Complement a Vulnerability Management Program with PowerShell

GIAC (GSEC) Gold Certification

Author: Colm Kennedy
Advisor: Stephen Northcutt
Date of Acceptance:

Abstract

A vulnerability management program is a critical task that all organizations should be running. Part of this program involves the need to patch systems regularly and to keep installed software up to date. Once a vulnerability program is in place organizations need to remediate discovered vulnerabilities quickly. Occasionally some discovered vulnerabilities are false positives. The problem with false positives is that manually vetting them is time-consuming. There are tools available, which assist in showing what patches may be missing, like SCCM, but can be rather costly. For organizations concerned that these types of programs hurt their budgets, there are free options available. PowerShell is free software that, if utilized, can complement an organization's vulnerability management program by assisting in scanning for unpatched systems. This paper presents a PowerShell script that provides Administrators with further insight into what systems are unpatched and streamlines investigations of possible false positives, with no additional cost.

1. Introduction

With the increasing likelihood that more dumps of Windows exploits from Shadow Brokers are on the way, the need for patching and fixing vulnerabilities on systems has become increasingly important. The WannaCry vulnerability shows the importance of keeping the software on systems updated and patched. As stated by Microsoft on the release date of WannaCry, "Most of the exploits that were disclosed fall into vulnerabilities that are already patched in our supported products" (M. Team, 2017). When the WannaCry announcement came out, those organizations that regularly patch had little to do because a previous patch resolved the issue. With over two hundred thousand systems impacted with this ransomware, organizations that did not keep up with patches had to scramble to get them installed on the impacted systems (B. Brenner, 2017). The importance of monthly patching of all systems plays a key role in preventing outbreaks, similar to WannaCry, using future releases of Windows exploits by the Shadow Brokers.

A commonly accepted concept within vulnerability management is that "Exploits keep coming, so vigilance and routine patching are vital" (CyberTrend, 2016). A vulnerability management program can help determine what systems get patched and where vulnerabilities exist in the environment. Part of every vulnerability program includes scanning and reporting on vulnerable software on systems. Occasionally some of these scans produce false positives that require further investigation. The problem with investigating false positives is it typically requires manual vetting of software versions installed or if the systems are missing Microsoft patches. A bigger issue arises when multiple systems need further investigation. This process of vetting can take many hours to accomplish. Tools are available, which can assist in showing what patches are missing, like SCCM, but some are costly. PowerShell is a free and effective software when utilized correctly. It gives an administrator another tool to scan for unpatched systems.

The goal of this paper is to show how a small script in PowerShell can aid in the investigation of many systems and report information in an organized manner that will help streamline investigations of possible false positives. The script in this paper will assist in identifying missing patches or old versions of software installed. The PowerShell script in this paper sends the results to a CSV file that lists the system name, the operating system installed, the number of missing critical and important patches, last boot time of the system, Chrome version, Firefox Version, Java Version, Adobe Flash Version, and Adobe Reader Version. The format of the CSV file makes it easy to browse over all systems scanned to make a quick determination if more investigation needs to take place for each system. An important part of this script is that it needs to be transferrable to any environment with little effort.

2. Survey Results

Discovering if there is a need for a PowerShell script that determines if a system is up to date is an important part of this paper. Gathering this information was accomplished using a Survey published online with twenty-seven responses recorded. Five questions were asked in this survey to get a better understanding of how information gathering, similar to the script in this paper, took place. The list of questions is as follows:

1. What tools do you use to report on installed applications like Chrome, Firefox, Adobe Reader, Adobe Flash on a system?
2. What tools do you use to report on how many critical patches are available to be installed on a system?
3. How much money and time do you spend collectively on this type of information gathering?
4. Would you be interested in a script that could report this to you in a CSV file?
5. Are you comfortable using open source? What are your concerns?

Question one showed many organizations using SCCM as their tool to report on installed applications in their respective environments. Other organizations used a variation of free software like WMIC and PowerShell to relatively cheap alternatives Lansweeper and PDQ Inventory (both under $1000). The results of question one are in Chart 1.

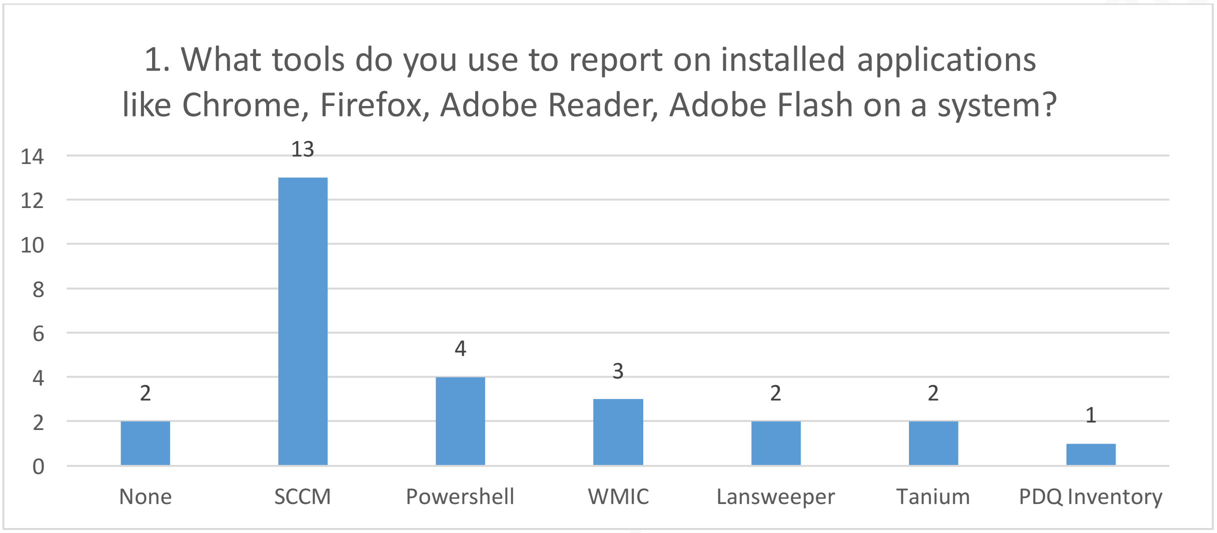


Chart 1 – Answers to Question One

The second question results, as seen in Chart 2, again show a majority of responders using SCCM for missing patches. Ten of the responders are using other products that cost money including Qualys, Nessus, Rapid7, and Shavlick. Four of the organizations are using freeware, and one person is manually checking to see if systems are missing critical patches.

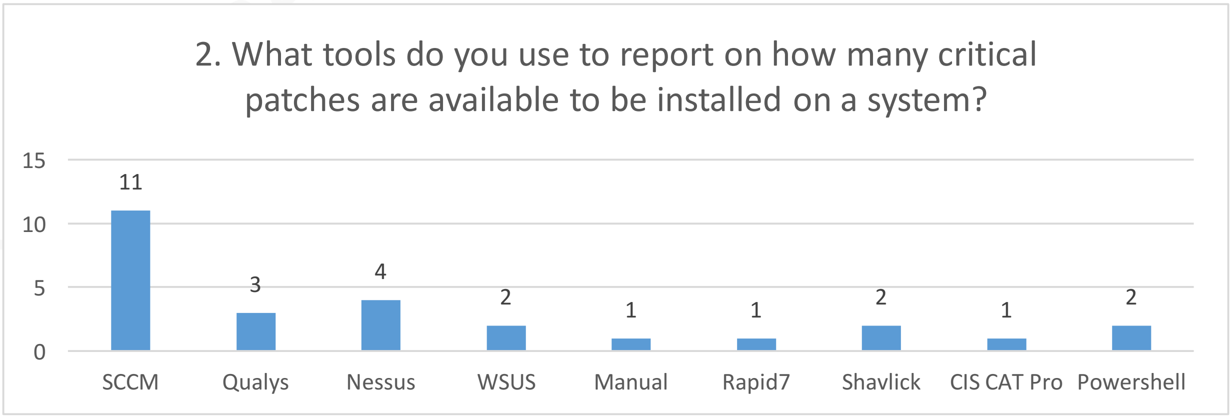


Chart 2 – Answers to Question Two

Question three has two parts, with the first part looking at how much time is spent on information gathering on missing patches on systems and software versions installed. As shown in Chart 3, organizations spent a lot of time on this type of information gathering. Even those using SCCM felt they were spending too much time gathering information and confirming that information with other tools.

Colm Kennedy

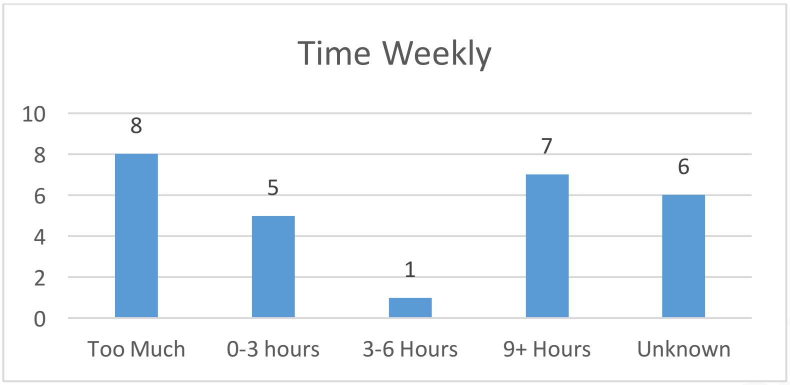


Chart 3 – Answers to Question Three

The second part of question three looks at how much money organizations are spending on tools that return missing patches on systems and what versions of applications are on a system. These results of this question are in Chart 4. Although this varied significantly, it is worth noting that the general feeling was they were spending too much time and money on gathering information on missing patches and installed software.

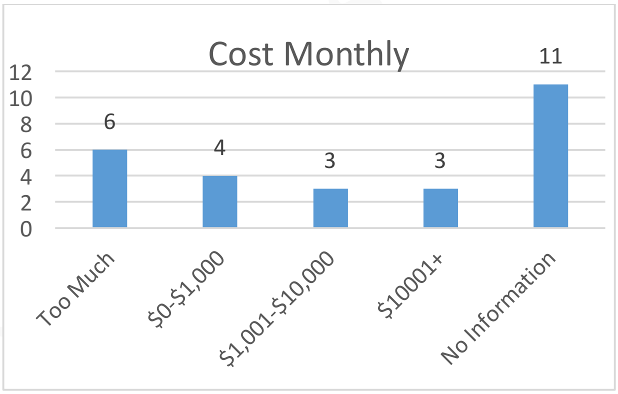


Chart 4 – Answers to Question Three

Question four shows the responses from the twenty-seven that responded to the survey on if there is interest in a script that could report information gathering around patches and installed applications. The results, in Chart 5, show that 24 of them showed interest in seeing a PowerShell script that could assist in gathering this information and exporting to a CSV file.

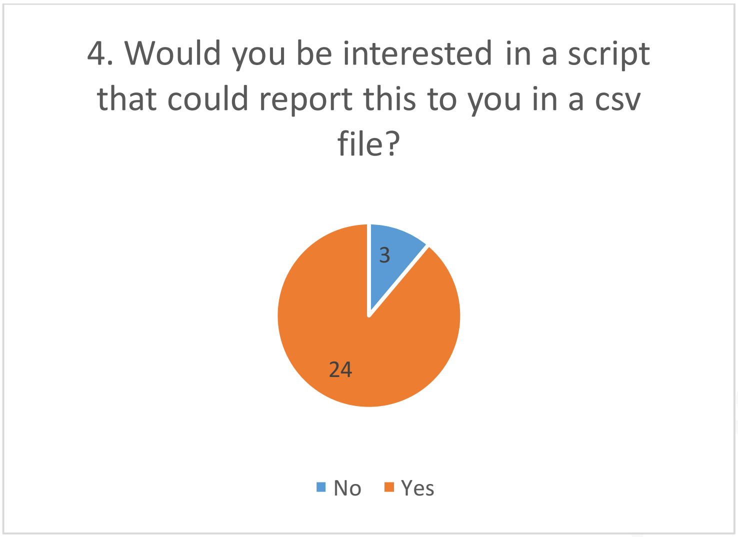


Chart 5 – Answer to Question Four

The fifth question asked if organizations have a comfort level using Open Source and if not, what their concerns would be. Only one answered no to this question, but that was due to an organizational issue that required all products used to have maintenance purchased with it.

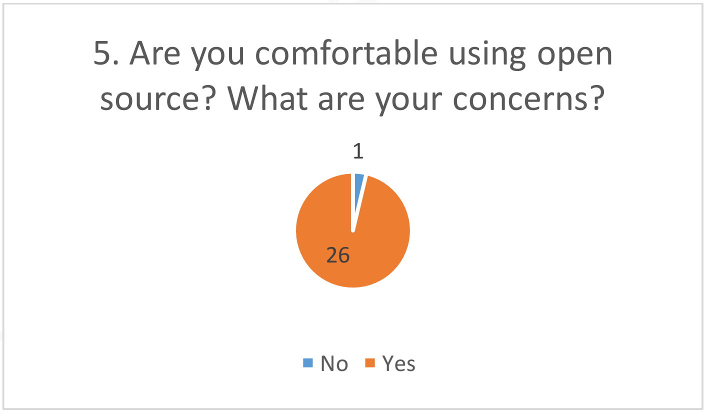


Chart 6 – Answer to Question Five

The overall response to the five questions shows that many organizations are using tools that cost money to gather missing patches and software versions installed on systems. There was much interest in additional tools and abilities to spot check results of other tools that can be easily followed and is transferable to other environments.

3. PowerShell

The best explanation of what PowerShell is comes from Microsoft's web page and says, "PowerShell is an automation platform and scripting language for Windows and Windows server that allows you to simplify the management for your systems" (What is PowerShell, 2017). PowerShell 1.0 was first released in 2006 for Windows XP and Server 2003 but required installation. Microsoft integrated PowerShell into future Operating system versions with PowerShell 2.0. The current version of PowerShell is version 5.1. PowerShell now comes built into all Windows installs. PowerShell scripts can be found online using Google searches and on many other exchanges like technet.microsoft.com, Stackoverflow.com, and PowerShell.org. All sites have avenues to get assistance in creating or editing a PowerShell script.

For this research, PowerShell Integrated Scripting Environment (ISE) simplified the creation of this script. The PowerShell ISE environment gives the option of a split screen that allows for editing of the script while running tests in a command line window. Using the ISE environment made the troubleshooting of the script much easier to work on during the creation of this PowerShell script.

The idea was to create a script that was easy to follow for beginners with PowerShell and to make a script that would help those Administrators with a limited budget for information gathering of missing Microsoft patches and installed software versions. The script also reports on the operating system installed and the last boot time of the system. The last boot time of a system assists in identifying the last time this system received patches successfully. Using last boot time as an indicator is effective because most critical and important security patches require a reboot for installation of the patch to be completed. If a system has not rebooted in over six months, it is likely that this system has not successfully patched in that time. The lack of a recent reboot on the system would raise suspicion for an administrator to investigate that system further manually.

The five applications that are reported by this script are often seen on systems and not updated frequently. Chrome and Firefox by default auto-update, but if not used, stay unpatched. If not used for long enough, some of these programs do not allow for updates and require a reinstallation of the product. Java, Adobe Flash, and Adobe Reader are notoriously left unpatched, and vulnerabilities are regularly released.

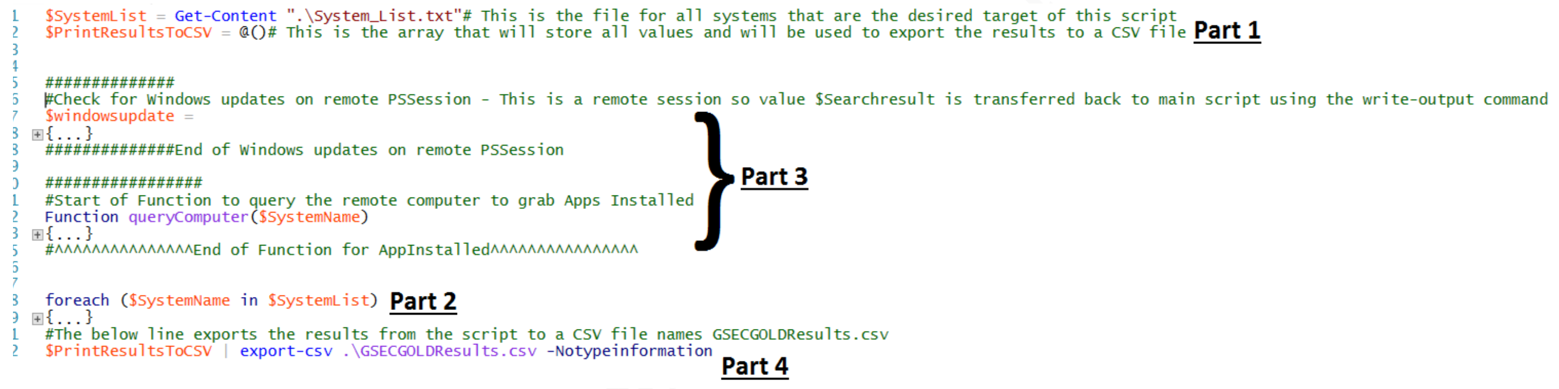
4. The Script

4.1 Pre-Requisites for script

With PowerShell, there are some requirements for scripts to run successfully. For this script to work properly in an environment, a few services need to be running on the remote systems. These services are the Windows Remote Management (WinRM service) and the Remote Registry service. Opening PowerShell on the local system as a user that has Administrator access on the remote systems allows for the script to function properly. This particular script has been successfully tested on Windows 7, Windows 10, Windows Server 2003, Windows Server 2008 and Windows Server 2012 with all above requirements set. For learning purposes and to edit this script, it is best run from an elevated permission with PowerShell ISE and using the run script button at the top of the program. Two items need to be adjusted in the script to be able to work, one is the location of the System_List.txt file in the top line of the script and the second is the location of the export CSV file, GSECGOLDResults.csv, at the bottom of the script. These two are adjustable to wherever an administrator prefers. Another option is to leave the script as is. The Administrator then needs to place the System_List.txt file in the C:\Windows directory and the script creates the output file in the same directory. Although without local administrator rights, it might not be possible to write to the Windows folder. If that is an issue, the administrator should change and run the system_list file and the gsecgoldresults export file, from the desktop location.

4.2 Functions of the Script

The script contains four main parts. The script begins with the importing of the system list and the array variable is defined for later storage of the results of the scans (Part1). The second part is the main body of the script with a loop for each system on the imported list. This loop in the main body is called a 'foreach' loop. Within this main body is the third part where the main script calls for three different functions. One is a WMI call to return the Operating system and Last Boot Time. The second is a PSSession or remote PowerShell session that allows for the Windows updates information to be returned. The third part of the main script is a function call that makes a remote registry connection to return application versions installed. These three functions make up most of the script and do all of the work to gather the desired results. The fourth part is the export to CSV command that sends all the results stored in the array to the CSV file defined on the last line of the script. Figure 1 shows the four parts of the script with the functions, and the main body minimized. The content of these sections is provided in more detail in the following sections.



```
$SystemList = Get-Content ".\System_List.txt"# This is the file for all systems that are the desired target of this script
$PrintResultsToCSV = @()# This is the array that will store all values and will be used to export the results to a CSV file


##############
#Check for Windows updates on remote PSSession - This is a remote session so value $Searchresult is transferred back to main script using the write-output command
$windowsupdate =
{...}
##############End of Windows updates on remote PSSession

#################
#Start of Function to query the remote computer to grab Apps Installed
Function queryComputer($SystemName)
{...}
#^^^^^^^^^^^^^^^End of Function for AppInstalled^^^^^^^^^^^^^^^


foreach ($SystemName in $SystemList)
{...}
#The below line exports the results from the script to a CSV file names GSECGOLDResults.csv
$PrintResultsToCSV | export-csv .\GSECGOLDResults.csv -Notypeinformation
```

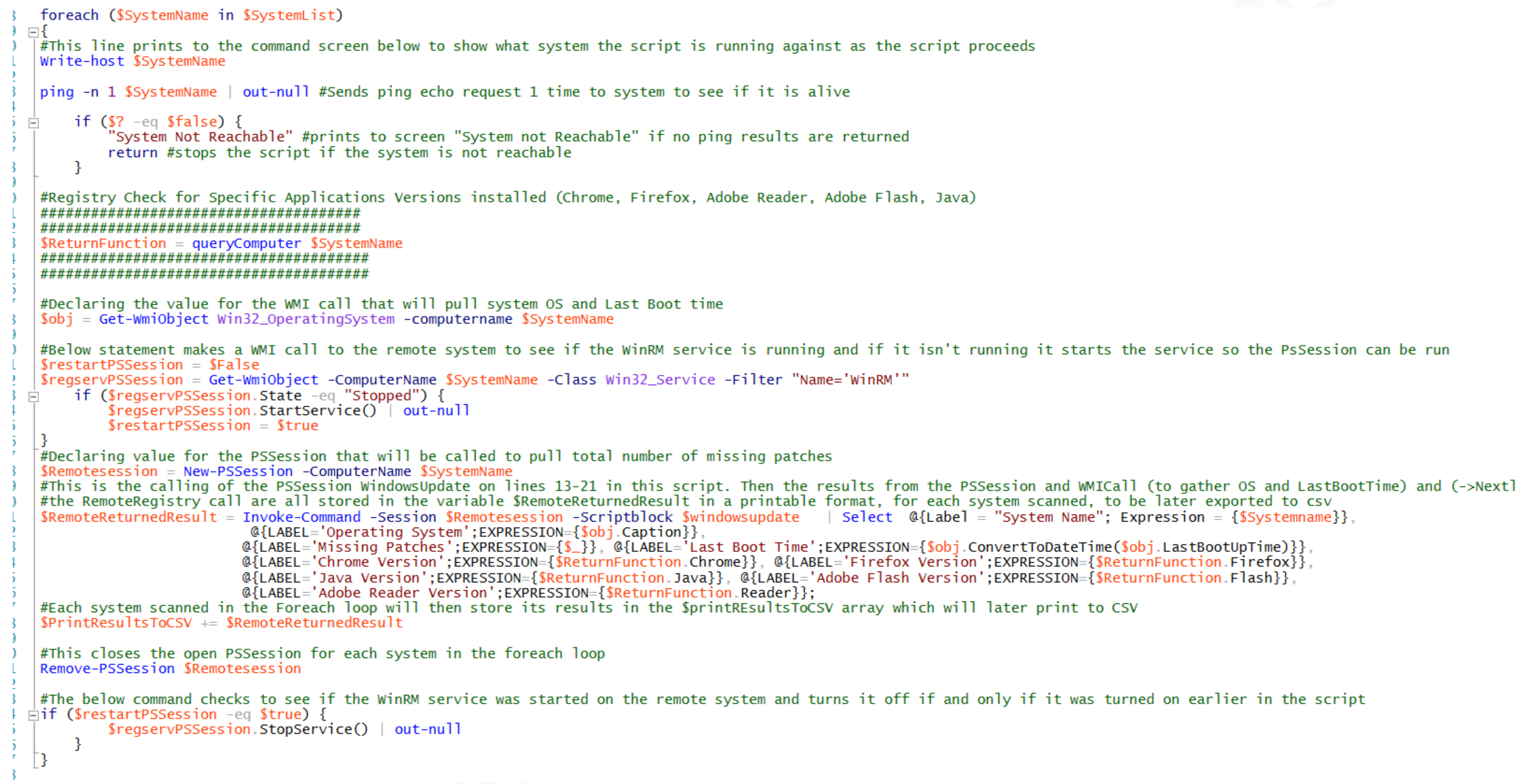
gure 1 – Overall view of script

4.2.1 The Script – Part 1

The script starts with the importing of a text file that has the systems listed, one system per line, which is to be scanned by the script. The import is done using the Get-Content command that then stores this list of systems in the variable called $SystemList. A requirement for this script is that the file is named System_List.txt. This file needs to be in the proper directory before running the script. This file contains the list of systems that are getting scanned. The variable $SystemList needs to point to text files location. By default, it is looking for the file in the C:\Windows directory. Next, in the script is the array variable that stores and then export the results to a CSV file on the final line of the script. It is called $PrinterResultsToCSV. Figure 1 above shows where Part 1 is noted.

4.2.2 The Script – Part 2

The script then continues into Part 2, or the main body and starts with the use of the 'foreach' loop. This loop takes the systems stored in the $SystemList variable and runs through the contents of Part 2 of the script for each system on the list. It does this by placing each system into the variable $SystemName for the script to run against then puts the next system on the list into that variable name. The loop runs until scanning of all systems on the list is complete.



```
foreach ($SystemName in $SystemList)
{
#This line prints to the command screen below to show what system the script is running against as the script proceeds
Write-host $SystemName

ping -n 1 $SystemName | out-null #Sends ping echo request 1 time to system to see if it is alive

    if ($? -eq $false) {
        "System Not Reachable" #prints to screen "System not Reachable" if no ping results are returned
        return #stops the script if the system is not reachable
    }

#Registry Check for Specific Applications Versions installed (Chrome, Firefox, Adobe Reader, Adobe Flash, Java)
######################################
######################################
$ReturnFunction = queryComputer $SystemName
#######################################
#######################################

#Declaring the value for the WMI call that will pull system OS and Last Boot time
$obj = Get-WmiObject Win32_OperatingSystem -computername $SystemName

#Below statement makes a WMI call to the remote system to see if the WinRM service is running and if it isn't running it starts the service so the PsSession can be run
$restartPSSession = $False
$regservPSSession = Get-WmiObject -ComputerName $SystemName -Class Win32_Service -Filter "Name='WinRM'"
    if ($regservPSSession.State -eq "Stopped") {
        $regservPSSession.StartService() | out-null
        $restartPSSession = $true
}
#Declaring value for the PSSession that will be called to pull total number of missing patches
$Remotesession = New-PSSession -ComputerName $SystemName
#This is the calling of the PSSession WindowsUpdate on lines 13-21 in this script. Then the results from the PSSession and WMICall (to gather OS and LastBootTime) and (->Nextl
#the RemoteRegistry call are all stored in the variable $RemoteReturnedResult in a printable format, for each system scanned, to be later exported to csv
$RemoteReturnedResult = Invoke-Command -Session $Remotesession -Scriptblock $windowsupdate   | Select  @{Label = "System Name"; Expression = {$Systemname}},
                         @{LABEL='Operating System';EXPRESSION={$obj.Caption}},
                        @{LABEL='Missing Patches';EXPRESSION={$_}}, @{LABEL='Last Boot Time';EXPRESSION={$obj.ConvertToDateTime($obj.LastBootUpTime)}},
                        @{LABEL='Chrome Version';EXPRESSION={$ReturnFunction.Chrome}}, @{LABEL='Firefox Version';EXPRESSION={$ReturnFunction.Firefox}},
                        @{LABEL='Java Version';EXPRESSION={$ReturnFunction.Java}}, @{LABEL='Adobe Flash Version';EXPRESSION={$ReturnFunction.Flash}},
                        @{LABEL='Adobe Reader Version';EXPRESSION={$ReturnFunction.Reader}};
#Each system scanned in the Foreach loop will then store its results in the $printREsultsToCSV array which will later print to CSV
$PrintResultsToCSV += $RemoteReturnedResult

#This closes the open PSSession for each system in the foreach loop
Remove-PSSession $Remotesession

#The below command checks to see if the WinRM service was started on the remote system and turns it off if and only if it was turned on earlier in the script
if ($restartPSSession -eq $true) {
        $regservPSSession.StopService() | out-null
    }
}
```

gure 2 – Foreach Loop and main body of script

Inside the 'foreach' loop, as seen in Figure 2 above, each system name is sent to the screen as the script scans each system. The system name displays on the screen, so the person running the script knows which system is currently being scanned. Each system is then sent a ping echo request one time to see if that system is alive. If it is not, it returns to the screen "System not Reachable" message and the script stops with no results. That system then requires a removal from the system list and the script rerun.

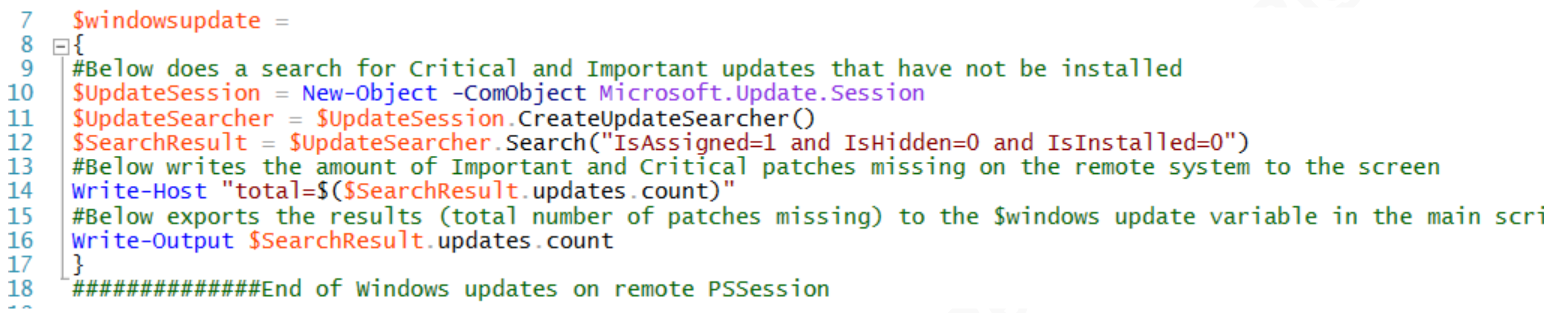
The next portion of the main body is to define the value that returns the results from the function 'QueryComputer.' Calling this also passes the name of the next system on the list. The contents of 'QueryComputer' are explained later in Part 3 of the script, but this is what returns the versions of installed applications for each system. The variable $obj is a WMI call to each remote system and returns the operating system version and the last boot time.

As previously stated, a pre-requisite for this script to function properly is the requirement for the WinRM service to be running. In this section, a check is made using a WMI call to the system to check to see if the WinRM service is enabled or stopped. If the service is not running, this script then enables that service. The variable that requires the WinRM service is

$RemoteSession. This variable creates a remote PowerShell session (or PSSession) on the remote system. On the following line of the script, the variable $RemoteReturnedResult calls the PSSession $Windowsupdate. The $Windowsupdate function returns Windows update results to the $RemoteReturnedResult variable. Within this variable, all results are piped to be stored temporarily in a format that makes the later CSV export easier to read. The results are piped by using a Select statement and labeling each column of the CSV file in single quotes and the values of each column in the ‘expression’ results. With all values for that system stored, the variable $RemoteReturnedResult adds its results to the array $PrintResultsToCSV to store that system’s results until scanning finishes for all systems. Before the script moves on to the next system within the loop, it checks to see if the script started the WinRM service previously and if so, it stops that service. If the script did not start the service, it leaves it enabled. The point is to return the status of the WinRM service to its original state.

4.2.3 The Script – Part 3

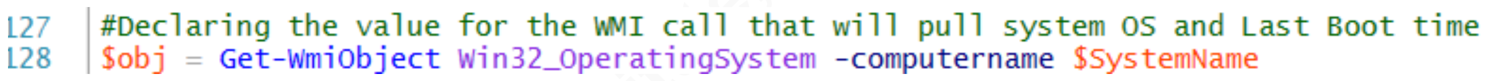
The third part of this script contains three functions that are each called from within the main body. The first portion is the PSSession that is called using the variable named $Windowsupdate. As seen, in Figure 3, it starts by declaring a new object for checking for windows updates. This object then creates the variable $updatesearcher that calls for the method ‘createupdatesearcher.' This method allows the script to perform operations that involve Windows updates (IUpdateSession, 2017). Using the $updatesearcher variable the script then uses the method ‘search’ to run the search for updates on a system (IUpdateSearcher, 2017). This work then adds up the total number of critical and important updates that are found to be missing on a system and returns that number to the main script. The value also is printed to the command screen under the system name. The format of this print to the screen is ‘Total=#,' with the # representing the total number of critical and important patches missing.



```
$windowsupdate =
{
#Below does a search for Critical and Important updates that have not be installed
$UpdateSession = New-Object -ComObject Microsoft.Update.Session
$UpdateSearcher = $UpdateSession.CreateUpdateSearcher()
$SearchResult = $UpdateSearcher.Search("IsAssigned=1 and IsHidden=0 and IsInstalled=0")
#Below writes the amount of Important and Critical patches missing on the remote system to the screen
Write-Host "total=$($SearchResult.updates.count)"
#Below exports the results (total number of patches missing) to the $windows update variable in the main scri
Write-Output $SearchResult.updates.count
}
##############End of Windows updates on remote PSSession
```

gure 3 – PSSession

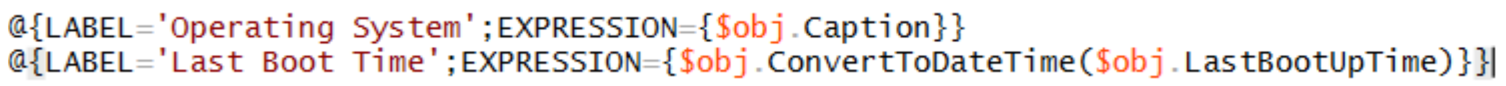
The next function called is the WMIC command that gathers the results that return the operating system and last boot time of that system, as seen in Figure 4 below.



```
#Declaring the value for the WMI call that will pull system OS and Last Boot time
$obj = Get-WmiObject Win32_OperatingSystem -computername $SystemName
```

gure 4 – WMI call to remote machine

The variable in Figure 4, $obj is then called and stored in the main body of the script in the $RemoteReturnedResult variable using the lines shown in Figure 5.



```
@{LABEL='Operating System';EXPRESSION={$obj.Caption}}
@{LABEL='Last Boot Time';EXPRESSION={$obj.ConvertToDateTime($obj.LastBootUpTime)}}
```

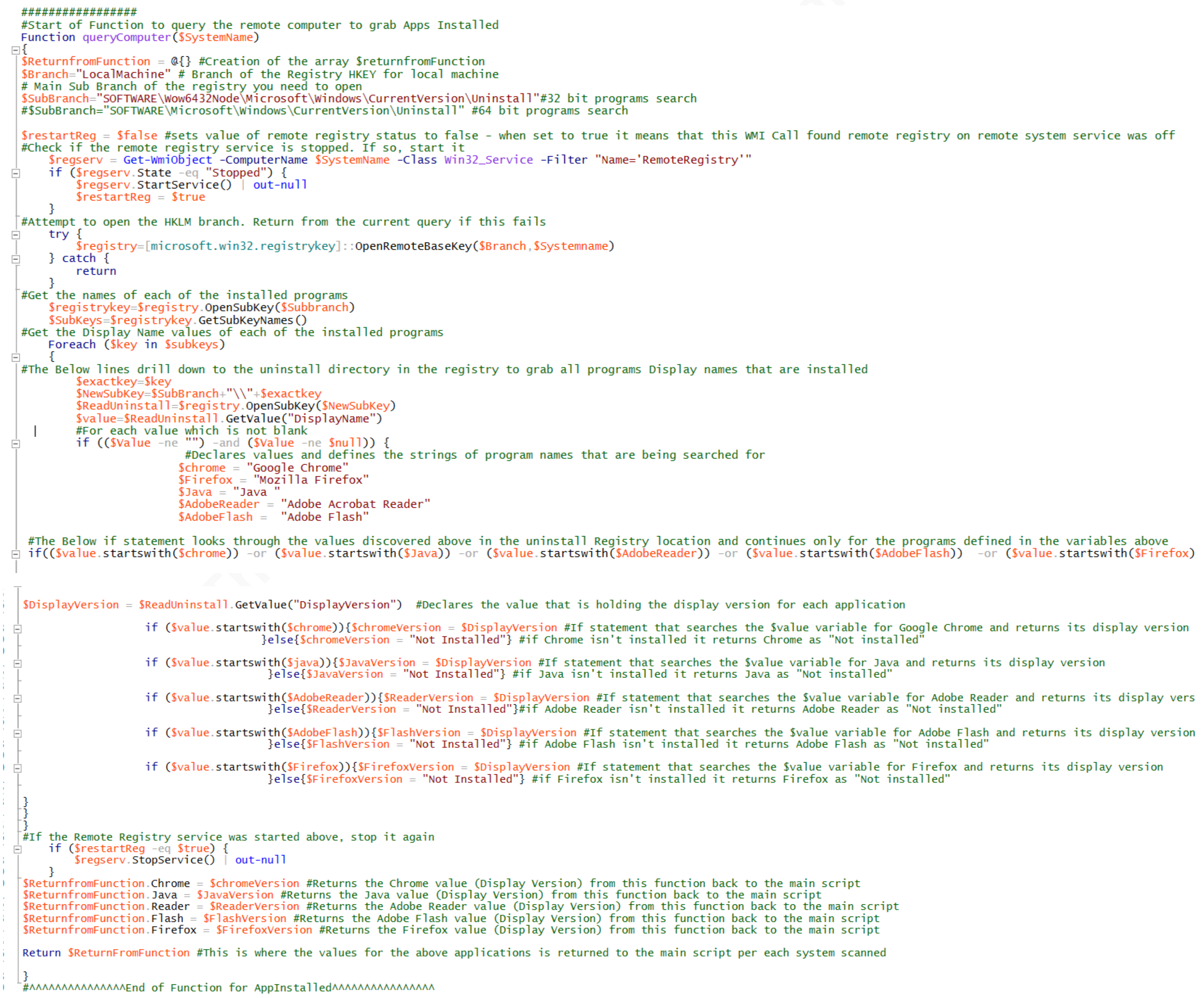
gure 5 – WMI Call to remote system for Operating system and Last boot time

The third function in this script is named querycomputer, shown in Figure 6. It starts with setting the variable $ReturnfromFunction as an array. This variable is used at the end of this function to pass the installed applications values to the central part of the script. Next, the two directories in the registry that the function looks at to gather information are defined. The $Branch variable specifies to look in the Local Machine branch of the system. The $SubBranch variable defines the registry key path to use for the script. By default, it is looking in the directory where 32-bit programs reside. It can be easily adjusted to scan the 64-bit directory by removing the comment symbol (#) in front of the greened out $SubBranch variable and placing it in front of the non-greened out $SubBranch variable.

To test to make sure that the remote registry is running a WMIC command is run to check the status of the service. The command turns the Remote Registry service on so a query

Colm Kennedy

can run against the remote systems registry. Once enabled, the script attempts a connection to the remote registry in line thirty-nine. If that is successful, the function continues to tunnel down into the registry retrieving all the ‘Display name’ values in the ‘uninstall’ folder and stores them in the variable $subkeys.



```
#################
#Start of Function to query the remote computer to grab Apps Installed
Function queryComputer($SystemName)
{
$ReturnfromFunction = @{} #Creation of the array $returnfromFunction
$Branch="LocalMachine" # Branch of the Registry HKEY for local machine
# Main Sub Branch of the registry you need to open
$SubBranch="SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall"#32 bit programs search
#$SubBranch="SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall" #64 bit programs search

$restartReg = $false #sets value of remote registry status to false - when set to true it means that this WMI Call found remote registry on remote system service was off
#Check if the remote registry service is stopped. If so, start it
    $regserv = Get-WmiObject -ComputerName $SystemName -Class Win32_Service -Filter "Name='RemoteRegistry'"
    if ($regserv.State -eq "Stopped") {
        $regserv.StartService() | out-null
        $restartReg = $true
    }
#Attempt to open the HKLM branch. Return from the current query if this fails
    try {
        $registry=[microsoft.win32.registrykey]::OpenRemoteBaseKey($Branch,$Systemname)
    } catch {
        return
    }
#Get the names of each of the installed programs
    $registrykey=$registry.OpenSubKey($Subbranch)
    $SubKeys=$registrykey.GetSubKeyNames()
#Get the Display Name values of each of the installed programs
    Foreach ($key in $subkeys)
    {
#The Below lines drill down to the uninstall directory in the registry to grab all programs Display names that are installed
        $exactkey=$key
        $NewSubKey=$SubBranch+"\\"+$exactkey
        $ReadUninstall=$registry.OpenSubKey($NewSubKey)
        $value=$ReadUninstall.GetValue("DisplayName")
  |     #For each value which is not blank
        if (($Value -ne "") -and ($Value -ne $null)) {
                        #Declares values and defines the strings of program names that are being searched for
                       $chrome = "Google Chrome"
                       $Firefox = "Mozilla Firefox"
                       $Java = "Java "
                       $AdobeReader = "Adobe Acrobat Reader"
                       $AdobeFlash =  "Adobe Flash"

 #The Below if statement looks through the values discovered above in the uninstall Registry location and continues only for the programs defined in the variables above
 if(($value.startswith($chrome)) -or ($value.startswith($Java)) -or ($value.startswith($AdobeReader)) -or ($value.startswith($AdobeFlash))  -or ($value.startswith($Firefox)


$DisplayVersion = $ReadUninstall.GetValue("DisplayVersion")  #Declares the value that is holding the display version for each application

                  if ($value.startswith($chrome)){$chromeVersion = $DisplayVersion #If statement that searches the $value variable for Google Chrome and returns its display version
                                    }else{$chromeVersion = "Not Installed"} #if Chrome isn't installed it returns Chrome as "Not installed"

                  if ($value.startswith($java)){$JavaVersion = $DisplayVersion #If statement that searches the $value variable for Java and returns its display version
                                     }else{$JavaVersion = "Not Installed"} #if Java isn't installed it returns Java as "Not installed"

                  if ($value.startswith($AdobeReader)){$ReaderVersion = $DisplayVersion #If statement that searches the $value variable for Adobe Reader and returns its display vers
                                     }else{$ReaderVersion = "Not Installed"}#if Adobe Reader isn't installed it returns Adobe Reader as "Not installed"

                  if ($value.startswith($AdobeFlash)){$FlashVersion = $DisplayVersion #If statement that searches the $value variable for Adobe Flash and returns its display version
                                     }else{$FlashVersion = "Not Installed"} #if Adobe Flash isn't installed it returns Adobe Flash as "Not installed"

                  if ($value.startswith($Firefox)){$FirefoxVersion = $DisplayVersion #If statement that searches the $value variable for Firefox and returns its display version
                                     }else{$FirefoxVersion = "Not Installed"} #if Firefox isn't installed it returns Firefox as "Not installed"

}
}
}
#If the Remote Registry service was started above, stop it again
    if ($restartReg -eq $true) {
        $regserv.StopService() | out-null
    }
$ReturnfromFunction.Chrome = $chromeVersion #Returns the Chrome value (Display Version) from this function back to the main script
$ReturnfromFunction.Java = $JavaVersion #Returns the Java value (Display Version) from this function back to the main script
$ReturnfromFunction.Reader = $ReaderVersion #Returns the Adobe Reader value (Display Version) from this function back to the main script
$ReturnfromFunction.Flash = $FlashVersion #Returns the Adobe Flash value (Display Version) from this function back to the main script
$ReturnfromFunction.Firefox = $FirefoxVersion #Returns the Firefox value (Display Version) from this function back to the main script

Return $ReturnFromFunction #This is where the values for the above applications is returned to the main script per each system scanned

}
#^^^^^^^^^^^^^^^End of Function for AppInstalled^^^^^^^^^^^^^^^
```

gure 6 –QueryComputer function that returns installed application versions

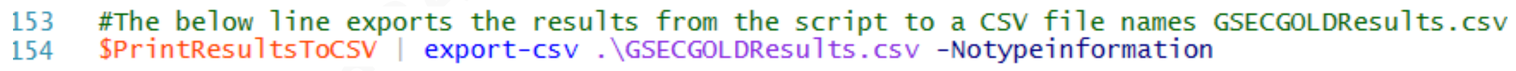
A loop is then run for all values within $subkeys variable one at a time to find the desired applications. It does this by placing each value of $subkeys into the variable $Key and then

Colm Kennedy

running through the rest of the function. In lines fifty-six through sixty-one the five applications are defined and given corresponding variable names. An 'if statement' is then set to continue with the function only if the $Key value starts with one of the five applications specified above. Then for each application an if statement is declared to return either the installed version of that software or if not found the value "Not Installed" is returned. At the bottom of the function, in lines ninety through ninety-four, the results are sent back to the main script using reference variables to pass values out of a function. Also, at the end of this function is a follow-up to stop the remote registry service if previously started, returning the service of the remote system to its original state.

4.2.4 The Script – Part 4

The fourth part of the script is the export to a CSV file of all the data collected. The export of the results takes the array that was defined at the beginning of the script and pipes it to a file named GSECGOLDResults.csv. The export runs by using the export-CSV command with the $PrintResultsCSV array, shown in Figure 7. This command contains all the results stored in it during the main scripts loop through each system.

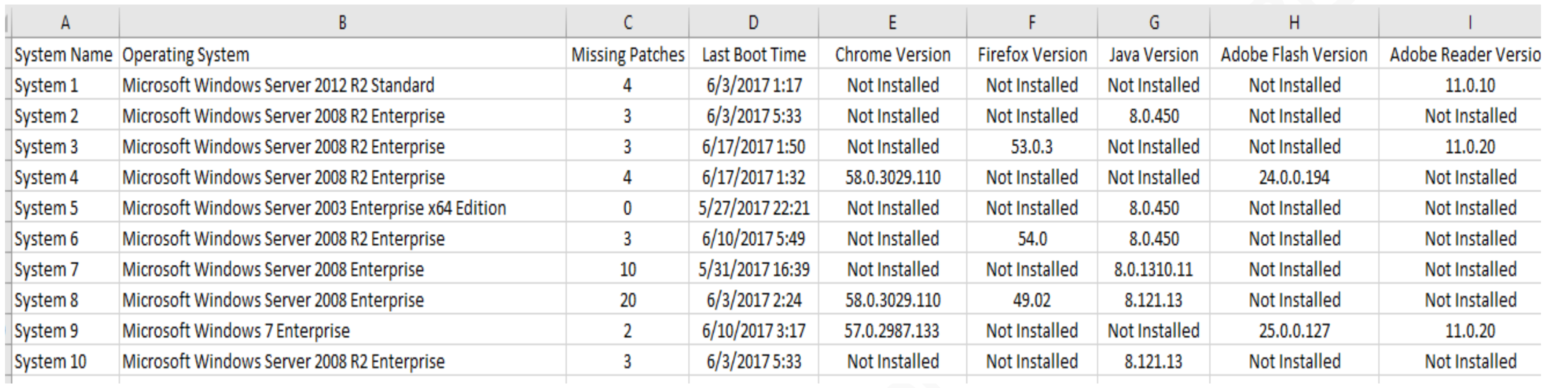


```
#The below line exports the results from the script to a CSV file names GSECGOLDResults.csv
$PrintResultsToCSV | export-csv .\GSECGOLDResults.csv -Notypeinformation
```

gure 7 – Export to CSV file all results

4.2.5 The Script – Results

After the script has completed scanning all systems in the system_list.txt and the results get exported to a CSV file named GSECGOLDResults.CSV the file can be opened to reveal the returned data. The results, shown in Figure 8, are easy to browse across to see what systems need further investigation. Also, it gives a quick way to see if any software is outdated and should be uninstalled or updated to the latest version.



A	B	C	D	E	F	G	H	I
System Name	Operating System	Missing Patches	Last Boot Time	Chrome Version	Firefox Version	Java Version	Adobe Flash Version	Adobe Reader Versio
System 1	Microsoft Windows Server 2012 R2 Standard	4	6/3/2017 1:17	Not Installed	Not Installed	Not Installed	Not Installed	11.0.10
System 2	Microsoft Windows Server 2008 R2 Enterprise	3	6/3/2017 5:33	Not Installed	Not Installed	8.0.450	Not Installed	Not Installed
System 3	Microsoft Windows Server 2008 R2 Enterprise	3	6/17/2017 1:50	Not Installed	53.0.3	Not Installed	Not Installed	11.0.20
System 4	Microsoft Windows Server 2008 R2 Enterprise	4	6/17/2017 1:32	58.0.3029.110	Not Installed	Not Installed	24.0.0.194	Not Installed
System 5	Microsoft Windows Server 2003 Enterprise x64 Edition	0	5/27/2017 22:21	Not Installed	Not Installed	8.0.450	Not Installed	Not Installed
System 6	Microsoft Windows Server 2008 R2 Enterprise	3	6/10/2017 5:49	Not Installed	54.0	8.0.450	Not Installed	Not Installed
System 7	Microsoft Windows Server 2008 Enterprise	10	5/31/2017 16:39	Not Installed	Not Installed	8.0.1310.11	Not Installed	Not Installed
System 8	Microsoft Windows Server 2008 Enterprise	20	6/3/2017 2:24	58.0.3029.110	49.02	8.121.13	Not Installed	Not Installed
System 9	Microsoft Windows 7 Enterprise	2	6/10/2017 3:17	57.0.2987.133	Not Installed	Not Installed	25.0.0.127	11.0.20
System 10	Microsoft Windows Server 2008 R2 Enterprise	3	6/3/2017 5:33	Not Installed	Not Installed	8.121.13	Not Installed	Not Installed

gure 8 – CSV Results

This format saves much time looking through these results instead of having to manually check each system for missing patches or old software versions. In this example, the two systems that would need some additional searches would be System 8 and System 7. With both missing more Microsoft patches than expected, further investigation would occur. However, because of this script, the administrator only needs to check these two systems and not all ten for missing Microsoft patches. Thus, saving time and money avoiding the need to check each of the other eight systems manually.

5. Life after Administrator Leaves

An important part of any script that gets created or used by an administrator in an organization is what happens when that person leaves. It is always important to create scripts that are transferable and easy to follow for those that do not know the scripting language. In this case within the PowerShell script, there are many notes to detail what is happening as it proceeds. This paper also stands as a guide to follow what is going on in the script and how it can be adjusted to fit the organization's needs at any given time. A new PowerShell user should make notes in all scripts so that others can reference what the thought process is or was when writing a script. It is important to point out that the creation of this script started with no experience in PowerShell. The script was pieced together using knowledge gained from walking through other scripts, understanding the logic using these scripts accompanying notes, Google searches and lots of trial and error.

6. Conclusion

PowerShell is a powerful tool that all administrators should learn. It allows administrators the ability to quickly scan systems to gather pertinent information that helps in day-to-day activities. Using the script provided as an example, it saves a lot of manual effort and time and gives a quick overview of system status without the need for additional agents installed like SCCM. Using this script, an administrator can also change what software you are searching for on remote systems. All that is required is a quick switch to the application variable values from line 61-66 on the script.

Using this script and other PowerShell scripts allows an administrator time to work on other tasks while eliminating the need for manual checks of individual systems for information gathered in these scripts. If results are in a readable format, as is the case with the results of the script in this paper, it allows for quick review and serves its purpose of saving time parsing through possible false positives.

This script is intended for administrators with budget constraints or those looking for additional, no cost alternatives to gathering information. As proven in the survey, there is a need for a script like this to assist Administrators in collecting information without the need to spend money on tools or time installing agents across an environment. PowerShell comes built in on supported Microsoft Windows operating systems. With the right scripts, PowerShell can give administrators enough insight into their systems to get the job completed promptly.

References

COM Objects and Interfaces. (n.d.). Retrieved May 22, 2017, from https://msdn.microsoft.com/en-us/library/windows/desktop/ms690343(v=vs.85).aspx

Creating an Array of Custom Objects in Powershell. (n.d.). Retrieved May 22, 2017, from http://www.get-blog.com/?p=82

Guys, T. S. (2011, November 13). Use PowerShell to Quickly Find Installed Software. Retrieved May 22, 2017, from https://blogs.technet.microsoft.com/heyscriptingguy/2011/11/13/use-powershell-to-quickly-find-installed-software/

Guys, T. S. (2013, December 11). Use PowerShell to Create Remote Session. Retrieved May 22, 2017, from https://blogs.technet.microsoft.com/heyscriptingguy/2013/12/11/use-powershell-to-create-remote-session/

IUpdateSearcher interface. (n.d.). Retrieved May 22, 2017, from https://msdn.microsoft.com/en-us/library/windows/desktop/aa386515(v=vs.85).aspx

IUpdateSession interface. (n.d.). Retrieved May 22, 2017, from https://msdn.microsoft.com/en-us/library/windows/desktop/aa386854(v=vs.85).aspx

Oliness. (2012, June 22). List Remote Applications and exclude common applications. Retrieved May 22, 2017, from https://gallery.technet.microsoft.com/scriptcenter/List-Remote-Applications-4c2720d4/view/Discussions#content

Parankewich, S. (2016, January 07). Get Last Computer Boot Time or Up Time With PowerShell. Retrieved May 22, 2017, from http://powershellblogger.com/2016/01/get-last-computer-boot-time-or-up-time-with-powershell/

Prox, B. (2016, June 23). Finding Pending Updates Using PowerShell. Retrieved May 22, 2017, from https://mcpmag.com/articles/2016/06/23/finding-pending-updates.aspx

Team, M. (2017, April 14). Protecting customers and evaluating risk. Retrieved May 22, 2017, from https://blogs.technet.microsoft.com/msrc/2017/04/14/protecting-customers-and-evaluating-risk/

Team, M. (2017, May 12). Customer Guidance for WannaCrypt attacks. Retrieved May 22, 2017, from https://blogs.technet.microsoft.com/msrc/2017/05/12/customer-guidance-for-wannacrypt-attacks/

What is PowerShell? (n.d.). Retrieved May 22, 2017, from https://msdn.microsoft.com/en-us/powershell/mt173057.aspx

Cybertrend. (2016, January 21). Why Vulnerability Management Matters. Retrieved May 22, 2017, from https://www.secureworldexpo.com/industry-news/why-vulnerability-management-matters-0

Brenner, Bill. WannaCry: the ransomware worm that didn’t arrive on a phishing hook. Naked Security. Sophos. Retrieved 18 May 2017, from https://nakedsecurity.sophos.com/2017/05/17/wannacry-the-ransomware-worm-that-didnt-arrive-on-a-phishing-hook/