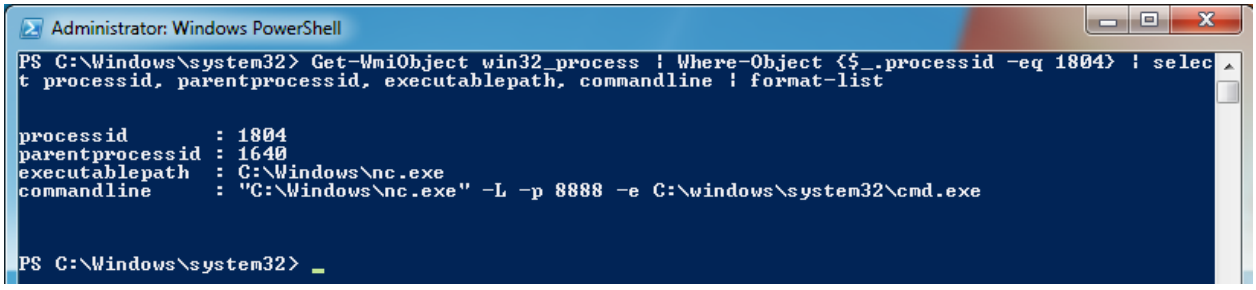


Figure 12 WMI process information for PID 1804

Saving this output to a variable will allow investigators to combine this data with additional information about each process gathered with different tools.

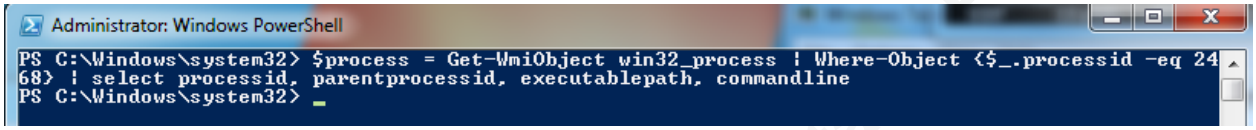


Figure 13 Saving WMI process information to variable

WMI can query the same information of each parent process using a PowerShell loop to provide a complete picture of process lineage. This lineage will help an investigator determine if there is anything suspicious regarding the creation of the process in question. Figure 14 shows the process lineage for PID 2468 using the WMI process data variable created in figure 13.

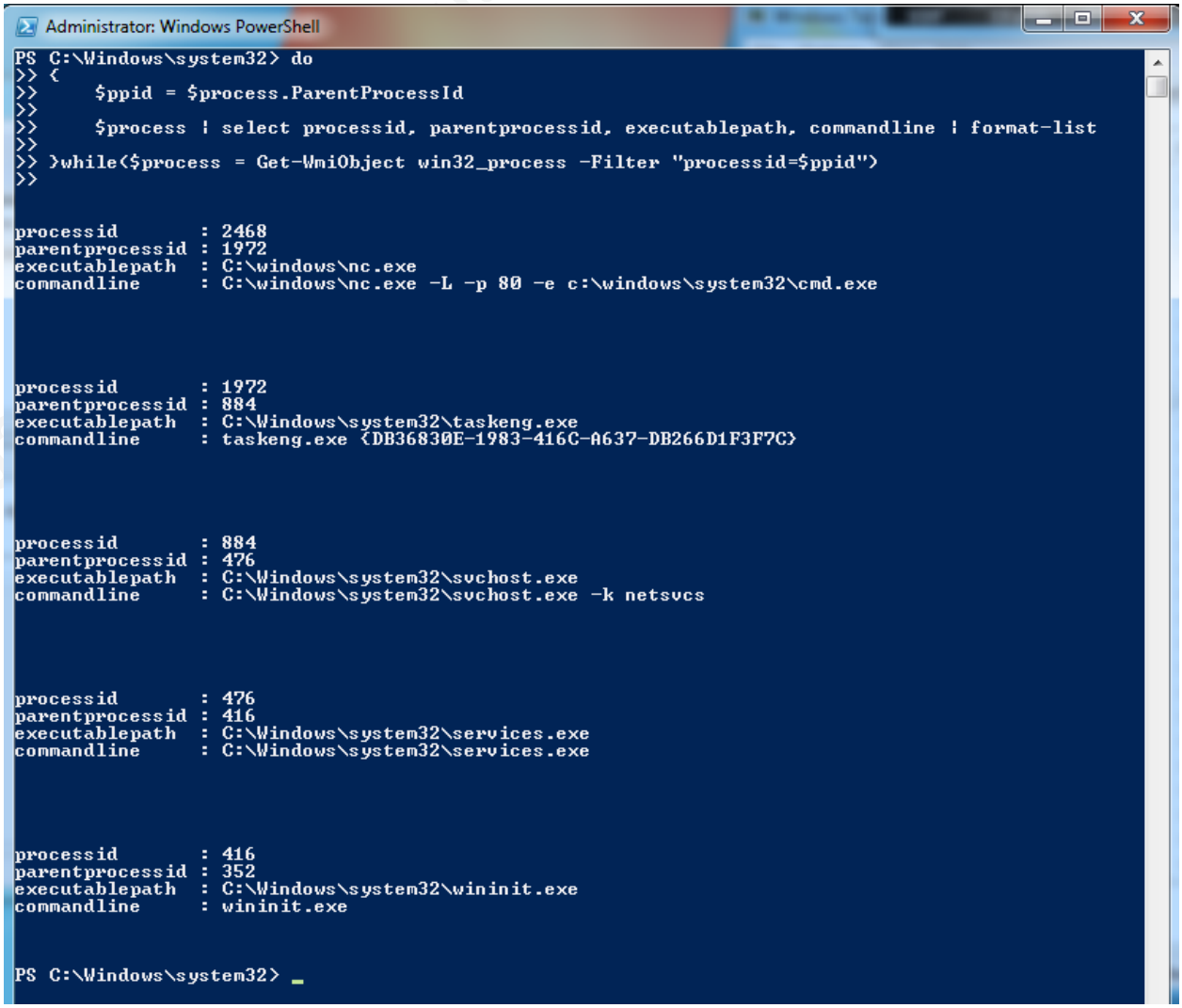


Figure 14 Process lineage for PID 2468

3.2.2. Username

The process' username is relevant because it specifies what permissions the process has and what activity it can perform. For example, if a process is running as standard user "Bob," it will be able to access and modify any of Bob's files. However, it won't be able to access Alice's data, nor will it be able to perform system configuration changes, such as creating new services. Conversely, processes running as an administrator or the System account can access all files and implement any configuration changes. Additionally, most executables run as an expected user, so any deviation could be an indicator of malicious activity and is worth investigating further. For example, only normal users should be running web browsers, and only System should be running lsass.exe, so if a user is running lsass.exe or System is running a web browser, this is abnormal and worthy of further examination.

Running Tasklist with the verbose option, /v, when executed with administrator privileges, will display the user context of every process. Tasklist can output this information in Comma Separated Value (CSV) format, which can be easily converted to PowerShell objects using the ConvertFrom-Csv PowerShell cmdlet.

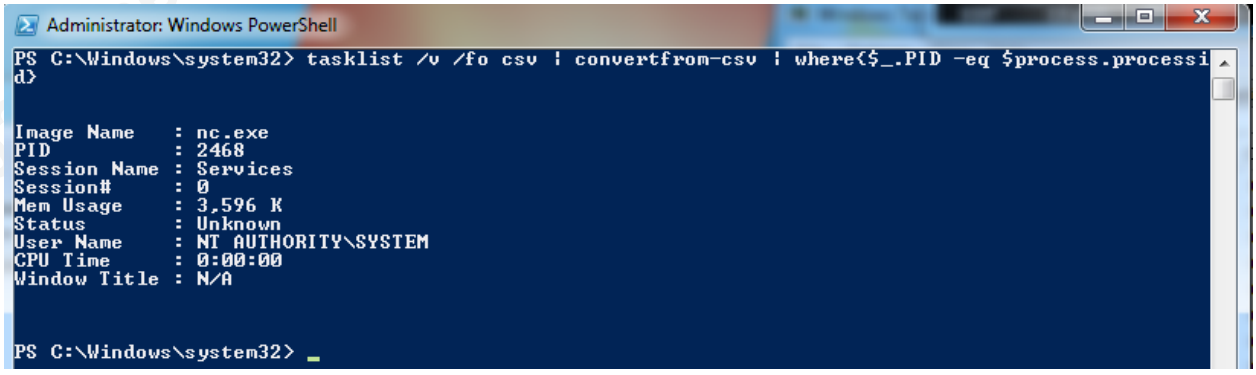


Figure 15 Tasklist data for PID 2468 after conversion to PowerShell object

Once PowerShell converts the data, the User Name field can be extracted and added to the process variable created earlier. Now the data is combined and can be displayed in one coherent output, as shown in figure 16 below.

```
PS C:\Windows\system32> tasklist /v /fo csv | convertfrom-csv | where{$_.PID -eq $process.processi
d} | select "User Name"

User Name
---------
NT AUTHORITY\SYSTEM


PS C:\Windows\system32> $process | Add-Member NoteProperty Username (tasklist /v /fo csv | Convert
From-Csv | where {$_.PID -eq $process.processid} | select -expand "User Name")
PS C:\Windows\system32> $process


processid       : 2468
parentprocessid : 1972
executablepath  : C:\windows\nc.exe
commandline     : C:\windows\nc.exe -L -p 80 -e c:\windows\system32\cmd.exe
Username        : NT AUTHORITY\SYSTEM



PS C:\Windows\system32> _
```

Figure 16 Username information from Tasklist added to WMI data for PID 2468

3.2.3. Process List

An investigator can combine the above techniques to display the process name, PID, PPID, username, executable path, and command-line arguments of every running process on the computer with a PowerShell loop. This process list would help during the initial review of a system to determine if any process on the machine looks suspicious.

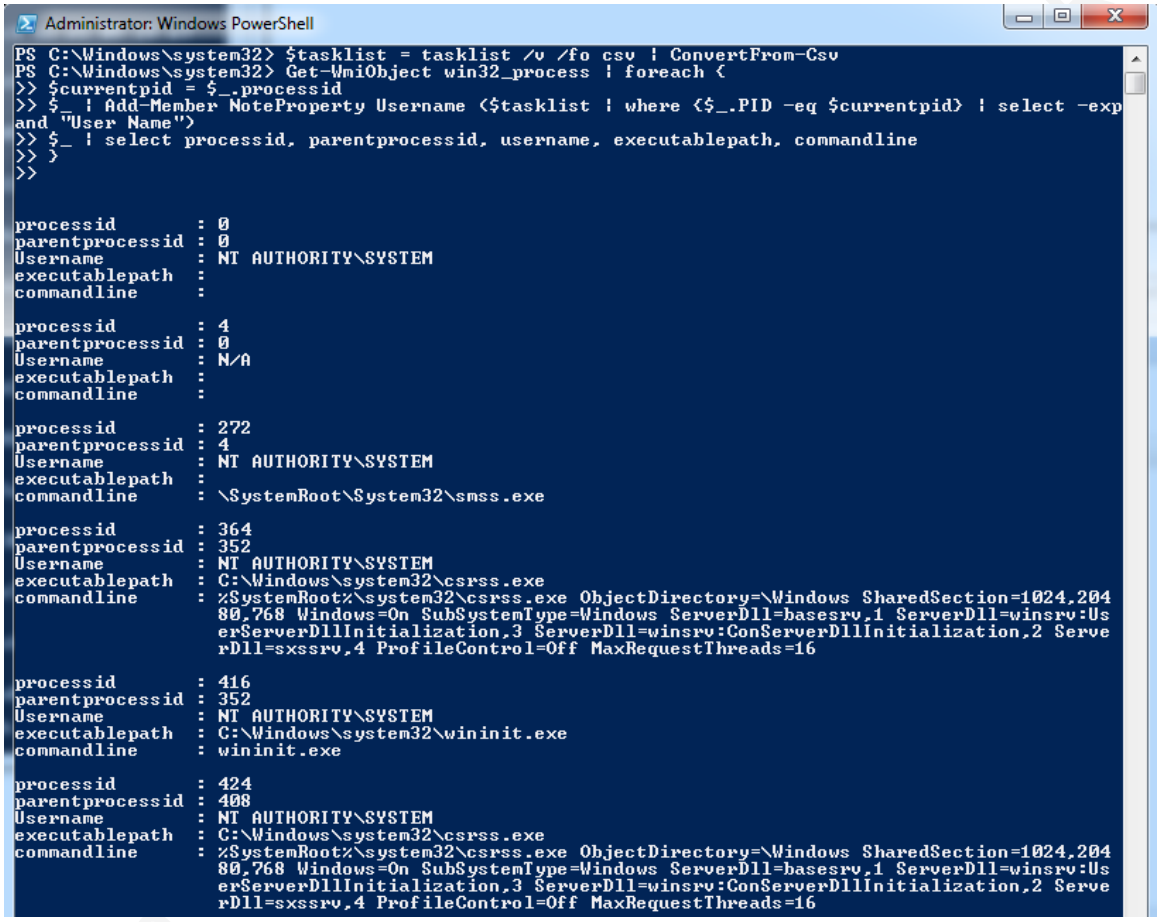


Figure 17 Combining and displaying Tasklist and WMI information for each running process

To display the data as a table, a PowerShell table object can be created, and each process's data can be added as a new row to the table as described on Microsoft's website (Microsoft, 2012). In this example, the script creates a function to create the table if necessary and add rows to the table. The script then uses a loop to call the function on each process to add it to the table. The script then displays the table on the screen.

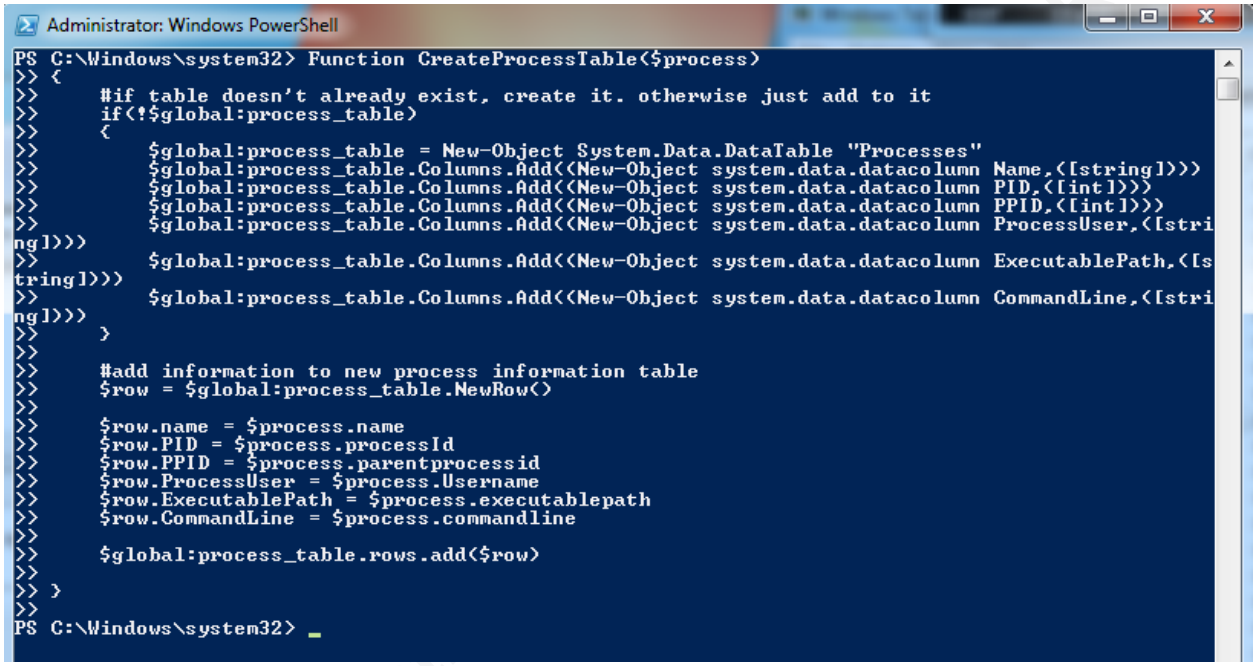


Figure 18 Function for creating the process list table

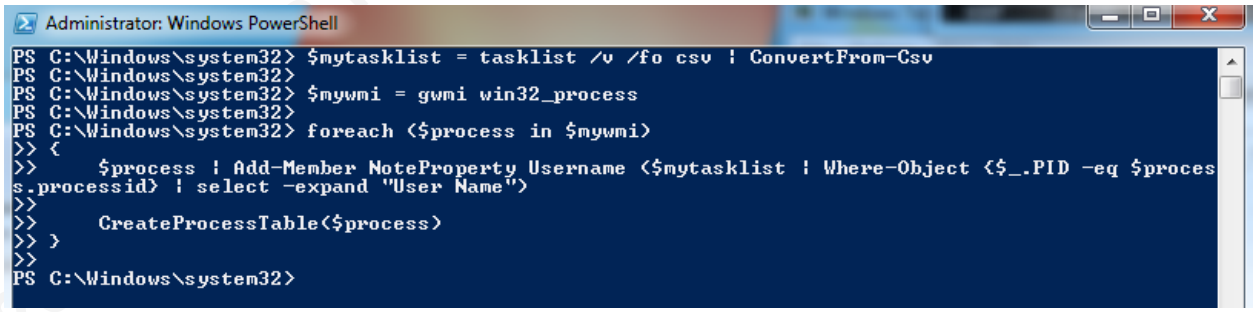


Figure 19 Combining process data and adding process information to the table

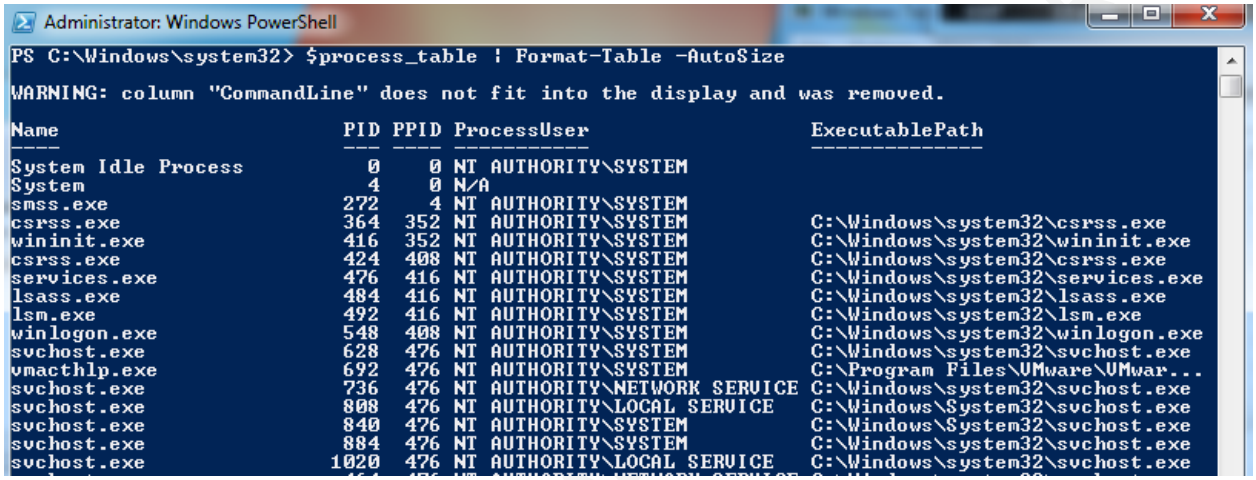


Figure 20 Displaying the process list table (Note: The command-line data is not displayed in this screenshot due to space constraints)

3.2.4. Process Tree

As mentioned previously, process lineage is a meaningful datapoint in determining if a process is suspicious. User Kazun on Microsoft's Technet forum provides a PowerShell function to create a simple process tree (Kazun, 2013).

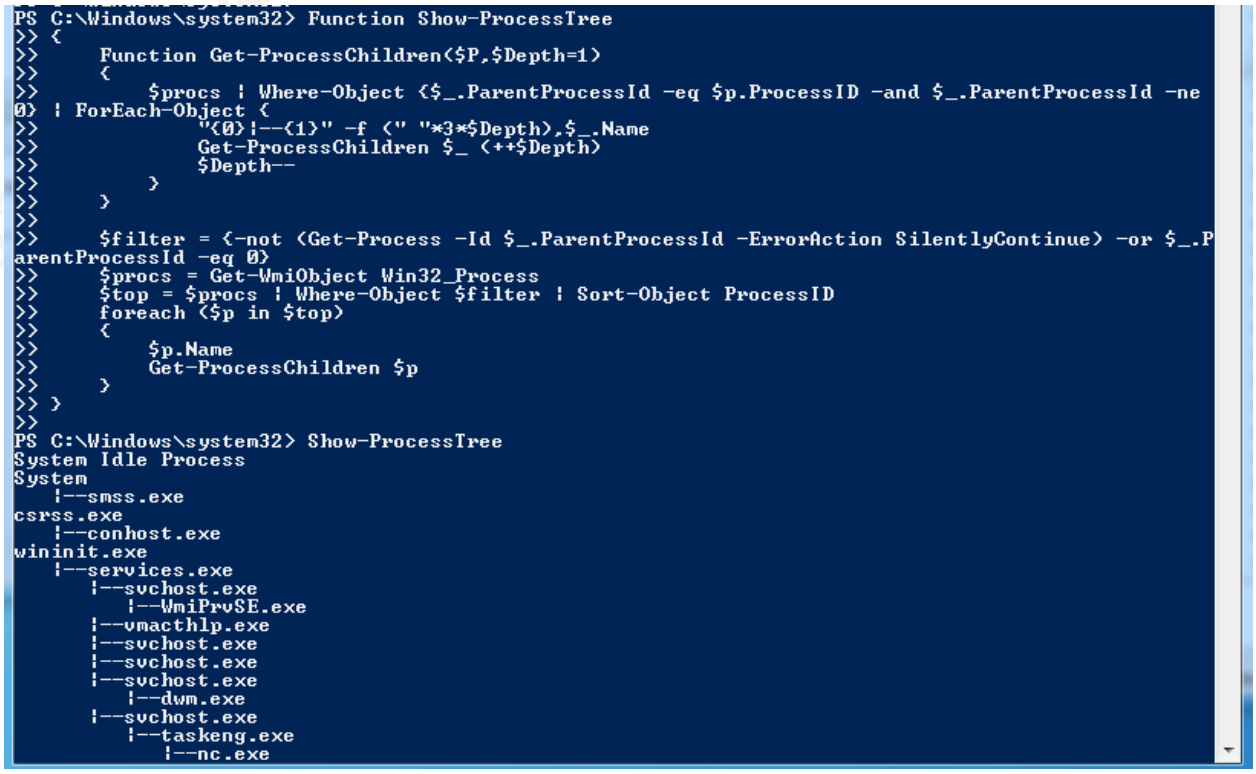


Figure 21 Kazun's process tree function and results

With some minor adjustments to this function and the process table function mentioned above, these functions can be combined to display the process name, PID, PPID, username, executable path, and command-line arguments of each running process in tree form.

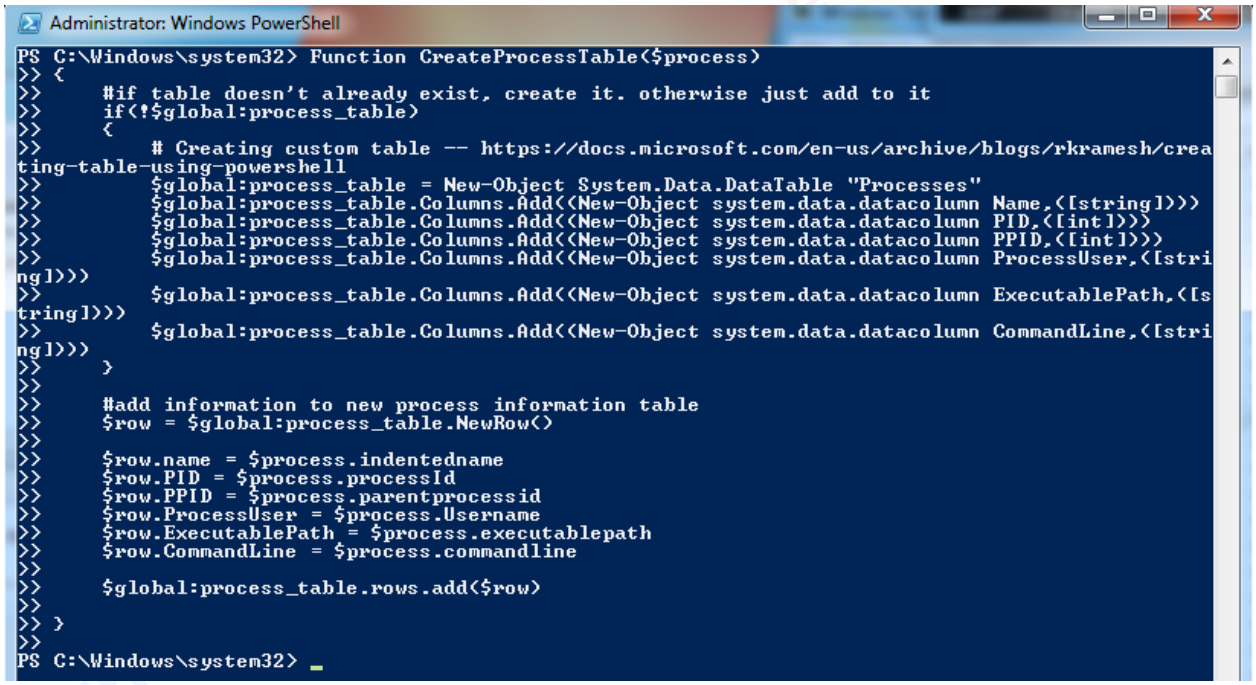


Figure 22 Function creating the process table, same as figure 18

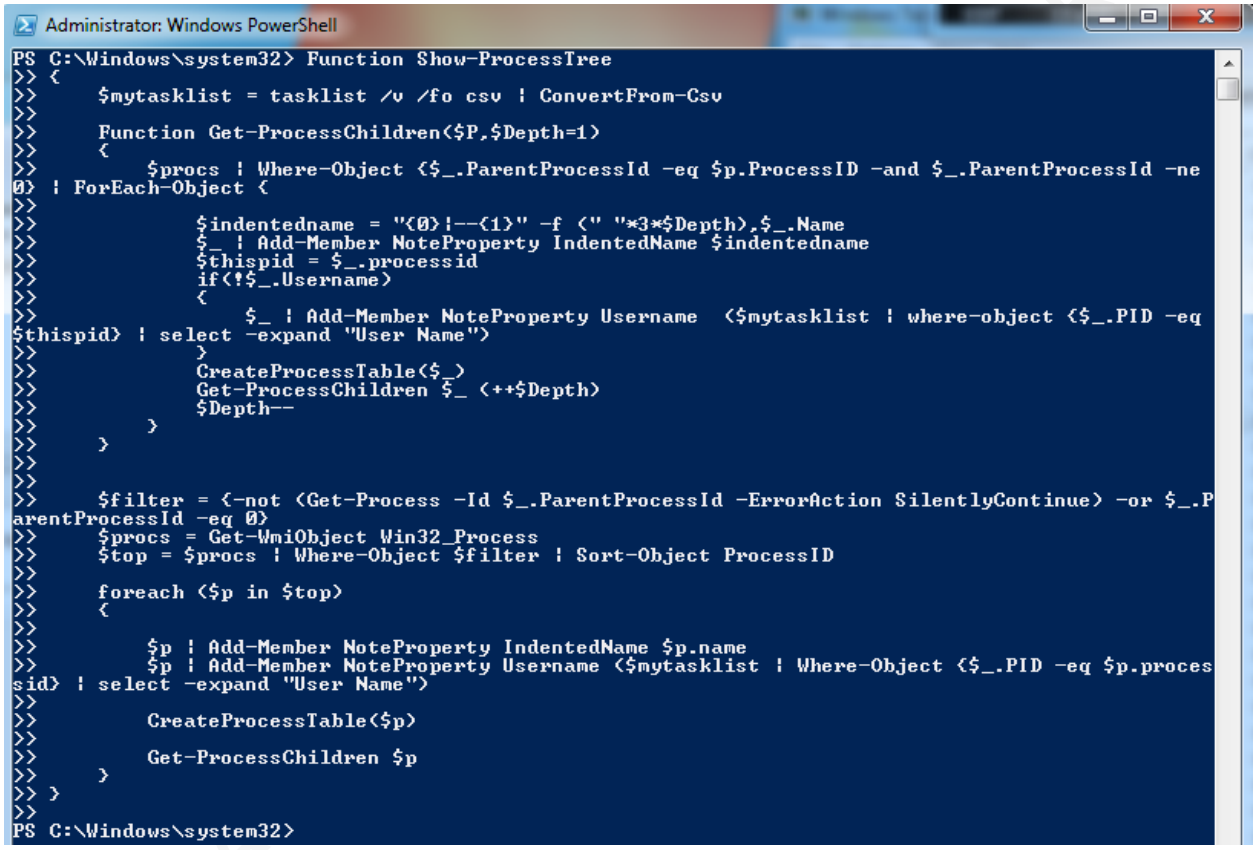


Figure 23 New Process Tree function

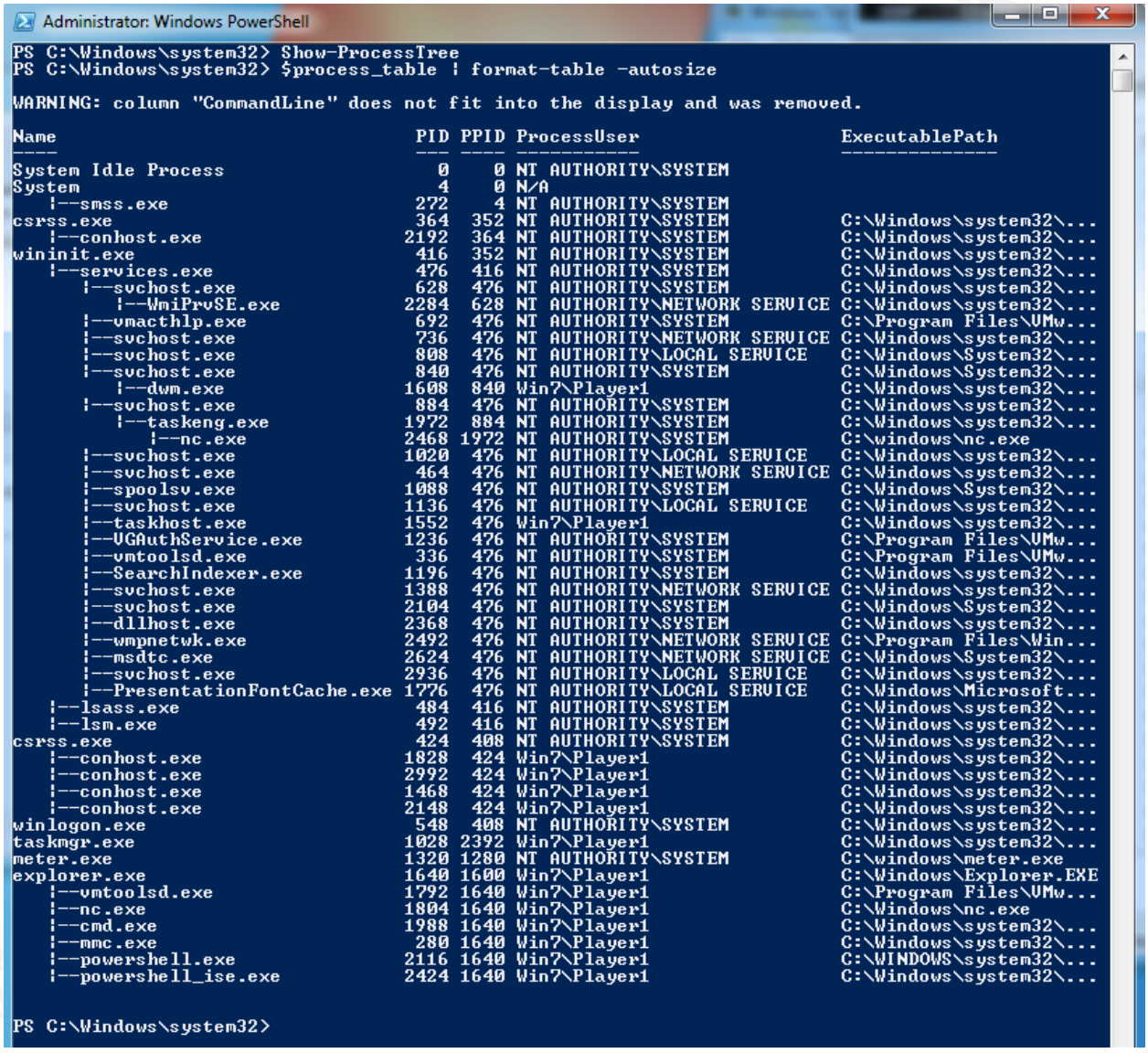


Figure 24 Display new process tree (Note: due to size constraints, the CommandLine column is not displayed in this screenshot)

3.2.5. Network Data

Besides information about the running process itself, another important set of data to collect is any network activity associated with it. Malware often attempts network communication, such as listening for inbound connections from the attacker or reaching out to Command and Control (C2) servers to check for new instructions. This paper uses netstat to gather this information instead of the more PowerShell-friendly Get-NetTCPConnection cmdlet due to the desire to make this script backward compatible

with PowerShell Version 2.0. Get-NetTCPConnection wasn't available until PowerShell Version 3.0.

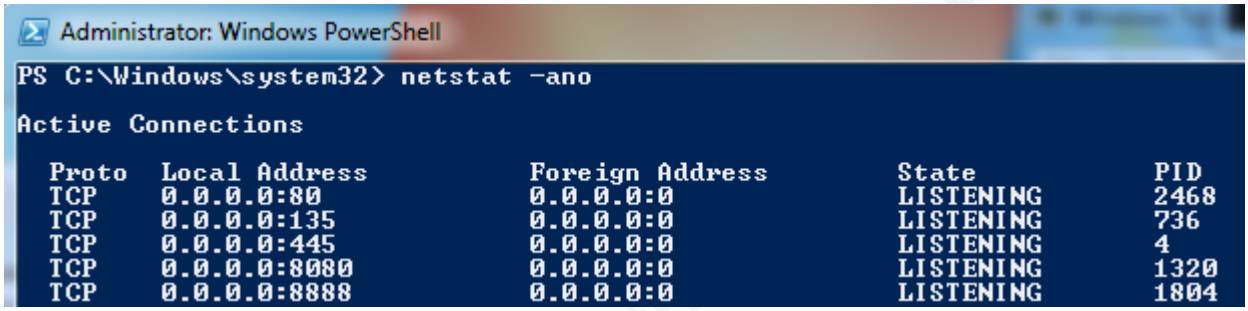


Figure 25 Raw Netstat output

Netstat does not provide output formatting options and returns the results as a series of strings, therefore piping the output to PowerShell's Select-String cmdlet is required to obtain only the lines associated with the PID in question. However, it is worth noting that using Select-String with just the PID will match anything in the output strings. If the PID happens to match part of a port number or IP address, that line will also return. The Select-String query can be enhanced with regular expressions to overcome this challenge. The PID is always the final column in the netstat output and is preceded by whitespace, so if an investigator searches for the PID as the last part of a line and after a word boundary, they can ensure the search matches just the PID. For example, when searching for PID 2468, the select-string query will be "\b2468$". A variable can also be used in place of the hardcoded PID 2468.

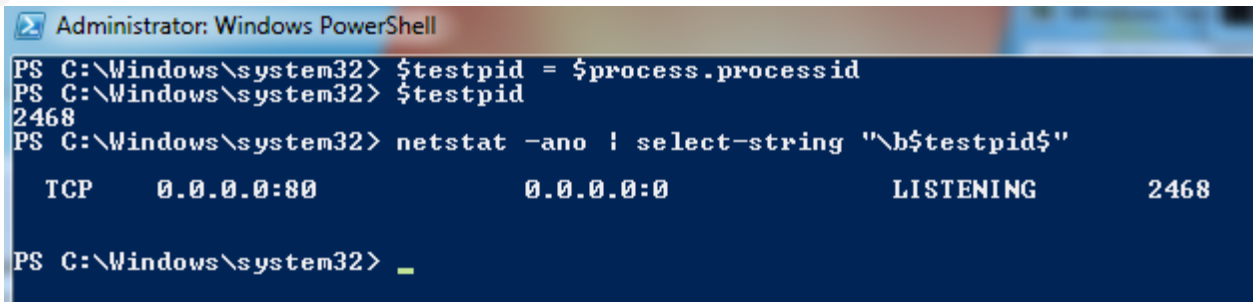


Figure 26 Netstat output for the PID held in $process variable, PID 2468 in this example

3.2.6. Loaded Dynamic Link Libraries

Dynamic Link Libraries (DLLs) are files containing pre-written code that programs can use to ease the burden on application developers and to ensure multiple

programs performing the same action have the same experience (Microsoft, 2019). Microsoft provides many DLLs with Windows, so their existence is not unusual. However, a malicious actor can also create DLLs, which can be either injected into or loaded by a running process (Phan, 2015). These malicious DLLs, once loaded by a process, can provide whatever functionality a malicious actor wants.

PowerShell's Get-Process cmdlet, when executed as an administrator, contains a listing of each DLL loaded by a process, the path it was loaded from, the company that created it, the file version, and more. Figure 27 shows the information for DLLs loaded by PID 2468.

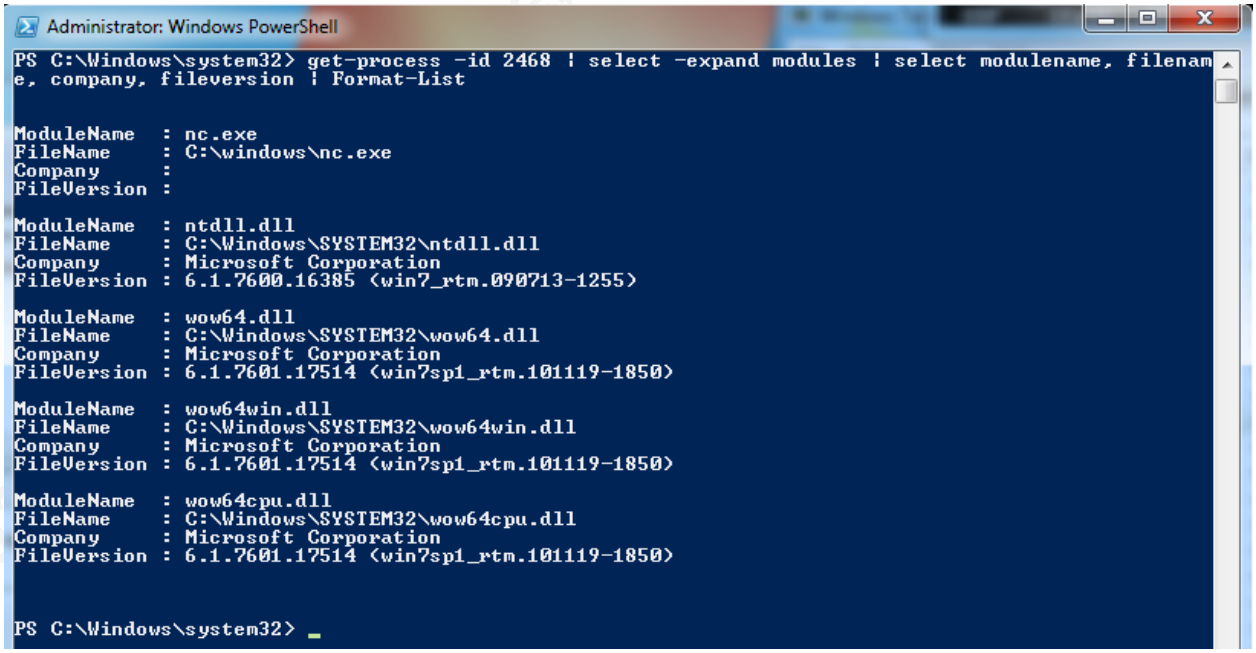


Figure 27 DLL and associated information loaded by PID 2468

Saving this data to a variable, as shown in figure 28, permits the addition of each DLL's file hash to the PowerShell Object. Once combined, both sets of data can be displayed in one view, as seen in figure 33.

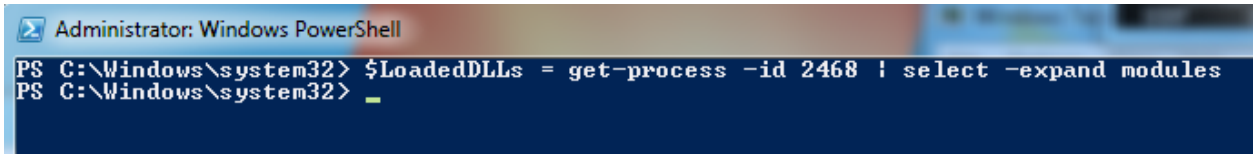


Figure 28 Saving DLL information to a PowerShell variable

3.2.7. File Hashes

File hashes are like fingerprints uniquely identifying a file or string. Variable sized data is fed to a hashing function, and a fixed size string is output, representing the hash (Hoffman, 2018). Calculating the hash of the executable and DLLs allows an investigator to use a tool, such as VirusTotal, to search for antivirus results of the files without having to upload them.

Using CertUtil with the -hashfile option, a file path, and a hashing algorithm, will obtain the hash of the given file with the given hashing algorithm. The output format of the hash generated by CertUtil has changed over the years. On Windows 7, there is a space after every two hex characters, but on Windows 10, CertUtil outputs the hash without spaces.

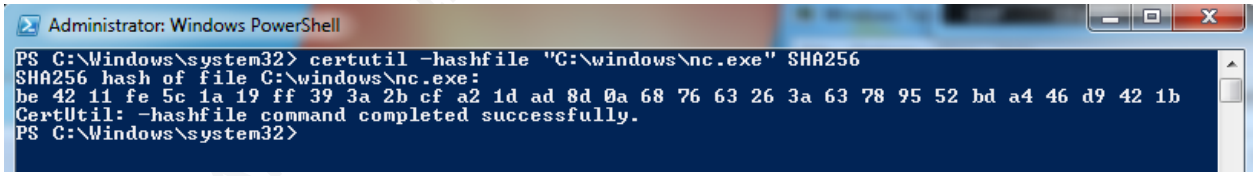


Figure 29 CertUtil output on Windows 7

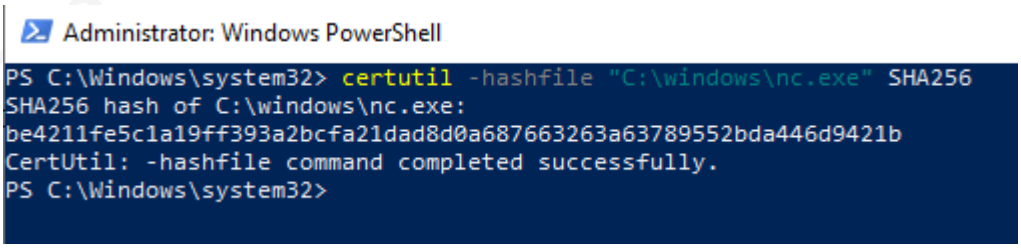


Figure 30 CertUtil output on Windows 10

As seen in figures 29 and 30, there are extraneous lines of data, and the hash output is not normalized. PowerShell can be used to select just the line containing the hash and then remove the spaces by using its string replacement functionality. Adding this additional logic will produce normalized results regardless of the operating system.

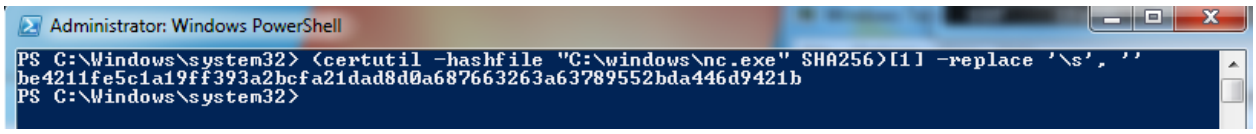


Figure 31 CertUtil Normalized on Windows 7

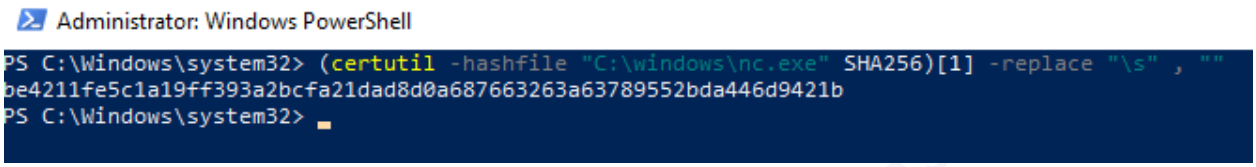


Figure 32 CertUtil Normalized on Windows 10

To collect the hashes of each loaded DLL, loop through the DLL variable and add a new member to the object containing the hash calculated by CertUtil. Figure 33 shows the collection and addition of the DLL hashes to the DLL variable and displays the results.

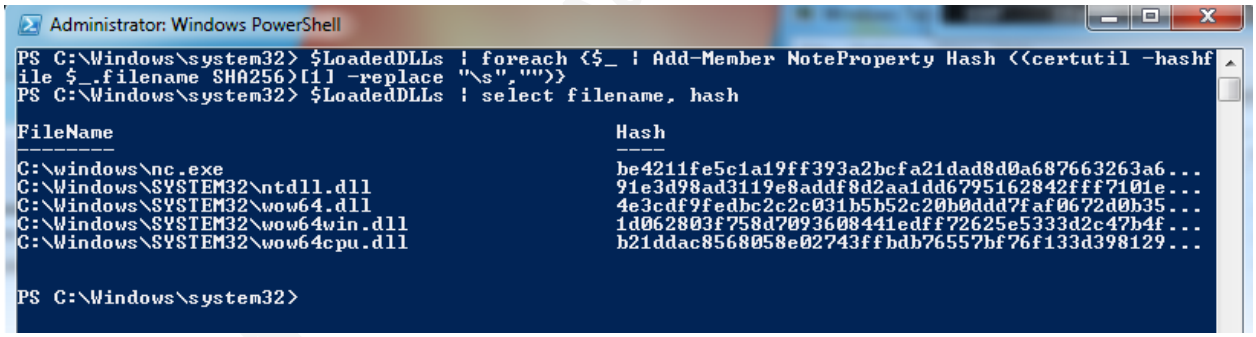


Figure 33 DLL hashes calculated, added to the PowerShell object, and displayed

3.3. VirusTotal Analytics

VirusTotal is a website that offers free antivirus scanning of submitted files and websites using over 70 antivirus scanners and shares the data with VirusTotal's partners and community (VirusTotal, n.d.). VirusTotal also offers the ability to search for antivirus results based on file hash. A user might prefer this option for numerous reasons, including a desire to keep company files private or not wanting to tip off a potential attacker that someone is researching their malware. If a user queries VirusTotal for a file hash that hasn't been scanned by their service, no results will return.

PowerShell's Invoke-RestMethod cmdlet can be used to submit file hashes to VirusTotal's API. Before using the API, an API key must be acquired by creating an account on their site, which will provide a free "public" level API key. This key needs to be embedded as an HTTP header with queries to the API. It is worth noting that this public API key comes with request quota restrictions. At the time of this writing,

VirusTotal limits the public API to four requests per minute, 1,000 requests per day, and 30,000 requests per month (VirusTotal, n.d.). The example script includes a 15-second sleep timer to ensure it doesn't exceed these thresholds when looping through requests to the VirusTotal API.

It is also worth noting that unlike every other application and cmdlet discussed thus far, PowerShell's Invoke-RestMethod Cmdlet is only available in PowerShell Version 3.0 and higher, which comes standard with Windows 8. To run this part of the script on Windows 7, PowerShell will need to be updated. Otherwise, this command must be executed from Windows 8 or newer.

VirusTotal's API accepts a web request with the hash appended to the end of the URL and the API key embedded as a request header field. In figure 34, a single request to the API is sent where $hash is the variable containing a string of the hash being queried, and $vtAPIKey is a variable containing the API key.

```
Invoke-restmethod https://www.virustotal.com/api/v3/files/$hash -Headers @{"x-apikey"=$vtAPIKey}
```

Figure 34 Sending request to VirusTotal's API using PowerShell

Invoke-RestMethod returns a PowerShell object containing multiple sub-attributes that hold the results from the query. The properties of interest are the last_analysis_date and last_analysis_stats, which include the date VirusTotal last scanned the file in question and the results from the scans. However, the last_analysis_date provided is in Linux Epoch time format, requiring conversion to a human-readable format before being displayed to the user.

In figure 35, every loaded DLL is iterated through, and each file that is not from Microsoft is submitted to VirusTotal. The decision not to submit DLLs from Microsoft to VirusTotal was made to speed up script execution due to the VirusTotal queries-per-minute limitation. If this behavior is not desired, the check to determine if the file is from Microsoft can be removed.

```powershell
#This is the master variable that will hold the data for all dll lookups
$VTLookupResults = @()

foreach($dll in $LoadedDLLs)
    {
        #Don't Look up DLLs created by Microsoft. This was done to reduce the number of lookups
        if ($dll.Company -ne "Microsoft Corporation")
        {
            $hash = $dll.hash

            $test = Invoke-restmethod https://www.virustotal.com/api/v3/files/$hash -Headers @{"x-apikey"=$vtAPIKey}



            #Convert Last Analysis Date from Epoch time to human readable time
            [datetime]$origin = '1970-01-01 00:00:00'
            $LastAnalysisDate = $origin.AddSeconds($test.data.attributes.last_analysis_date)

            #Create a temporary variable to store just the VirusTotal results we want, to be added to master variable
            $results = $test.data.attributes.last_analysis_stats
            $results | Add-Member NoteProperty Filename $dll.filename
            $results | Add-Member NoteProperty Hash $hash
            $results | Add-Member NoteProperty LastAnalysisDate $LastAnalysisDate

            #Add results to master variable
            $VTLookupResults += $results

            #VirusTotal API is rate limited (4 per minute with Free API). This Call to sleep ensures we don't break that limit
            sleep($vtsleep)
        }
    }
```

Figure 35 PowerShell loop to query VirusTotal for each DLL not from Microsoft

```
Administrator: Windows PowerShell
PS C:\Windows\system32> $VTLookupResults


confirmed-timeout : 0
failure           : 1
harmless          : 0
malicious         : 44
suspicious        : 0
timeout           : 1
type-unsupported  : 1
undetected        : 27
Filename          : C:\windows\nc.exe
Hash              : be4211fe5c1a19ff393a2bcfa21dad8d0a687663263a63789552bda446d9421b
LastAnalysisDate  : 5/14/2020 8:54:11 PM



PS C:\Windows\system32>
```

Figure 36 Final results from VirusTotal

3.4. Persistence Mechanisms

For malware to survive a system reboot, it needs to be configured for persistence, so that it will execute automatically in the future (Fortuna, 2017). There are multiple ways a piece of malware can accomplish this. This paper focuses on four primary methods: Scheduled Tasks, System Services, Registry AutoStart Locations, and Startup

Folders. These items may exist even if the executable in question is not currently running as a process.

3.4.1. Scheduled Tasks

Windows Task Scheduler is "like an alarm clock that you can set, to start a procedure under specified circumstances" (Arntz, 2015). Scheduled tasks can be set to start at a specific time, at repeatable time intervals, upon system boot, user login, and more. When creating a scheduled task, the creator can also specify a program to execute, the user the program should run as, and that the program automatically bypass User Access Control (UAC) prompts (Arntz, 2015).

Using Schtask with the /query and /v options gathers all information about each scheduled task, and the /fo CSV option outputs the data as comma-separated values. Piping this output to ConvertFrom-Csv in PowerShell turns the data into PowerShell objects allowing easy searching and filtering of the data. To search for scheduled tasks that run the executable in question, pipe the converted PowerShell object into Where-Object searching for any matches of the executable path within the "Task to Run" field.

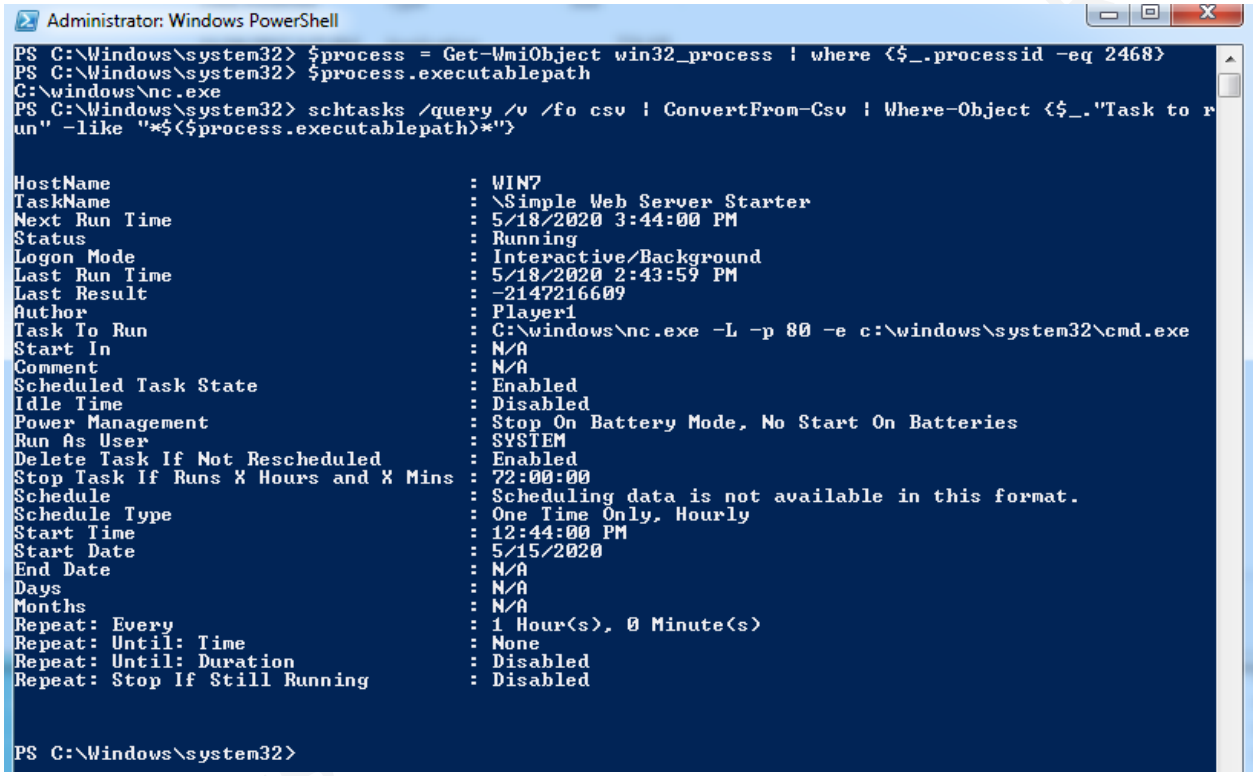


Figure 37 Scheduled Task results for tasks associated with executable of PID 2468

Also, note the search string includes a beginning and ending '*' character in the path. This wildcard character ensures that results match that have data before or after the executable path, as shown in Figure 37. Otherwise, Where-Object will only return if the executable path matches exactly, with no extra data. This example task would not have matched without the wildcards.

3.4.2. Windows Services

Windows services are "long-running executable applications that run in their own Windows sessions" (Microsoft, 2017). Services can be configured to start on system startup as a specified user and provide no user interface (Microsoft, 2017). Creating services requires administrator-level privileges on the system (Heddings, Understanding and Managing Windows Services, 2019).

WMI's win32_service class provides detailed information about every installed service such as: service name, startup type, the path to the executable, any command-line

arguments passed to it, the user it runs as, if it's currently running, and more. This information can be piped to Where-Object to filter the results based on the executable in question.

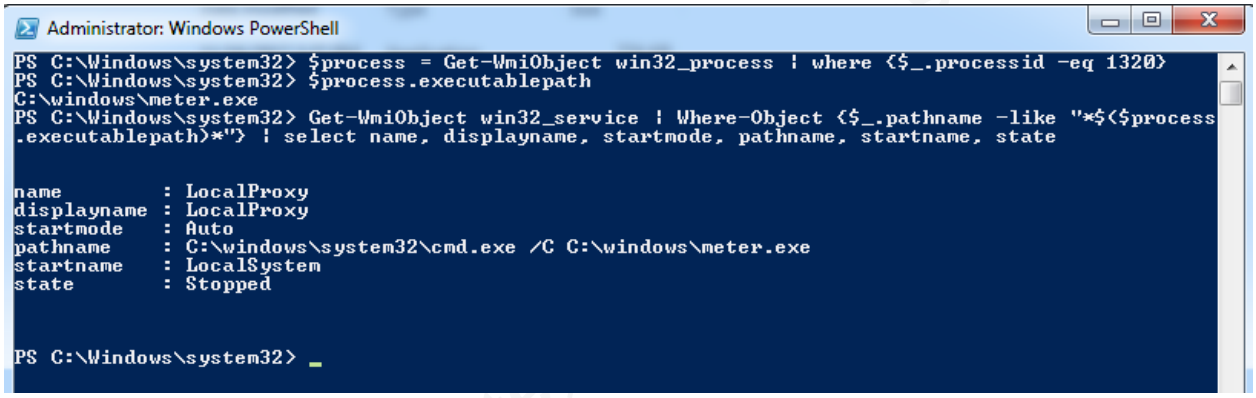


Figure 38 Windows service data associated with the executable of PID 1320

3.4.3. Windows Registry Auto Start Locations

The Windows registry has multiple locations where applications can be registered to start automatically on system boot or user login. The most notable of these locations are the Run and RunOnce registry keys. These keys "cause programs to run each time that a user logs on" (Microsoft, 2018). The value of these keys is the command-line invocation for a program, including the path and program arguments. The paths to applications within these keys could be the fully defined path, such as C:\windows\system32\notepad.exe, or use environment variables as part of the path, such as %windir%\system32\notepad.exe. The RunOnce keys are automatically deleted before the system executes its value, while the Run keys are permanent (Microsoft, 2018). Many other registry locations can also be used for persistence.

Sysinternal's Autorunsc program was executed to obtain a list of known registry keys used for persistence. Each registry key was saved into a PowerShell array to be fed to Reg for searching. This list is provided in Appendix A of this paper.

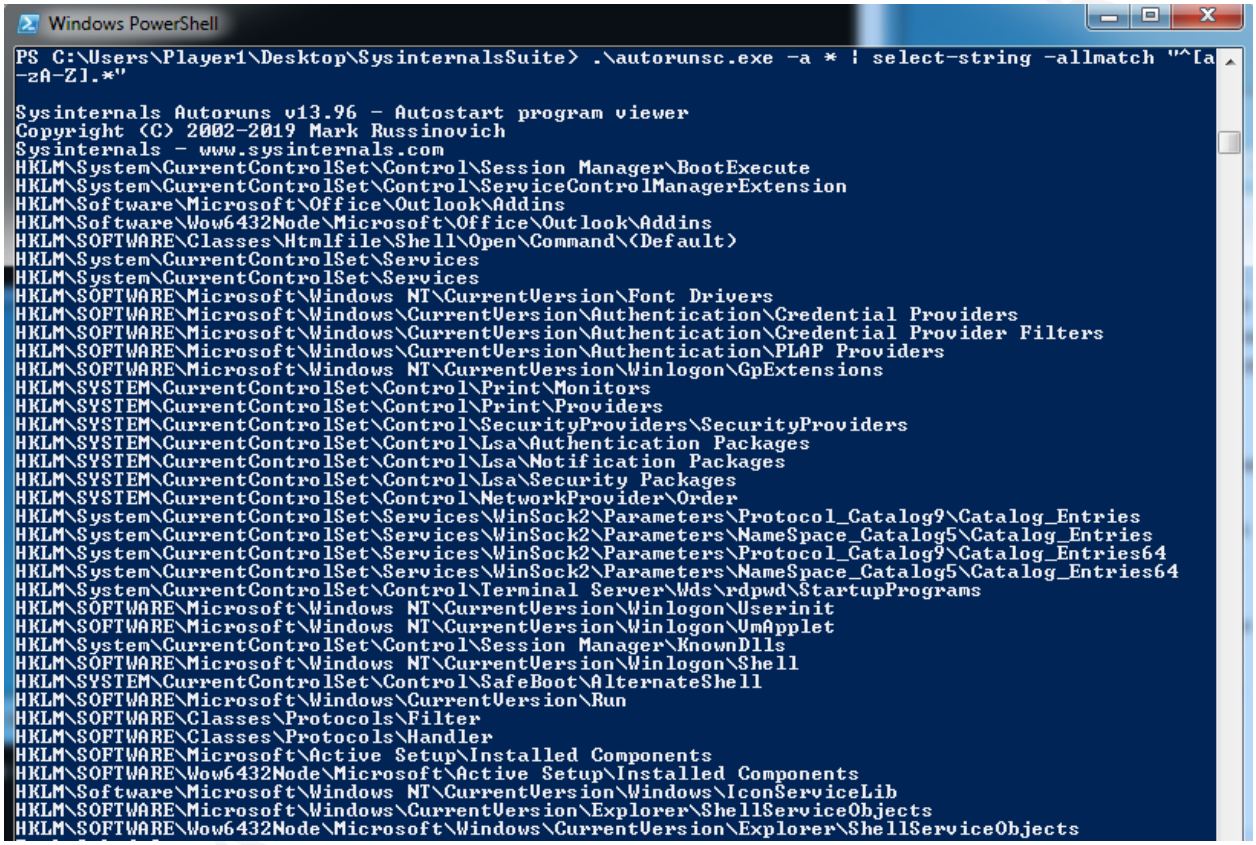


Figure 39 Autorunsc registry key output

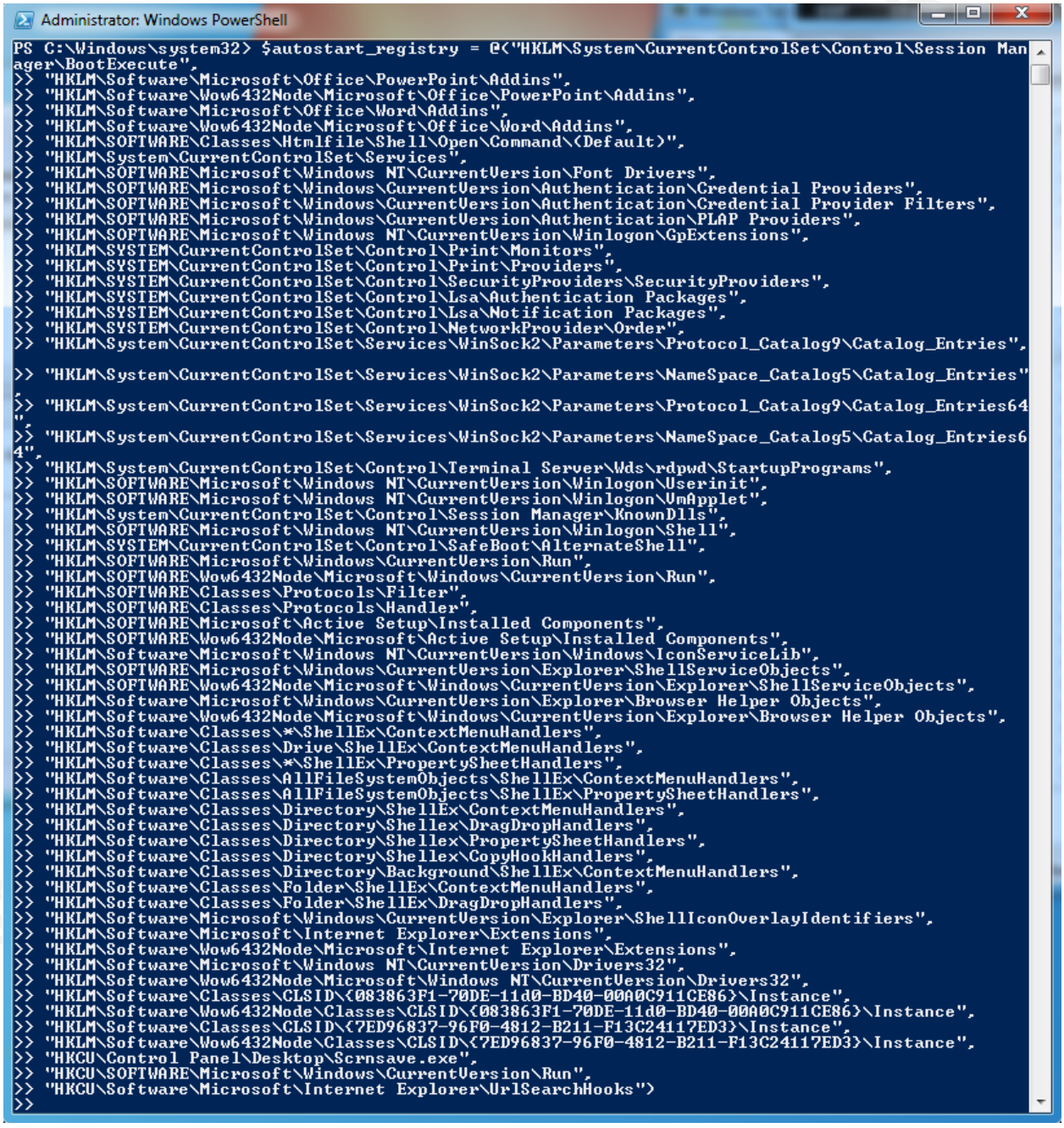


Figure 40 Adding each registry key to an array

In Figure 41, Reg is used to search each registry key for the use of the "C:\Windows\meter.exe" executable. Reg found one result in the HKEY_LOCAL_MACHINE\system\CurrentControlSet\Services\LocalProxy key.

Author Name, email@addressm

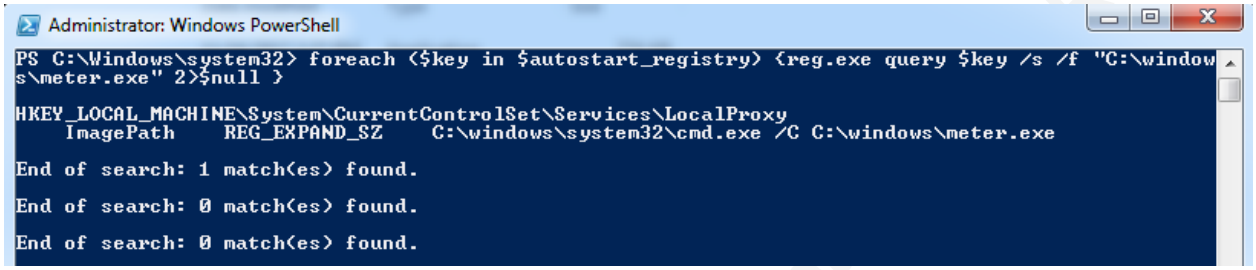


Figure 41 Using Reg to search the registry

Reg's output is also plain text and requires the use of PowerShell to clean up the data and present it in a more organized manner. In figure 42, an empty PowerShell array is created to hold the final results and another variable is created to hold the raw output from Reg. Reg always finishes the output with a blank line followed by a line that reads, "End of search: x match(es) found," which will interfere with the otherwise standardized output. The output can be trimmed by re-defining the variable holding the raw Reg results to include all the data except these last two lines. Then this variable is run through a For loop, creating a new PowerShell object and assigning properties to it for the registry key found and the value that matched. This object is then added to the final results array, and the final results array outputs all the data found.

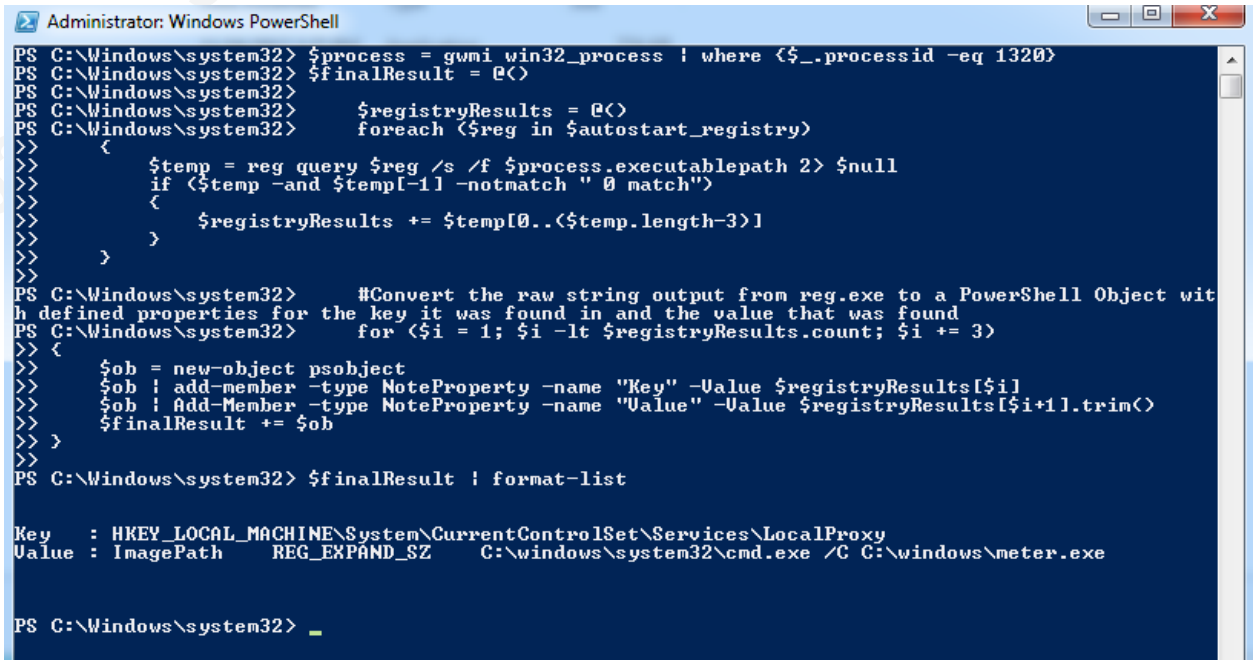


Figure 42 Creating a PowerShell object out of normalized Reg output

3.4.4. Startup Folders

Startup folders are hidden system folders in Windows that automatically execute programs and shortcuts placed within them upon user login (Azeria, n.d.). Each user has a copy of this folder, located at "C:\Users\<username>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup," which will execute the programs it contains whenever that specific user logs on. Additionally, a system-wide startup folder is located at "C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup," which will run its contents whenever any user logs on to the system.

PowerShell's Get-ChildItem cmdlet displays the contents of these directories. However, shortcuts are most common in these directories, and Get-ChildItem cannot show what program executes when the shortcut is activated. To obtain this information, a temporary ComObject must be used to take the path of a shortcut and reveal the target it runs upon execution. Once done, this data can be compared to the executable path in question. Figure 43 shows the file that is executed when the Management Interface shortcut is activated.

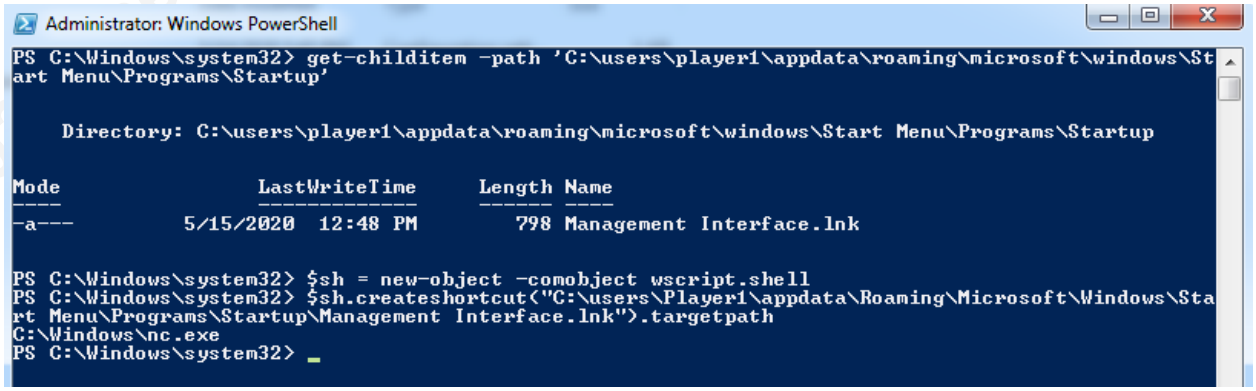


Figure 43 Creating ComObject to reveal shortcut target

To search all startup folders for references to the executable in question, enclose the above logic into a loop that searches each startup folder and save the results into a variable for later viewing. This method is shown in figure 44.

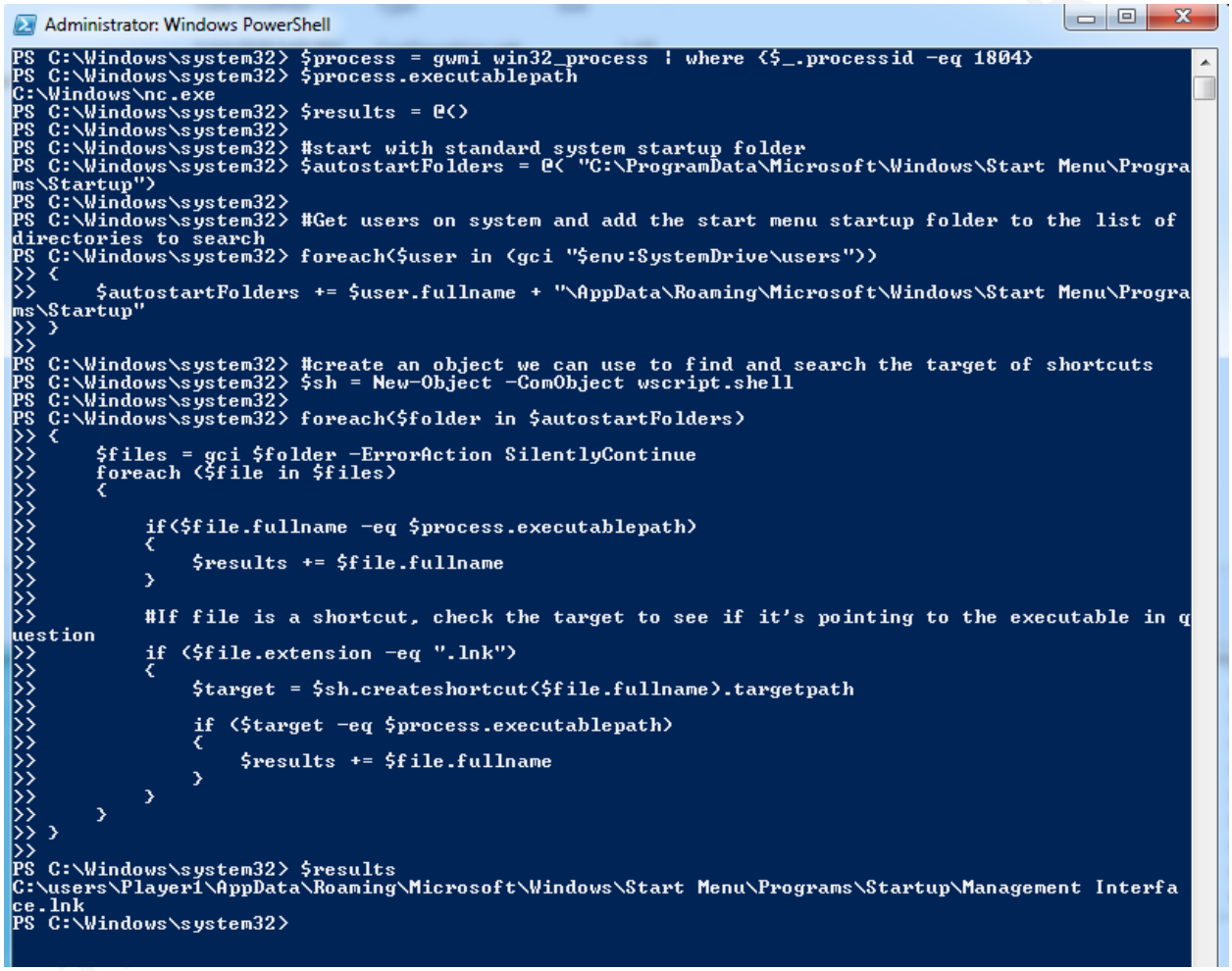


Figure 44 Searching all startup folders for the executable used by PID 1804 and displaying the results

3.5. File and System Data

Similar to persistence methods, the following items may also exist whether a process is running or not. PowerShell can provide all the data desired in this section.

3.5.1. File Creation Time

The date and time the executable was created on the system can be useful for several reasons. First, if the executable is trying to masquerade as a system executable, its creation time may be the same as another system file (Silveira, 2010). Second, many system files are created and updated at roughly the same time, so if the creation time is

much more recent or older than other system files, it may be malicious and is worth further investigation.

Get-ChildItem collects much more data than it displays by default, including the file creation time in local and Coordinated Universal Time (UTC). To access this data, simply pipe the output from Get-ChildItem to Select-Object and specify those fields.

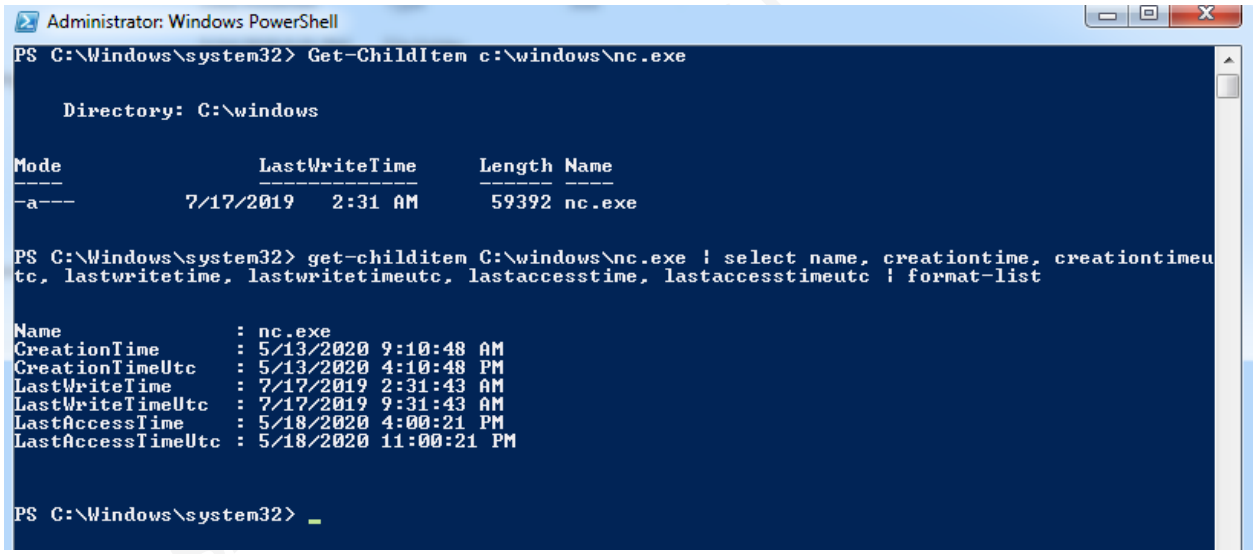


Figure 45 Using Get-ChildItem to view multiple timestamps of file

3.5.2. Files Created at Nearly the Same Time as Executable

It is common for malware and malicious actors to drop or create multiple files as part of their regular operation, which can help an investigator gain insight into the inner workings of the malware (Zeltser, 2018). These files may include scripts and configurations used by the executable in question or may be the target of various persistence mechanisms to obfuscate the relationship to the main malicious executable. For example, a scheduled task is created to run a batch file, which in turn executes the malicious executable. Searching scheduled tasks for the executable won't work, but searching for files created around the executable will return the batch file, which would provide a lead for further investigation.

Using a combination of WMI's win32_LogicalDisk class, Get-ChildItem, and a loop, we can recursively search each local hard drive for files created within a specified

timeframe of the executable in question. To obtain the list of local disks, use the PowerShell command Get-WMIObject win32_LogicalDisk and pipe the output to Where-Object searching for any drive that has a DriveType of three, meaning it is a local disk (Microsoft, 2018). Piping this output to Select-Object and expanding the property DeviceID returns a list of just the drive letters. Assigning this result to a variable enables the use of these drives within a ForEach loop.

```
PS C:\Windows\system32> $drives = Get-WmiObject win32_logicaldisk | where {$_.drivetype -eq 3} | s
elect -expand deviceid
PS C:\Windows\system32> $drives
C:
PS C:\Windows\system32>
```

Figure 46 Using WMI to determine local drives and saving the results to a variable

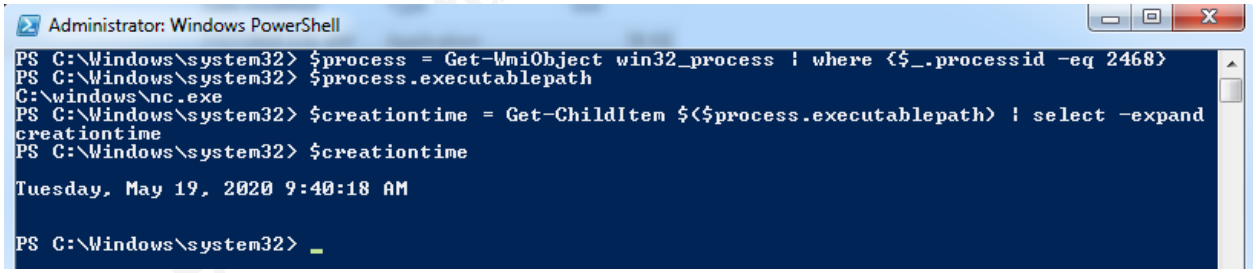


Figure 47 Saving the executable creation timestamp to a variable

Now, within a ForEach loop, recursively list all files on each drive, piping the results into Where-Object searching for files created within a time window of the file in question using the addminutes method. Figure 48 shows the search for files created five minutes before or after the creation of c:\windows\nc.exe.

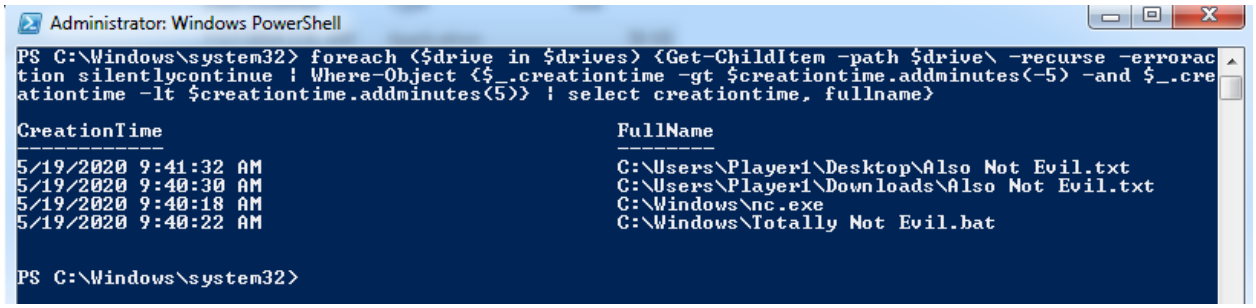


Figure 48 Files created five minutes before or after the creation of nc.exe

3.6. Information Gathering by File Path

In addition to searching for information based on running processes, much of the same data can be gathered from a given file path, including the file hash, creation time, and files created around the same time as the file in question. Searching by file path would be of great benefit if an investigator discovers a malicious process on one system and wants to search other computers for the same executable, even if the process is not currently running. These checks can be done in the same manner listed above, simply by replacing the $process.executablepath variable with a string of the executable path directly.

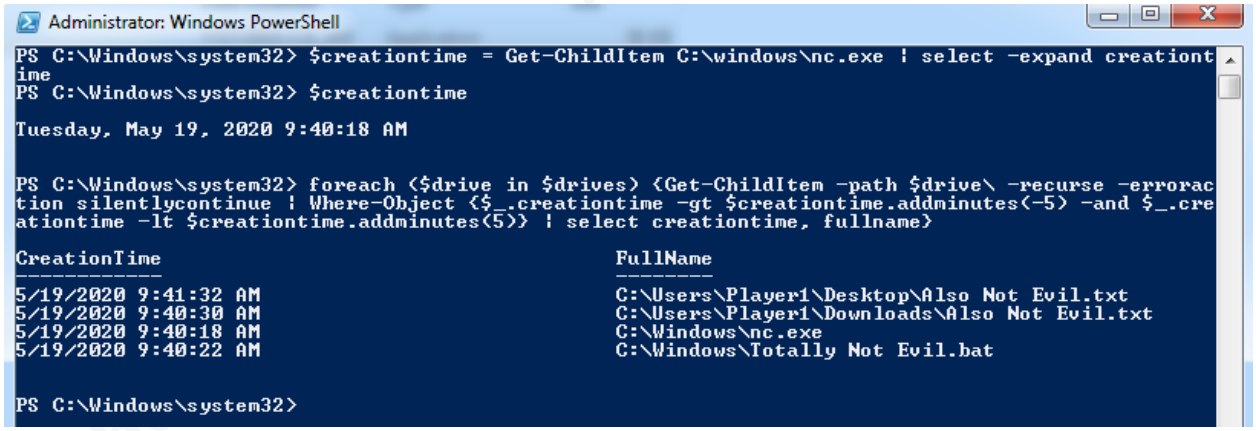


Figure 49 Files created five minutes before or after the creation of nc.exe

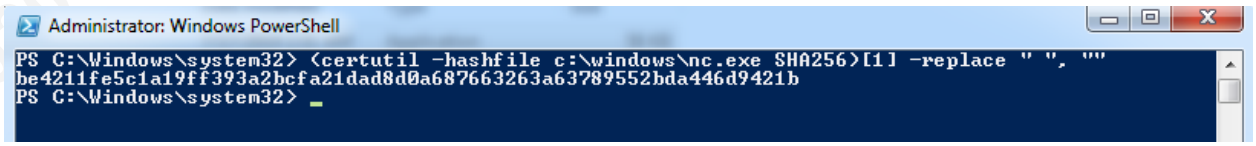


Figure 50 Finding the file hash of nc.exe

However, PowerShell and CertUtil will raise exceptions if the file does not exist. These exceptions can be overcome by putting Get-ChildItem into an If statement and saving the results in a variable. If the file does not exist, the If statement will fail, and it will execute any code in the Else statement, such as a custom error stating the file does not exist.

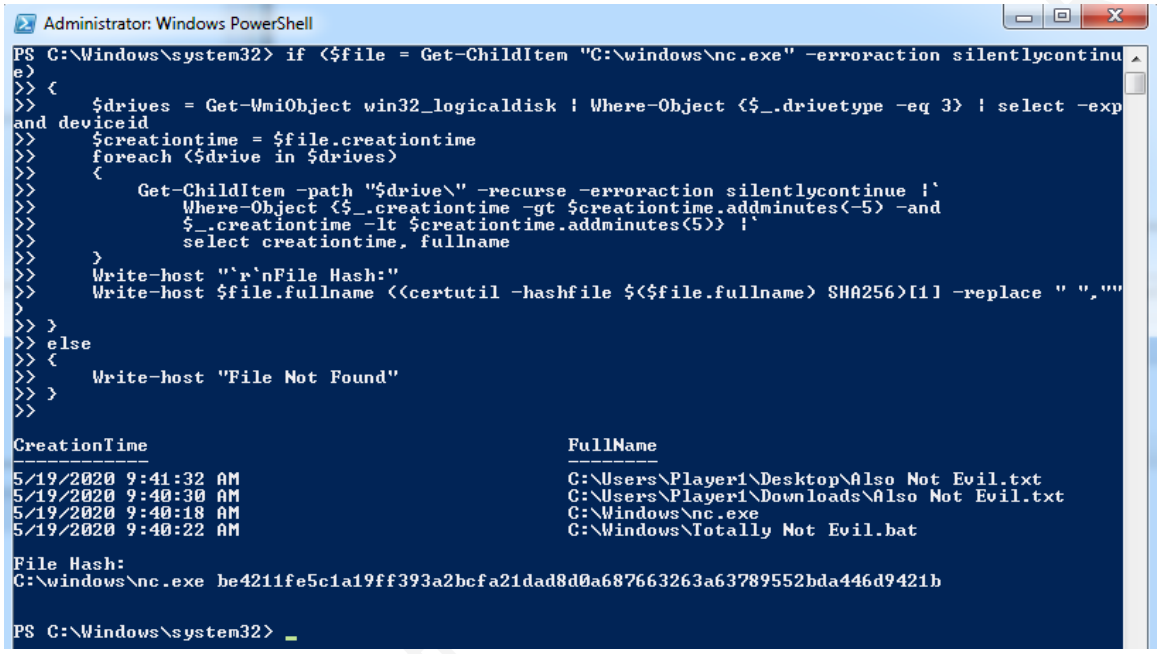


Figure 51 Results if the file exists

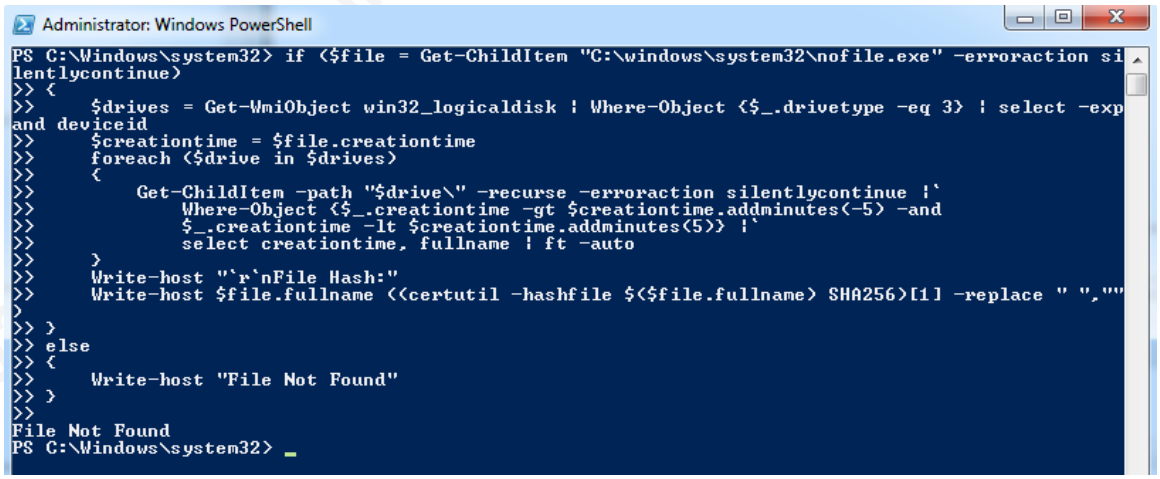


Figure 52 Results if the file does not exist

In figure 51, only the original file searched, "C:\windows\nc.exe," is hashed. This decision was made because of the possibility of hundreds of files matching the search criteria and hashing each one could significantly increase execution time.

WMI can be used to see if the executable at the given path is currently running as a process. If so, each of the data gathering methods mentioned thus far can be executed on the process. WMI may return zero, one, or multiple processes, so it is recommended to

run the initial call to Get-WMIObject as part of an If statement as described above, then iterating through each result based on PID. Figure 53 shows the information for two processes using the same executable obtained with this method.

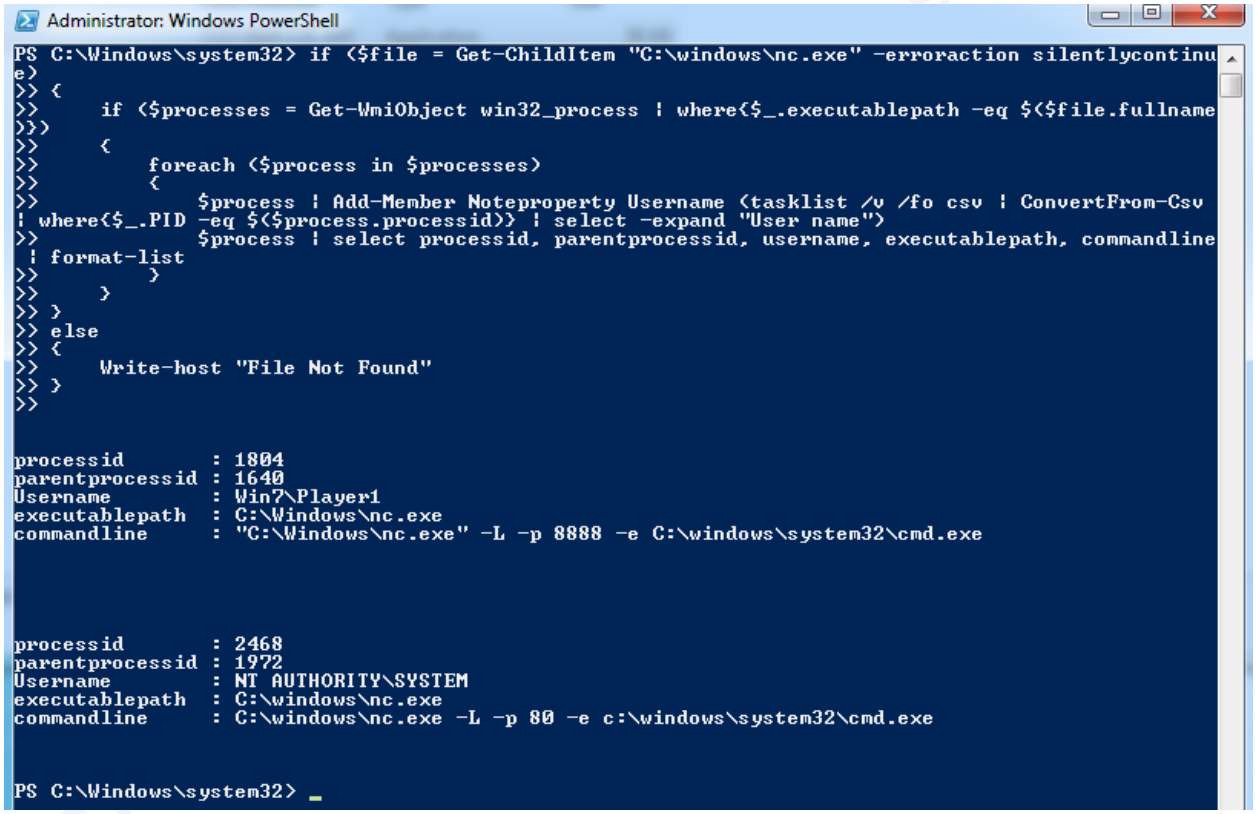


Figure 53 Using WMI to get information for all processes associated with nc.exe

3.7. Collecting Data from Remote Machines

Collecting the desired data from a remote system is simple using PowerShell's Invoke-Command cmdlet, which comes built-in to every version of PowerShell. There are two prerequisites to using Invoke-Command: 1) the system receiving the connection needs to enable Windows Remote Management (WinRM), and 2) the Trusted Hosts list on the system initiating the connection must contain the destination system's hostname or IP address.

WinRM is Microsoft's implementation of the WS-Management Protocol and enables PowerShell to be used remotely (Microsoft, 2018). WinRM is not enabled by default and must be configured before PowerShell can be used on it remotely.

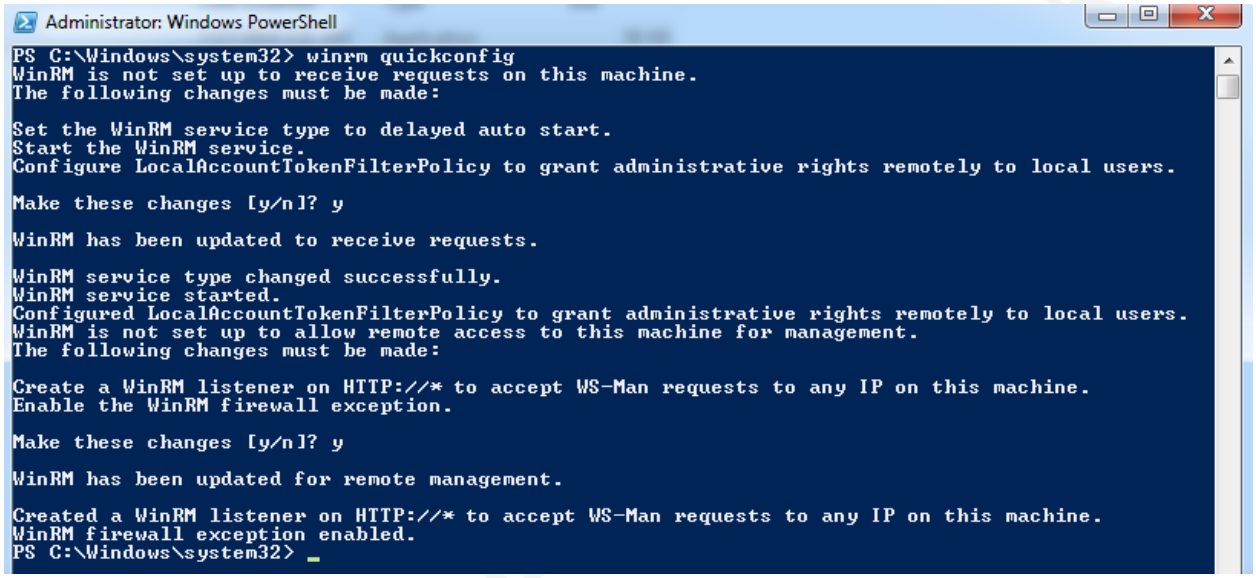


Figure 54 Setting up WinRM on the system receiving the connection

The Trusted Hosts list is, as the name implies, a list of hosts that the computer trusts. The local system must trust a remote system before it will allow a user to connect to the remote system with PowerShell. An administrator can add the "*" wildcard to the Trusted Hosts list to enable connections to any remote system, which is what was done in the lab for this paper. However, for security purposes, it is recommended that an administrator only add systems to the list that are specifically known and trusted.

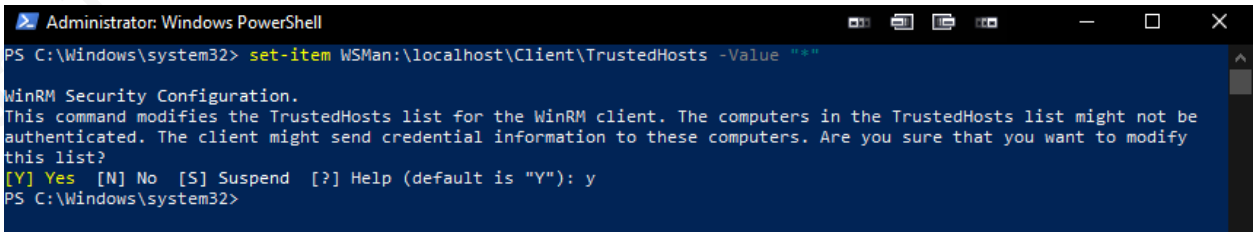


Figure 55 Setting the Trusted Hosts list on the system initiating the connection

PowerShell's Invoke-Command cmdlet runs commands or scripts on local and remote systems (Microsoft, n.d.). Multiple commands and full scripting logic can be provided when using the -ScriptBlock parameter.

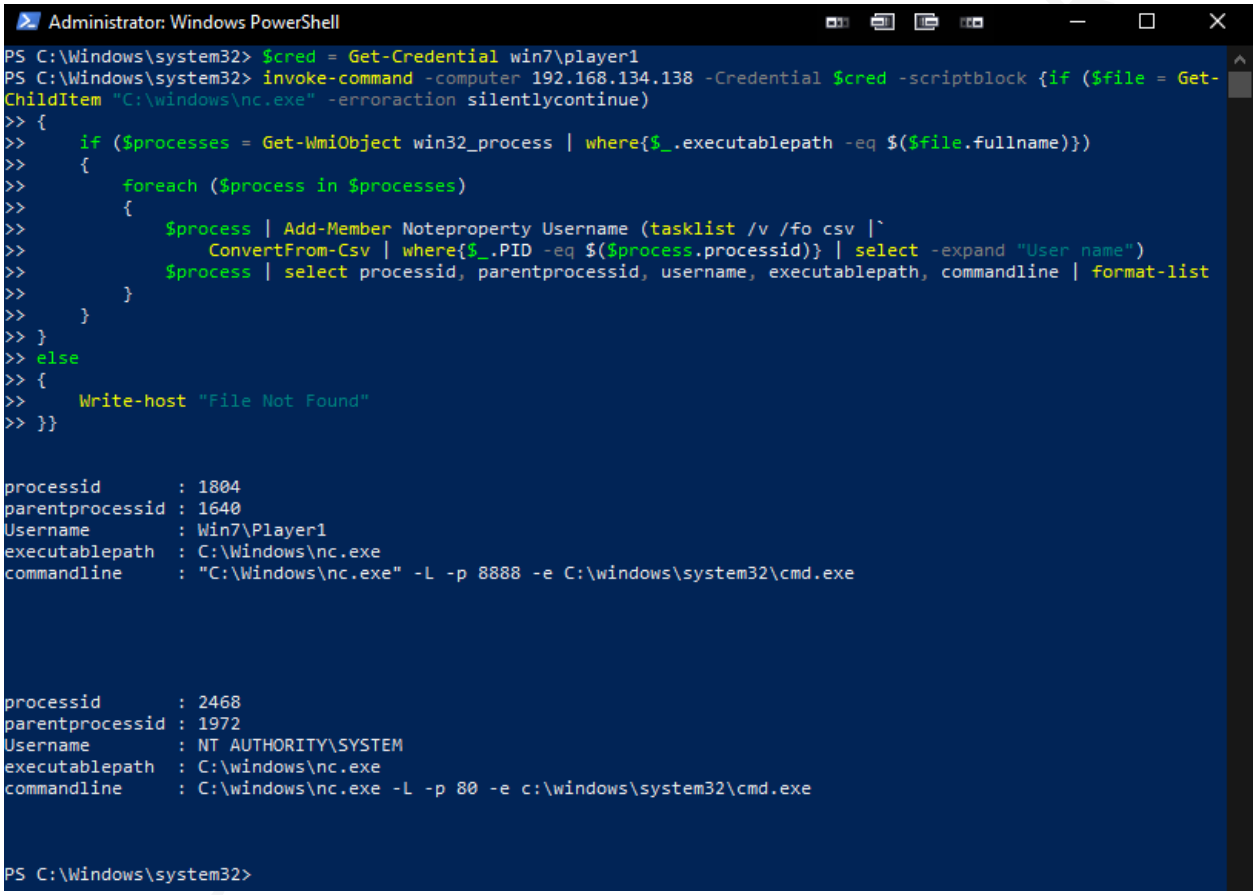


Figure 56 Running a file path search on the remote system

4. Findings and Limitations

The programs and scripts run in the previous section all worked across each version of Windows tested, both locally and remotely, except for checking VirusTotal results. Checking VirusTotal was unsuccessful on Windows 7 and Windows 2008R2 due to the Invoke-RestMethod cmdlet being unavailable on PowerShell Version 2.0.

The main limitation encountered was that using PowerShell to gather and combine this data took a significant amount of time compared to running each command individually. As shown in figures 59 and 60, when executed individually, Tasklist and Get-WMIObject Win32_process took a combined total of 214 milliseconds. When ran in a script to combine the output, it took a total of 840 milliseconds to create the process tree, and 522 milliseconds to create the flat process list, as shown in figures 57 and 58,

respectively. On a busier system, this difference would be even more noticeable. The increase in time is most likely the result of cross-referencing and extracting process data from multiple commands before displaying the results. The process tree must also perform a recursive lookup for each process, further increasing the amount of time required by the script.

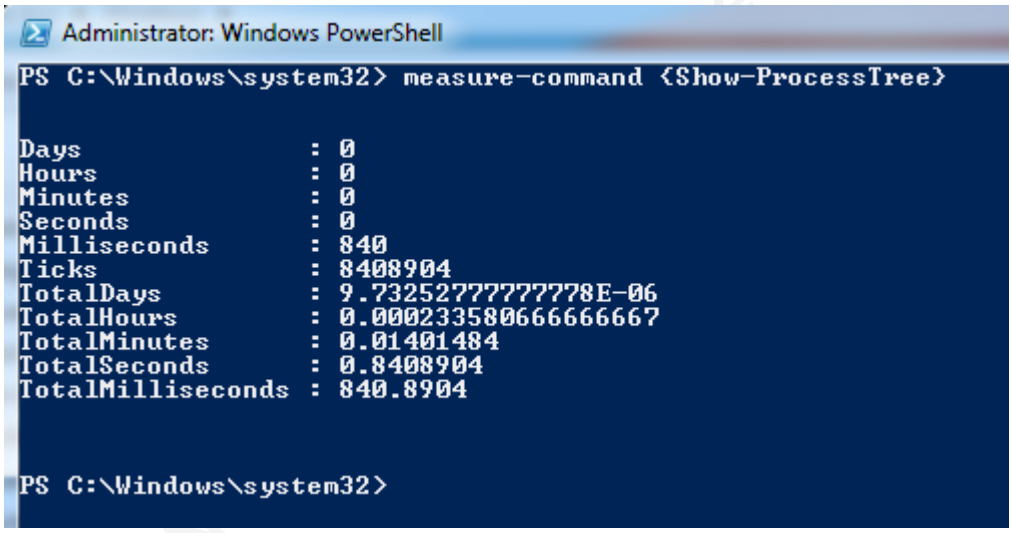


Figure 57 Process Tree function time elapsed on execution

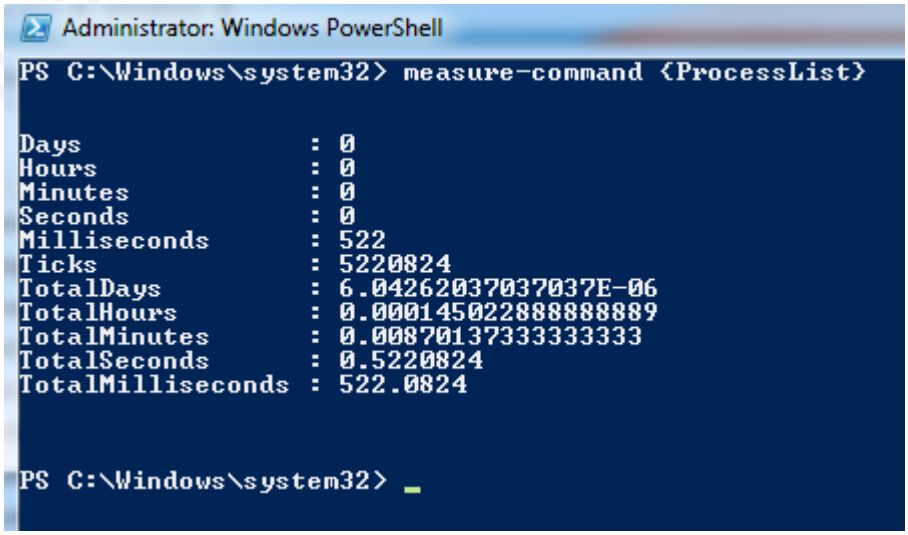


Figure 58 Process List function time elapsed on execution

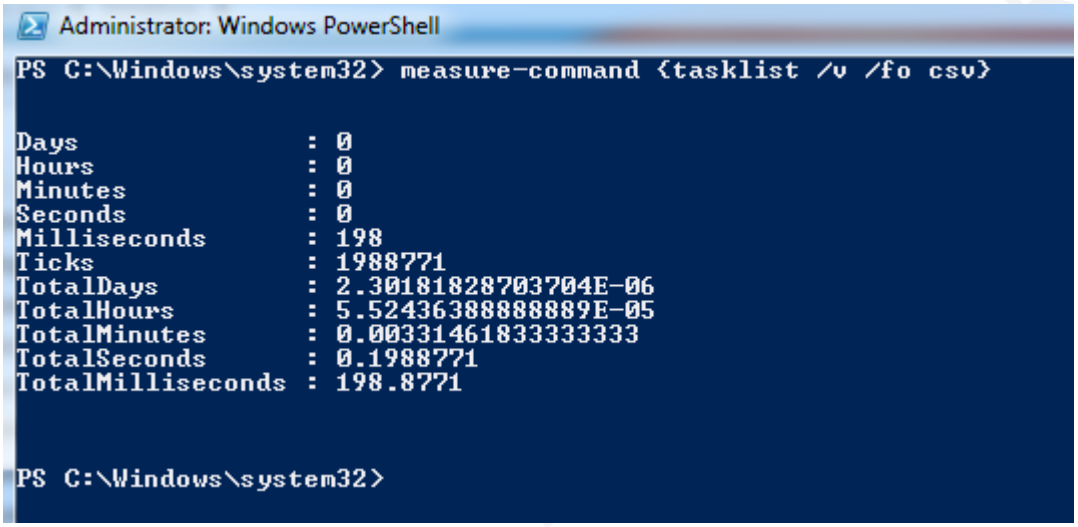


Figure 59 Tasklist execution time when ran individually

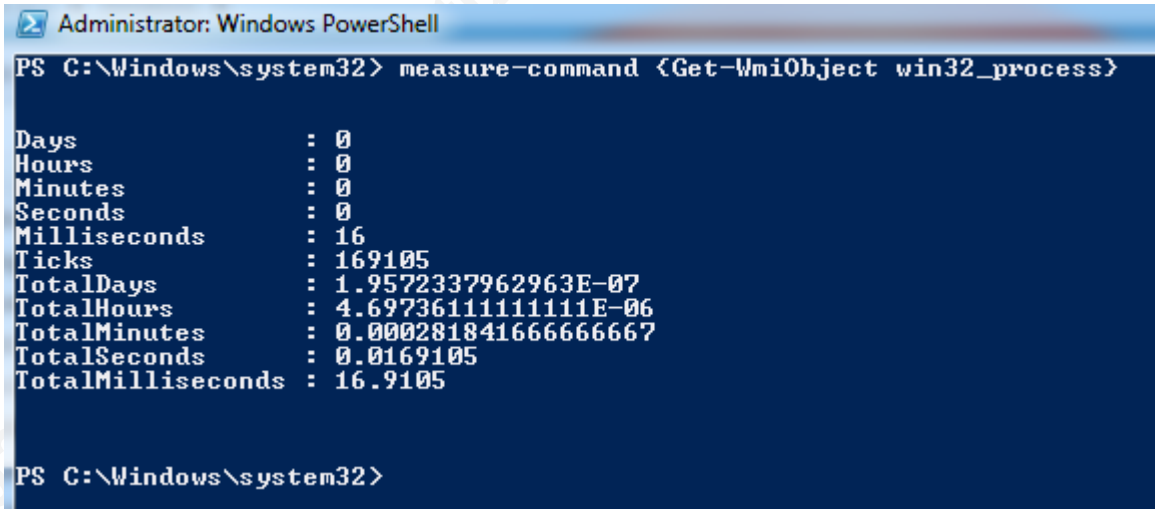


Figure 60 WMI execution time when ran individually

Additionally, searching the filesystem for files created around the same time as the file in question took 14 seconds and 759 milliseconds on the Windows 7 machine with no additional software installed or files added. This search could take significantly longer on systems with many more files to look through.

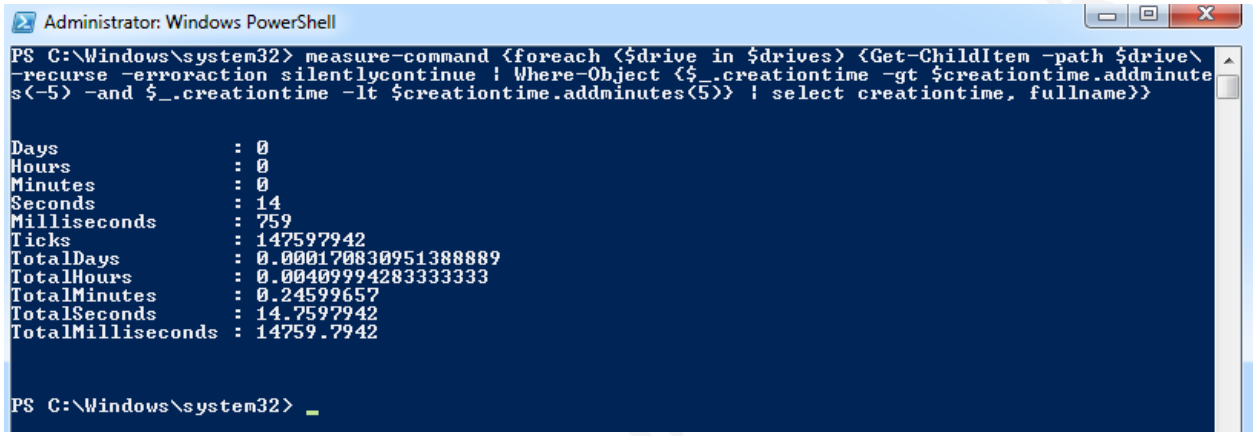


Figure 61 Filesystem search time elapsed

Furthermore, while not within this paper's scope, it is worth noting that there are no built-in Windows tools that allow an investigator to view the details of file handles opened by a process. The closest one can get with built-in tools is PowerShell's Get-Process cmdlet, which will show how many handles a process has open but not what they are.

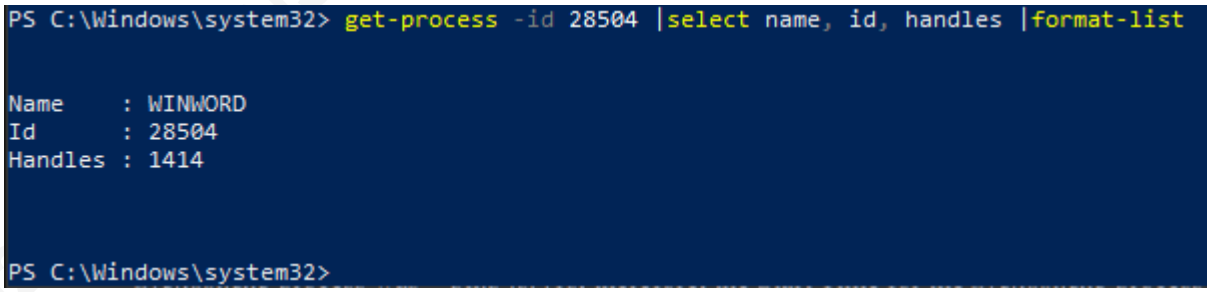


Figure 62 Get-Process showing number of file handles opened by a process

The full script created to accomplish all of these data-gathering tasks has been provided in Appendix B. This script is merely an example of one way to accomplish these tasks. There may be different and more efficient methods using these same techniques.

5. Conclusion

Despite the additional time required to combine and parse the output from several commands, it is entirely possible to obtain the data an investigator wants using only tools built-in to Windows. This capability can provide personnel working in environments that

are strict on additional software the opportunity to continue incident response investigations mostly unimpeded. Furthermore, automating the collection and combining of data for a specific process or file can help prevent crucial data from getting lost in the noise of running and reviewing each command manually.

Author Name, email@addressm

References

Arntz, P. (2015, March 23). *Scheduled Tasks*. Retrieved May 24, 2020, from Malwarebytes: https://blog.malwarebytes.com/cybercrime/2015/03/scheduled-tasks/

Azeria. (n.d.). *Persistence*. Retrieved May 24, 2020, from Azeria Labs: https://azeria-labs.com/persistence/

Fortuna, A. (2017, July 06). *Malware persistence techniques*. Retrieved May 24, 2020, from Andrea Fortuna: https://www.andreafortuna.org/2017/07/06/malware-persistence-techniques/

Heddings, L. (2019, October 15). *Sysinternals Pro: What Are the SysInternals Tools and How Do You Use Them?* Retrieved May 24, 2020, from How-to Geek: https://www.howtogeek.com/school/sysinternals-pro/lesson1/

Heddings, L. (2019, April 30). *Understanding and Managing Windows Services*. Retrieved May 24, 2020, from How-To Geek: https://www.howtogeek.com/school/using-windows-admin-tools-like-a-pro/lesson8/

Hoffman, C. (2018, August 23). *What Are MD5, SHA-1, and SHA-256 Hashes, and How Do I Check Them?* Retrieved May 24, 2020, from How-To Geek: https://www.howtogeek.com/67241/htg-explains-what-are-md5-sha-1-hashes-and-how-do-i-check-them/

Kazun. (2013, March 15). *How to recursively print process, parent process, grand parent process, great grand parent process?* Retrieved May 24, 2020, from Microsoft Technet: https://social.technet.microsoft.com/Forums/windowsserver/en-US/87b5e231-4832-43ca-92ed-0ab70b6e6726/how-to-recursively-print-process-parent-process-grand-parent-process-great-grand-parent-process?forum=winserverpowershell

Lee, R., & Pilkington, M. (2018). *Find Evil - Know Normal.* Retrieved May 26, 2020, from SANS: https://digital-forensics.sans.org/media/SANS_Poster_2018_Hunt_Evil_FINAL.pdf

Author Name, email@addressm

Microsoft. (2006, July 18). *Microsoft Acquires Winternals Software - Stories*. Retrieved May 24, 2020, from Microsoft: https://news.microsoft.com/2006/07/18/microsoft-acquires-winternals-software/

Microsoft. (2006, June 30). *WMI Scripting Primer: Part 1*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/scripting-articles/ms974579(v=msdn.10)

Microsoft. (2010, September 20). *Install Windows PowerShell 2.0*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/previous-versions/appfabric/ff637750(v=azure.10)

Microsoft. (2012, February 01). *Creating table using Powershell*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/archive/blogs/rkramesh/creating-table-using-powershell

Microsoft. (2017, October 16). *certutil*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/certutil

Microsoft. (2017, March 30). *Introduction to Windows Service Applications*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications

Microsoft. (2017, October 16). *netstat*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/netstat

Microsoft. (2017, October 16). *reg*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/reg

Microsoft. (2017, October 16). *Tasklist*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/tasklist

Microsoft. (2018, May 31). *Processes and Threads - Win32 apps.* Retrieved from Microsoft: https://docs.microsoft.com/en-us/windows/win32/procthread/processes-and-threads

Author Name, email@addressm

Microsoft. (2018, May 31). *Run and RunOnce Registry Keys*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows/win32/setupapi/run-and-runonce-registry-keys

Microsoft. (2018, May 31). *Schtasks.exe - Win32 apps*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows/win32/taskschd/schtasks

Microsoft. (2018, May 31). *Win32_LogicalDisk class - Win32 apps*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows/win32/cimwin32prov/win32-logicaldisk

Microsoft. (2018, May 31). *Windows Management Instrumentation - Win32 apps*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page

Microsoft. (2018, May 31). *Windows Remote Management - Win32 apps*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/windows/win32/winrm/portal

Microsoft. (2019, December 17). *What is a DLL?* Retrieved May 24, 2020, from Microsoft: https://support.microsoft.com/en-us/help/815065/what-is-a-dll

Microsoft. (2020, May 22). *What is PowerShell?* Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7

Microsoft. (n.d.). *Invoke-Command*. Retrieved May 24, 2020, from Microsoft: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/invoke-command?view=powershell-7

Nolan, R., Baker, M., Branson, J., Hammerstein, J., Rush, K., Waits, C., & Schweinsberg, E. (2005, September). *First Responders Guide to Computer Forensics: Advanced Topics.* Retrieved May 23, 2020, from Software Engineer Institute - Carnegie Mellon University: https://resources.sei.cmu.edu/asset_files/Handbook/2005_002_001_14432.pdf

Phan, T. (2015, December 10). *A Crash Course in DLL Hijacking*. Retrieved May 24, 2020, from Fortinet: https://www.fortinet.com/blog/industry-trends/a-crash-course-in-dll-hijacking.html

Author Name, email@addressm

Silveira, C. (2010, August 18). *Benefits of using multiple timestamps during timeline analysis in digital forensics*. Retrieved May 24, 2020, from SANS: https://www.sans.org/blog/benefits-of-using-multiple-timestamps-during-timeline-analysis-in-digital-forensics/

VirusTotal. (n.d.). *How it works*. Retrieved May 24, 2020, from VirusTotal: https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works

VirusTotal. (n.d.). *What is the difference between the public API and the private API?* Retrieved May 24, 2020, from VirusTotal: https://support.virustotal.com/hc/en-us/articles/115002119845-What-is-the-difference-between-the-public-API-and-the-private-API-

Weyne, F. (2016, September). *Analyzing malicious office documents*. Retrieved May 26, 2020, from Uperesia: https://www.uperesia.com/analyzing-malicious-office-documents

Wilson, E. (2015, September 14). *Backwards Compatibility in PowerShell*. Retrieved May 24, 2020, from Microsoft: https://devblogs.microsoft.com/scripting/backwards-compatibility-in-powershell/

Yonts, J. (2014, August 25). *Digging for Malware: Suspicious Filesystem Geography*. Retrieved from Malicious Streams: http://www.malicious-streams.com/resources/articles/DGMW1_Suspicious_FS_Geography.html

Zeltser, L. (2018). Malware Analysis Fundamentals. *FOR610 | Reverse-engineering Malware: Malware Analysis Tools and Techniques*. SANS.

Author Name, email@addressm

Appendix A – Autostart Registry Keys

HKLM\System\CurrentControlSet\Control\Session Manager\BootExecute

HKLM\Software\Microsoft\Office\PowerPoint\Addins

HKLM\Software\Wow6432Node\Microsoft\Office\PowerPoint\Addins

HKLM\Software\Microsoft\Office\Word\Addins

HKLM\Software\Wow6432Node\Microsoft\Office\Word\Addins

HKLM\SOFTWARE\Classes\Htmlfile\Shell\Open\Command\(Default)

HKLM\System\CurrentControlSet\Services

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Font Drivers

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credential Providers

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credential Provider Filters

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\PLAP Providers

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GpExtensions

HKLM\SYSTEM\CurrentControlSet\Control\Print\Monitors

HKLM\SYSTEM\CurrentControlSet\Control\Print\Providers

HKLM\SYSTEM\CurrentControlSet\Control\SecurityProviders\SecurityProviders

HKLM\SYSTEM\CurrentControlSet\Control\Lsa\Authentication Packages

HKLM\SYSTEM\CurrentControlSet\Control\Lsa\Notification Packages

HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\Order

HKLM\System\CurrentControlSet\Services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries

HKLM\System\CurrentControlSet\Services\WinSock2\Parameters\NameSpace_Catalog5\Catalog_Entries

HKLM\System\CurrentControlSet\Services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries64

HKLM\System\CurrentControlSet\Services\WinSock2\Parameters\NameSpace_Catalog5\Catalog_Entries64

Author Name, email@addressm

HKLM\System\CurrentControlSet\Control\Terminal Server\Wds\rdpwd\StartupPrograms
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\VmApplet
HKLM\System\CurrentControlSet\Control\Session Manager\KnownDlls
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Run
HKLM\SOFTWARE\Classes\Protocols\Filter
HKLM\SOFTWARE\Classes\Protocols\Handler
HKLM\SOFTWARE\Microsoft\Active Setup\Installed Components
HKLM\SOFTWARE\Wow6432Node\Microsoft\Active Setup\Installed Components
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\IconServiceLib
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellServiceObjects
HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Explorer\Shel
lServiceObjects
HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects
HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Explorer\Browser
Helper Objects
HKLM\Software\Classes*\ShellEx\ContextMenuHandlers
HKLM\Software\Classes\Drive\ShellEx\ContextMenuHandlers
HKLM\Software\Classes*\ShellEx\PropertySheetHandlers
HKLM\Software\Classes\AllFileSystemObjects\ShellEx\ContextMenuHandlers
HKLM\Software\Classes\AllFileSystemObjects\ShellEx\PropertySheetHandlers
HKLM\Software\Classes\Directory\ShellEx\ContextMenuHandlers
HKLM\Software\Classes\Directory\Shellex\DragDropHandlers
HKLM\Software\Classes\Directory\Shellex\PropertySheetHandlers
HKLM\Software\Classes\Directory\Shellex\CopyHookHandlers
HKLM\Software\Classes\Directory\Background\ShellEx\ContextMenuHandlers
HKLM\Software\Classes\Folder\ShellEx\ContextMenuHandlers
HKLM\Software\Classes\Folder\ShellEx\DragDropHandlers

Author Name, email@addressm

HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers

HKLM\Software\Microsoft\Internet Explorer\Extensions

HKLM\Software\Wow6432Node\Microsoft\Internet Explorer\Extensions

HKLM\Software\Microsoft\Windows NT\CurrentVersion\Drivers32

HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Drivers32

HKLM\Software\Classes\CLSID\{083863F1-70DE-11d0-BD40-00A0C911CE86}\Instance

HKLM\Software\Wow6432Node\Classes\CLSID\{083863F1-70DE-11d0-BD40-00A0C911CE86}\Instance

HKLM\Software\Classes\CLSID\{7ED96837-96F0-4812-B211-F13C24117ED3}\Instance

HKLM\Software\Wow6432Node\Classes\CLSID\{7ED96837-96F0-4812-B211-F13C24117ED3}\Instance

HKCU\Control Panel\Desktop\Scrnsave.exe

HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

HKCU\Software\Microsoft\Internet Explorer\UrlSearchHooks

Appendix B – Sample Script

```
param([string]$computer=$null,[int]$p=$null,[switch]$tree,[int]$search=$null,[string]$filepath=$null,$cred,[switch]$table=$null,[switch]$list=$null,$outpath=$null,[switch]$vt=$null,[switch]$h=$null,[switch]$help=$null,[switch]$raw=$null)

$vtsleep = 15
$vtAPIKey = "" #Insert your VirusTotal API key here


if($h -or $help)
{
   $command = $MyInvocation.MyCommand

   write-host @"

   Usage: $command [options]

   When executed without options, will display a list of all running processes as a flat list.

   When executed given a PID or Filepath, will search the file system for the process' lineage, loaded DLLs, Netstat results,
   scheduled tasks, services, Autostart Registry and folder matches, and file timestamps. Additional Data can be retrieved with more options.

   Main Options:

   <no option>          Displays all running processes as a flat list. Not compatible with search, -p, filepath, or vt.
   -tree                Display All running processes in tree format. Not compatible with search, -p, filepath, or vt. Defaults to Table output format
   -p [PID]             Process ID to get information for. Not compatible with -filepath. Takes precendence over -filepath.
   -filepath [filename]   Find information related to [filename]. If running as a process, get process information as well. Not Compatible with -p.

   Additional Options:
   -computer            Computer Name or IP address to run this script against
   -cred [var]          Supply your own Get-Credential variable for the script to use when accessing remote systems
   -search [min]        Search the filesystem for other files created within [min] minutes of the file or executable in quetsion
   -vt                  Check Non-Microsoft DLLs against VirusTotal. Valid API key must be set within script.
                        VirusTotal lookups will always be done from the local system, even if script is executed against a remote machine.

Author Name, email@addressm
```

```
   -h                Display this Help message
   -help             Display this help message

   Output Format Options:
   -table            Display results in Format-Table format. Not Compatible with -list
option
   -list             Display results in Format-List format. Not compatible with -tree option.
This is the default option if none other specified.
   -outpath [path]       Save each type of result as separate CSV file in [path]
   -raw              Output results without formatting. Use this if you want to save to your
own variable to use or parse separately.

   Usage Examples:
   $command
   $command -tree
   $command -p 1492
   $command -p 1492 -search 5 -vt
   $command -filepath "C:\windows\system32\svchost.exe" -table
   $command -computer 192.168.1.10 -p 2468 -vt -search 5 -table -outpath "C:\temp\"

"@

   exit
}

#check if given the outpath variable, and if so, append a slash to the end if the path given
doesn't end with one.
if($outpath)
{
   if($outpath[-1] -ne "\")
   {
      $outpath += "\"
   }
}

#This function encompassases all of the checks to be run regardless of option. This is
wrapped in a function so that it
# can easily be executed against a remote machine with the invoke-command function.
Function FullScript($p, $tree, $search, $filepath)
{

$global:process_Table = $null

#Expects Get-WmiObject win32_process formatted process
#Creating a custom table to hold the consolidated process information from multiple
commands


Author Name, email@addressm
```

```
Function CreateProcessTable($process)
{
   #if table doesn't already exist, create it. otherwise just add to it
   if(!$global:process_table)
   {
      # Creating custom table -- https://docs.microsoft.com/en-
us/archive/blogs/rkramesh/creating-table-using-powershell
      $global:process_table = New-Object System.Data.DataTable "Processes"
      $global:process_table.Columns.Add((New-Object system.data.datacolumn
Name,([string])))
      $global:process_table.Columns.Add((New-Object system.data.datacolumn
PID,([int])))
      $global:process_table.Columns.Add((New-Object system.data.datacolumn
PPID,([int])))
      $global:process_table.Columns.Add((New-Object system.data.datacolumn
ProcessUser,([string])))
      $global:process_table.Columns.Add((New-Object system.data.datacolumn
ExecutablePath,([string])))
      $global:process_table.Columns.Add((New-Object system.data.datacolumn
CommandLine,([string])))
   }

   #add information to new process information table
   $row = $global:process_table.NewRow()

   $row.name = $process.indentedname
   $row.PID = $process.processId
   $row.PPID = $process.parentprocessid
   $row.ProcessUser = $process.Username
   $row.ExecutablePath = $process.executablepath
   $row.CommandLine = $process.commandline

   $global:process_table.rows.add($row)

}

Function GetTasklist()
{
   #get process info from tasklist to get user name data
   return (tasklist /v /fo csv | convertfrom-csv)
}


#found at https://social.technet.microsoft.com/Forums/windowsserver/en-US/87b5e231-
4832-43ca-92ed-0ab70b6e6726/how-to-recursively-print-process-parent-process-grand-
parent-process-great-grand-parent-process?forum=winserverpowershell


Author Name, email@addressm
```

```
###STart technet code. adding my own along the way. look at process-tree-technet script
to get original code

Function Show-ProcessTree
{
   $mytasklist = GetTasklist

   Function Get-ProcessChildren($P,$Depth=1)
   {
      $procs | Where-Object {$_.ParentProcessId -eq $p.ProcessID -and
$_.ParentProcessId -ne 0} | ForEach-Object {

         $indentedname = "{0}|--{1}" -f (" "*3*$Depth),$_.Name
         $_ | Add-Member NoteProperty IndentedName $indentedname
         $thispid = $_.processid
         if(!$_.Username)
         {
            $_ | Add-Member NoteProperty Username  ($mytasklist | where-object
{$_.PID -eq $thispid} | select -expand "User Name")
         }
         CreateProcessTable($_)
         Get-ProcessChildren $_ (++$Depth)
         $Depth--
      }
   }


   $filter = {-not (Get-Process -Id $_.ParentProcessId -ErrorAction SilentlyContinue) -or
$_.ParentProcessId -eq 0}
   $procs = Get-WmiObject Win32_Process
   $top = $procs | Where-Object $filter | Sort-Object ProcessID

   foreach ($p in $top)
   {

      $p | Add-Member NoteProperty IndentedName $p.name
      $p | Add-Member NoteProperty Username ($mytasklist | Where-Object {$_.PID -eq
$p.processid} | select -expand "User Name")

      CreateProcessTable($p)

      Get-ProcessChildren $p
   }
}

##########END PROCESS TREE CODE FROM TECHNET


Author Name, email@addressm
```

```
Function MyProcessList
{

   $mytasklist = GetTasklist

   $mywmi = gwmi win32_process

   foreach ($process in $mywmi)
   {
      $process | Add-Member NoteProperty Indentedname $process.name
      $process | Add-Member NoteProperty Username ($mytasklist | Where-Object
{$_.PID -eq $process.processid} | select -expand "User Name")

      CreateProcessTable($process)
   }

}


#Powershell V2 compatible way to get scheduled tasks.
Function GetTasks($process){
   $tasks = schtasks /query /fo csv /v | convertfrom-csv

   if ($taskResult = $tasks | Where-Object {$_."Task to run" -like
$process.executablepath + "*"} )
   {
      $taskResult
   }
}


Function SearchRegistry($process){

   $finalResult = @()

   $registryResults = @()
   foreach ($reg in $autostart_registry)
   {
      $temp = reg query $reg /s /f $process.executablepath 2> $null
      if ($temp -and $temp[-1] -notmatch " 0 match")
      {
         $registryResults += $temp[0..($temp.length-3)]
      }
   }


Author Name, email@addressm
```

```
   #Convert the raw string output from reg.exe to a PowerShell Object with defined
properties for the key it was found in and the value that was found
   for ($i = 1; $i -lt $registryResults.count; $i += 3)
{
   $ob = new-object psobject
   $ob | add-member -type NoteProperty -name "Key" -Value $registryResults[$i]
   $ob | Add-Member -type NoteProperty -name "Value" -Value
$registryResults[$i+1].trim()
   $finalResult += $ob
}

   return $FinalResult
}

Function AutostartFolders($process)
{
   $results = @()

   #start with standard system startup folder
   $autostartFolders = @( "C:\ProgramData\Microsoft\Windows\Start
Menu\Programs\Startup")

   #Get users on system and add the start menu startup folder to the list of directories to
search
   foreach($user in (gci "$env:SystemDrive\users"))
   {
      $autostartFolders += $user.fullname +
"\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup"
   }

   #create an object we can use to find and search the target of shortcuts
   $sh = New-Object -ComObject wscript.shell

   foreach($folder in $autostartFolders)
   {
      $files = gci $folder -ErrorAction SilentlyContinue
      foreach ($file in $files)
      {

         if($file.fullname -eq $process.executablepath)
         {
            $results += $file.fullname
         }
```

```
        #If file is a shortcut, check the target to see if it's pointing to the executable in
question
        if ($file.extension -eq ".lnk")
        {
           $target = $sh.createshortcut($file.fullname).targetpath

           if ($target -eq $process.executablepath)
           {
              $results += $file.fullname
           }
        }
     }
   }

   return $results
}

Function SearchFilesystem($process){
   $drives = gwmi win32_logicaldisk | Where-Object {$_.drivetype -eq 3} | select -
expand deviceid
    $suspectExecutable = gci $process.executablepath -ErrorAction SilentlyContinue

    $results = @()

   if (!$suspectExecutable)
   {
     return
   }

   if (!$search)
   {
      return $suspectExecutable
   }

   if($search -gt 0)
   {
      foreach ($drive in $drives)
      {
         #note the added '\' to the gci command. this is required for the filesystem search
will fail.
         $results += gci -recurse "$drive\" -ErrorAction SilentlyContinue | Where-Object
{$_.creationtime -gt $suspectExecutable.creationtime.addminutes(-$search) -and
$_.creationtime -lt $suspectExecutable.creationtime.addminutes($search)} | select
creationtime, fullname
      }
   }
```

Author Name, email@addressm

```
    return $results
}

#All Registry autostart locations used by autorunsc64.exe (version 13.96 executed on
windows 10)
$autostart_registry = @("HKLM\System\CurrentControlSet\Control\Session
Manager\BootExecute",
"HKLM\Software\Microsoft\Office\PowerPoint\Addins",
"HKLM\Software\Wow6432Node\Microsoft\Office\PowerPoint\Addins",
"HKLM\Software\Microsoft\Office\Word\Addins",
"HKLM\Software\Wow6432Node\Microsoft\Office\Word\Addins",
"HKLM\SOFTWARE\Classes\Htmlfile\Shell\Open\Command\(Default)",
"HKLM\System\CurrentControlSet\Services",
"HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Font Drivers",
"HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credential
Providers",
"HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credential
Provider Filters",
"HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\PLAP
Providers",
"HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon\GpExtensions",
"HKLM\SYSTEM\CurrentControlSet\Control\Print\Monitors",
"HKLM\SYSTEM\CurrentControlSet\Control\Print\Providers",
"HKLM\SYSTEM\CurrentControlSet\Control\SecurityProviders\SecurityProviders",
"HKLM\SYSTEM\CurrentControlSet\Control\Lsa\Authentication Packages",
"HKLM\SYSTEM\CurrentControlSet\Control\Lsa\Notification Packages",
"HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\Order",
"HKLM\System\CurrentControlSet\Services\WinSock2\Parameters\Protocol_Catalog9\C
atalog_Entries",
"HKLM\System\CurrentControlSet\Services\WinSock2\Parameters\NameSpace_Catalog
5\Catalog_Entries",
"HKLM\System\CurrentControlSet\Services\WinSock2\Parameters\Protocol_Catalog9\C
atalog_Entries64",
"HKLM\System\CurrentControlSet\Services\WinSock2\Parameters\NameSpace_Catalog
5\Catalog_Entries64",
"HKLM\System\CurrentControlSet\Control\Terminal
Server\Wds\rdpwd\StartupPrograms",
"HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit",
"HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\VmApplet",
"HKLM\System\CurrentControlSet\Control\Session Manager\KnownDlls",
"HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell",
"HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell",
"HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run",
"HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Run",
"HKLM\SOFTWARE\Classes\Protocols\Filter",


Author Name, email@addressm
```

"HKLM\SOFTWARE\Classes\Protocols\Handler",
"HKLM\SOFTWARE\Microsoft\Active Setup\Installed Components",
"HKLM\SOFTWARE\Wow6432Node\Microsoft\Active Setup\Installed Components",
"HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\IconServiceLib",
"HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellServiceObject
s",
"HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Explorer\Sh
ellServiceObjects",
"HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper
Objects",
"HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Explorer\Browse
r Helper Objects",
"HKLM\Software\Classes*\ShellEx\ContextMenuHandlers",
"HKLM\Software\Classes\Drive\ShellEx\ContextMenuHandlers",
"HKLM\Software\Classes*\ShellEx\PropertySheetHandlers",
"HKLM\Software\Classes\AllFileSystemObjects\ShellEx\ContextMenuHandlers",
"HKLM\Software\Classes\AllFileSystemObjects\ShellEx\PropertySheetHandlers",
"HKLM\Software\Classes\Directory\ShellEx\ContextMenuHandlers",
"HKLM\Software\Classes\Directory\Shellex\DragDropHandlers",
"HKLM\Software\Classes\Directory\Shellex\PropertySheetHandlers",
"HKLM\Software\Classes\Directory\Shellex\CopyHookHandlers",
"HKLM\Software\Classes\Directory\Background\ShellEx\ContextMenuHandlers",
"HKLM\Software\Classes\Folder\ShellEx\ContextMenuHandlers",
"HKLM\Software\Classes\Folder\ShellEx\DragDropHandlers",
"HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentif
iers",
"HKLM\Software\Microsoft\Internet Explorer\Extensions",
"HKLM\Software\Wow6432Node\Microsoft\Internet Explorer\Extensions",
"HKLM\Software\Microsoft\Windows NT\CurrentVersion\Drivers32",
"HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Drivers32",
"HKLM\Software\Classes\CLSID\{083863F1-70DE-11d0-BD40-
00A0C911CE86}\Instance",
"HKLM\Software\Wow6432Node\Classes\CLSID\{083863F1-70DE-11d0-BD40-
00A0C911CE86}\Instance",
"HKLM\Software\Classes\CLSID\{7ED96837-96F0-4812-B211-
F13C24117ED3}\Instance",
"HKLM\Software\Wow6432Node\Classes\CLSID\{7ED96837-96F0-4812-B211-
F13C24117ED3}\Instance",
"HKCU\Control Panel\Desktop\Scrnsave.exe",
"HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run",
"HKCU\Software\Microsoft\Internet Explorer\UrlSearchHooks")

Function GetProcessLineage($process)
{

Author Name, email@addressm

```
    #clearing process table variable to prevent conflicts from multiple runs based off file
path
    Clear-Variable process_ table -scope Global

  do
  {
    $mytasklist = GetTasklist
    $ppid = $process.ParentProcessId

    $process | Add-Member NoteProperty IndentedName $process.name
    $process | Add-Member NoteProperty Username ($mytasklist | Where-Object
{$_.PID -eq $process.processid} | select -expand "User Name")



    CreateProcessTable($process)

  }while($process = Get-WmiObject win32_process -Filter "processid=$ppid")

}


Function GetNetworkActivity($p)
{
  $results = @()

  #To make this script PowerShell v2.0 compatible, we need to use netstat and manually
parse the output
  #instead of using the newer Get-NetTCPConnection and Get-NetUDPEndpoint
Cmdlets
  $net = netstat -ano | select-string "\b$p$"
  if($net)
  {
    foreach ($line in $net)
    {
      $temp = $line -split '\s+'

      $ConnectionObject = New-Object -TypeName psobject
      $ConnectionObject | Add-Member -MemberType NoteProperty -Name PID $p
      $ConnectionObject | Add-Member -MemberType NoteProperty -Name Protocol
$temp[1]

      #using substring to handle IPv6 Addresses
      $ConnectionObject | Add-Member -MemberType NoteProperty -Name
LocalAddress $temp[2].substring(0, $temp[2].lastindexof(':'))
```

Author Name, email@addressm

```
        $ConnectionObject | Add-Member -MemberType NoteProperty -Name LocalPort
$temp[2].split(':')[-1]

        if ($temp[1] -eq "TCP")
        {
           $ConnectionObject | Add-Member -MemberType NoteProperty -Name
ForeignAddress $temp[3].substring(0, $temp[2].lastindexof(':'))
           $ConnectionObject | Add-Member -MemberType NoteProperty -Name
ForeignPort $temp[3].split(':')[-1]
           $ConnectionObject | Add-Member -MemberType NoteProperty -Name State
$temp[4]
        }
        else
        {
           $ConnectionObject | Add-Member -MemberType NoteProperty -Name
ForeignAddress $null
           $ConnectionObject | Add-Member -MemberType NoteProperty -Name
ForeignPort $null
           $ConnectionObject | Add-Member -MemberType NoteProperty -Name State
$null
        }

        $results += $ConnectionObject

     }
  }
  return $results
}

Function GetFileTimestamps($process)
{

  return (gci $process.executablepath -ErrorAction SilentlyContinue | select Name,
CreationTime, CreationTimeUtc, LastWriteTime, LastWriteTimeUtc, LastAccessTime,
LastAccessTimeUtc)

}

Function GetService($process)
{
  $services = Get-WmiObject win32_service

  if ($ServiceResult = $Services | Where-Object {$_.Pathname -like "*" +
$process.executablepath + "*"} )
  {
     return $ServiceResult


Author Name, email@addressm
```

```
    }

}

#expecting powershell's Get-Process process object
Function GetDLLs($gp_process)
{
    $LoadedDLLs = $gp_process | select -expand modules

    #get SHA256 hash of each DLL loaded into file using certutil. Replaces any spaces
produced by certutil (common in older OSes)
    $LoadedDLLs | foreach {$_ | Add-Member NoteProperty Hash ((certutil -hashfile
$_.filename SHA256)[1] -replace '\s','')  }
    return $LoadedDLLs
}

Function GetPersistenceInfo($process)
{
    $filetimes = GetFileTimestamps $process
    $TaskResults = GetTasks $process

    $ServiceResults = GetService $process

    $registryResults = SearchRegistry $process

    $AutostartFolderResults = AutostartFolders $process

    if ($search)
    {
        $FilesystemSearchResults = SearchFilesystem $process
    }

    #If above functions didn't return results, force variable to have something to maintain
result array order
    if(!$taskresults){$TaskResults = ""}
    if(!$ServiceResults){$ServiceResults = ""}
    if(!$registryResults){$registryResults = ""}
    if(!$autostartFolderResults){$autostartFolderResults = ""}
    if(!$FilesystemSearchResults){$FilesystemSearchResults = ""}
    if(!$filetimes){$filetimes = ""}

    $results =
$taskresults,$ServiceResults,$registryResults,$AutostartFolderResults,$filetimes,$Filesy
stemSearchResults

    return $results


Author Name, email@addressm
```

```
}

Function GetSpecificProcessInfo($p)
{

   $gp_process = get-process -id $p

   $process = Get-WmiObject win32_process -filter "processid=$p"

   $LoadedDLLs = GetDLLs $gp_process

   $NetworkResults = GetNetworkActivity $p

   GetProcessLineage $process #the results of this are added to $global:process_table

   if(!$NetworkResults){$NetworkResults = ""}

   $ResultArray = $global:Process_table,$LoadedDLLs,$NetworkResults

   if(!$filepath)
   {
      $persistenceResults = GetPersistenceInfo($process)
      $resultArray += $persistenceResults
   }



   return $ResultArray
}


######################START OF ACTUAL SCRIPT
PROCESSING#########################################


#If the user passes the -tree option, list all processes in tree format, otherwise write a
simple list of all processes sorted by PPID
   if (!$p -and $tree -and !$filepath)
   {
      Show-ProcessTree
   }
   elseif (!$p -and !$filepath)
   {
      MyProcessList
   }


Author Name, email@addressm
```

```
#if given a PID, and it exists, get info about it. otherwise return an error message stating
PID doesn't exist
   if($p -and (get-process -id $p -ErrorAction SilentlyContinue) -and !$filename)
   {


     return GetSpecificProcessInfo $p

   }
   elseif ($p -and !$filename)
   {
   write-host "Process ID doesn't exist"
   return $false
   }

   if ($filepath)
   {


      #creating a new object with a property named ExecutablePath so we can use all the
same functions
      $filenameContainer = New-Object -TypeName psobject
      $filenameContainer | Add-Member NoteProperty Executablepath  $filepath

      $persistenceResults = GetPersistenceInfo $filenameContainer

      $AllProcessResultsArray = @()

      $RunningProcesses = Get-WmiObject win32_process | where {$_.executablepath -
like $filepath}

      if ($RunningProcesses)
      {
         foreach ($process in $RunningProcesses)
         {
            $AllProcessResultsArray += GetSpecificProcessInfo $process.processid
         }
      }


      $fileresults = $persistenceResults
      $fileresults += $AllProcessResultsArray

      return $fileresults
```

Author Name, email@addressm

```
    }



} ###This is the ending bracket for "FullScript" Function


Function VirusTotal($LoadedDLLs)
{
   #This is the master variable that will hold the data for all dll lookups
   $VTLookupResults = @()

   foreach($dll in $LoadedDLLs)
   {
      # Don't Look up DLLs created by Microsoft. This was done to reduce the number of
lookups
      if ($dll.Company -ne "Microsoft Corporation")
      {
         $hash = $dll.hash

         #Try to check Virus Total for hash results. if no results found, return that
information instead
         try
         {

            $test = Invoke-restmethod https://www.virustotal.com/api/v3/files/$hash -
Headers @{"x-apikey"=$vtAPIKey} -ErrorAction SilentlyContinue

            #Convert Last Analysis Date from Epoch time to human readable time
            [datetime]$origin = '1970-01-01 00:00:00'
            $LastAnalysisDate =
$origin.AddSeconds($test.data.attributes.last_analysis_date)

            #Create a temporary variable to store just the VirusTotal results we want, to be
added to master variable
            $results = $test.data.attributes.last_analysis_stats
            $results | Add-Member NoteProperty Filename $dll.filename
            $results | Add-Member NoteProperty Hash $hash
            $results | Add-Member NoteProperty LastAnalysisDate $LastAnalysisDate
         }
         catch [System.Net.WebException] #Do this if Virus Total gives back a 404 error
meaning no data found
         {
```

Author Name, email@addressm

```
            $results = New-Object -TypeName psobject
            $results | Add-Member NoteProperty Filename $dll.filename
            $results | Add-Member NoteProperty Hash $hash
            $results | Add-Member NoteProperty LastAnalysisDate "Results not found"

        }
        catch
        {
            Write-host "Invoke-restmethod cmdlet not found"
        }
        #Add results to master variable
        $VTLookupResults += $results

        #VirusTotal API is rate limited (4 per minute with Free API). This call to sleep
ensures we don't break that limit
        sleep($vtsleep)
      }
   }

   return $VTLookupResults
}

Function FormatOutput($results, $vtresults)
{
   <#
      When processing the $Results variable, the array will contain data in different
elements depending on if this script was
      executing searching for a PID or file path. Below details what information is stored
in which element of the array.

      by PID
      (result data by index)

      0 = ProcessLineage
      1 = LoadedDLLs
      2 = Network Results
      3 = Scheduled Task Results
      4 = Service Results
      5 = Registry Results
      6 = Autostart Folder Results
      7 = File Timestamps
      8 = Filesystem Search Results

      By Filepath
      (Result data by index)



Author Name, email@addressm
```

```
    0 = Scheduled Task Results
    1 = Service Results
    2 = Registry Results
    3 = Autostart Folder Results
    4 = File Timestamps
    5 = Filesystem Search Results
    6 = Process Lineage
    7 = LoadedDLLs
    8 = Network Results

    NOTE:: when searching by filepath, it also checks for running processes executing
from that path, which is where 6-8 come in.
                it's possible for multiple processes to be running from that path, so there
might be multiple sets of data related to 6-8. This
    is why these elements were moved to the end of the array, so we can iterate through
these three elements as many times as there
    where processes running.

  #>

  #Process and output all process information
  if (!$p -and $tree -and !$filepath)
  {

    if ($list)
    {
      $global:process_table | select name, pid, ppid, processuser, executablepath,
commandline | fl
    }
    else
    {
      $global:process_table | select name, pid, ppid, processuser, executablepath,
commandline | ft -AutoSize -wrap
    }
    if ($outpath)
    {
      $global:process_table | select name, pid, ppid, processuser, executablepath,
commandline | export-Csv -NoTypeInformation ($outpath + "ProcessList.csv")
    }

  }
  elseif (!$p -and !$filepath)
  {

    if ($table)
    {
```

Author Name, email@addressm

```
        $global:process_table | sort ppid | select name, pid, ppid, processuser,
executablepath, commandline | ft -AutoSize -wrap
      }
      else
      {
        $global:process_table | sort ppid | select name, pid, ppid, processuser,
executablepath, commandline
      }
      if ($outpath)
      {
        $global:process_table | sort ppid | select name, pid, ppid, processuser,
executablepath, commandline | export-Csv -NoTypeInformation ($outpath +
"ProcessList.csv")
      }
   }

   if($p)
   {
      if($table)
      {
        Write-Host "Process Lineage in Reverse Order"
        $results[0] | Format-Table -auto -wrap

        Write-Host "Process Loaded DLLs"
        $results[1] | select modulename, filename, hash, company, fileversion | Format-
Table -auto -Wrap

        if($results[2])
        {
          Write-Host "Network Information"
          $results[2] | Format-Table -auto -wrap
        }
        else
        {
          Write-Host "No Network Activity Associated with PID'r `n"
        }

        if($results[3])
        {
          Write-Host "Scheduled Task Information"
          $results[3] | select Hostname, Taskname, "Next Run Time", Status, "Last Run
Time", Author, "Task to Run", "Run As User", "Schedule Type" | Format-Table -
AutoSize -wrap
        }
        else
        {
```

Author Name, email@addressm

```
            Write-Host "No Scheduled Tasks`r`n"
        }

        if($results[4])
        {
            Write-Host "Service Information"
            $results[4] | select SystemName, name, DisplayName, StartMode, Pathname,
StartName, State | format-table -auto -Wrap
        }
        else
        {
            write-host "No Matching Services`r`n"
        }
        if($results[5])
        {
            Write-Host "Autostart Registry Keys Found"
            $results[5] | select key, value | format-table -auto -wrap
        }
        else
        {
            Write-Host "No Autostart Registry Keys Found"
        }
        if($results[6])
        {
            Write-Host "Autostart Folder Results`r`n"
            $results[6]
            Write-Host "`r`n"
        }
        else
        {
            Write-Host "No Matching Autostart Folder Entries`r`n"
        }
        if($results[7])
        {
            Write-Host "Executable Timestamps"
            $results[7] | select name, creationtime, creationtimeutc, lastwritetime,
lastwritetimeutc, lastaccesstime, lastaccesstimeutc | format-table -auto -wrap
        }
        else
        {
            Write-host "Couldn't get executable timestamp information. File doesn't
exist.`r`n"
        }
        if($results[8])
        {
            Write-Host "Files created within $search minutes of the executable in question"


Author Name, email@addressm
```

```
            $results[8] | select creationtime, Fullname | format-table -AutoSize -wrap
        }
        else
        {
            Write-Host "Search not requested or no files crearted within search timeframe
of executable in question'r `n"
        }

    }
    elseif($list -or !$outpath)
    {
        Write-Host "Process Lineage in Reverse Order"
        $results[0] | format-list

        Write-Host "Process Loaded DLLs"
        $results[1] | select modulename, filename, hash, company, fileversion | Format-
List

        if($results[2])
        {
            Write-Host "Network Information"
            $results[2] | Format-List
        }
        else
        {
            Write-Host "No Network Activity Associated with PID'r `n"
        }

        if($results[3])
        {
            Write-Host "Scheduled Task Information"
            $results[3] | select Hostname, Taskname, "Next Run Time", Status, "Last Run
Time", Author, "Task to Run", "Run As User", "Schedule Type" | Format-list
        }
        else
        {
            Write-Host "No Scheduled Tasks`r`n"
        }

        if($results[4])
        {
            Write-Host "Service Information"
            $results[4] | select SystemName, name, DisplayName, StartMode, Pathname,
StartName, State | format-list
        }
        else
```

Author Name, email@addressm

```
        {
          write-host "No Matching Services`r`n"
        }
        if($results[5])
        {
          Write-Host "Autostart Registry Keys Found"
          $results[5] | select key, value | format-list
        }
        else
        {
          Write-Host "No Autostart Registry Keys Found"
        }
        if($results[6])
        {
          Write-Host "Autostart Folder Results`r`n"
          $results[6] | format-list
          Write-Host "`r`n"
        }
        else
        {
          Write-Host "No Matching Autostart Folder Entries`r`n"
        }
        if($results[7])
        {
          Write-Host "Executable Timestamps"
          $results[7] | select name, creationtime, creationtimeutc, lastwritetime,
lastwritetimeutc, lastaccesstime, lastaccesstimeutc | format-list
        }
        else
        {
          Write-host "Couldn't get executable timestamp information. File doesn't
exist.`r`n"
        }
        if($results[8])
        {
          Write-Host "Files created within $search minutes of the executable in question"
          $results[8] | select creationtime, Fullname | format-list
        }
        else
        {
          Write-Host "Search not requested or no files crearted within search timeframe
of executable in question'r `n"
        }
      }
      if($outpath)
      {
```

Author Name, email@addressm

```
        $results[0] | export-csv -NoTypeInformation "$outpath$p-ProcessLineage.csv"

        $results[1] | select modulename, filename, hash, company, fileversion | export-csv
-NoTypeInformation "$outpath$p-LoadedDLLs.csv"

        if($results[2])
        {
          $results[2] | export-csv -NoTypeInformation "$outpath$p-
NetworkInformation.csv"
        }
        else
        {
          Write-Host "No Network Activity Associated with PID'r `n"
        }

        if($results[3])
        {
          $results[3] | select Hostname, Taskname, "Next Run Time", Status, "Last Run
Time", Author, "Task to Run", "Run As User", "Schedule Type" | export-csv -
NoTypeInformation "$outpath$p-ScheduledTasks.csv"
        }
        else
        {
          Write-Host "No Scheduled Tasks`r`n"
        }

        if($results[4])
        {
          $results[4] | select SystemName, name, DisplayName, StartMode, Pathname,
StartName, State | export-csv -NoTypeInformation "$outpath$p-ServiceInformation.csv"
        }
        else
        {
          write-host "No Matching Services`r`n"
        }
        if($results[5])
        {
          $results[5] | select key, value | export-csv -NoTypeInformation "$outpath$p-
AutostartRegistryKeys.csv"
        }
        else
        {
          Write-Host "No Autostart Registry Keys Found"
        }
        if($results[6])
        {
```

Author Name, email@addressm

```
            $results[6] | export-csv -NoTypeInformation "$outpath$p-
AutostartFolderResults.csv"
        }
        else
        {
            Write-Host "No Matching Autostart Folder Entries`r`n"
        }
        if($results[7])
        {
            $results[7] | select name, creationtime, creationtimeutc, lastwritetime,
lastwritetimeutc, lastaccesstime, lastaccesstimeutc | export-csv -NoTypeInformation
"$outpath$p-ExecutableTimestamps.csv"
        }
        else
        {
            Write-host "Couldn't get executable timestamp information. File doesn't
exist.`r`n"
        }
        if($results[8])
        {
            $results[8] | select creationtime, Fullname | export-csv -NoTypeInformation
"$outpathPID-$p-FilesystemSearchResults.csv"
        }
        else
        {
            Write-Host "Search not requested or no files crearted within search timeframe
of executable in question'r `n"
        }
    }
  }

  #The elements in the array were re-ordered when given the filepath option. This was
because multiple processes could be running from the same file
  # Which makes the length of the array variable while the rest is static. By putting it at
the end, we can work through the dynamic length with a loop.
  elseif($filepath)
  {
    if($table)
    {
        if($results[0])
        {
            Write-Host "Scheduled Task Information"
            $results[0] | select Hostname, Taskname, "Next Run Time", Status, "Last Run
Time", Author, "Task to Run", "Run As User", "Schedule Type" | Format-Table -
AutoSize -wrap
        }
```

Author Name, email@addressm

```
        else
        {
           Write-Host "No Scheduled Tasks`r`n"
        }

        if($results[1])
        {
           Write-Host "Service Information"
           $results[1] | select SystemName, name, DisplayName, StartMode, Pathname,
StartName, State | format-table -auto -Wrap
        }
        else
        {
           write-host "No Matching Services`r`n"
        }
        if($results[2])
        {
           Write-Host "Autostart Registry Keys Found"
           $results[2] | select key, value | format-table -auto -wrap
        }
        else
        {
           Write-Host "No Autostart Registry Keys Found"
        }
        if($results[3])
        {
           Write-Host "Autostart Folder Results`r`n"
           $results[3]
           Write-Host "`r`n"
        }
        else
        {
           Write-Host "No Matching Autostart Folder Entries`r`n"
        }
        if($results[4])
        {
           Write-Host "Executable Timestamps"
           $results[4] | select name, creationtime, creationtimeutc, lastwritetime,
lastwritetimeutc, lastaccesstime, lastaccesstimeutc | format-table -auto -wrap
        }
        else
        {
           Write-host "Couldn't get executable timestamp information. File doesn't
exist.`r`n"
        }
        if($results[5])


Author Name, email@addressm
```

```
        {
          Write-Host "Files created within $search minutes of the executable in question"
          $results[5] | select creationtime, Fullname | format-table -AutoSize -wrap
        }
        else
        {
          Write-Host "Search not requested or no files crearted within search timeframe
of executable in question'r `n"
        }

        #Check if any results from the file running as a process. if so, process the data,
otherwise skip it and move on.
        if($results[6])
        {
          ###start running process processing
          $i = 6 #starting at 6 to match the index position of results array
          do
          {
            Write-Host "Process Lineage in Reverse Order"
            $results[$i] | Format-Table -auto -wrap

            Write-Host "Process Loaded DLLs"
            $results[$i+1] | select modulename, filename, hash, company, fileversion |
Format-Table -auto -Wrap

            if($results[$i+2])
            {
              Write-Host "Network Information"
              $results[$i+2] | Format-Table -auto -wrap
            }
            else
            {
              Write-Host "No Network Activity Associated with PID'r `n"
            }

            $i += 3

          }while($i -lt $results.count)
        }
        else
        {
          Write-Host "File not running as process"
        }

      }
      elseif($list -or !$outpath)


Author Name, email@addressm
```

```
    {

      if($results[0])
      {
        Write-Host "Scheduled Task Information"
        $results[0] | select Hostname, Taskname, "Next Run Time", Status, "Last Run
Time", Author, "Task to Run", "Run As User", "Schedule Type" | Format-list
      }
      else
      {
        Write-Host "No Scheduled Tasks`r`n"
      }

      if($results[1])
      {
        Write-Host "Service Information"
        $results[1] | select SystemName, name, DisplayName, StartMode, Pathname,
StartName, State | format-list
      }
      else
      {
        write-host "No Matching Services`r`n"
      }
      if($results[2])
      {
        Write-Host "Autostart Registry Keys Found"
        $results[2] | select key, value | format-list
      }
      else
      {
        Write-Host "No Autostart Registry Keys Found"
      }
      if($results[3])
      {
        Write-Host "Autostart Folder Results`r`n"
        $results[3] | format-list
        Write-Host "`r`n"
      }
      else
      {
        Write-Host "No Matching Autostart Folder Entries`r`n"
      }
      if($results[4])
      {
        Write-Host "Executable Timestamps"
```

Author Name, email@addressm

```
        $results[4] | select name, creationtime, creationtimeutc, lastwritetime,
lastwritetimeutc, lastaccesstime, lastaccesstimeutc | format-list
    }
    else
    {
        Write-host "Couldn't get executable timestamp information. File doesn't
exist.`r`n"
    }
    if($results[5])
    {
        Write-Host "Files created within $search minutes of the executable in question"
        $results[5] | select creationtime, Fullname | format-list
    }
    else
    {
        Write-Host "Search not requested or no files crearted within search timeframe
of executable in question'r `n"
    }

    #Check if any results from the file running as a process. if so, process the data,
otherwise skip it and move on.
    if($results[6])
    {
        ###start running process processing
        $i = 6 #starting at 6 to match the index position of results array
        do
        {
            Write-Host "Process Lineage in Reverse Order"
            $results[$i] | format-list

            Write-Host "Process Loaded DLLs"
            $results[$i+1] | select modulename, filename, hash, company, fileversion |
Format-List

            if($results[$i+2])
            {
                Write-Host "Network Information"
                $results[$i+2] | Format-List
            }
            else
            {
                Write-Host "No Network Activity Associated with PID'r `n"
            }
            $i += 3
        }while($i -lt $results.count)
    }
```

Author Name, email@addressm

```
        else
        {
            Write-Host "File not running as process"
        }

    }
    if($outpath)
    {
        $justFilename = ($filepath.split('\')[-1]).split('.')[0]

        if($results[0])
        {
            $results[0] | select Hostname, Taskname, "Next Run Time", Status, "Last Run
Time", Author, "Task to Run", "Run As User", "Schedule Type" | export-csv -
NoTypeInformation "$outpath$justFilename-ScheduledTasks.csv"
        }
        else
        {
            Write-Host "No Scheduled Tasks`r`n"
        }

        if($results[1])
        {
            $results[1] | select SystemName, name, DisplayName, StartMode, Pathname,
StartName, State | export-csv -NoTypeInformation "$outpath$justFilename-
ServiceInformation.csv"
        }
        else
        {
            write-host "No Matching Services`r`n"
        }
        if($results[2])
        {
            $results[2] | select key, value | export-csv -NoTypeInformation
"$outpath$justFilename-AutostartRegistryKeys.csv"
        }
        else
        {
            Write-Host "No Autostart Registry Keys Found"
        }
        if($results[3])
        {
            $results[3] | export-csv -NoTypeInformation "$outpath$justFilename-
AutostartFolderResults.csv"
        }
        else
```

Author Name, email@addressm

```
        {
          Write-Host "No Matching Autostart Folder Entries`r`n"
        }
        if($results[4])
        {
          $results[4] | select name, creationtime, creationtimeutc, lastwritetime,
lastwritetimeutc, lastaccesstime, lastaccesstimeutc | export-csv -NoTypeInformation
"$outpath$justFilename-ExecutableTimestamps.csv"
        }
        else
        {
          Write-host "Couldn't get executable timestamp information. File doesn't
exist.`r`n"
        }
        if($results[5])
        {
          $results[5] | select creationtime, Fullname | export-csv -NoTypeInformation
"$outpath$justFilename-FilesystemSearchResults.csv"
        }
        else
        {
          Write-Host "Search not requested or no files crearted within search timeframe
of executable in question'r `n"
        }

        #Check if any results from the file running as a process. if so, process the data,
otherwise skip it and move on.
        if($results[6])
        {
          ###start running process processing
          $i = 6 #starting at 6 to match the index position of results array
          do
          {
            #Get the PID of the first process found. Just trying to get the PID field from
results gives an array of all PIDs within the process
            # lineage, so we have t specify just the first PID in the array, which is the pid
of the exectuable in quetsion
            $currentPID = ($results[$i][0].pid)[0]

            $results[$i] | export-csv -NoTypeInformation "$outpath$justFilename-
$currentPID-ProcessLineage.csv"

            $results[$i+1] | select modulename, filename, hash, company, fileversion |
export-csv -NoTypeInformation "$outpath$justFilename-$currentPID-LoadedDLLs.csv"

            if($results[$i+2])
```

Author Name, email@addressm

```
                {
                    $results[$i+2] | export-csv -NoTypeInformation "$outpath$justFilename-
$currentPID-NetworkInformation.csv"
                }
                else
                {
                    Write-Host "No Network Activity Associated with PID $ currentPID'r `n"
                }

                $i += 3

            }while($i -lt $results.count)
        }
        else
        {
            Write-Host "File not running as process"
        }

    }
}

if($vt -and $vtresults)
{
    if($table)
    {
        $vtresults | select Filename, LastAnalysisDate, Undetected, type-unsupported,
malicious, suspicious, failure, timeout, harmless, hash | Format-Table -AutoSize -wrap
    }
    elseif($list -or !$outpath)
    {
        $vtresults | select Filename, LastAnalysisDate, Undetected, type-unsupported,
malicious, suspicious, failure, timeout, harmless, hash | Format-list
    }
    if($outpath)
    {
        if($p)
        {
            $vtresults | select Filename, LastAnalysisDate, Undetected, type-unsupported,
malicious, suspicious, failure, timeout, harmless, hash | Export-Csv -NoTypeInformation
"$outpath$p-VirusTotal.csv"
        }
        elseif($filepath)
        {
            $vtresults | select Filename, LastAnalysisDate, Undetected, type-unsupported,
malicious, suspicious, failure, timeout, harmless, hash | Export-Csv -NoTypeInformation
"$outpath$justFilename-VirusTotal.csv"
```

Author Name, email@addressm

```
            }
        }
    }
    elseif($vt -and !$vtresults)
    {
        Write-host "No Results from Virus Total. This likely means all DLLs loaded are
from Microsoft or the file is not a running process."
    }




}



if ($computer)
{
    if(!$cred)
    {
        $cred = Get-Credential
    }

    $temp = Invoke-Command -ComputerName $computer -cred $cred -ScriptBlock
${function:FullScript} -ArgumentList $p,$tree,$search,$filepath

    if($vt -and $p)
    {
      $vtresults = VirusTotal $temp[1]
    }
    elseif($vt -and $filepath)
    {
        if($temp[7])
        {
            $vtresults = VirusTotal $temp[7]
        }
    }

    if($temp -ne $false)
    {
        FormatOutput $temp $vtresults
    }
}
else
{
```

Author Name, email@addressm

```
    $temp = FullScript $p $tree $search $filepath

    if($vt -and $p)
    {
        $vtresults = VirusTotal $temp[1]
    }
    elseif($vt -and $filepath)
    {
        if($temp[7])
        {
            $vtresults = VirusTotal $temp[7]
        }
    }

    if($raw)
    {
        $temp
    }
    if(($temp -ne $false) -and !$raw)
    {
        FormatOutput $temp $vtresults
    }
}
```