**Project Report**

**Monitoring Computer Information and Settings**

Jefferey Lipford, Mosope Onifade, Shea Moore

CSCI 590

## 

# **Background information**

In line with Department of Energy orders and the National Institute of Technology and Standards (NIST), specific regulatory cybersecurity policies need to take place. These are documented in [NIST Special publication 800-54 Rev4](https://csrc.nist.gov/publications/detail/sp/800-53/rev-4/final), but ultimately boil down to data collection and verification of different nodes on information systems. This program works towards satisfying the information collection portion of this requirement..

# **Purpose of the Project**

The purpose of this project is to fulfill the information collection requirement of the cyber security policies SRNL is required to adhere to by using PowerShell scripts to return information about the computer. This utility is usually carried out using a suite of third party programs; however, industrial control systems and other operational technology systems are negatively impacted by additional network load imposed by these third party vendor applications. Our project also aims to solve the problem of added network load by developing the program "in house".

More specifically, our industry mentor, Mackenize [sic] Morris, gave us a list of seventeen "artifacts" – pieces of information from the computer, such as, for example, the process list – to retrieve. Each artifact's information is stored in one or more individual files. Additionally, there is a single large CSV file that stores information from all artifacts.

# **Project Description and Features**

Our project consists of sixteen PowerShell scripts: Out-ArtifactsCsv.ps1, which is the main script, and fifteen secondary scripts that Out-ArtifactsCsv.ps1 calls, with names such as Get-Artifact1.ps1, Get-Artifact2.ps1, Get-SecurityPolicy.ps1, etc. Each script produces at least one file. The Get-Artifact scripts produce files related to the artifacts they are trying to retrieve; for example, Get-Artifact2.ps1 produces a small TXT file containing the list of processes running on the computer. Each Get-Artifact script also returns some information to Out-ArtifactsCsv.ps1, which includes that information in a CSV file, Artifacts.csv.

**Out-ArtifactsCsv**

This is the main file that calls the fifteen secondary Get-Artifact scripts. It looks like this:

Param([String] $Path) # Path where files will be saved

# Such as 'C:\Users\The User\Documents\'

If($Path -eq $Null -or $Path -eq '') {

$Path = '.\' # Current directory

}

If(-not (Test-Path $Path)) {

Throw('Invalid path argument')

}

If(-not $Path.EndsWith('\')) {

$Path += '\'

}

Write-Host '--------------- Working'

$head = 'Hostname,Make,Model,MAC Addresses,'

$info = .\Get-Artifact1 -Path $Path

Write-Host -NoNewline '.'

$head += 'Process List,'

$info += .\Get-Artifact2 -Path $Path

Write-Host -NoNewline '.'

... And so on for artifacts 3, 4, ..., 17 ...

$head += "`n"

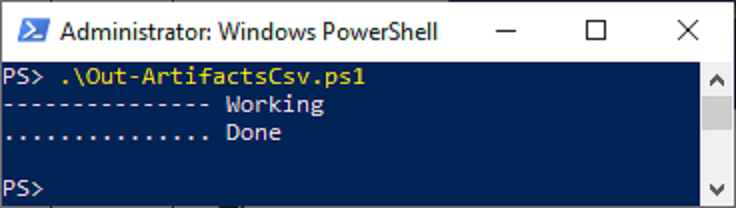
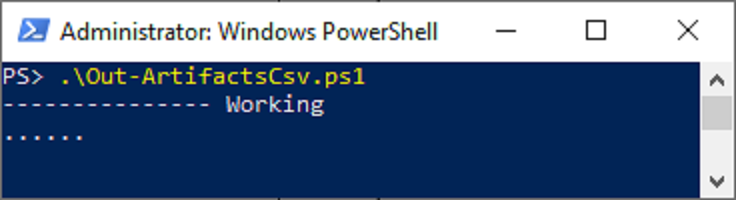
$info += "`n"

($head + $info) | Out-File -Encoding UTF8 -FilePath ($Path + 'Artifacts.csv')

Write-Host " Done`n"

$Path is the single parameter of the script. It determines the directory where Artifacts.csv and the other output files will be stored. If no $Path is supplied, the current directory (.\) is used. If a $Path is supplied, the Test-Path cmdlet makes sure the directory exists, and if not, an error is thrown. Because the filename, "Artifacts.csv," will be appended to $Path later, a backslash is appended if one is not already there.

This script has a simple progress bar. The following are images of the script running (left) and finished (right) with no $Path specified.



Before calling the first Get-Artifact script, Write-Host '--------------- Working' produces the first line of that progress bar.

The string $head stores the first row of the CSV file. $info stores the second row. Commas in these strings (unless they are within pairs of quotation marks) represent cell boundaries. Each time a Get-Artifact script is called, the information it returns is appended to $info. Then, a single period is printed in the progress bar.

At the end, $head and $info are concatenated and written to Artifacts.csv in the directory indicated by $Path using the Out-File cmdlet. We use UTF-8 encoding because the default encoding does work correctly.

**Common features of each Get-Artifact script**

All Get-Artifact scripts (Get-Artifact1.ps1, Get-Artifact2.ps1, Get-SecurityPolicy.ps1, etc.) look something like this:

Param([String] $Path) # Path where files will be saved

# Such as 'C:\Users\The User\Documents\'

If($Path -eq $Null -or $Path -eq '') {

$Path = '.\' # Current directory

}

If(-not (Test-Path $Path)) {

Throw('Invalid path argument')

}

If(-not $Path.EndsWith('\')) {

$Path += '\'

}

**...**

$Output = **...**

$txtOutput =

(Get-CimInstance -Class Win32\_ComputerSystem | ForEach-Object { $\_.Name }) + ','

($txtOutput + $Output + "`n") | Out-File -Encoding UTF8 -FilePath ($Path + '**...**.txt')

return $Output

Each of these scripts both outputs a file and returns a similar string, which is either printed to the console if a user called the script in isolation or is used in Artifacts.csv if Out-ArtifactsCsv.ps1 called the script. These scripts differ primarily in what they do at the places marked "**...**," which determine what output is generated and the filename of the output file. They also have some other minor differences.

$Path is the single parameter of the script. It determines the directory where the output file or files will be stored. The code for $Path is the same as in Out-ArtifactsCsv.ps1.

$Output stores whatever String will be returned and either printed to the console or written to Artifacts.csv. The assignment of $Output usually looks something like this:

$Output = '"' + (Call-SomeCommand | Out-String).Replace('"', "'") + '",'

Some cmdlet is called, or some program is run, or some variable or variables produced earlier in the program are referenced. Cmdlets often return objects other than Strings, so data is often piped to the Out-String cmdlet for conversion to a String.

Cells in CSV files are delimited with commas, but commas within pairs of quotation marks (") do not delimit cells. For example, the following could represent three cells in a CSV file: "This is a cell","This is a cell, too, with 2 commas","This is a 3rd cell". We decided to standardize our Get-Artifact scripts by making each one represent a cell in the same way: as data preceded by a quotation mark and followed by a quotation mark and a comma, regardless of whether the data could contain a comma. Therefore, we replace any quotation marks in output with single-quotation marks (').

The line $txtOutput = (Get-CimInstance -Class Win32\_ComputerSystem | ForEach-Object { $\_.Name }) + ',' retrieves the host name of the computer that the script is being run on and stores the result in a variable named $txtOutput. The Get-CimInstance cmdlet is used to retrieve information from the system including desktop settings, bios version, processor, disk, local time and much more. ForEach-Object { $\_.Name }) specifies the object that should be returned and ensures that it only returns text with no notation surrounding it.This is done so that the text file produced will start with the hostname of the computer.

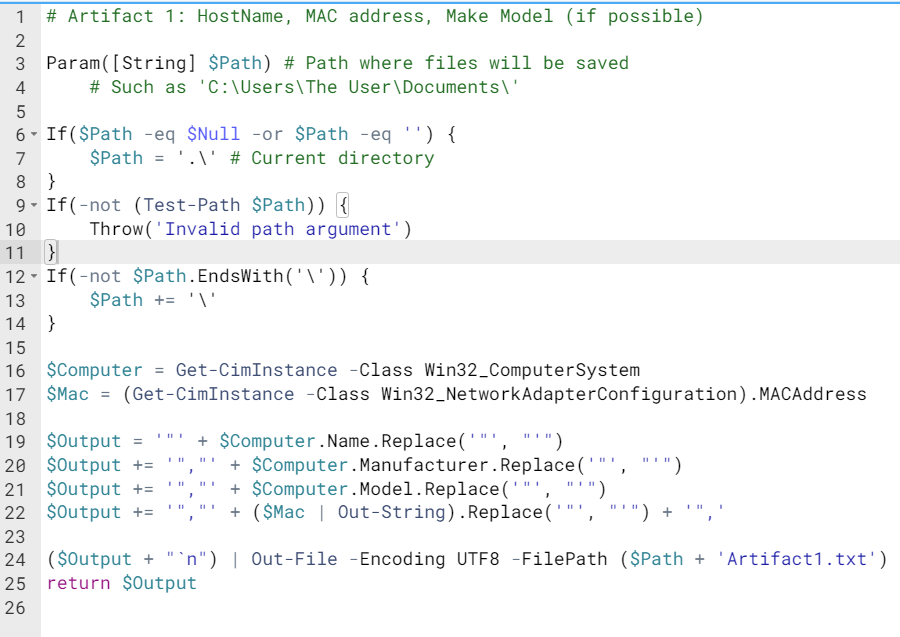
The line ($txtOutput + $output + "`n") | Out-File -Encoding UTF8 -FilePath ($Path + 'Artifact1.txt') (where Artifact1.txt is replaced with a different filename for each script) makes up the output that is stored in a text file. At the end of each artifact, the host name of the computer and the content of the artifact are concatenated together and stored in a text file. This is so that a user of this program can look specifically at the information for each artifact.

Finally, $Output is returned.

We explain each Get-Artifact script in more detail below; please refer to the included source code and output files while reading.

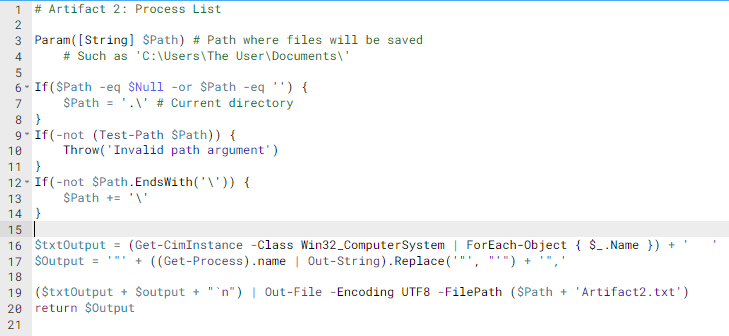
**Artifact 1: HostName, Make, Model, and MAC Addresses**

Artifact 1 is fairly simple. Get-CimInstance -Class Win32\_ComputerSystem returns an object containing lots of information including the computer's name, primary owner, domain, bytes of physical memory, model, and manufacturer. Out of this we get the name, manufacturer (make), and model, which we place in different cells in the CSV file. Get-CimInstance -Class Win32\_NetworkAdapterConfiguration likewise returns an object containing lots of information, out of which we get the MAC Addresses, which we place in a fourth cell.



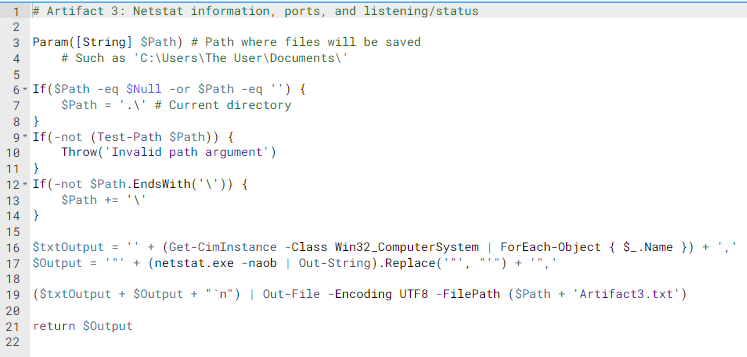
**Artifact 2: Process List**

Artifact 2 is also simple. Get-Process returns an array of System.Diagnostics.Process objects. Out of that array we get an array of just the Names of the processes. We then convert that array to a single String, in which there is a newline after each process name.



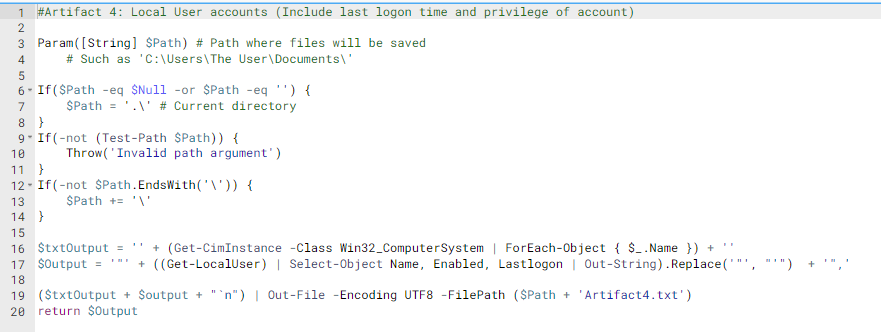
**Artifact 3: Netstat information, ports, and listening/status**

Artifact 3 simply calls netstat.exe -naob and converts the result to a String. According to Netstat's help information, -a "displays all connections and listening ports," -n "displays addresses and port numbers in numerical form," -o "displays the owning process ID associated with each connection," and -b "displays the executable involved in creating each connection or listening port." Our industry mentor told us to use those four flags. Without using any flags, Netstat had taken several minutes to run, much more time than the other artifacts.



**Artifact 4: Local User accounts (Include last logon time and privilege of account)**

Artifact 4 retrieves local user account information using the cmdlet Get-LocalUser. This cmdlet gets default built-in user accounts, local user accounts created by the user, and local accounts that are connected to local Microsoft accounts.We then specify the objects needed with Select-Object Name, Enabled, Lastlogon. This shows that the local user objects returned should contain the account name, whether or not the account is enabled and the last logon time for the account.



**Security Policy**

Get-SecurityPolicy.ps1 covers multiple artifacts (5, 8, part of 12, and 17) that were listed separately in our industry mentor's requirements. It turned out that a single script could easily retrieve the information for these separate artifacts.

First, we write an empty String to SecurityPolicy.txt in the folder indicated by $Path. We then use secedit.exe to append the Security Policy to SecurityPolicy.txt. secedit.exe normally writes two brief lines to the console, but Out-Null prevents this.. The beginning of the file looks something like this:

[Unicode]

Unicode=yes

[System Access]

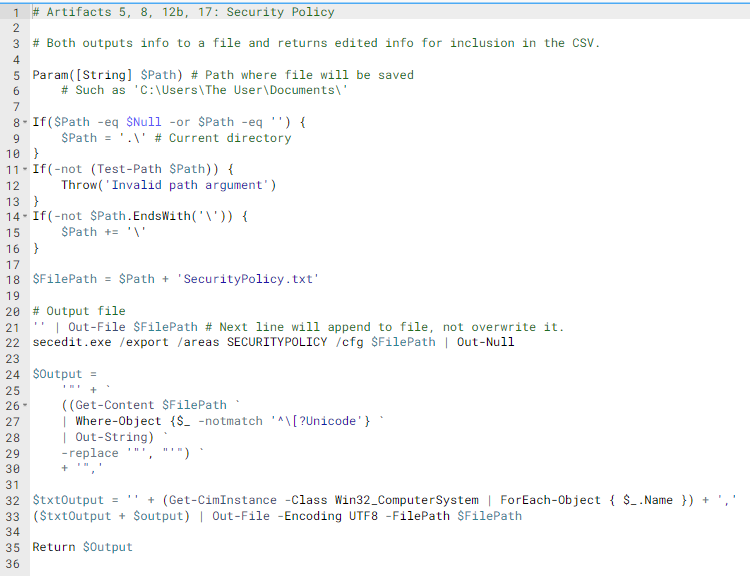
MinimumPasswordAge = 0

MaximumPasswordAge = 42

...

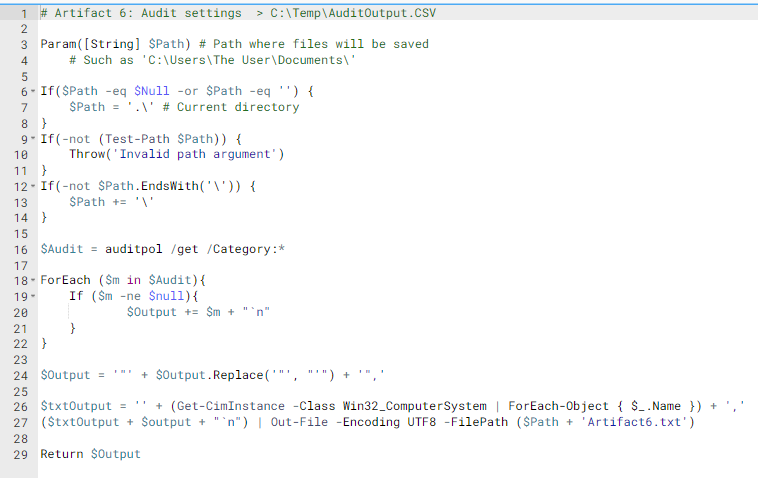
Our industry mentor asked us to remove the first two lines about Unicode. We first use the Get-Content cmdlet to read the file as an array of lines of text. We then pipe that array to Where-Object {$\_ -notmatch '^\[?Unicode'}. Here, we keep only the lines that do not match the regular expression '^\[?Unicode'. The first two lines are removed, but the rest remain. We then convert the array to a single String with Out-String.

Before returning $Output, we save ($txtOutput + $Output) to SecurityPolicy.txt, overwriting that file again.



**Artifact 6: Audit settings**

Artifact 6 retrieves the systems auditing settings using auditpol. Windows uses audit policies to define account limits as well as determine which events are recorded in the system’s Security logs. For this Artifact, we return all categories and subcategories available through this cmdlet. These categories include logon/logoff settings, object access, system settings, use privileges, and policy changes.



**Artifact 7: Windows Event Log**

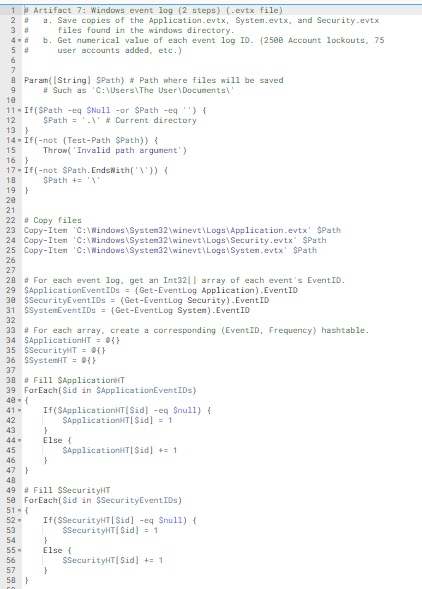
The purpose of this artifact is to copy three event log files – Application.evtx, System.evtx, and Security.evtx – and create three corresponding tables of the frequencies of event IDs in those files. To save space, for the remainder of this section we will refer only to the System event log, but everything said about it is applicable to the other two.

First, we copy C:\Windows\System32\winevt\Logs\System.evtx to the directory indicated by $Path, using the Copy-Item cmdlet. We do not read from or write to this newly created file again.

Next, we use (Get-EventLog System).EventID to get an integer array of event IDs from the System event log. We then initialize an empty hash table with syntax @{}. For each event ID, if it does not exist as a key in the hash table, we add it as a key and set the value of that key to 1; otherwise, we increment the value of that key. The resulting table looks something like this:

| Keys: Event IDs | Values: Frequencies |
| --- | --- |
| 50037 | 9 |
| 50036 | 10 |
| ... | ... |

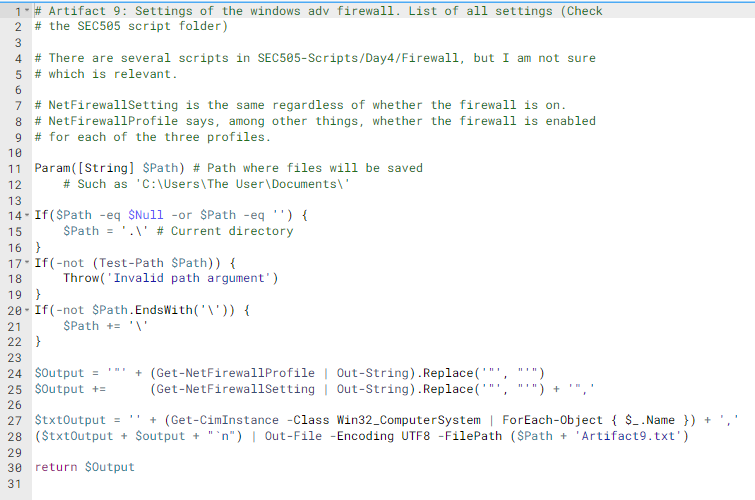
$Output is formatted such that Event IDs and Event ID Frequencies are saved in separate cells of the CSV.





**Artifact 9: Settings of the Windows Advanced Firewall**

This artifact is very simple. The Get-NetFirewallProfile and Get-NetFirewallSetting cmdlets get the information we need. Output is formatted and saved normally.



**User Rights**

This script is similar to Get-SecurityPolicy.ps1. It covers the part of Artifact 12 that Get-SecurityPolicy.ps1 did not cover. As in Get-SecurityPolicy.ps1, secedit.exe saves information in a file, and information about Unicode is removed by using Where-Object with a regular expression.

However, at this point, the User Rights information contains a bunch of cryptic sequences of characters starting with "\*S." It looks something like this:

[Privilege Rights]

SeNetworkLogonRight = \*S-1-1-0,\*S-1-5-32-544,\*S-1-5-32-545,\*S-1-5-32-551,\*S-1-5-32-583

SeBackupPrivilege = \*S-1-5-32-544,\*S-1-5-32-551

...

Our industry mentor asked us to replace IDs such as "\*S-1-1-0" with meaningful text. For example, "\*S-1-1-0" actually means "Everyone." The User Rights information should look like this:

[Privilege Rights]

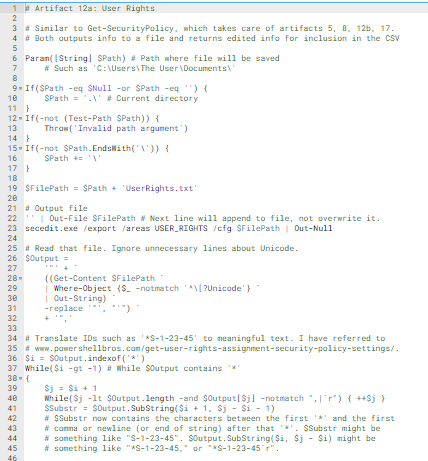
SeNetworkLogonRight = Everyone,BUILTIN\Administrators,BUILTIN\Users,BUILTIN\Backup Operators,BUILTIN\Device Owners

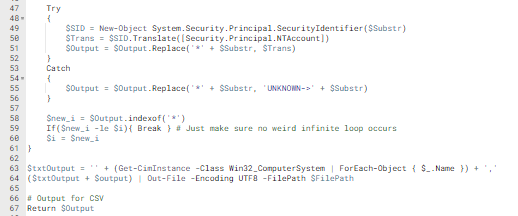
SeBackupPrivilege = BUILTIN\Administrators,BUILTIN\Backup Operators

...

A single ID can be replaced using the Translate method in an instance of the System.Security.Principal.SecurityIdentifier class. In a large, complicated loop, we repeatedly find an untranslated ID, translate it, replace every instance of it in the String with its translation, and continue the loop unless there is no untranslated ID left.

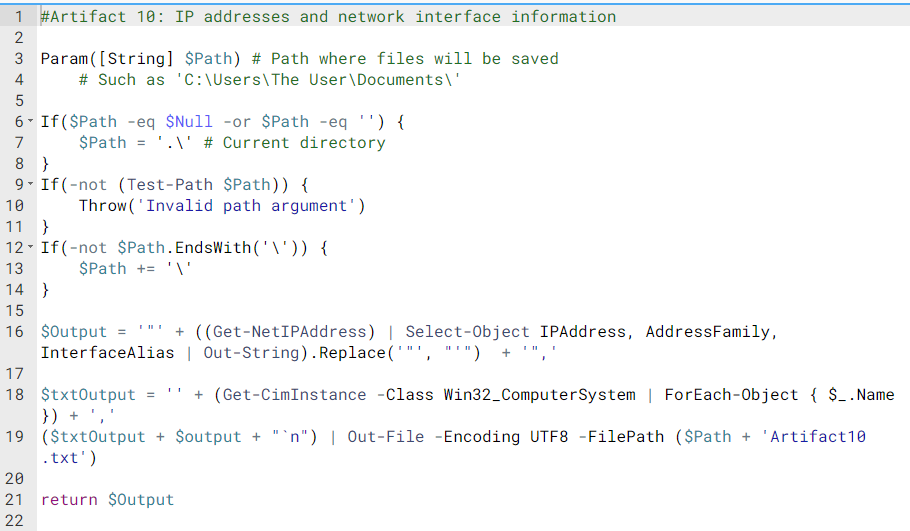
The rest of this script works as normal.





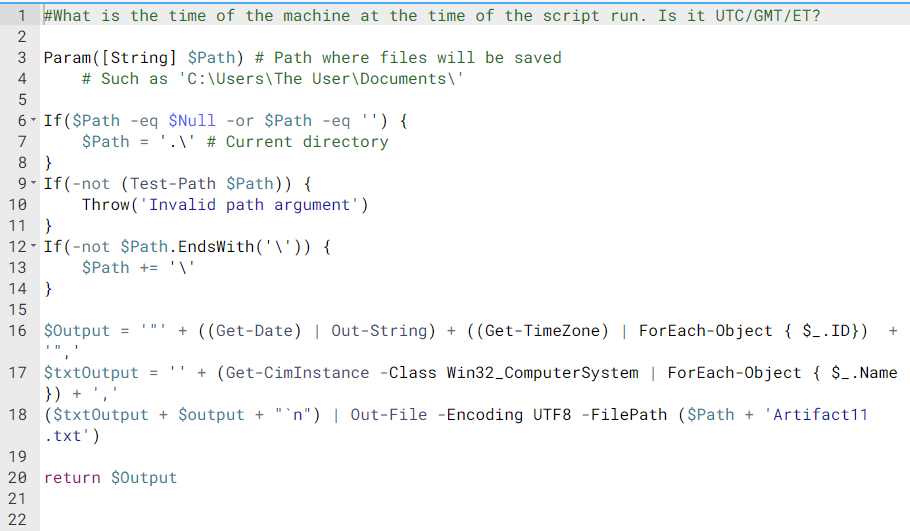
**Artifact 10: IP addresses and network interface information**

This artifact uses the Get-NetIPAddress cmdlet which returns Ip address configuration. This includes IPv4 addresses, IPv6 addresses and the IP interfaces the IP interfaces are associated to. This cmdlet shows why it is important to specify parameters because if they are not specified in this case, the cmdlet returns the entire IP configuration of the computer. We specify the parameters we would like to be returned with Select-Object IPAddress, AddressFamily, InterfaceAlias. This shows we only need the IP address, address family and interface alias.



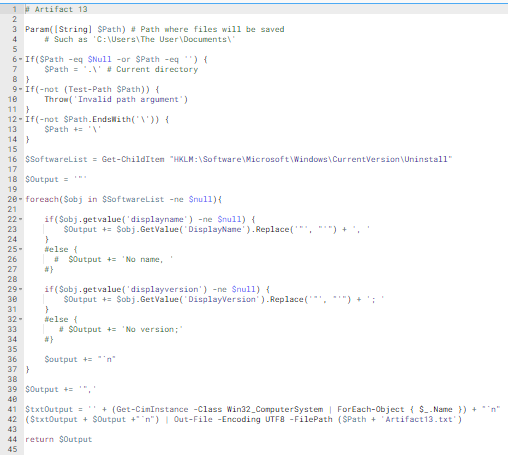
**Artifact 11: Time of the machine when the script is run**

Artifact 11 returns the date, time and timezone on the machine at the time that the machine is being run. The cmdlet Get-Date and Get-TimeZone are concatenated using the following syntax (Get-Date) | Out-String) + ((Get-TimeZone) | ForEach-Object { $\_.ID}). For Get-TimeZone we need to specify ForEach-Object { $\_.ID} because this ensures that the result is only text with no odd notations surrounding it.



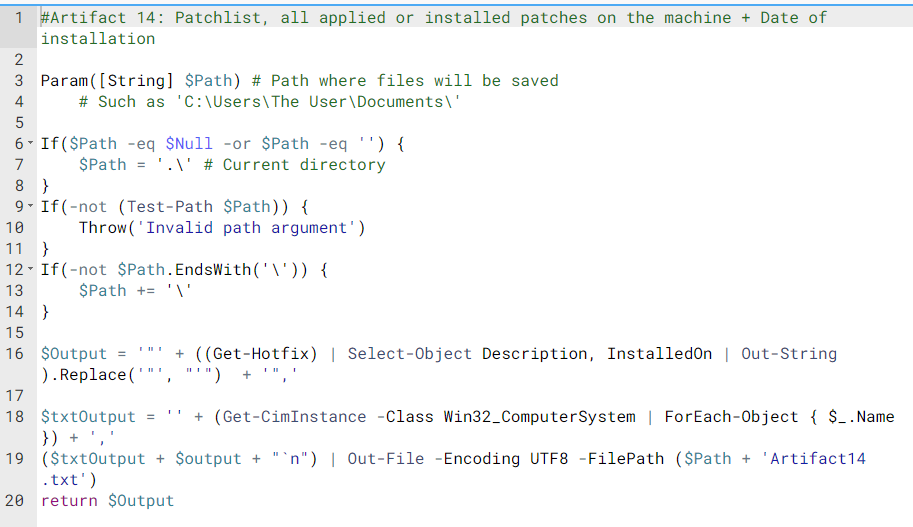
**Artifact 13: SoftwareList (all installed items on machine)**

Artifact 13 is used to return a list of all software names and versions installed on the machine. This script uses a HKEY file path which directs the Artifact to the Windows Registry tree. This tree contains information about drivers, programs, and other configuration data used by all users on the system. Since some items returned using this path are null, this artifact also uses a loop which replaces null values with either No name or No version.



**Artifact 14: Patchlist, all applied or installed patches on the machine + Date of installation**

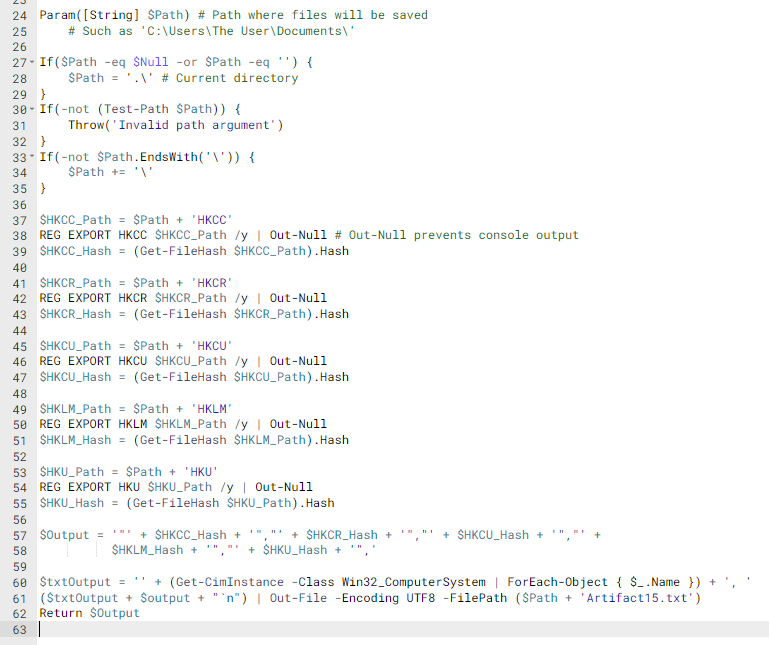
Artifact 14 is used to get a list of installed patches on the machine. This Artifact uses the cmdlet Get-Hotfix which returns system update information including its source, description, id, the user it was installed by, and the installation date. For our project, we only want information pertaining to the update’s description and date. To do this, we use the Select-Object cmdlet which allows you to limit return data based on specific objects.



**Artifact 15: Copies and Hashes of the Five Registry Hives**

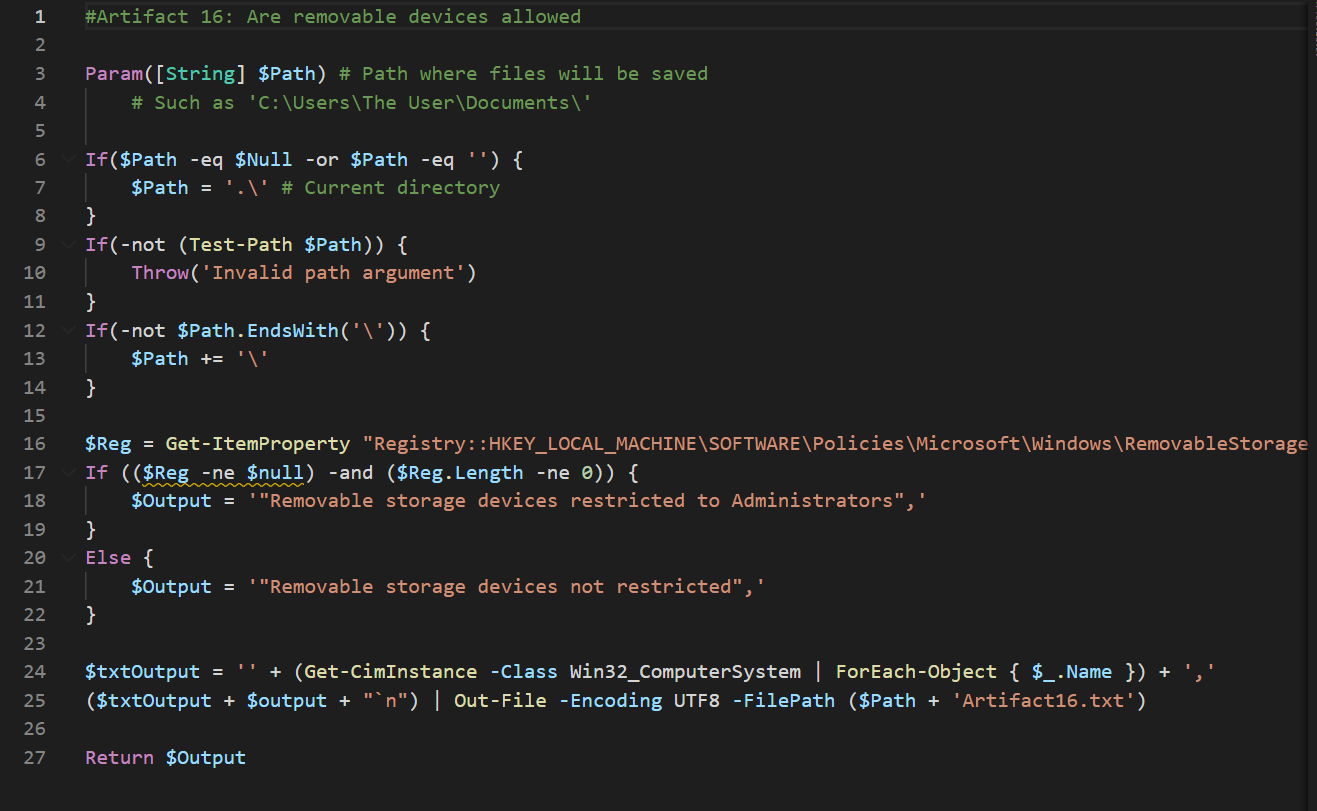
There are five "hives" in the Registry: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS, and HKEY\_CURRENT\_CONFIG. They are abbreviated HKCR, HKCU, KHLM, HKU and HKCC. Neither the hives nor the full registry are actual files on the computer, so we cannot simply copy them with Copy-Item as in Artifact 7. Instead, we use REG EXPORT (with /y flag to "force overwriting the existing file without prompt") to save hives to the directory indicated by $Path. After saving a hive, we use the Get-FileHash cmdlet to get a hash code from the file. The produced files are many megabytes in size, so we only return the hash codes for inclusion in the CSV.

This artifact takes the longest to run because the registry contains lots of information.



**Artifact 16:**

Artifact 16 is used to determine if the user is allowed to use removable devices such as USB drives or CDs. This script uses the cmdlet Get-ItemProperty with a Registry file path to retrieve a property item value located within Windows Local Group Policy Editor (this editor is only available to view if the system has Windows 10 Pro installed). Our targeted value is the policy, “Devices: Allow to format and eject removable media” located under the Security Options folder in Windows Security Settings. Since there is no way to directly grab the text value from the Policy Editor, we had to use an if statement to measure the length of the value returned. If the length is 0, then there are no user restrictions for removable devices. If the length is not equal to 0, then removable media is only allowed by Administrators.



# **Project Challenges**

* Learning PowerShell. Powershell was a new language to all the members of the team at the time we started the project. This meant that the first 2 weeks was mainly for us to research and practice using Powershell and this continued throughout the process of working on the project.
* Lacking knowledge of details of the Windows operating system, such as the organization of the Registry.
* Our industry mentor, Mackenize [sic] Morris, had to buy Jeffrey Windows 10 Pro so he could download the Group Policy PowerShell module. However, after more research, it turned out Jeffrey did not need that module. Nevertheless, it was by downloading it and trying to use it that Jeffrey figured out that he did not need it and should use something else. We also needed Windows 10 Pro to open the Windows Local Group Policy Editor

1. **Possible Future Extension of the Program**

Our program only runs on a single computer. It would be more useful if it could run remotely on multiple computers simultaneously. Our industry mentor or someone else could expand upon our program to make this possible. Our program is set up to be extensible in this way; the Artifacts.csv file is organized into two rows specifically so that if our program were extended, multiple computers could be included, each on its own row.

1. **What We Learned**

This project taught us lots about the IT side of computer science, which we have not covered much in our previous coursework. We learned PowerShell, an entirely new language, and got some exposure to the inner workings of the Windows operating system.

Also, through this project, we got experience working in a team to create a large program over a long period of time. Few of our other courses give us practice doing this; Software Engineering is the other main one that does. While working on it, we did plenty of research in order to figure out for ourselves how to solve problems with minimal supervision by knowledgeable professionals. Our industry mentor intentionally avoided giving us too much help in order to make the project more like ones we might encounter in the real world.

Overall, we think we benefited substantially from this course.