UTILIZING WINDOWS POWERSHELL FOR HOST-BASED IDS LOG MONITORING

CHARALABOS VAIRLIS

UNDERGRADUATE THESIS

Supervisor: Ioannis Mavridis, Associate Professor

Applied Informatics Department

UNIVERSITY OF MACEDONIA
ECONOMIC AND SOCIAL SCIENCES

Thessaloniki

June 2015

Abstract

The aim of this thesis was to study Windows PowerShell, identify its capabilities and utilize it for making a fully functional Host-Based Intrusion Detection System.

PowerShell is a command-line shell and scripting environment developed by Microsoft. The main purpose of PowerShell is to enhance and automate the management capabilities of Windows operating systems and the applications that run on Windows.

The theoretical background of the thesis is concentrated on Windows Server 2012 R2 operating system, PowerShell and PowerShell ISE scripting environment. The sources for the thesis were gathered both from literary and internet sources.

The practical section of the thesis is comprised of different self-defined modules and scripts written on *PowerShell ISE* for event log monitoring purposes, SQL Server Database access and Graphical User Interface (GUI) development for the visualization of events, using SQL Server 2012 Express and PowerShell 4.0 as the only tools. The custom modules and scripts include tasks for parsing event log and store events in database, scheduled task for filling the database with new events every ten minutes, tasks for connecting PowerShell and database and visualization of database information using windows forms and charts.

All management tasks in the thesis were performed in a virtual environment. The virtual environment was provided by GRNET's cloud service. It consisted of one virtual machine with Windows Server 2012 R2 installed.

Results of this study revealed that PowerShell has unlimited capabilities. It can be used by advanced users for dealing with their everyday tasks and by developers to make fully functional tools in a simpler environment than Visual Studio or other C# programming language.

Results of this study also provide images and snapshots of the author's tool that follow the standards of fully functional Host-Based Intrusion Detection System.

Acknowledgements

I would first like to thank my professor for the wonderful support and experience I had at University of Macedonia. He was available any time I needed help, he taught me how to overcome the problems of the real world and show me how to use and utilize new technologies. He guided me and encouraged me when I felt lost in the world of security and he taught me to focus on detail.

Moreover, I would also like to thank my parents for their support during my studies. They were beside me, every single time I needed them and they were willing to help me and calm me when I felt stressed. Without them I could not find the motivation and the energy to fulfill that dream of mine.

Table of Contents

List of Figures

List of Acronyms

GUI	Graphical User Interface
PS	PowerShell
ISE	Integrated Scripting Environment
SQL	Structured Query Language
CMD	Windows Command Prompt
ID	Identification/Identity/Identifier
IDS	Intrusion Detection System
CMDLET	Lightweight command that's used in Windows PowerShell Environment
CLI	Command-Line Interface
AD	Active Directory
OU	Organizational Unit
WMF	Windows Management Framework
WMI	Windows Management Instrumentation
OS	Operating System
OU	Organizational Unit
GM	Get-Member (PS cmdlet)
HTML	Hypertext Markup Language
XML	Extensible Markup Language
JSON	JavaScript Object Notation
CSS	Cascading Style Sheets
ID	Intrusion Detection
IDS	Intrusion Detection System
NIDS	Network Based Intrusion Detection System
HIDS	Host Based Intrusion Detection System

Chapter 1. Introduction

1.1 Theoretical Background

Microsoft has continually evolved its technology and has introduced some tools that can be used for advanced administration, data analysis and security.[1] These tools including Microsoft Event Viewer and Windows Firewall implement a Graphical User Interface (GUI) environment where you can hit some clicks to achieve a certain result. Nonetheless GUI tents to become more and more complex and this means that we go to a tone of clicking, spending a lot of time and sometimes without result. Furthermore, some advanced tasks are may be supported by the GUI. To overcome these problems and to help users work with something more interactive, in 2006 Microsoft introduced Windows PowerShell.

Windows PowerShell is the tool that transforms the mouse click experience to a keyboard click experience. It is awesome, powerful, and free. It allows administrators and auditors to gather information about active directory, access control list, event log, firewall, user accounts, group accounts, the domain and many others. It also provides full access to all the .NET Framework classes.

PowerShell has two components. A command-line console host application and an Integrated Scripting Environment (ISE). It consists of modules and modules contain commands (we call them cmdlets) and these cmdlets allow you to do things. PowerShell implements an object-oriented environment and this means that most of the cmdlets return objects and the console represents information of the objects as text. PowerShell is an interactive shell. This means that it is an environment where you can think about what do you want, you type it and you get it.

Microsoft, also wanted to make a tool become extensible. *PowerShell ISE* provides an environment where the developers can make self-defined modules, cmdlets and scripts and even build GUI applications.

Microsoft provides the largest client-side operating system on the planet. It is likely that an intrusion detection Analyst will be using a type of Microsoft Operating System as his main workstation. In the past many tools were developed in order to perform intrusion analysis. Hence, intrusion analysis can be performed without a lot of tools. One is the best tool that Microsoft created and this is the Windows PowerShell.

PowerShell as an analysis language can use the administrative capability to perform monitoring tasks, communicate with SQL Server Database and other security technologies such as: Firewall configuration, Active Directory and Windows Event Logs. In order to take advantage of the monitoring capability of PowerShell, an Analyst will need to learn how to script and use programmatic logic, which in PowerShell is not difficult [1].

1.2 Outline of the Thesis

This thesis is organized as follows. Chapter 2 provides background information about some basics concerns of security, emphasizing the Intrusion Detection Systems. Chapter 3 provides a brief presentation of the basic PowerShell components. In Chapter 4 some basics PowerShell Security concerns will be specified. Chapter 5 provides an overview of the author's Host-based IDS system implementation,

including all the PowerShell code that has been developed. Chapter 6 discusses the results of the implementation, providing screenshots with description. Finally Chapter 7 summarizes the PowerShell capabilities and also suggestions are presented for future work in upcoming evaluations.

1.3 Scope of Thesis and Implemented Tools

This thesis is aimed at utilizing a Host-Based Intrusion Detection System with only one tool in use, Windows PowerShell.

Chapter 2. Background

One of the fundamental concerns of computer operating systems is the security. Organizations have to be thoughtful about security if they want their infrastructure (*networks & computers*) to be secured. In this chapter we are going to refer about the terms of the *Security Policy* and the *Layered Security* that the organizations may adopt. Moreover we are going to analyze what an Intrusion Detection System (IDS) is, how it differs from an Intrusion Prevention System (IPS) and more specifically we are going to provide the components and a schema of Host-Based IDS.

2.1 Security Policy

A security policy identifies the rules and procedures that all persons accessing computer resources must adhere to in order to ensure the confidentiality, integrity and availability of data and resources [2]. In other words it includes all the necessary procedures that an organization has to follow in order to be protected.

2.2 Layered Security

Organizations need to focus on the information they are protecting and these days, the only way to do this, is to build layers of security around the organization infrastructure. In effect, they need to create a defense-in-depth solution [3].
Figure 1 gives a typical representation of security layers that organizations should consider.

LAYERED SECURITY MATRIX

SECURITY LAYER	PREVENTIVE CONTROLS								DETECTIVE CONTROLS					
	User Access Control	Network Access Control	Encryption	System Hardening	Software Patching and Updates	Malware Detection/Prevention	Security Awareness Training	Policies & Procedures	Change Control	Security Configuration Management	Log Monitoring	File Integrity Monitoring	Vulnerability Management	Incident Alerting
Physical Layer			✔	✔			✔	✔	✔					✔
Network Perimeter		✔	✔	✔	✔	✔			✔	✔	✔	✔	✔	✔
Local Area Network		✔	✔	✔	✔	✔			✔	✔	✔	✔	✔	✔
Wide Area Network		✔	✔	✔	✔	✔			✔	✔	✔	✔	✔	✔
Virtualization Hypervisor	✔			✔	✔	✔			✔	✔	✔	✔	✔	✔
Virtualization Virtual Network	✔	✔	✔	✔	✔	✔			✔	✔	✔	✔	✔	✔
Host System	✔			✔	✔	✔			✔	✔	✔	✔	✔	✔
Application	✔			✔	✔	✔			✔	✔	✔	✔	✔	✔
Intellectual Property	✔		✔								✔	✔		✔
Entrusted Data (e.g. PII)	✔		✔								✔	✔		✔
Sensitive Databases	✔		✔								✔	✔		✔
Sensitive Files	✔		✔								✔	✔		✔

Figure 1. Typical security layers organizations should consider [3]

All we can see in figure 1 is that all of the components of an enterprise computer infrastructure can be divided into 12 layers *(on the left side)*. After this, there are some security controls *(moving from left to right)* that the organizations can follow in order

to prevent or detect anomalies on their systems. As an example, in order to detect anomalies on Applications, organization can adopt security controls such as: Change Control, Security Configuration Management, Log Monitoring, File Integrity Monitoring, Vulnerability Management and Incident Alerting.

2.3 Intrusion Detection

Intrusion Detection can be defined as the act of detecting actions that attempt to compromise the confidentiality, integrity or availability of a resource. More specifically, the goal of intrusion detection is to identify entities attempting to subvert in-place security controls [4].

In computer science terms, Intrusion Detection is the process of monitoring the events occurring in a computer system or network and analyzing then to identify possible incidents, which are violations or imminent threats of violation of computer security policies [5].

2.3.1 Intrusion Detection System

Intrusion Detection Systems (IDS) are an important part of a layered security defense. [6]. An IDS is software that automates the intrusion detection process [5].

IDSs can be classified into two main categories:

- Host-Based IDSs: HIDS systems evaluate information found on a single or multiple host systems, including contents of operating systems, system and applications files [7].
- Network Based IDSs: NIDS systems evaluate information captured from network communications, analyzing the stream of packets which travel across the network [7].

2.3.2 IDS vs. IPS

Intrusion Prevention Systems (IPS) follow the same process of gathering and analyzing events, with the added ability to prevent an abusive activity in real time [4].

2.3.3 Host-Based Intrusion Detection System (HIDS)

Host-Based IDS systems detect attacks for an individual system, using system logs and other operating system audit trails [7].

2.3.4 Components of a HIDS

A HIDS system, or generally an IDS system consists of three major functional components namely [8]:

- An information source that provides a stream of event records *(data source)*.
- An analysis engine that finds signs of intrusion *(analysis engine)*.
- A response component that generates reactions based on the outcome of the analysis engine *(response engine)*.

These major components can be enhanced by adding a data storing engine and a visualization engine that will visualize all the information gathered. This extended approach of an IDS system components can be displayed as a flow, as shown in figure 2.

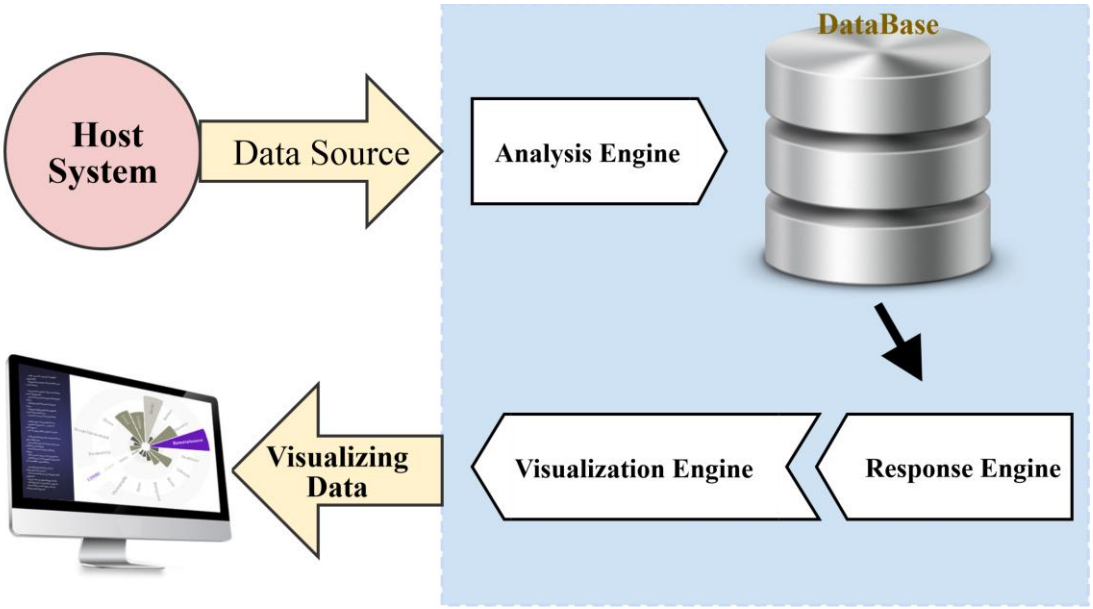


Figure 2. Simple IDS

Figure 2 provides an example of a simple IDS system. The Data Source *(events)* come from a host system *(e.g. Windows 8.1 machine)*; Analysis Engine analyzes and stores the data in a file or in a database; Response Engine collects the data and generates critical information, statistics and details. Finally the data has been analyzed and Visualization Engine is responsible to display the result of the detection.

2.4 Summary

Nowadays it is very important to maintain a high level security to ensure safe and trusted communications of information between various organizations. Data communications over the internet and any other network is always under threat of intrusions and misuses. So Intrusion Detection Systems have become a needful component in terms of computer and network security.

Chapter 3. Brief Presentation of PowerShell

3.1 Introduction to Windows PowerShell

While casual users may know about the Windows Command Prompt (CMD), very few have ever heard about **Windows PowerShell**. A short explanation should be that *PowerShell* is an object-oriented interface tool that is intended to replace the CMD and deliver more power and control over the Windows operating system.

In this chapter we are going to analyze what PowerShell is, what new comes with this tool and why should we try to learn it. After preparing our console with the latest core packages (WMF), we will see in more detail the basic components of PowerShell. These will be the powerful help system, cmdlets and aliases, providers, modules, objects & members (methods and properties), pipeline, running cmdlets, running statements and loops. Finally, we are going to introduce some key points of making tools with PowerShell.

3.1.1 What is Windows PowerShell

To give you a better understanding of PowerShell, we should first define what a *shell* is. In computer science, a *shell* is basically a user interface that gives you access to various services of an operating system. A shell can be command-line based or it can include a Graphical User Interface (GUI).

Windows PowerShell is a task-based command-line shell and scripting language developed by Microsoft for purposes of task automation, configuration management and especially for system administration. [9]

Built-on the .NET Framework, Windows PowerShell helps IT Professionals and power users control and automate the administration of the Windows Operating System and Applications that run on Windows. [9]

Microsoft delivers PowerShell within Windows Operating System installed by default in most versions. PowerShell 3.0 and later versions have two components: the standard, text-based console host (powershell.exe) and the more visual Integrated Scripting Environment (ISE; powershell_ise.exe). [10]

3.1.2 How PowerShell differs from the Command Prompt Processor

Windows PowerShell is actually very different from the Windows Command Prompt (CMD). It uses different commands, known as cmdlets in PowerShell. Many system administration tasks – from managing the registry to WMI – are exposed via PowerShell cmdlets, while they aren't accessible via the Command Prompt.

PowerShell makes use of pipes, just as Linux and other Unix-like systems do. Pipes allow you to pass the output of the cmdlet to the input of another cmdlet, using multiple cmdlets in sequence to manipulate the same data. Contrary to the Unix-like systems, which can only text (streams of characters), PowerShell pipes objects between cmdlets. This allows PowerShell to share more complex data between cmdlets.

In addition, PowerShell offers a scripting environment where you can create complex scripts for managing Windows systems in a much easier way than you could with the Command Prompt.

The Command Prompt is essentially an environment that copies all of the various DOS commands you would find on a DOS system. It does not support objects, it is painfully limited, it cannot access many Windows system administration features and it is more difficult to learn it, to mention just a few of its limitations.

PowerShell embeds all of the classic Command Prompt DOS commands. This means you can run commands such us *ipconfig, dir or cd* in PowerShell giving the same results with Command Prompt.

Finally we can probably understand that PowerShell came to the fore to replace the classic CMD and to give more power to advanced users and administrators to do things in more powerful and easy way.

3.1.3 New elements of Windows PowerShell

The new thing with PowerShell is that it is interactive. An environment where you can go through and explore the system, try several commands, be able to get things done, then you can execute more complex tasks repeatedly by putting all commands in a script and run them all at once. Then, as we reduce our repetitive strain injury, now we can pass to the next level and make modules where we can parameterize our needs to use for wide range of things and then we can share with others to manage thousands of machines. This model is called the Admin Development Model. [11]

What is important here, is that you are in an interactive environment so that you can explore and honestly have fun. Microsoft has transformed the mouse click experience to a keyboard click experience. At this point, we have to emphasize that we are unaccustomed to Windows environments having an interactive shell, a real-time solving problems tool. We have always had basic commands but that was not really solving our problems. This means that now we are able to solve your problems, then literally copy-paste or hit save and now we can automate it. It does not require anymore than have fun finding and solving problems. [11]

3.1.4 Why should I learn PowerShell?

That is the key question and we will answer it.
The answer is that everybody's scenario is different.

Initially Microsoft spent a lot of time talking to customers trying to understand their scenarios and put them into the GUI innovation. But for many different types of users, the GUI did not help and they had to go to a tone of clicking to perform their tasks. In other words, GUI tends to become more and more complex and as a result not functional.

For instance, if it takes you five minutes to create a new user in Active Directory (AD) and assuming you are filling in a lot of the fields, that is a reasonable estimate, you will never get any faster than that. One hundred users will take five hundred minutes – there is no way to make the process go any quicker, short of learning to type and click faster.

There is nothing wrong with using the GUI. Anybody can use this for things you just need to do once. For example, creating a site link in AD, GUI is great, but when you have business challenges that can be met, such as to set the mailbox limits of a

particular Organizational Unit (OU) in AD, the GUI doesn't support that and one does not have to expect Microsoft to solve that problem because they are not supposed to know all of our issues. [11]

Microsoft continues to build GUI consoles, but many of those are executing PowerShell commands behind the scenes. That approach forces the company to make sure that every possible thing you can do with the product is accessible through PowerShell. [12]

So, PowerShell, is the way that you can deal with any kind of task, as many times as you will be requested to, reducing repetitiveness by writing and using scripts and modules, giving always best results, with one tool in use.

Generally speaking, PowerShell:

- Is easy to learn
- Collects useful information that GUI even cannot see
- Deals with complex tasks in an easy way
- Enables Remoting
- Has Fun

3.1.5 PowerShell Versions

Microsoft first official release of PowerShell was in 2006 with PowerShell 1.0. After this, PowerShell 2.0 and PowerShell 3.0 were major releases, with many new important features. Some of the v2 and v3 new features are PowerShell Remoting (WS-Management), Background Jobs (PSJobs), Modules (for creating self-contained reusable units), Eventing (for managing system events), ISE (GUI-based host for scripting), Scheduled Jobs, Help Update, New commands, New cmdlets, New providers, New operators, and so on.

One year after PowerShell 3.0 release, in 2013 PowerShell 4.0 was released and was integrated into Windows 8.1 and Windows Server 2012 R2. PowerShell 4.0 is also available for older versions such as Windows 7 by downloading and installing the Windows Management Framework 4.0 (see next section).

Finally in April 2014, Microsoft released a preview of PowerShell 5.0 with Windows Management Framework Core 5.0 package. In February 2015 the latest WMF 5.0 Preview was announced with 2 stable and 5 experimental new scenarios. [13]

This thesis it was written under the PowerShell 4.0 era. If you are working on Windows 7 which has PowerShell 2.0 installed by default, you can read about updating your console in next section. If, however, you cannot update your console to the latest version, you can begin with typing “Get-Help” or explore the available commands by typing “Get-Command”.

3.1.6 Where to find and run PowerShell?

There are several ways to find and run PowerShell executable files.

- On older versions of Windows you can navigate from *Start Menu: All Programs > Accessories > Windows PowerShell. You can also select Run from Start Menu, type PowerShell.exe, and hit Enter to open the PowerShell console application.*
- On Windows 8.1 and Windows Server 2012, *hold the Windows key on your keyboard and press R to get the Run dialog box. Or, press and release the Windows key, and start typing "powershell" to quickly get to the PowerShell icons.*

If you running a 32-bit OS, you have probably only 32-bit PowerShell applications. In contrast, 64-bit OS have both 64-bit and 32-bit versions, and the 32-bit versions include "x86" in their icon names. This means you have to select and run the appropriate version of PowerShell in order to have best functionality. [14]
Once you have found PowerShell application, click to run it. Have in mind that in order to have full administrative access to receive best results from PowerShell you have to run the program with administrator privileges ("Run as Administrator").
After running PowerShell you are able to see the console, as shown in figure 3.

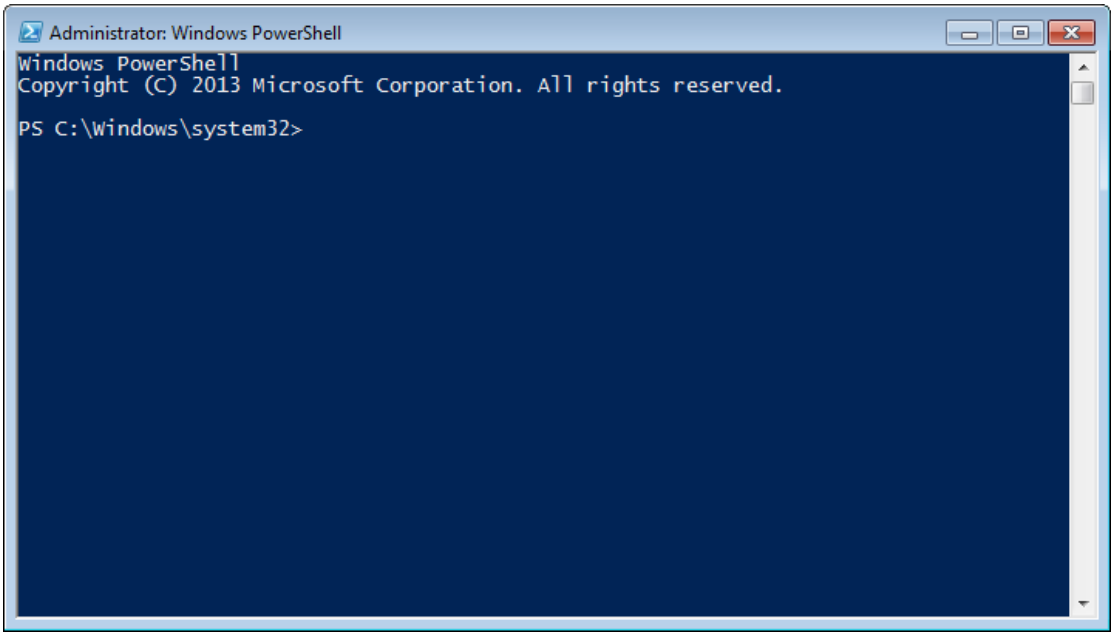


Figure 3. Console after opening PowerShell as an Administrator.

As you can see, the top of the window shows you the current user running PowerShell. *Administrator: Windows PowerShell* allows you to do administration. Just below of the top border of the window there is information that shows you that Windows PowerShell runs in the console and then there's a "***PS***" that differs from the usual CMD. Current working directory comes after the "PS" prompt and this is how you can start using PowerShell.

3.2 Preparing PowerShell

3.2.1 Update Windows Management Framework 4.0

In order to use a fully functionally Windows PowerShell, you have to download and install the latest Windows Management Framework core packages.

If you are running a Windows 8.1 or Windows Server 2012 R2 machine, you have the latest Windows Management Framework packages installed by default and you can proceed to the next section of this chapter.

Windows Management Framework 4.0 (WMF 4.0) makes updates management functionality available for installation on Windows 7 SP1, Windows Server 2008 R2 SP1 and Windows Server 2012.

For this installation sample needs, we are going to download the WMF 4.0 from Microsoft download center and then we install it on our Windows 7 machine. Figure 4 shows the specific WMF 4.0 on Microsoft Download Center. [15]

Figure 4. Downloading WMF 4.0 from microsoft.com.[15]

WMF 4.0 – Install Instructions

1. **Download the correct package for your operating system and architecture**. The following architectures are supposed.
   - Windows 7 SP1
     - x64: Windows6.1-KB2819745-x64-MultiPkg.msu
     - x86: Windows6.1-KB2819745-x86.msu
   - Windows Server 2008 R2 SP1
     - x64: Windows6.1-KB2819745-x64-MultiPkg.msu
   - Windows Server 2012
     - x64: Windows8-RT-KB2799888-x64.msu
2. **Close all Windows PowerShell windows**.
3. **Uninstall any other copies of WMF 4.0, including any prerelease copies or copies in other languages.**

To install WMF 4.0 from Windows Explorer (or File Explorer in Windows Server 2012)

1. Navigate to the folder into which you downloaded the MSU file
2. Double-click the MSU to run it.

You can find full instructions for the installation of WMF 4.0 procedure at the end of the document [15].

TIP

PowerShell requires .NET Framework v4 at a minimum, and it prefers to have the latest and greatest version of the framework that you can get. Microsoft recommends also installing at least .NET Framework v3.5 SP1 and .NET Framework v4.0 to get the maximum functionality from the shell.[10]

After downloading and installing WMF 4.0 you can run PowerShell as Administrator and type "Get-Host" to verify that the new core installed successfully, as is shown in figure 5.

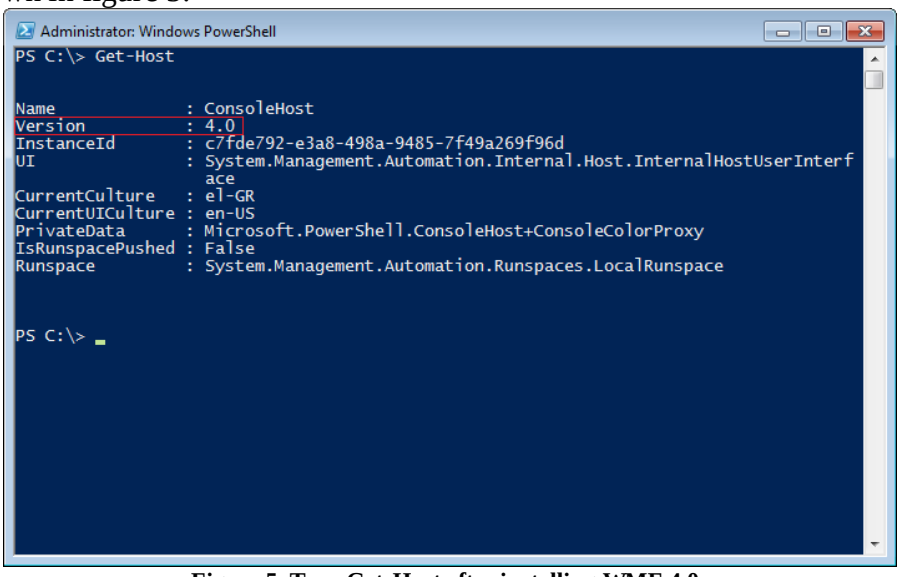


Figure 5. Type Get-Host after installing WMF 4.0.

3.2.2 Customize the shell

Before you go further, take a few minutes to customize the shell.

Click the control box (that's the PowerShell icon in the upper left of the console window) or right click at the top of the window border and select Properties from the menu.

In the dialog box that appears, browse through the various tabs to change the font, window colors, window size and position, and so forth.

TIP

Make sure that both the Window Size and Screen Buffer have the same Width values.[10]

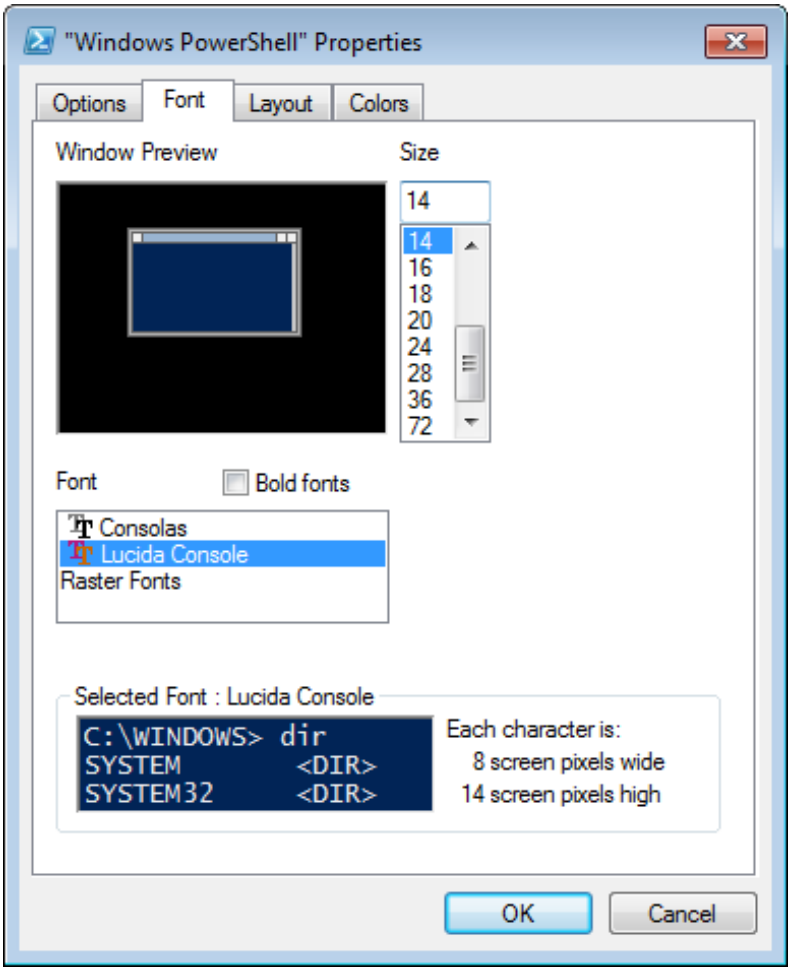


Figure 6. Windows PowerShell properties dialog box.

3.3 PowerShell Console Host Application Components

3.3.1 Powerful Built-in Help System

This is probably the most important section of this chapter. When you go to PowerShell for the first time, you are going to stare at the console and you will not know what to do. You look at the prompt wondering what you are going to do.

The answer is in the: **The Help System**. [16]

3.3.1.1 Why do you need help

Help System has been created to help you. You do not have to memorize things. Using the Help System facilitates figuring out how to do things. With Help System, you are going to learn how to learn and then you can use those techniques over and over again.

3.3.1.2 Update Help Feature

Help is basically the documentation of PowerShell. Microsoft used to ship the help with the product and it turned out that there were problems with that. When they had to change things on the product, many differences between the product and the documentation appeared and the users has confused about what instructions to follow or how to proceed. Furthermore, the Help System is huge. Enterprises do not need the help file installed on all servers and client computers. They only need help on one machine which the administrator uses.

Thus, due to these problems, Microsoft moved to an **Updatable Help Model**. It is a structural help system based on metadata, but all the help text is available for download from the internet. So, you can download it every single day to make sure you have the latest and greatest help.

For updating Help System, there is a cmdlet called "Update-Help". Make sure you are able to connect to the Internet to do this. As you can see in figure 7, we type "Update-Help -Force" and this is going to go out to the Internet and download all of the help files that you need for all the stuff that you have on system.

You can see the progress bar, too.

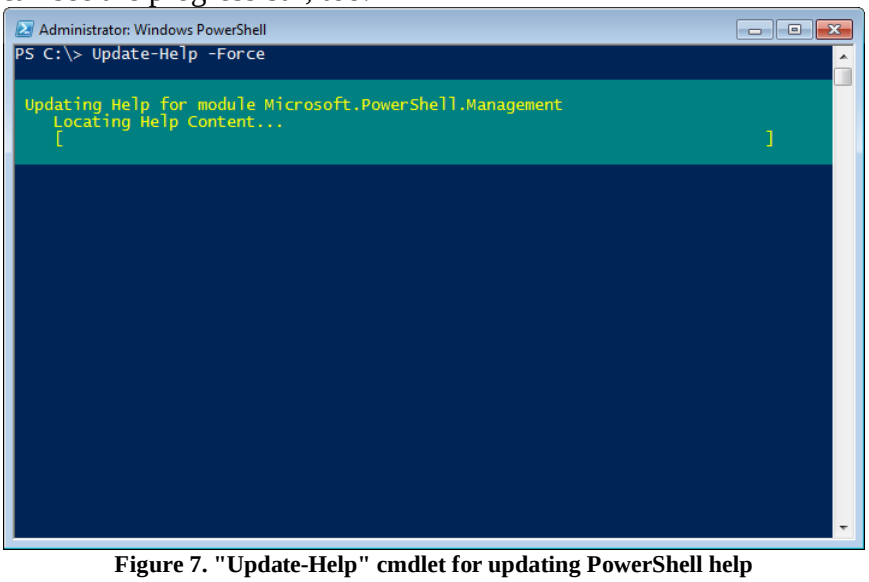


Figure 7. "Update-Help" cmdlet for updating PowerShell help

NOTE
Now, you can also save the help files you just downloaded to use them on a machine that it is not plugged on the Internet. Surely, having the internet connection does make you life much easier.

3.3.1.3 Discoverability with the Help System

Once we make sure we have uploaded help, we can start using "Get-Help" cmdlets in PowerShell console. Many people simply use "help" cmdlet. We can spot the differences between *Get-Help* and *help* or *man* cmdlet. To figure this difference out, we are going to type these commands and we probably can spot the difference.

Figure 8 shows the output from typing "Get-Help Get-Service" that shows the help file of "Get-Service" cmdlet.

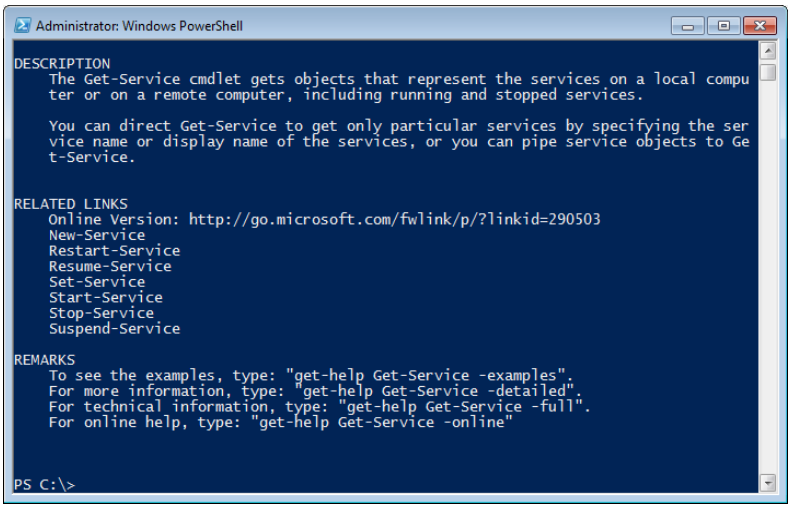


Figure 8. Output from typing "Get-Help Get-Service".

Figure 9 shows the output from typing "help Get-Service" that shows the help file of "Get-Service" cmdlet.

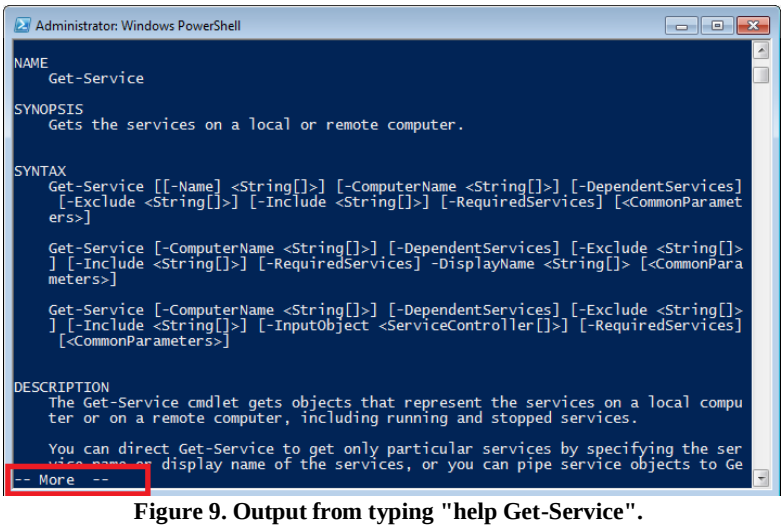


Figure 9. Output from typing "help Get-Service".

Figure 10 shows the output from typing "man Get-Service" that shows the help file of "Get-Service" cmdlet.

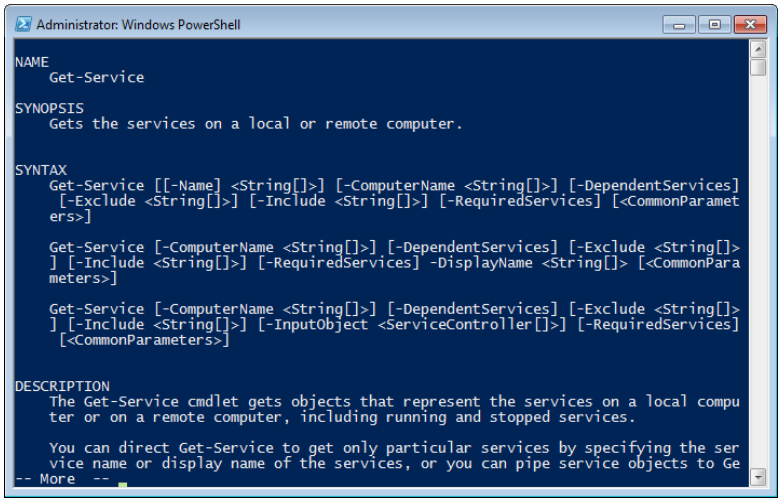


Figure 10. Output from typing "man Get-Service"

All we can see is that Help system gives us the *help text file* for the Get-Service cmdlet. So, the difference is what style or way we prefer reading this help file. By default, with "Get-Help" cmdlet *(figure 8)*, the console window shows us the text as one long, scrolling topic. On the other hand, if you prefer to see the first page of the help text file which fits to your console window and move through it with hitting enter or spacebar, you can rather use *help* or *man* cmdlet *(figure 9-10)*.

Help System is a great super discoverable great tool to help me find things. Now, we are going to point out some key uses of the Help System.

In figure 11 we are going to discover any cmdlets that can help us work with services.

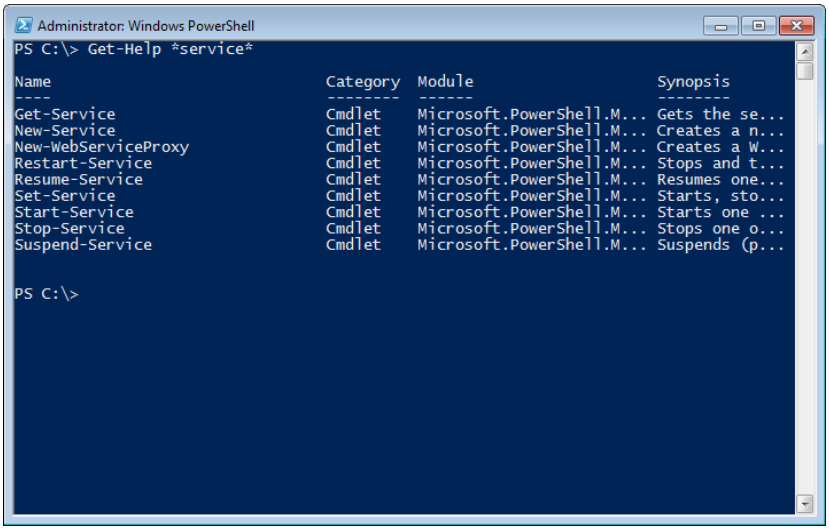


Figure 11. Typing "Get-Help *service*" outputs commands that can help us work with services.

TIP
The '' here is our best friend, it is like a wild card and helps us find any cmdlet that has inside the phrase we want.*

Other usage is to take all of the cmdlets that start with 'G' and contain the word service inside. As figure 12 shows, typing "Get-Help g*service*", it automatically gives us the Get-Service help file, because it found only one help file that matches with our search query.

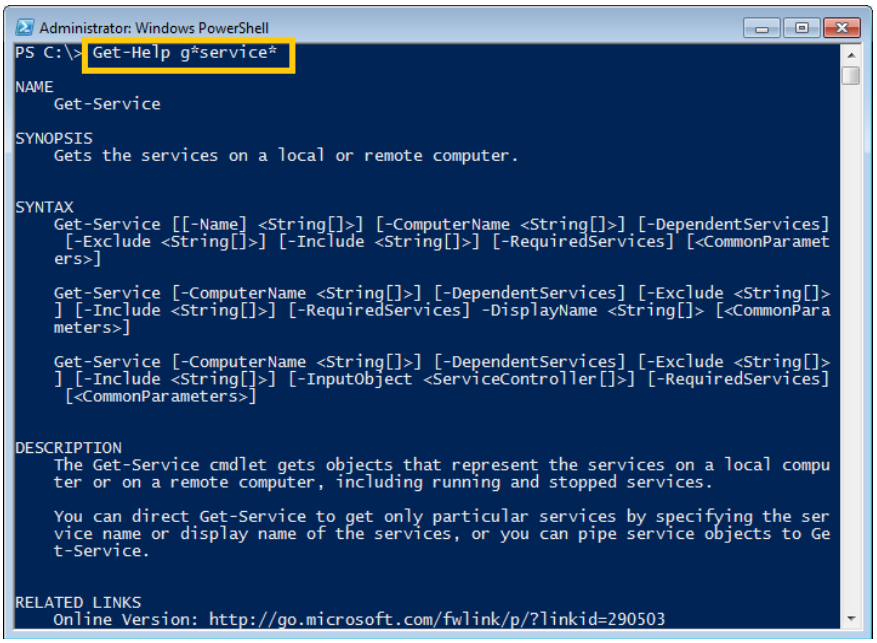


Figure 12. Output from typing "Get-Help g*service*".

Here is the idea of using Help System, and you are going to do this couple of times.

The idea is that you search for anything that you probably want to learn about. You can search for by noun, and by verb. You can search for something to deal with processes, or about services and a lot of other stuff.

So, **you think about what you want, you search for it by using Get-Help cmdlet and you get it.**

Another thing we have to point out is that if you type "Get-Help Get-Service", you get the **simple view** of the help file. This simple view gives you the Name, Synopsis, Syntax, Description and some related links and information. This is not the Full Help.

Once we have updated help, this help can be expanded upon to give us more help. Commands have parameters and we point out below some parameters for Get-Help cmdlet that will allow us to get more help.

- Detailed
- Full
- Examples
- ShowWindow

When you go to the detailed view, you can see the selected cmdlet Parameters are listed under Description of simple view, with an explanation of what they mean. There is also a Full Help view option. The full help is very similar to detailed, except when you are in the parameters, there are some special information, very important to us, but we are going to describe about these in next section. Except from the definition of the parameters, scrolling down the detailed and full help, you get examples. Examples are most of the time, very useful. People that worked so hard on these help files knew that there were going to be all kinds of different ways that a cmdlet can be used and they wanted to give us examples. Imagine how useful these examples are. Often times you will be trying to solve a business problem, you will find an example that is really close to, probably exactly what you wanted to do. Figure 13 shows that you can get only examples of the cmdlet you are interested for by typing "Get-Help Get-Service –Examples".

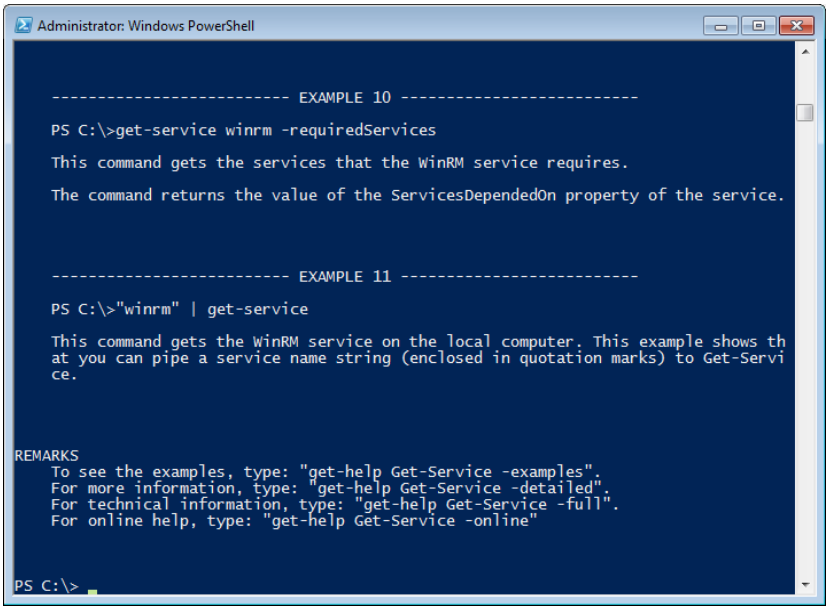


Figure 13

Microsoft made our life easier, learning all these deep level part of PowerShell. Exploring Help System let us discover that there are examples of sorting, filtering and getting kinds of information. Lot of times, lot of people go directly to examples.

We are going to spend so much time in the Help File, and there are many times you will want to copy paste from a help file, or you will want to read a help file while you are working on PowerShell. There is a way to do this, if you open two consoles concurrently. However, in PowerShell 3.0 and later versions, there is an easier way to do this, by using *-ShowWindow* parameter of Help System. This is a further view of Help System and if we type "Get-Help Get-Service -ShowWindow", this will open a window with the selected help file text, as shown in figure 14. There are some options for this window, to show only Examples or whatever we want, the Find textbox tool, where we can find an exact word inside the text, and the zoom function that help us get the help more easily. Also it fits better on our screen and it is easy to move around.

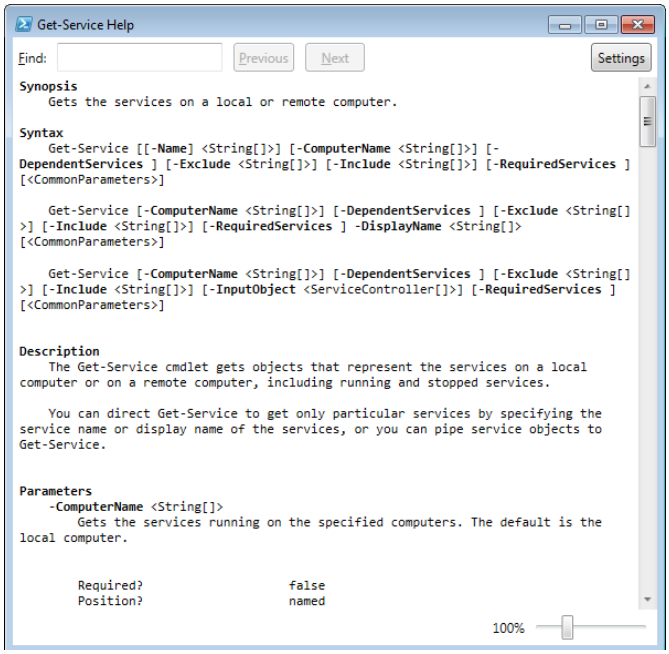


Figure 14. Output from typing "Get-Help Get-Service -ShowWindow".

TIP

*A nice cool thing of the console you might know is the **copy paste key**. The console does not allow Ctrl+C and Ctrl+V for copy and paste functions. The only way to use these functions is the **mouse right click key**. In particular, when you are in the PowerShell console, you can select a sequence of characters and hit right click for copy and then right click again for paste.*

3.3.1.4 Understanding Syntax

It is time to start explain Syntax of cmdlets. There is an easy way to start use the cmdlets, but sometimes you want a cmdlet to alter its output or to change what it is actually doing for you. For example, "Get-Service", it is a great cmdlet, but it is going to give you a list of all of your services. Maybe I just want the one called BITS, or something like that. So, the Syntax of the cmdlets, will explain to you the options that you have available to control the cmdlet.

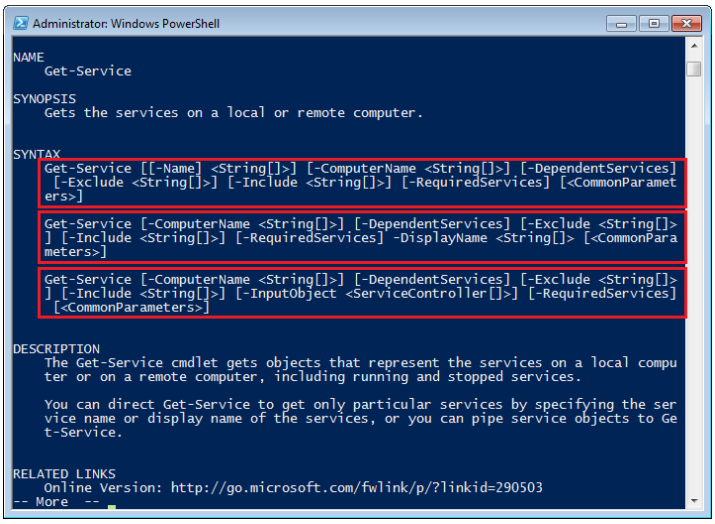


Figure 15. "Get-Service" cmdlet parameter sets.

Figure 15 shows the output from typing "Get-Help Get-Service". We will take a look at the syntax. We can see that there are three parameter sets. For every cmdlet there is different number of parameter sets. This is because every cmdlet have different types of syntax. For our case, the first one set has *-Name*, but down on the other you do not see *-Name*. The second one has *-DisplayName*, which you cannot see that at the other parameter sets.

The dashes "-" indicate that there are parameters. Parameters will allow us to change the results of what the cmdlet it is going to do.

We will now consider at the parameter *-Name*. Parameter syntax contain these characters "<[]>". We can see inside, a type of value followed by "[]". For our case after *-Name* we have a parameter with type of String. If we have these characters too "[]", this means that we can have multiple values, separated by a comma.

In order to test the syntax, we can type: "Get-Service -Name bits, w32time".

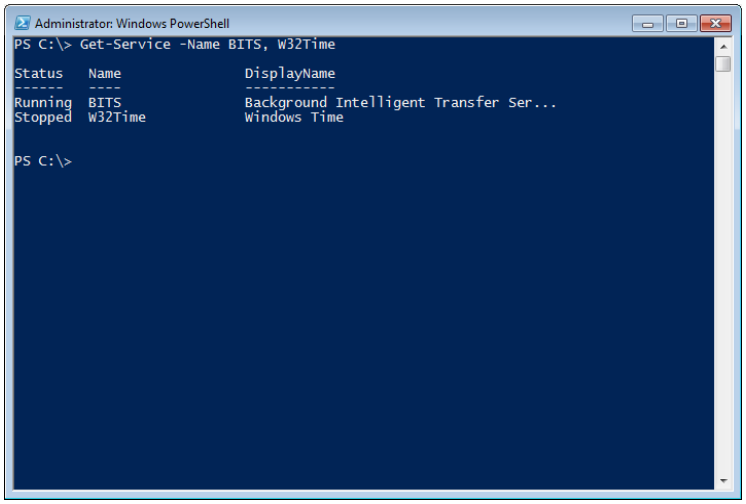


Figure 16. Output from typing "Get-Service -Name Bits, w32Time""

As figure 16 is shown, parameter *–Name* of "Get-Service" cmdlet accepts multiple inputs and outputs only the selected services that their names match with "BITS" and "w32Time".

TIP

It is good to say here that there is no conflict between wildcards and arrays, there are collections. We can have on our "Get-Service -Name" parameters something like: "Get-Service -Name b, w*". This will work fine.*

NOTE

You can get full information about wildcards by looking at the help file. Type: "Get-Help about_wildcards".

Other good thing we may point out for Syntax is the positional parameters. These are parameters like first *-Name* of the "help Get-Service" that are within [] brackets. If a parameter is inside these brackets, this means that if you just specify the value, you do not have to specify "-Name". More specifically, this means I can get the same outputs if I would type "Get-Service -Name bits", whether if I would type "Get-Service bits".

Now, it is upon everyone for how to use syntax. It is not bad to write self describing cmdlets, but in most cases, in cases of writing a script for example, you have to use all Parameters if you want your script to be absolutely readable.

NOTE
For our everyday tasks, we do not have to use all of the -Name or -Exclude or anything else parameters. However, there is a good way to exceed our speed limits. Try typing Get-Se and then hit the TAB key. This will automatically display Get-Service in your console. Continue using TAB keystroke after "-" to display parameters. Generally you can have your finger on the TAB key to be able to use it every moment and to get rid of wasting time.

3.3.1.5 Using Help to deal with a task Paradigm

As kind of a summary here we are going to show a way that help system and syntax help us to deal with a task.

Consider that we need to look for finding the newest 5 system errors out of our system log. Thus, we need to be able to look at log files. We are wondering if PowerShell has a cmdlet that could help us find more information about eventlog.

We type "Get-Help *eventlog*", and we take back plenty of cmdlets.

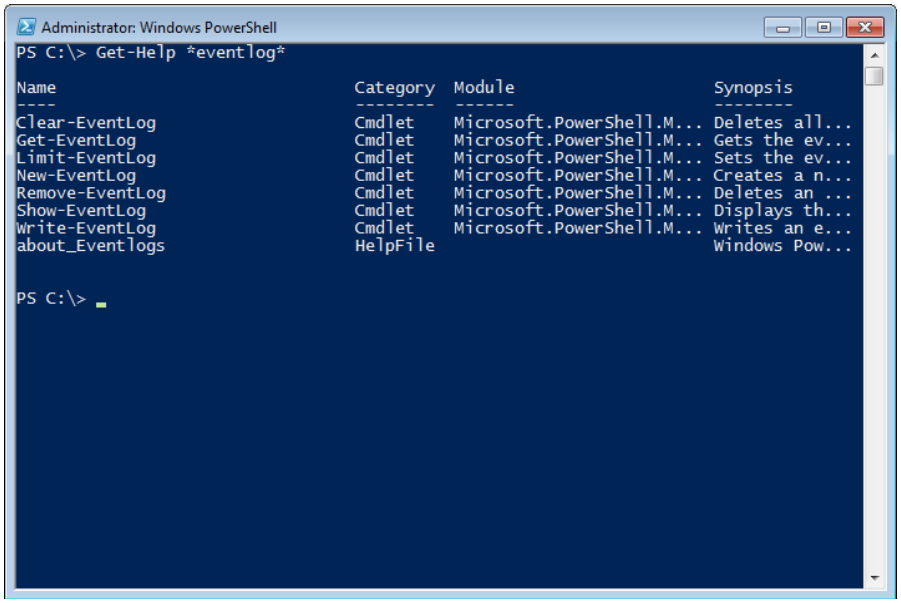


Figure 17. "Get-Help *eventlog*" outputs cmdlets that could help us work with eventlog.

Help System helped me discover that I can do many things for eventlogs like "Get-EventLog" or "Remove-EventLog", as shown in figure 17.

This is the point here. We are looking for some cmdlets to help us deal with eventlog, we type it and we get it. Now, we can just run *Get-EventLog* but it is better to take a look at the help file. So, we type "help Get-EventLog -Detailed", to take a look and the syntax and the definitions of the parameters, as shown in figure 18.

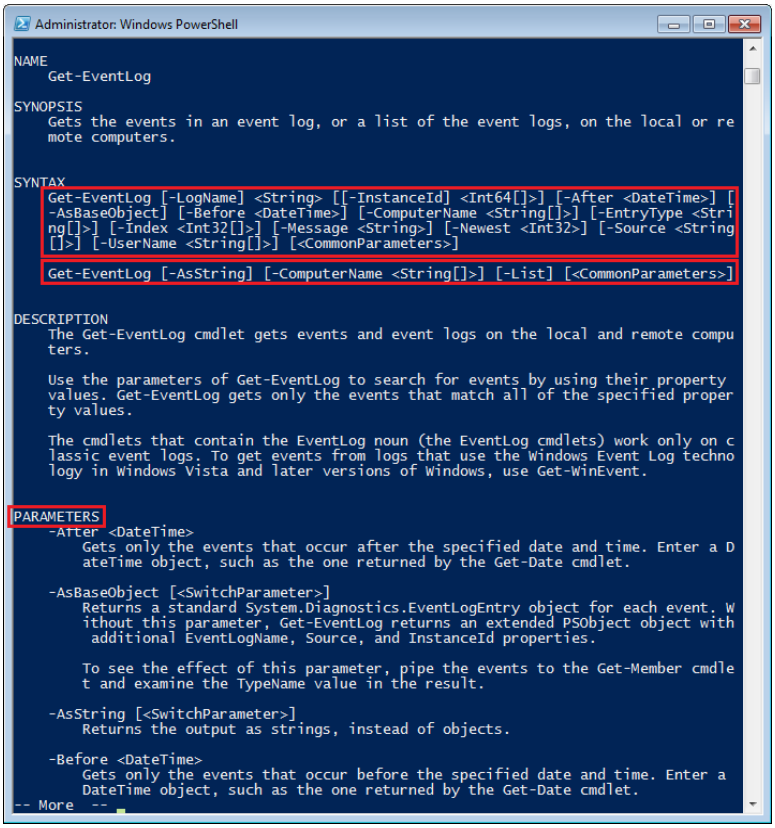


Figure 18. "help Get-EventLog -Detailed" to look at syntax and parameters.

So, we are going on the Syntax and we can see all of the parameters and if we do not understand something we can hit enter or spacebar to take a look down at the definitions of the parameters. You do not have to remember what all of the parameters mean, but many times we go to the list and we are reading the list of parameters first just to find out what capabilities we have. This seems like it may take a few extra minutes but it is worth it because we found that *newest* means we can get the newest entries in the log. Having a closer look at the eventlog help file, we realize that the -*LogName* parameter has to be filled to proceed to the cmdlet run. Other way to realize this it to type to the console "Get-EventLog" and hit enter; it asks for a LogName. We type System to get the entire system log entries.

But there is a more efficient way to get this, by using the help file, reading the syntax and parameters, looking at the examples and then we can start to make the cmdlet. Figure 19 provides a simple example of using "Get-EventLog" cmdlet to get the newest 5 system error events from eventlog.

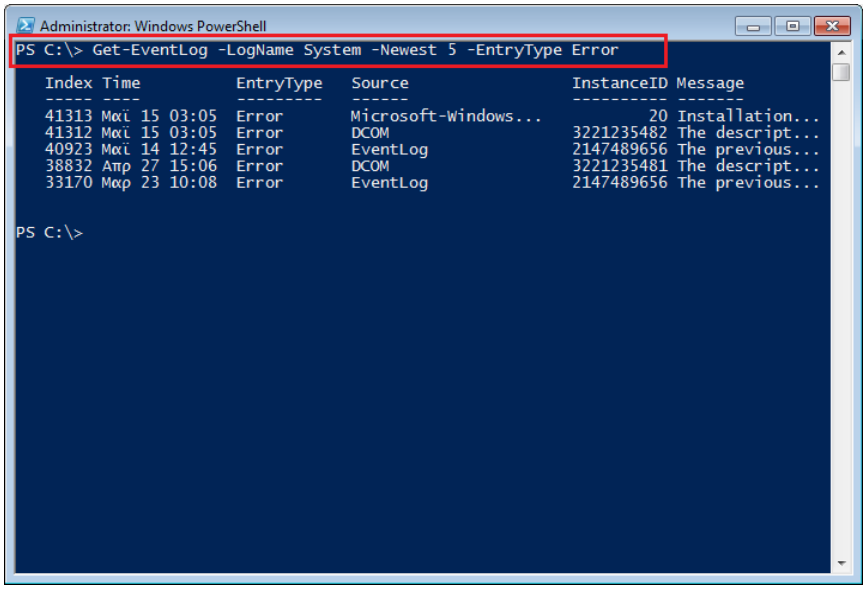


Figure 19. Output from typing "Get-EventLog -LogName System -Newest 5 -EntryType Error".

3.3.1.6 Conceptual Help

Concluding with Help System, we cannot pass by the **about_help**. Well, in time we typed "Get-Help *eventlog*", an *about_eventlogs* help file was displayed. These are the **conceptual topic** of PowerShell Help System. We can list all of about help files by typing "Get-Help about*", as shown in figure 20.

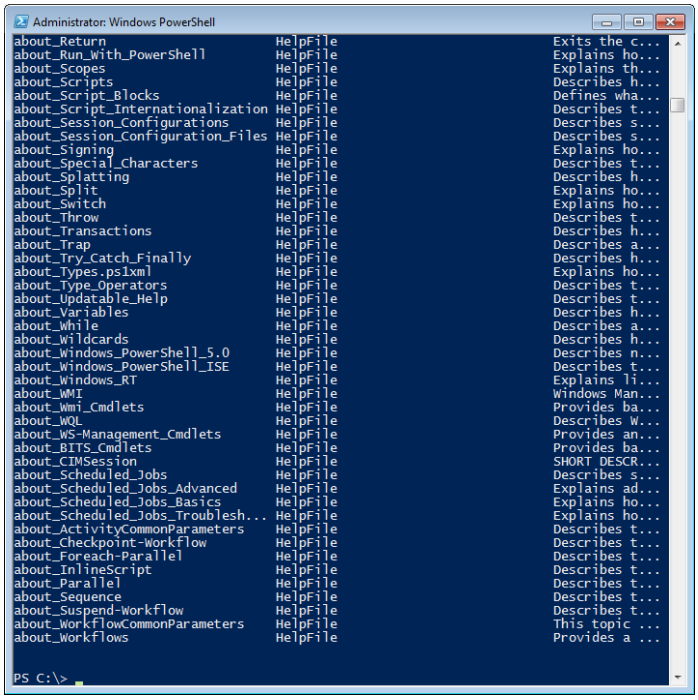


Figure 20. Getting all conceptual topics by typing "Get-Help about*"

This is actually a very good place to spend our time. PowerShell contains more than 100 conceptual help topics where you can read all of what you need to know. You can read about everything and these *"about"* help files are going to replace your bing or google searches. All you need to know is here, inside the PowerShell console.

3.3.2 Basic cmdlets & aliases

Once we have learned about using Help System, it is a good idea to search if there are cmdlets that contain the word "command". So we type "Get-Help *command*" and we find one of the most important cmdlets in PowerShell is one called "Get-Command", as shown in figure 21.

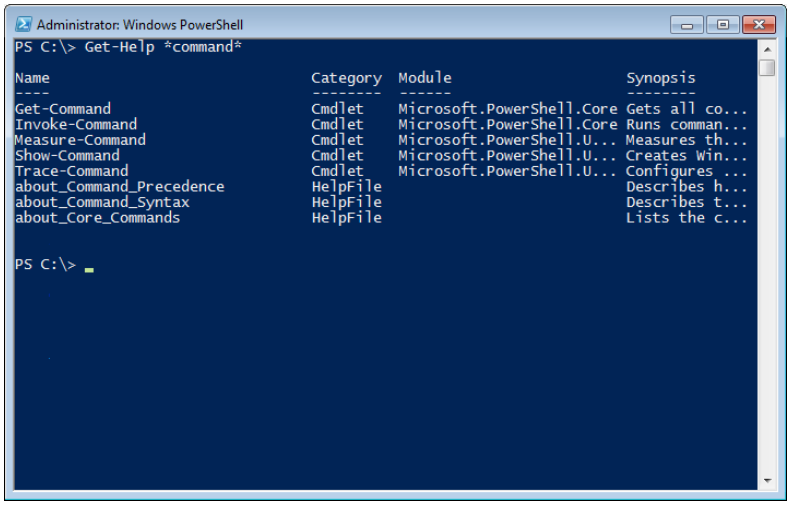


Figure 21. Output from typing "Get-Help *command*".

"Get-Command" is the cmdlet that will help you make the first steps to determine what you are able to do in PowerShell. We type "Get-Help Get-Command" to read the help text for *Get-Command* cmdlet, as shown in figure 22.

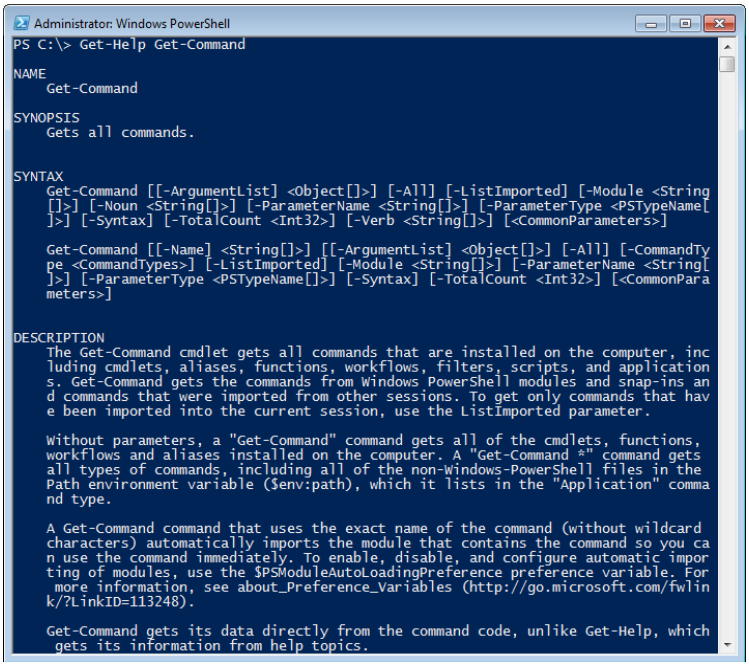


Figure 22. Getting the help for "Get-Command" cmdlet.

After reading the help text for "Get-Command" cmdlet, we can run it. Figure 23 provides the output from running "Get-Command". In fact "Get-Command" returns a

list of all commands installed on the system. Up to 400 cmdlets on PowerShell 3.0 are available to use.

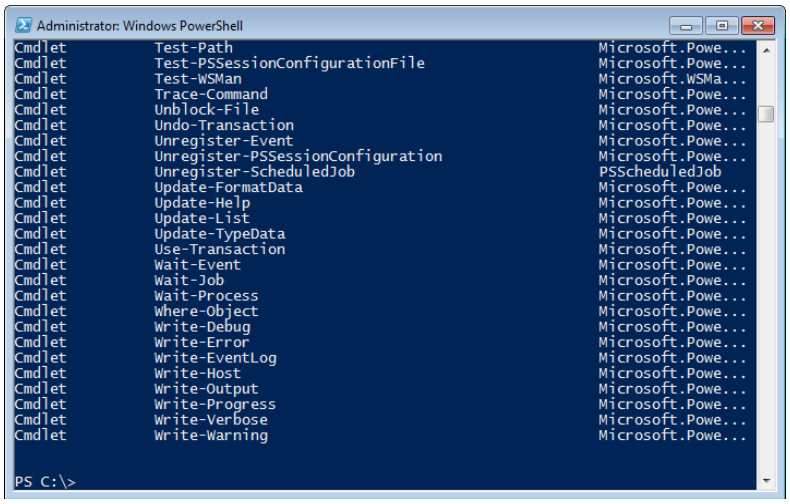


Figure 23. Output from running "Get-Command".

3.3.2.1 Verb-Noun Pattern

As we observe all of these cmdlets that *Get-Command* returned to us, it is easy to realize that all PowerShell cmdlets follow a **strict "Verb-Noun" pattern**. This means that, for memorization reasons, each cmdlet name consists of a standard verb hyphenated with a specific noun.

So, imagine getting the help of the verbs. You can type *Get-Verb*, and this will back all of the available verbs used on PowerShell and you are going to learn. Figure 24 provides the output from typing "Get-Verb". You can as well measure all these verbs by typing: "Get-Verb | measure". Those are the 98 things you can learn and then you are going to be able to think, type, and get what you want.

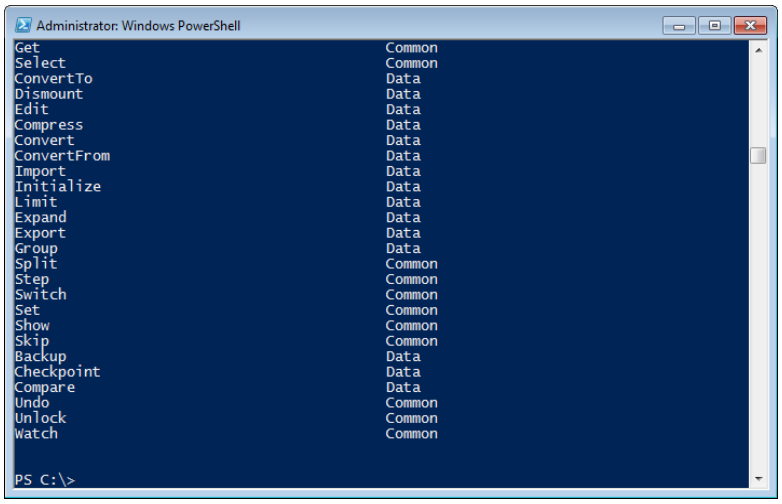


Figure 24. Output from running "Get-Verb".

3.3.2.2 Aliases: nicknames for commands

Although PowerShell cmdlets are easy to learn and to use, some cmdlets like "Set-WinDefaultInputMethodOverride" are a lot to type, even with TAB completion. So, in order to reduce waste of time writing long cmdlets again and again, we can use the aliases. [17]

An alias is nothing more than a nickname for a cmdlet. For example you can type "gvs" for *Get-Service* or "gps" for *Get-Process*. You can get a list of all built-in aliases by typing "Get-Alias", as shown in figure 25. We can realize that commands like cp and dir are aliases of PowerShell full name cmdlets. You can achieve the same results with Copy-Item and cp.

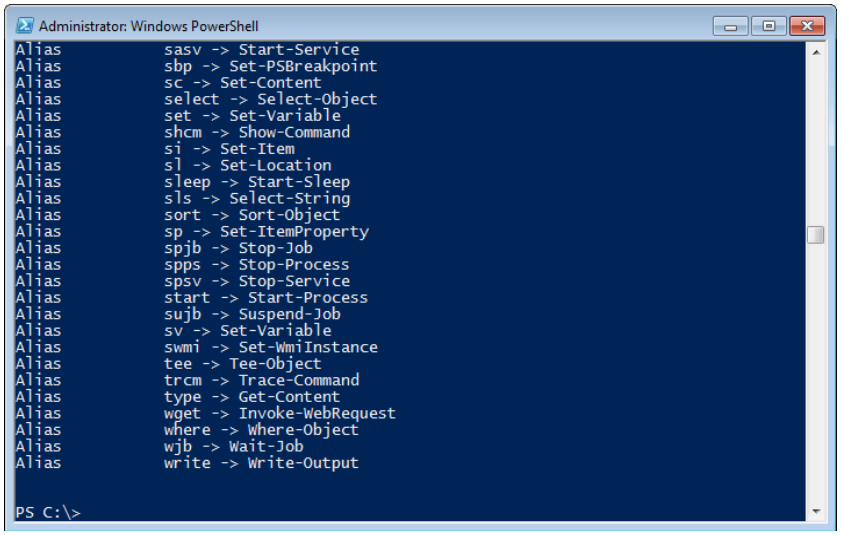


Figure 25. Output from running "Get-Alias".

3.3.3 PowerShell Providers

PowerShell provider, or PSProvider, is an adapter. [18] It is designed to take some kind of data storage and make it look like a disk drive. You can see a list of installed providers right within the shell by typing "Get-PSProvider", as shown in figure 26.

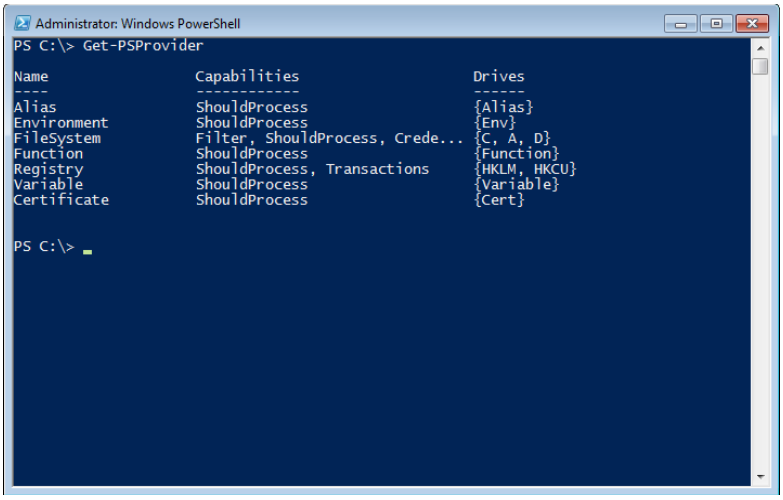


Figure 26. Output from running "Get-PSProvider".

In order we can view and manage data within PowerShell, we work with Providers. Providers are Microsoft .NET Framework-based programs that make the data in a specialized data store available in Windows PowerShell.

NOTE
You can get full information about providers by looking at the help file. Type: "Get-Help about_providers".

3.3.4 PowerShell Modules

When PowerShell 1.0 came out there were 192 cmdlets that we could work with. Then, PowerShell 2.0 came out and there were 236 things we could do. But we need to do more than that, we need to have something more extensible.

PowerShell team is responsible for the core PowerShell environment, the language, the syntax etc. PowerShell 3.0 and later versions came out with something new: plug-ins, originally called snap-ins, we now call them **modules**.

PowerShell module is a package that contains Windows PowerShell commands, such as cmdlets, providers, functions, workflows, variables and aliases. [19]

For those who are interesting of making the shell more flexible by adding more cmdlets, providers and functions, new modules can be written down and be imported in PowerShell.

TIP
PowerShell team created PS environment to be very easily extendable by giving the individual teams the ability to write their stuff and then ship and share their stuff.

NOTE
You can get full information about modules by looking at the helpfile. Type: "Get-Help about_modules" & "Get-Help about_PSSnapins".

Every PowerShell cmdlet belongs to a module. You can see a list of installed modules right within the shell by typing "Get-Module". This cmdlet will give back only modules that has been used in the current PS session. *Get-Module* cmdlet has a parameter called ListAvailable. Running "Get-Module -ListAvailable" we receive a list of all modules has been installed on the system.

As you can see in figure 27, there are currently 16 modules installed in the system. For each module there are commands that can be used.

We can list all commands of a module. For example, we want to list all of the cmdlets that Microsoft.PowerShell.Security module contains. Figure 28 provides a way to use "Get-Module" cmdlet in order to do this. We just use the most in use parameter called "Name" of *Get-Module* cmdlet and we call property *Exported Commands* for our case to list all of the cmdlets available from Microsoft.PowerShell.Security Module.

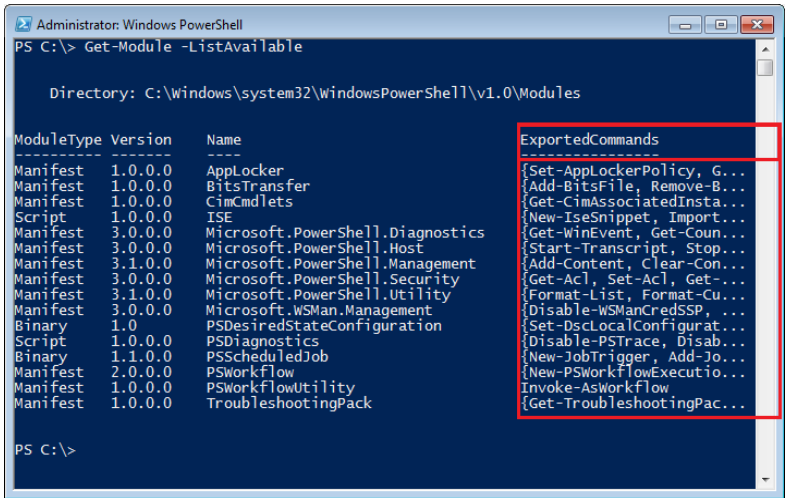


Figure 27. Output from running "Get-Module -ListAvailable".

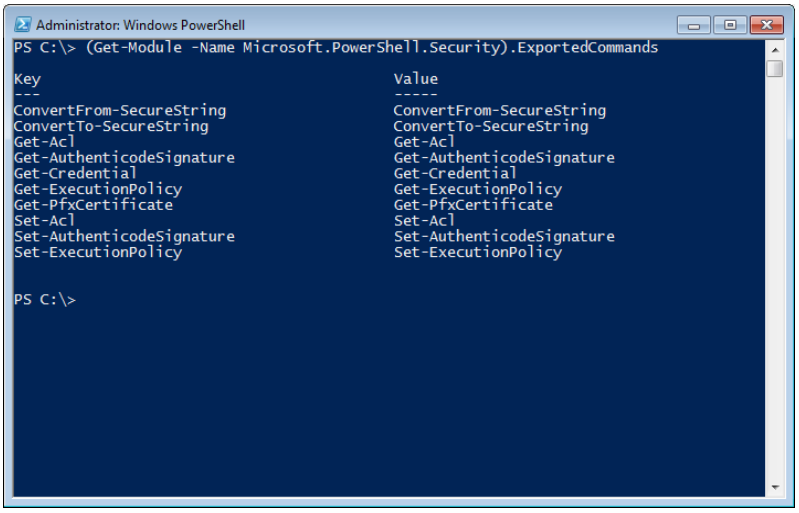


Figure 28. Provides a way to find the commands that the Security Module contains.

Concluding with modules, it is important to refer again that we use modules every single day by running each command. All the cmdlets belongs to a module and this is a way to group the cmdlets. Except from using the default built-in modules, PowerShell extends its power and let users to write their own cmdlets, put them all in a module and share modules with others.

3.3.5 The Pipeline: Connecting Commands

The pipeline is the character "|". It is located above the Enter key and it connects cmdlets to produce better results.

- It can be used as a sequence of pipes like:
  Get-Service | Select-Object Name, Status | Sort-Object Name
- It can be broken into several lines to increase readability, like:
  Get-Service |
  >> Select-Object Name, Status |
  >> Sort-Object Name

It is good to use the pipeline, to get specific information we want, and this will make us more effective with the results we want to get. We are going to start using the pipeline with a simple example.

Figure 29 provides a simple example of using the pipeline. In this time, PowerShell will go out and get a collection of service objects, it is going to grab all those service controller objects and then it is going to send them, one at a time across the pipeline and the pipeline will resolve the next function for each object.

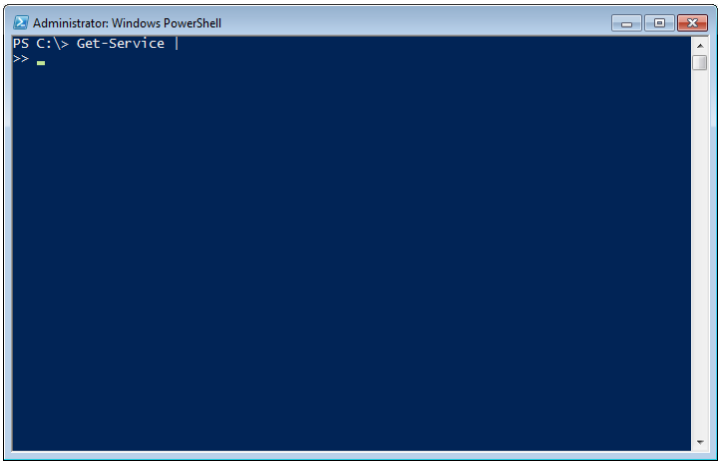


Figure 29. Simple example shows that "Get-Service" waits for piping.

Thus, to make this more clear. There is a cmdlet called Stop-Service and we can use it to stop a service. Figure 30 provides an example piping to "Stop-Service" cmdlet to stop a service. [20]

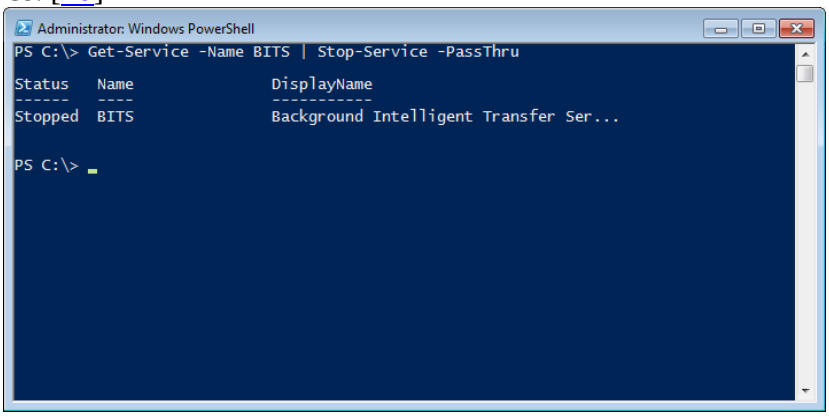


Figure 30. Piping a service to "Stop-Service" to stop the service.

This is going to get the *BITS Service*, it will send it across the pipeline. "Stop-Service" is going to grab and hold that service and do exactly what is going to do, it will stop the service BITS. With -PassThru parameter here, PowerShell is forced to output the newly modified object instead of hiding them.

Once we have tried to pipe a service to "Stop-Service" to stop the service, now we can use "Start-Service" to start the service, as shown in figure 31.

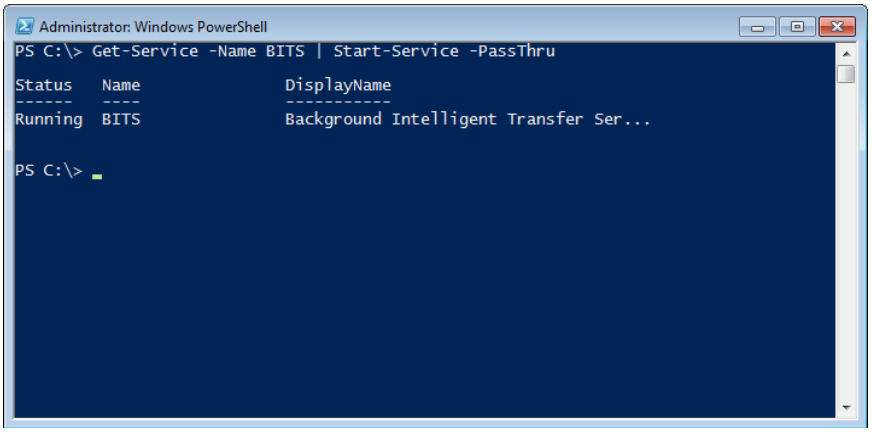


Figure 31. Piping a service to "Start-Service" to start the service.

That is a simple use of pipeline. But that concept, of taking something, sending it across the pipeline to have something else act on it, is very important. And think about it: you can continue with piping from one thing to the next and to the next and so on.

NOTE
BE CAREFUL WITH THE PIPELINE*: You have to have in mind, at this time that the pipeline is very* ***powerful****, sufficiently to allow you to make silly mistakes and generate problems.*

NOTE
BE CAREFUL WITH POWERSHELL*: PowerShell is a powerful tool that you want to be thoughtful about this. It allows you to do some damaging things, such us removing all the files from a file system, or stopping all the services, or stopping crucial processes like lsass and so forth. So, you have to be very careful, because very bad things can happen.*

3.3.6 Working with objects

As we have already mentioned, PowerShell implements an object-oriented environment. For example running the "Get-Service" cmdlet we get back objects of type Service. This means that most of the cmdlets in PowerShell return to as objects of some type.

Members of objects consist of properties and methods. For the representation of what an object contains, PowerShell give us a table with several columns that fit comfortably on the screen. If objects are like a giant table in memory, PowerShell only shows you a portion of that table on the screen. To learn more about an object you can use the pipeline, and pipe the object to "Get-Member". [21]

How can we get all the members of an object in PowerShell?
We pipe an object to "Get-Member" cmdlet.

3.3.6.1 Get-Member cmdlet

Get-Member is one of the most frequently used cmdlets and let us discover what an object contains. We most use the "gm" alias for "Get-Member".

NOTE
You can get full information about objects by looking at the help file. Type: "Get-Help about_objects". You can also "Get-Help Get-Member" to read the help text for Get-Member cmdlet.

For this time, we are going to work with Process objects. Figure 30 shows the way we use "Get-Member" cmdlet. Once we have opened a notepad process, then we type "Get-Process -Name notepad" to get back the process in the console. After this, we are going to pipe this to Get-Member.

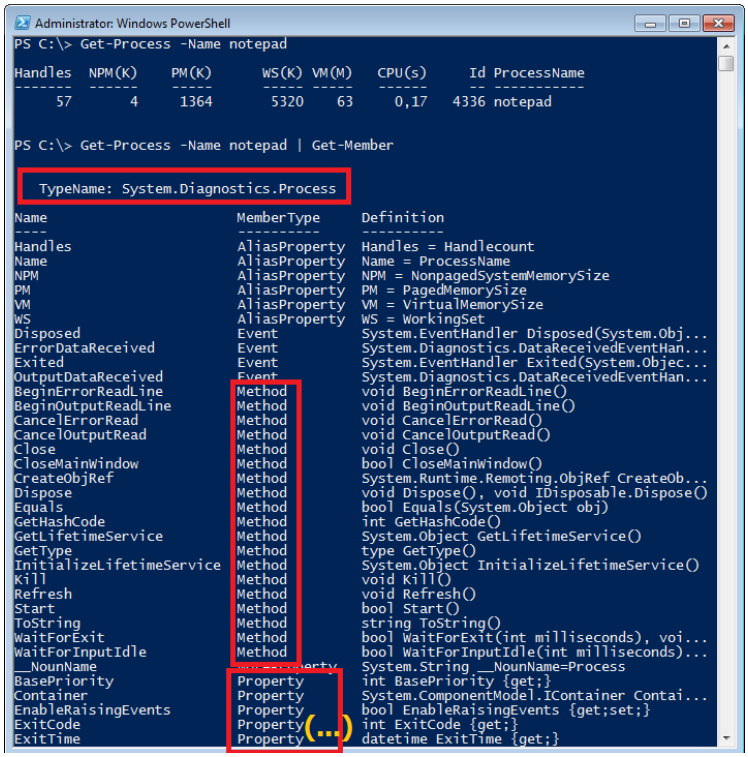


Figure 32. Getting members of a Process object.

As we can see in figure 32, after piping to Get-Member we first take TypeName of the object, which in this case is a Process object, and then there are up to 90 members within this object consist of methods and properties.

TIP
You can pipe to Measure-Object to retrieve the exact number of member of an object. In our case of Process object we can do the following:
PS C:\> Get-Process -Name notepad | Get-Member | Measure-Object

3.3.6.2 Using Properties and Methods

There are multiple ways to get access to an object within PowerShell to get information. Figure 33 shows a simple way to do this using the "dot method". Another way to get properties from an object is to use *Select-Object* or *Format-List* cmdlets, as shown in figures 34 and 35.

NOTE
You can get full information about Select-Object, Format-List and Format-Table cmdlets by looking at the help file. Type: "Get-Help Select-Object" or "Get-Help Format-List" or "Get-Help Format-Table" and read about syntax to use these cmdlets with the most efficient way.

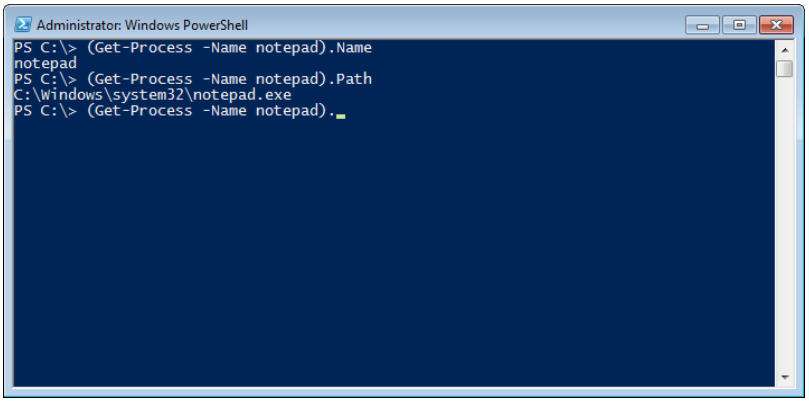


Figure 33. Getting a property from an object using "dot method".

From "Get-Member" of a Process object we realize that there is a Parameter called Name (with definition type of string) and another called Path. In order to get the values of these properties we can "call" each property by using the "dot method", as shown in figure 33.

Alternatively, we can use Select-Object, as shown in figure 34.

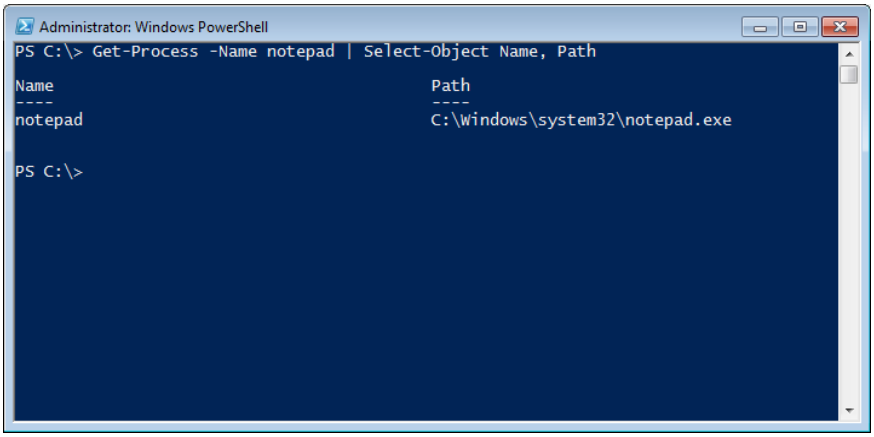


Figure 34. Getting a property from an object using "Select-Object" cmdlet.

"Select-Object" is going to select specified properties of an object or sets of objects. This means that "Select-Object" cmdlet can be also used to select unique objects from a collection of objects.

Another way to get the values of properties of an object is to use "Format-List" cmdlet. As shown in figure 35, we used "Format-List *" to display as a list, all properties that a process object currently contains.

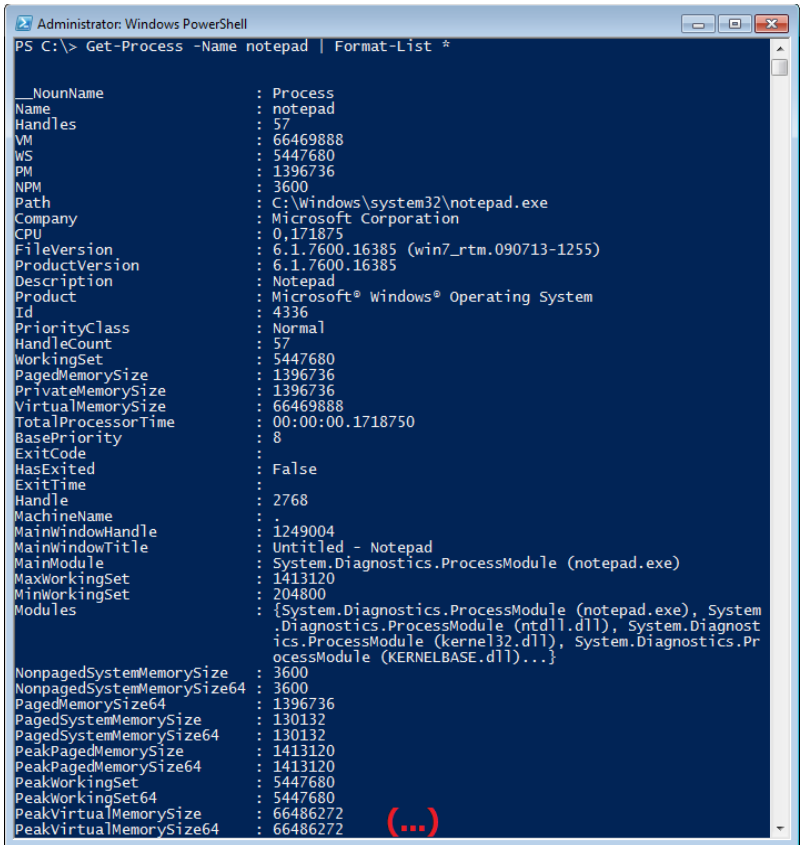


Figure 35. Using "Format-List *" to display all properties of an object as a list.

NOTE

*For the Format-List cmdlet we used wild card * as parameter to display all of the properties of the input from pipeline object. We could specify which properties would like to display, for example we could type: Get-Process -Name notepad | Format-List Name, Path.*

We have already seen that there are also methods within this Process object we could call. Once we piped a process to *Get-Member* we saw that there is a method of a Process object called *kill()*. As we could pipe our specific process to Stop-Process, we can achieve same results by calling kill() method of the Process object. Figure 36 provides a simple example of using *kill()* method ("calling with dot method") to kill a previously opened *notepad* process.

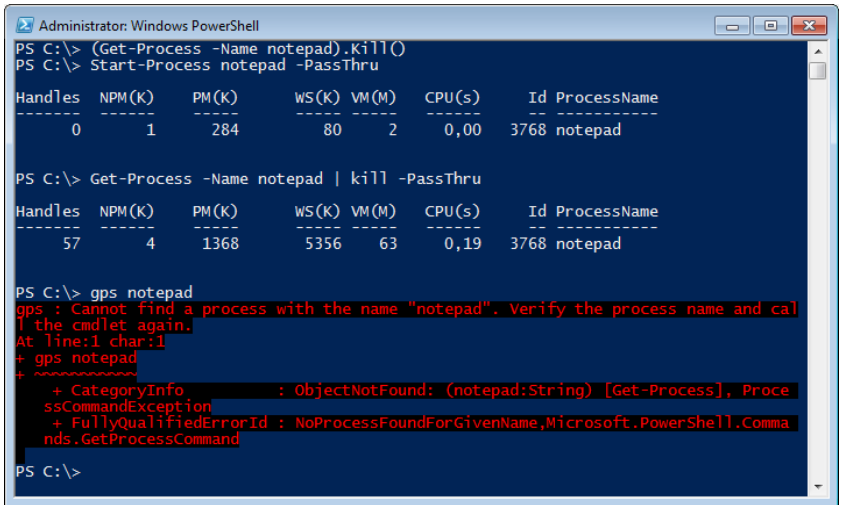


Figure 36. Using kill() method to kill a process.

All we can see in figure 36 is that we called the Process object method *kill()* to stop the specific process. After this we started again a new notepad process and then we kill the new process by piping to kill (Stop-Process alias) which stops the process again. After that, we used *gps* alias to "Get-Process notepad" and a warning message is been returned with information that there is no process with name notepad currently running to return process object.

We achieved getting some properties from a Process object with multiple ways. It is very important here to underline some basic differences. We have already mentioned that we can pipe an object to *gm* in order to get information about the object.
Figure 37 shows the result of piping the notepad process to gm. We are going to emphasize to some properties.

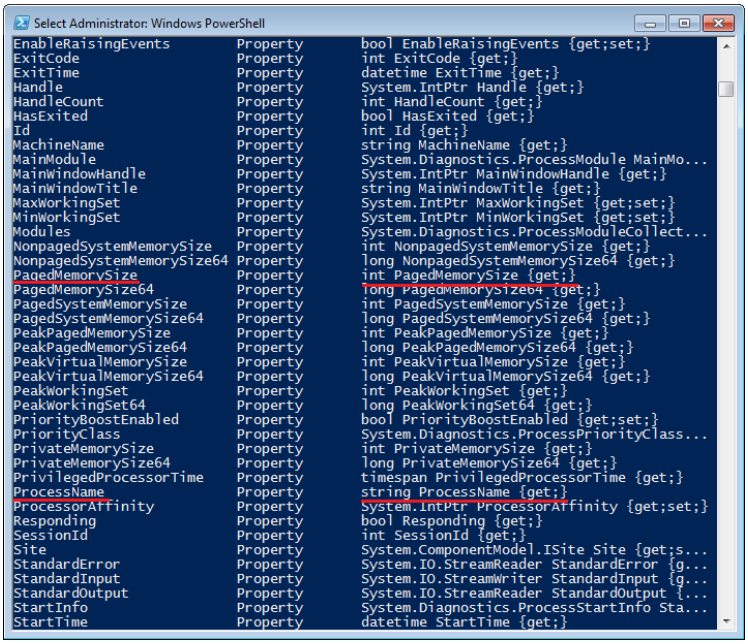


Figure 37. Closer look from piping a process object to get-member.

As we can see in figure 37, there is a property *ProcessName* which is the full definition of the property *Name* (Name is actually the AliasProperty for ProcessName) and it is been implemented to return a value of type string. As we can see, there is another property called *PagedMemorySize* which is been implemented to return a value of type int (integer). Figure 38 show that we use the "dot method" to retrieve the values of these properties.

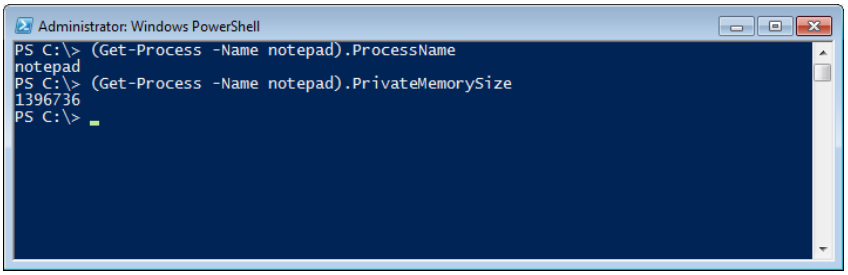


Figure 38. Getting properties from a process object

It is easy to understand that we received two property values, *a string for the ProcessName* property and then *an integer for the PrivateMemorySize* property.
What we are going to get if we pipe each of these properties to Get-Member again?

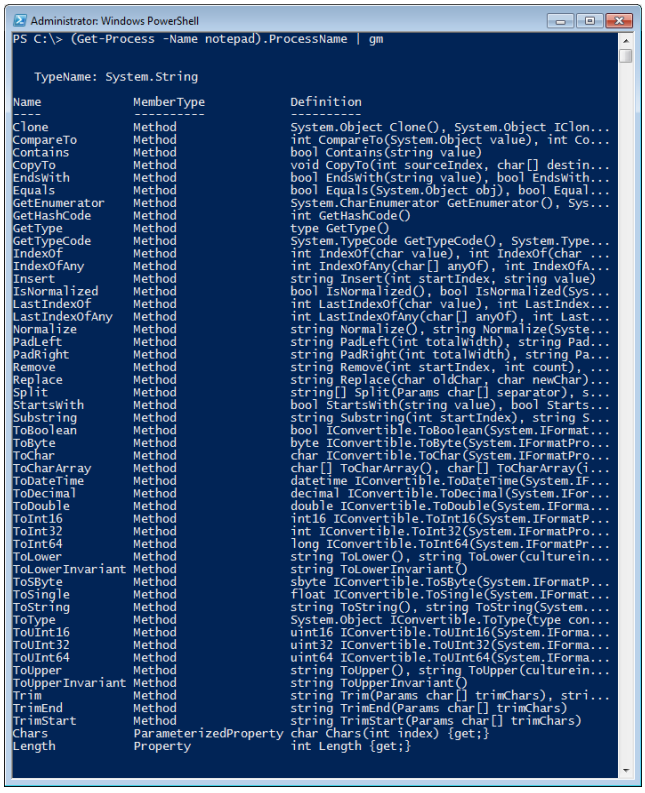


Figure 39. Piping a String Property to Get-Member

Figure 39 shows the output of piping to gm an object property which contains String. What we did here is we took a property value of a Process object (string value) and we piped it to Get-Member. By doing this, PowerShell returned that there is a String object came from the pipeline (TypeName: System.String) and then return all of the methods and properties that this String object contains.

Figure 40 shows the same procedure with the *PrivateMemorySize* property. PowerShell returned that there is an Int32 object came from the pipeline (TypeName: System.Int32) and then return all of the methods and properties that this int object contains.

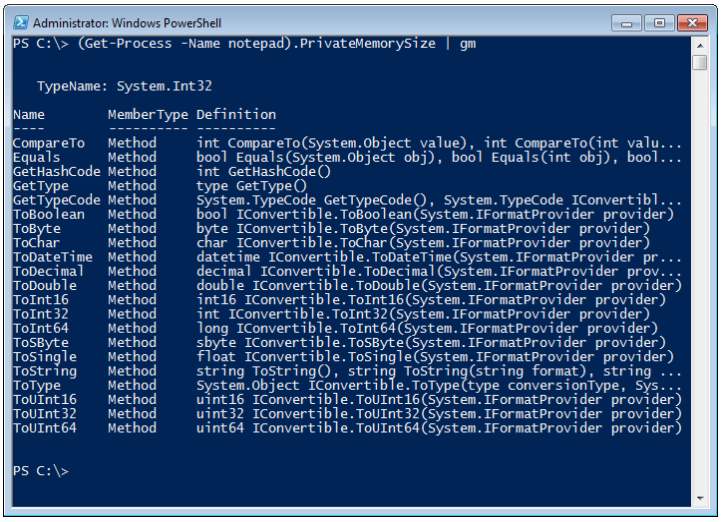


Figure 40. Piping an Integer Property to Get-Member

PowerShell help system informs us that PowerShell uses structured collections of information called objects to represent items. Most objects have properties and properties are the data that is associated with an object. Different types of object have different properties.

Each property of an object is represented as an object, too. This means that there are numerous standard types of Classes in .NET Framework that implement those types of information as objects. In our case System.String is a class that implements String objects and System.Int32 implements integer objects.

NOTE

You can get full information about properties of objects by looking at the help file. Type: "Get-Help about_properties".

After the "dot method", we used *Select-Object* to output specific property values. What we did, is to pipe the notepad process to Select-Object. Figure 41 shows the output from piping to Get-Member after Select-Object to discover what the output is after select-object cmdlet.

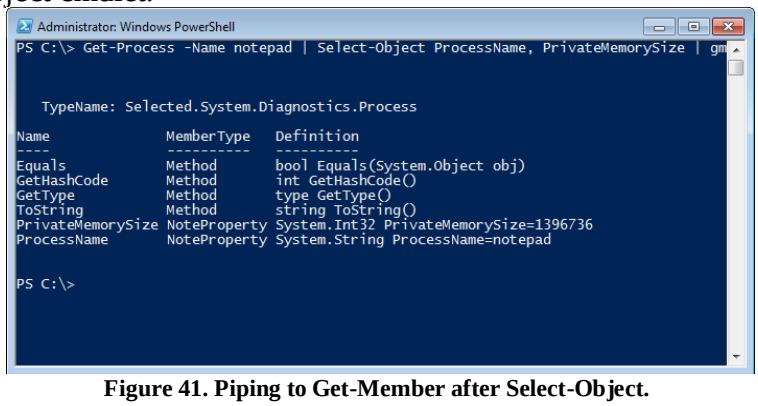


Figure 41. Piping to Get-Member after Select-Object.

As we can see in figure 41, the "Select-Object" cmdlet helps us display values from an object, it does not really return the actual properties values. In contrast, it outputs a different type of object of class Selected.System.Diagnostics.Process that contains two NoteProperties that came from the inputs of Select-Object cmdlet (ProcessName, PrivateMemorySize). This means that it filters only the specified properties and it returns a custom PS object to represent the item.

In conclusion, Windows is an object-oriented operating system. In fact, PowerShell implements .NET Framework classes and represents data as objects. In order to discover the type of an object and what exactly contains, we use the "Get-Member" cmdlet.

Working with objects within PowerShell gives us more power and flexibility. PowerShell can produce objects as the output of its commands and here's something we are going to analyze in the next section.

3.3.7 Piping Objects – Running cmdlets as an admin

In this section all we are going to see is some piping techniques. PowerShell piping may seem similar to how Unix and Linux shells work but in this chapter we are going to show that PowerShell's pipeline implementation is much richer and more modern.

With piping PowerShell gives us the ability pass objects over the pipeline and these are things we are going to see now.

3.3.7.1 Where-Object cmdlet

We have already referred to objects and how to discover things behind them. We have also understood that PowerShell's use of objects help us get rid of all text-manipulation overhead. This means that, although we do not have to waste time in parsing text to achieve results, we have to spend more time on focusing on cmdlet's syntax to be able to use PowerShell capabilities in the most efficient way.

When you are not able to get a cmdlet to do all of the filtering you need, you can turn to a core PowerShell cmdlet called "Where-Object" (we can use alias "where", too). This uses a generic syntax, and you can use it to filter any kind of object once you have retrieved it and put it into the pipeline. [22]

NOTE
You can get full information about using Where-Object cmdlet by looking at the help file. Type: "Get-Help Where-Object".

Consider that we want to get all processes currently running on our system. We use "Get-Process" for this. As we have seen a little earlier, there are up to 90 things that a process object contains. As we can see, Get-Process displays certain properties as columns, and we can see a property Handles. By piping Get-Process to gm we realize that there is always an integer value in this property.

Consider, for this time that we want to get all processes but only display these where the Handles are greater than 1400. "Where-Object" will help us deal with this task, as shown in figure 42.

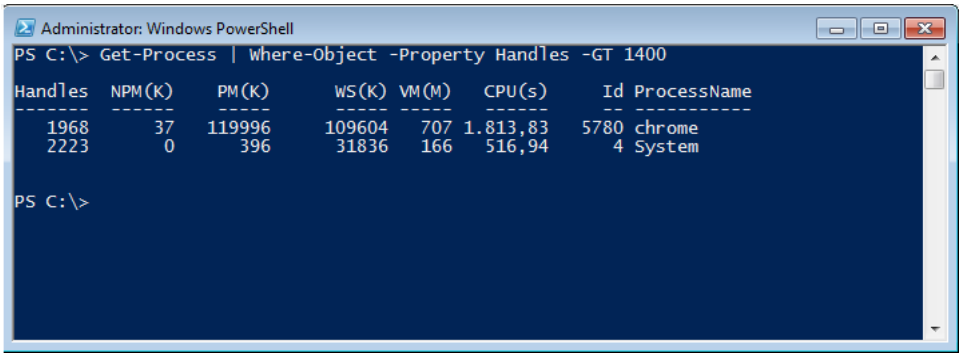


Figure 42. Using "Where-Object" cmdlet (1).

What we did, is we get all of the processes, and then we piped to "Where-Object" to display only these where the property handles is greater than 1400. We used -GT operator for this.

NOTE
You can get full information about operators by looking at the help file. Type: "Get-Help about_operators".

There are, actually, two different ways to construct a "Where-Object" command. So, instead of using the above construction, we can use "Where-Object" as shown in figure 43.

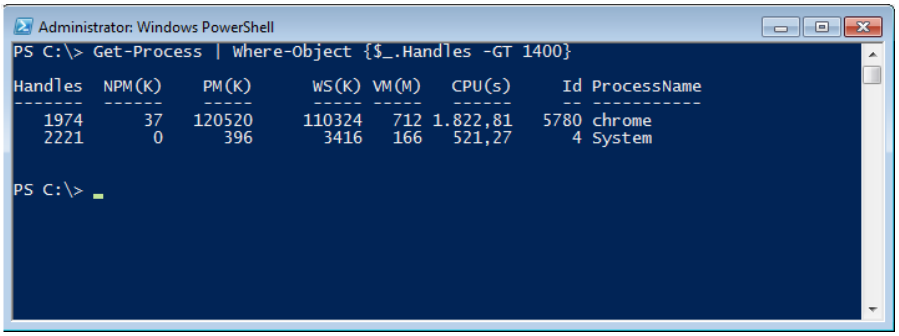


Figure 43. Using "Where-Object" cmdlet (2).

Another use of "Where-Object" cmdlet should be working with services to filter up only these where the status property is equals to Running. Figures 44 and 45 show two different ways to get only these service objects.

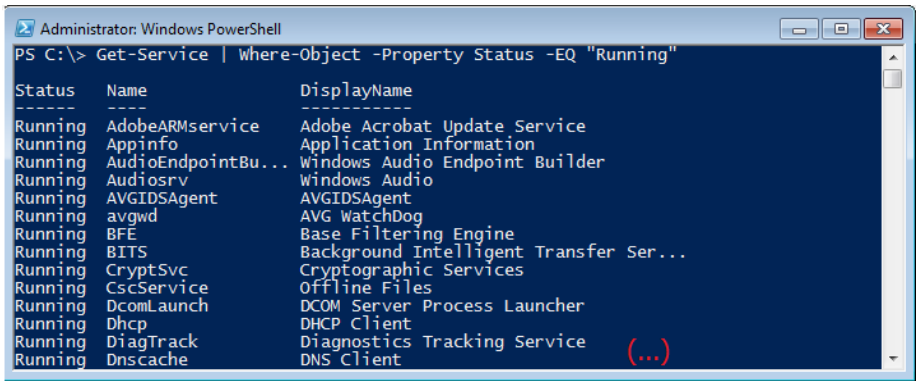


Figure 44. Using "Where-Object" cmdlet on Service Controller objects (-EQ).

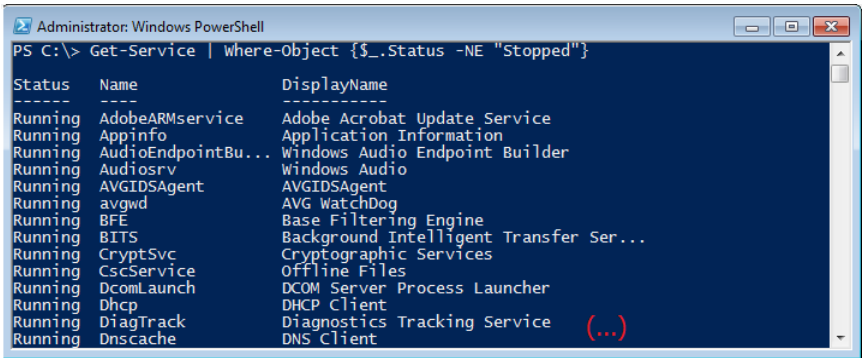


Figure 45. Using "Where-Object" cmdlet on Service Controller objects (-NE).

Another great use of "Where-Object" cmdlet could help us work with Management Objects (System.Management.ManagementObject).

Windows Management Instrumentation (WMI) classes help us find important information for our system, applications, networks, devices and other manageable components of the modern enterprise. [23]

There is a cmdlet in PowerShell called "Get-WmiObject" that help us discover our system's WMI (we often use alias "gwmi"). If we run "Get-WmiObject -List" we realize that we get back up to 1000 objects and it is hard to find and select which of these we are interesting for. Thus, we can use "Where-Object" to do our work.

Imagine we want to get only Management objects that will help us deal with something like *accounts*. There is a way to do this using "Where-Object" and operators as shown in figure 46.

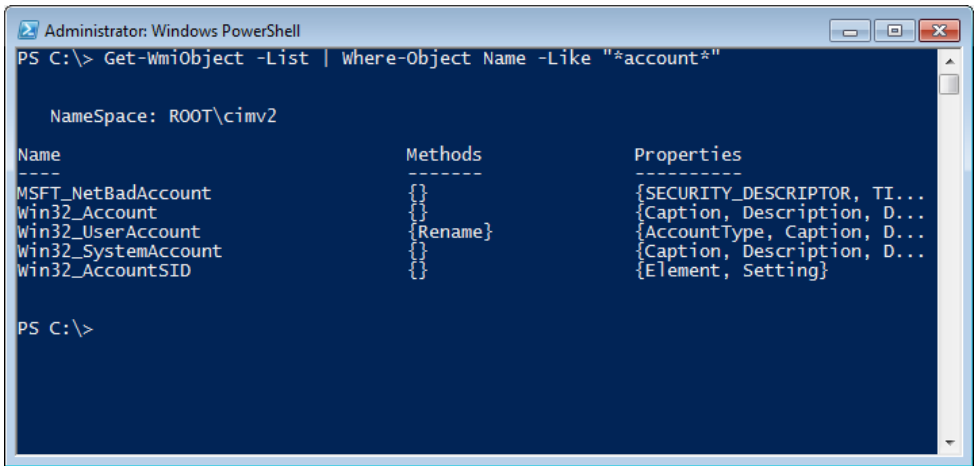


Figure 46. Using "Where-Object" to filter data from WMI objects.

NOTE

You can get full information about Windows Management Instrumentation (WMI) by looking at the help file. Type: "Get-Help about_wmi".

NOTE

You can get full information about getting wmi objects by looking at the help file. Type: "Get-Help Get-WmiObject" and read about syntax to use these cmdlets with the most efficient way.

NOTE

You can get full information about classes by looking at the help file. Type: "Get-Help about_classes".

3.3.7.2 Exporting/Importing CSV

"Export-Csv" is a great cmdlet that is going to let us have information of objects on our screen. Figure 47 shows a way to use this command. We type: "Get-Service | Export-Csv -Path C:\services.csv". "Export-Csv" is going to take all the information that services contain and export it to a Csv file at the selected path and filename. If you try to open this Csv file with notepad, you will probably not be able to read the information inside this. But in PowerShell, there is an easy way to *Import-Csv*. We type "Import-Csv -Path C:\services.csv" and all the data come back and it is formatted correctly, so we can actually work with it now.

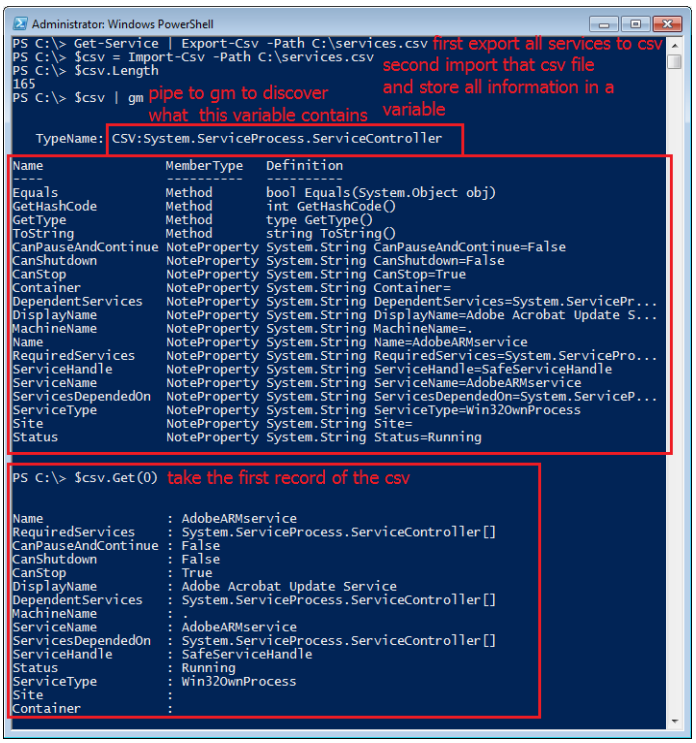


Figure 47. Using Export and Import Csv cmdlets.

NOTE

You can get full information about exporting or importing CSV cmdlets by looking at the help file. Type: "Get-Help Export-Csv" or "Get-Help Import-Csv" and read about syntax to use these cmdlets with the most efficient way.

3.3.7.3 Exporting/Importing XML

"Export-Clixml" is another great cmdlet that creates a generic command-line interface (CLI) *Extensible Markup Language* (XML) file. Instead of exporting the services to Csv, we want to export to xml all the processes of the computer we are on. We type: "Get-Process | Export-Clixml -Path C:\good.xml", as shown in figure 46.

NOTE

You can get full information about exporting or importing XML cmdlets by looking at the help file. Type: "Get-Help Export-Clixml" or "Get-Help Import-Clixml" and read about syntax to use these cmdlets with the most efficient way.

We just want to pretend that we took a snapshot of a perfectly good running machine processes. Pretend for a moment that we are working, now, on a non perfectly running machine and we open notepad and calc. Imagine here that instead of notepad and calc maybe there could be some malware processes. So we type "Start-Process calc & Start-Process notepad", as shown in figure 48.

These are simple examples of piping objects, but we can do some greater export stuff.

3.3.7.4 Compare-Object cmdlet

Consider we want to compare current running processes of two machines. Imagine there is a perfectly good running machine and all running processes are under control. So, we want to compare these processes with another machine current running processes to get brilliant information back. To do this we use the *Compare-Object* cmdlet, as shown in figure 48.

We type: *"Compare-Object -ReferenceObject (Import-Clixml -Path C:\good.xml) -DifferenceObject(Get-Process) -Property Name"*

NOTE

You can get full information about comparing objects cmdlets by looking at the help file. Type: "Get-Help Compare-Object" and read about syntax to use these cmdlet with the most efficient way.

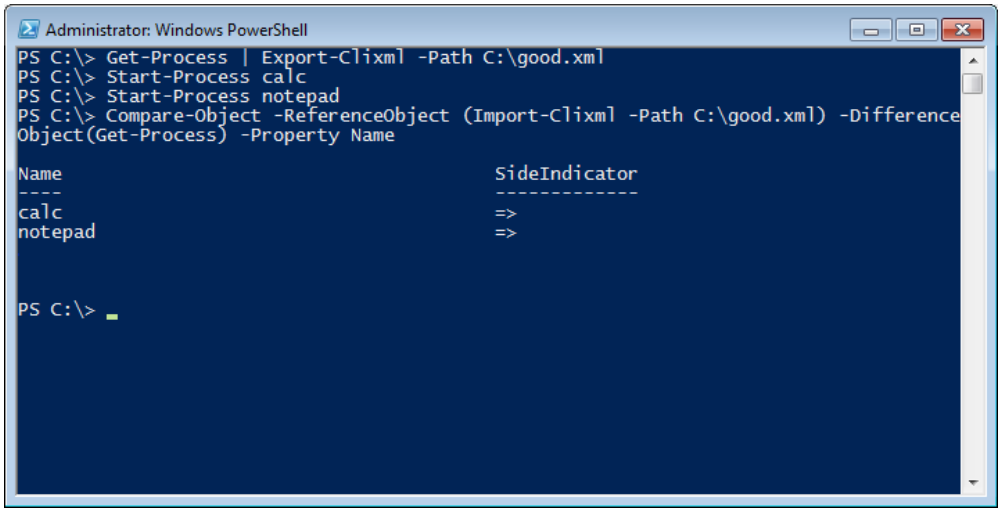


Figure 48. Using "Compare-Object" to compare two xml.

So, we have exported our process information with the pipeline and after this we used the "Compare-Object" cmdlet to compare the two machine running processes by property name. The side indicator look to the bad machine and it says that the calc and notepad are running on that and not to the good machine. [24]

Imagine comparing two machines for the software it is installed and making one machine look like other.

There are a lot of things we can do with building a pipeline. What we did above is because of PowerShell Live Objects and Object Adapters. We took all these objects and we compared against an XML file. Of course if you just try to compare these things for all their properties, you would realize that they are completely different. For this reason, we just compared them based upon their names.

Again, we think about what we want to do, we type it and the magical PowerShell is able to take these incredible complex stuff and present it to us in a very simple world.

3.3.7.5 Out-File cmdlet

Besides exporting out to Csv or xml, there are some other things we can do.
We type: "Get-Service | Out-File -Filepath C:\test.txt", as shown in figure 49. This will send all Service Objects to the specified text file. We continue by getting the content of this text file by using the "Get-Content" cmdlet.

NOTE
You can get full information about output to file cmdlets by looking at the help file. Type: "Get-Help Out-File" or "Get-Help Out-" and read about syntax to use these cmdlets with the most efficient way.*

TIP
*I'm wondering if there is something that would help me with getting the content from a text file. Help System is my friend, so I type: "Get-Help *content*" and I find that there is a cmdlet called "Get-Content"*

NOTE
You can get full information about getting the content from a file by looking at the help file. Type: "Get-Help Get-Content" and read about syntax to use these cmdlets with the most efficient way.

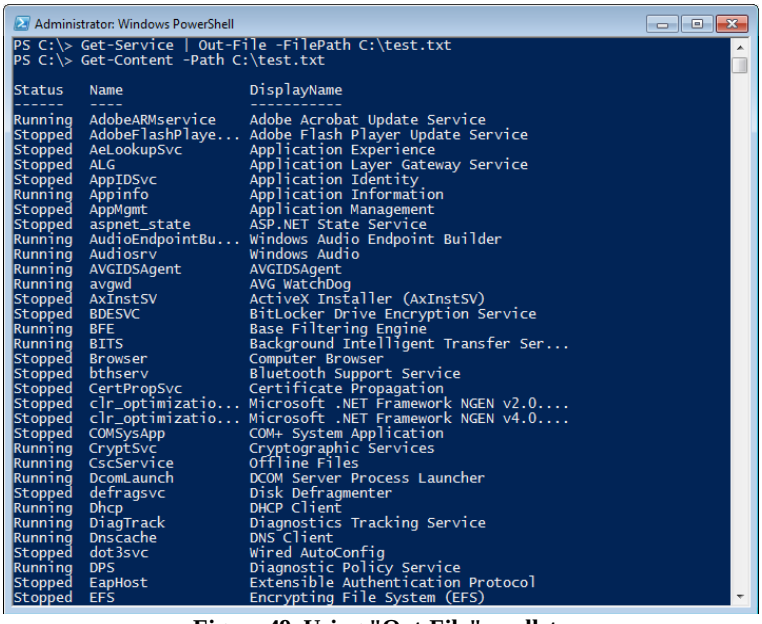


Figure 49. Using "Out-File" cmdlet.

3.3.7.6 ConvertTo cmdlets

As you can realize, there are also some *ConvertTo* cmdlets, we can use instead of export cmdlets.

In fact, **"export" is "convert" coupled with "output to a file"**.

When you do an export to Csv, you are done. When you do a "ConvertTo-Csv" you are leaving that stuff in the pipeline in case you are going to send this converted stuff to other things to do so.

NOTE
You can get full information about converting to cmdlets by looking at the help file. Type: "Get-Help ConvertTo" and read about syntax to use these cmdlets with the most efficient way.*

3.3.7.7 ConvertTo-Html cmdlet

"ConvertTo-Html" cmdlet produces well-formed, generic HTML that will display in any web browser. It is plain looking, but you can reference a CSS to specify more attractive formatting if desired.

NOTE
You can get full information about converting to HTML cmdlet by looking at the help file. Type: "Get-Help ConvertTo-Html" and read about syntax to use these cmdlets with the most efficient way. Note that you have a lot of options to make your htm file pretty.

Figure 50 provides a way to use combination of "ConvertTo-Html" and "Out-File" cmdlets.

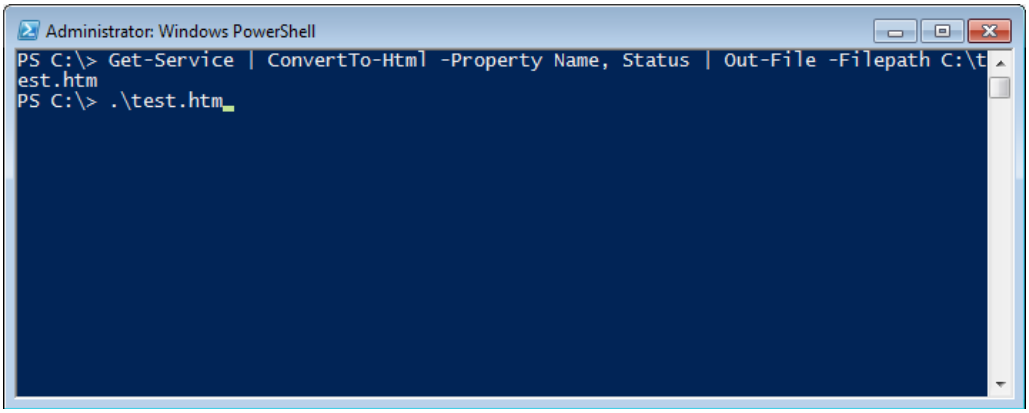
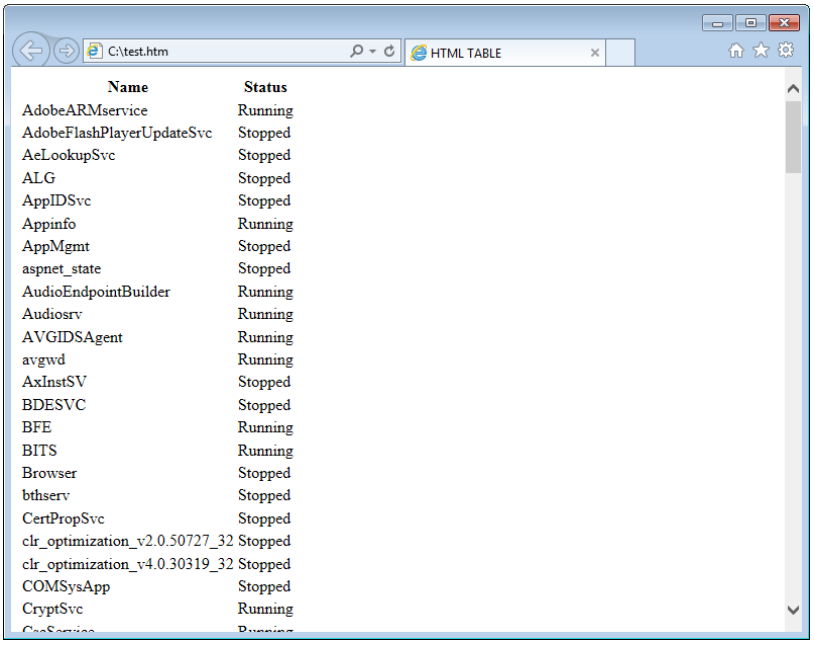


Figure 50. Using "ConvertTo-Html" cmdlet.

Firstly we typed: *"Get-Service | ConvertTo-Html -Property Name, Status | Out-File -Filepath C:\test.htm"*.
This will get all services, it will take the selected properties to be converted to html, convert and it will output this html file to the selected filepath with the selected filename. After this we execute this and PowerShell automatically opens a browser to display the file, as shown in figure 51.



Name	Status
AdobeARMservice	Running
AdobeFlashPlayerUpdateSvc	Stopped
AeLookupSvc	Stopped
ALG	Stopped
AppIDSvc	Stopped
Appinfo	Running
AppMgmt	Stopped
aspnet_state	Stopped
AudioEndpointBuilder	Running
Audiosrv	Running
AVGIDSAgent	Running
avgwd	Running
AxInstSV	Stopped
BDESVC	Stopped
BFE	Running
BITS	Running
Browser	Stopped
bthserv	Stopped
CertPropSvc	Stopped
clr_optimization_v2.0.50727_32	Stopped
clr_optimization_v4.0.30319_32	Stopped
COMSysApp	Stopped
CryptSvc	Running

Figure 51. Browser open to display the html file.

TIP
Another great new cmdlet you can try, coming with the latest's PowerShell versions (4.0 & 5.0) let us ***work with JSON*** *file formats, instead of using XML. You can get full information about working with JSON by looking at the help file. Type: "Get-Help ConverTo-Json" or "Get-Help ConvertFrom-Json" and read about syntax to use these cmdlets with the most efficient way.*

All these are going to give you an idea of how to get deeper in the pipeline and how to pipe from one command to another to achieve a certain result.

Now, it is time to emphasize on something we have written earlier. There are some dangerous shape of using "Stop*" or "Remove*" cmdlets. However here are also some very basic parameters we have to know about.

3.3.8 Confirm & WhatIf Parameters

NOTE
You can get full information about using -Confirm & -WhatIf parameters by looking at the help file. Type: "Get-Help Stop-Process" and "Get-Help Stop-Service" and read about syntax to use these cmdlets with the most efficient way.

In PowerShell, if you ever have to do something that you are uncertain of, you can always type -WhatIf. Figure 52 shows a way to use "WhatIf" parameter. It shows that if you request for a "WhatIf" I do something, PowerShell will tell you what your cmdlet would have do, without letting the cmdlet do it. [25]

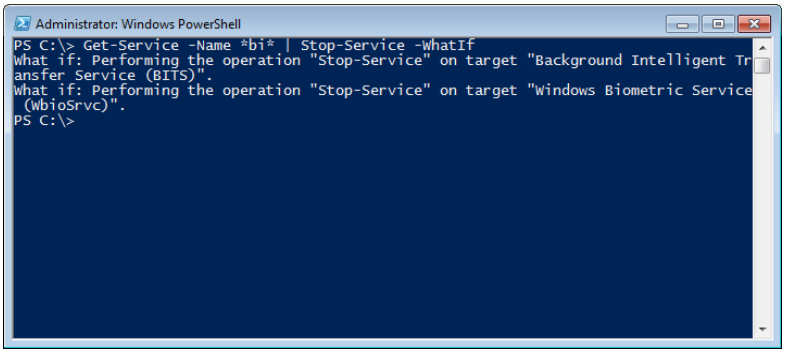


Figure 52. Simple WhatIf output.

In fact, "WhatIf" parameter provides a useful way to preview what a potentially dangerous cmdlet would have done to your computer, to make certain that you want to do that.

A similar parameter is "-Confirm". This parameter should be supported by any cmdlet that makes some kind of change to the system. It differs from "WhatIf" at the point that it not only informs us about what the cmdlet is going to do, it sends us a confirmation warning where we can decide if we want the cmdlet to be executed or not. Figure 53 shows an example of using *WhatIf* and *Confirm* parameters.

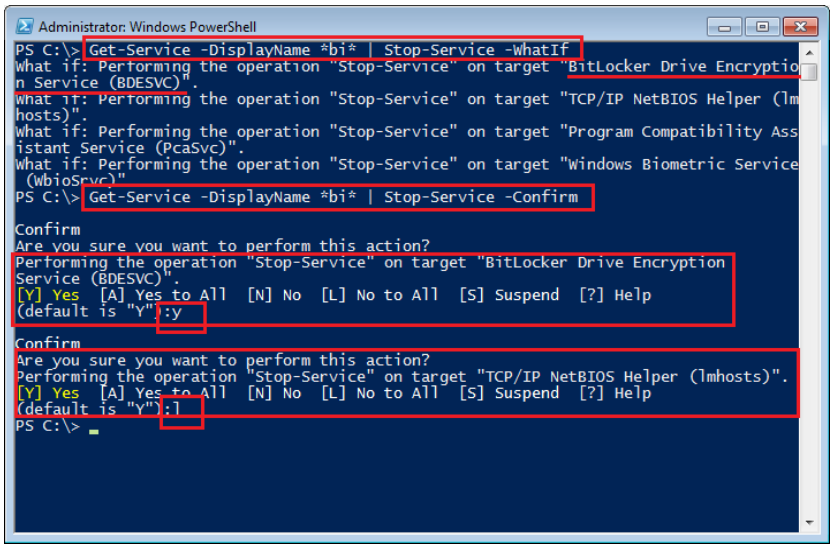


Figure 53. Working with WhatIf and Confirm parameters.

3.3.9 Running Conditions & Loops

As with other programming languages, PowerShell provide a way to create conditional statements and loops using comparison and logical operators. These statements can be implemented in both components of PowerShell (host console & scripting environment).

NOTE

You can get full information about construct if statements and loops by looking at the help file. Type: "Get-Help about_if" or "Get-Help about_switch" or "Get-Help about_do" or "Get-Help about_for" or "Get-Help about_foreach" or "Get-Help about_while" or "Get-Help about_break" and so on.

First of all, as we have already mentioned there are operators within PowerShell that we can use in order to construct conditional statements and loops.

In fact, all of these operators make questions that have only two possible answers; true or false. Figure 54 shows that Windows PowerShell can answer all these kind of statements.

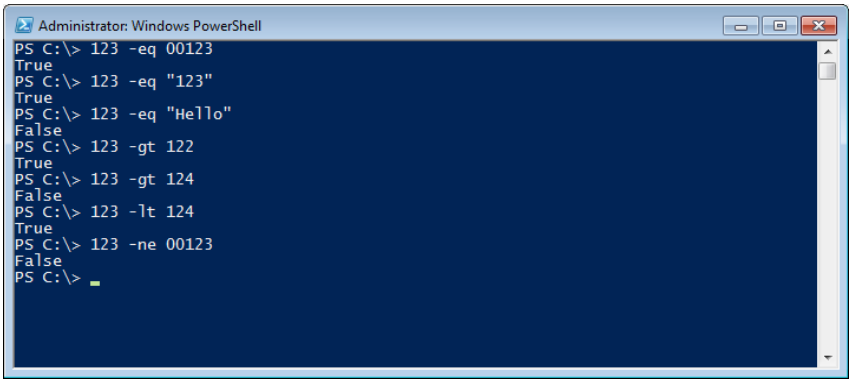


Figure 54. Simple statements that PowerShell can answer.

NOTE
You can get full information about comparison & logical operators by looking at the help file. Type: "Get-Help about_comparison_operators" and "Get-Help about_logical_operators".

NOTE
You can get full information about variables by looking at the help file. Type: "Get-Help about_variables".

Working with variables, conditional statements and loops we can compose intelligent PowerShell code capable of making decisions. In the following two figures we try to show the way that these functions work.

3.3.9.1 If – elseif –else statement

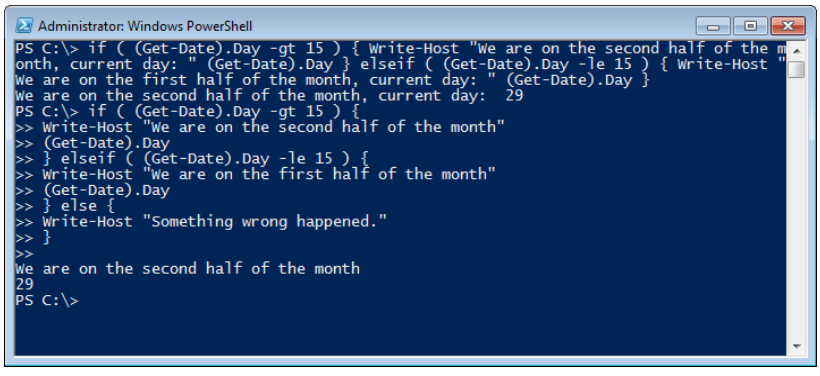


Figure 55. Simple example of using "if-elseif-else" conditional statements.

As we can see, in figure 55, we can write a conditional statement in one line, but we prefer to break it into several lines to increase readability.

What we did, is we constructed an *if statement* to return true if the current day (integer between 1 and 31) of the month is greater than 15, again return true if the

current day of the month is less or equal to 15 but print different output, and if both of these statements return false then print "Something wrong happened.".

In fact, this will take the current day as an integer and will implement the comparison operators, return true or false, and finally, print something.

3.3.9.2 Switch statement

Another great comparison can be implemented using *switch* statement, as shown in figure 56. What we did, is we passed the current way of the week (type of String) as switch parameter and constructed a switch statement that compares this value to each of the conditions. When the value of the switch parameter and the value of a condition match, an output will be printed.

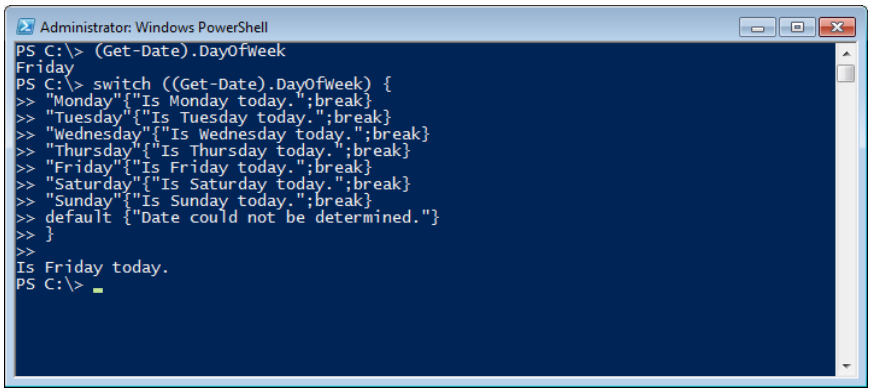


Figure 56. Simple example of using "switch" conditional statements.

Another simpler example of switch statement can be constructed as shown in figure 57.

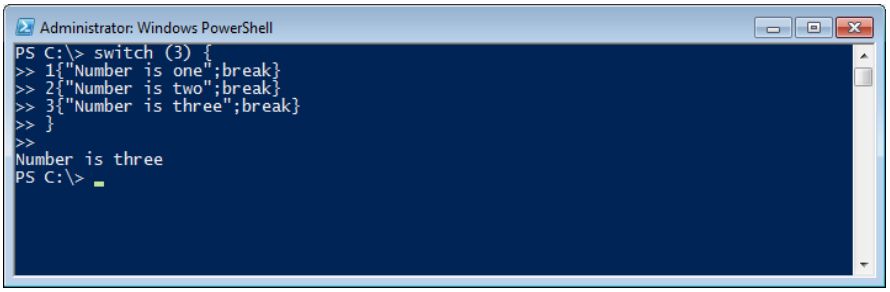


Figure 57. Simpler example of using "switch" conditional statements.

In order to extend conditional statements we have *loops*. There are 4 standard ways to compose a loop. We are going to work with some of these, giving examples.

3.3.9.3 While loop

Figure 58 shows the way we can construct while loops. We construct a *while loop* that executes the commands inside the blocks as long as the conditional test evaluates to true.

When *while loop* runs, Windows PowerShell evaluates the condition *($counter -le 5)* of the statement before entering the command block section. The condition portion of the statement resolves to either true or false. As long as the condition remains true, PowerShell reruns the commands inside the while block.

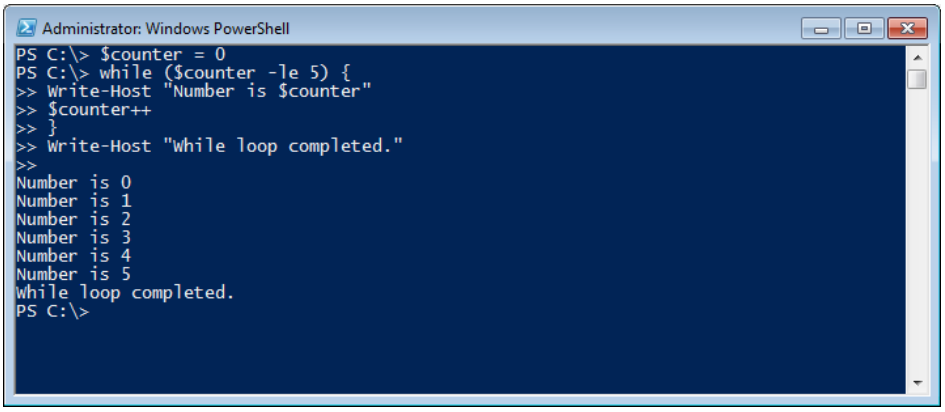


Figure 58. Simple example of using "while" loops.

3.3.9.4 For loop

Figure 59 shows the way we can construct for loops. We construct a simple *for loop* that runs commands inside the command block while the specified condition evaluates to true.

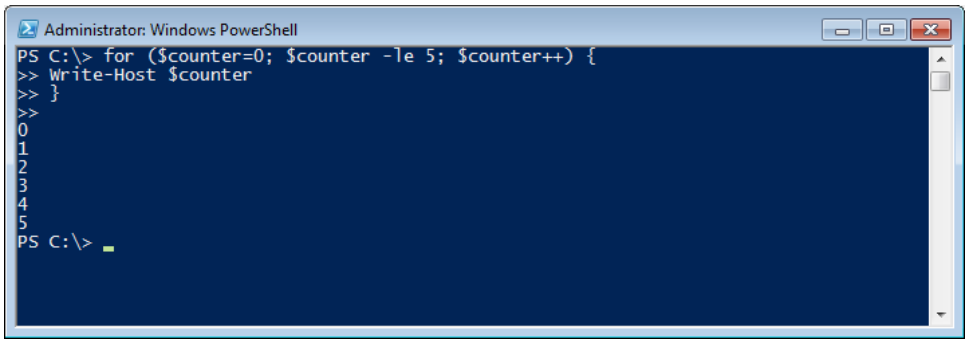


Figure 59. Simple example of using "for" loop.

Loops repeat particular PowerShell statements with the pipeline being one of the areas where you can benefit from loops. Most PowerShell commands wrap their results in arrays, and we will need loops when we want to examine single elements in an array more closely. In most cases, if we want to iterate all the values in an array, we use *foreach loops* as following.

3.3.9.5 Foreach loop to work with ArrayList

Figure 60 provides a way to work with loops and arrays. In fact, we are going to show a way that a *foreach loop* can be constructed to work with arrays.

Firstly, we get the newest 50 security eventlog entries (eventlogentry objects) from the eventlog and store all these in a variable. After this we are going to create an empty arraylist in which we are going to store only specific eventlog entries that have eventid *(instanceid)* to be equal with 4625.

TIP

You can use ForEach-Object cmdlet to process single objects of the PowerShell pipeline, such as to output the data contained in object properties as text or to invoke methods of the object. You can get full information about using ForEach-Object cmdlet by looking at the help file. Type: "Get-Help ForEach-Object".

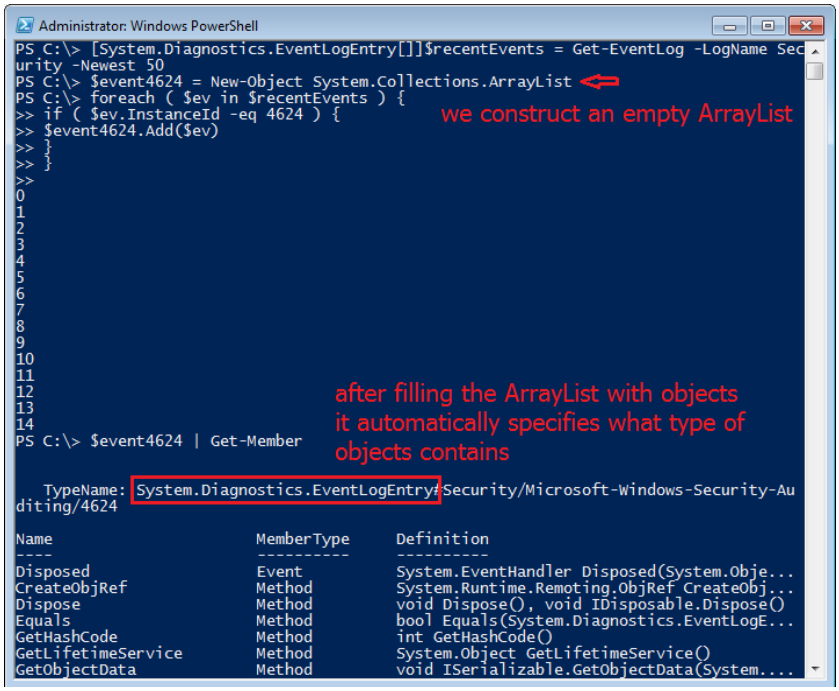


Figure 60. "foreach" loop can be constructed to work with arrays.

Concluding with conditional statements and loops, we cannot pass that all loops can exit ahead of schedule with the help of *Break* and skip the current loop cycle with the help of *Continue*.

Another thing we have to point out is that except from working in the PowerShell console host, we can use these techniques to deal with more complex tasks by putting all these in a script or more formally we can make modules that will contain functions. This is basically what we are going to describe in the next section.

3.4 Preparing for Toolmaking

As we have already mentioned, PowerShell consists of two components: the standard console host application (powershell.exe) and the more visual Integrated Scripting Environment (ISE; powershell_ise.exe). [10]

PowerShell ISE is an environment where you can write a bunch of commands consequently, put all these in a script and run all from there.

PowerShell toolmakers are focused on making reusable products, packaged tools that can complete a task. They used the term *toolmaking* instead of *scripting* because there is a key difference between the two. A script is something you make for yourself; it is often quick and ugly. A tool, one the other hand, has to be more structured and more resilient to errors. Tools need to be a bit more professional so you can share them. [26]

3.4.1 Why I should start making tools with PowerShell

Windows PowerShell provides a simpler, scripting-like environment where you can make tools, rather than moving into Visual Studio and a .NET Framework language like C#.

PowerShell Toolmakers still working on a professional environment that provides all the discipline and maturity of a developer—anticipating and handling errors, validating user input, and so forth.

PowerShell toolmakers work in a simpler environment than developers and often produce less-complex tools. They also can often tap into broad portions of the .NET Framework. [26]

3.4.1. PowerShell ISE

The Windows PowerShell Integrated Scripting Environment (ISE) is a graphical host application for Windows PowerShell. In *PowerShell ISE*, you can run commands and write, test, and debug scripts in a single Windows-based graphic user interface with multiline editing, tab completion, syntax coloring, selective execution, context-sensitive help, Show Command (compose commands in a window) and support for double-byte character sets and right-to-left languages. [27]

There are several ways to find and run *PowerShell ISE* executable files.

- On older versions of Windows you can navigate from *Start Menu: All Programs > Accessories > Windows PowerShell and then click Windows PowerShell ISE. You can also select Run from Start Menu, type PowerShell_ise.exe, and hit Enter to open the PowerShell console application.*
- On Windows 8.1 and Windows Server 2012, *hold the Windows key on your keyboard and press R to get the Run dialog box. Or, press and release the Windows key, and start typing "powershell" to quickly get to the PowerShell icons. Hit right click on the powershell.exe icon and then click on Run ISE as an Administrator.*

Figure 61 provides a snapshot of *PowerShell ISE* environment. You can navigate from Menu >> View to select panes that will be displayed. Thus, we can see that there is a scripting pane, a console pane and the commands pane. Also, there are buttons across the top that allows you to fit the scripting and console pane as you want.

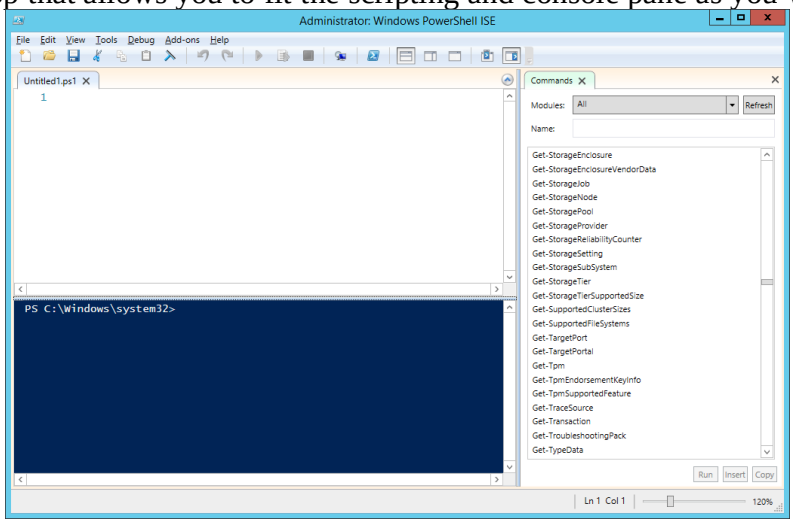


Figure 61. PowerShell Integrated Scripting Environment.

3.4.2 Running functions

Functions are self-defined new commands consisting of general PowerShell building blocks. Like cmdlets, functions can have parameters. The parameters can be named, positional, switch, or dynamic parameters. Function parameters can be read from the command line or from the pipeline. Functions can return values that can be displayed, assigned to variables or passed to other functions or cmdlets. You can create a function that works just like a cmdlet without using C# programming. [28]

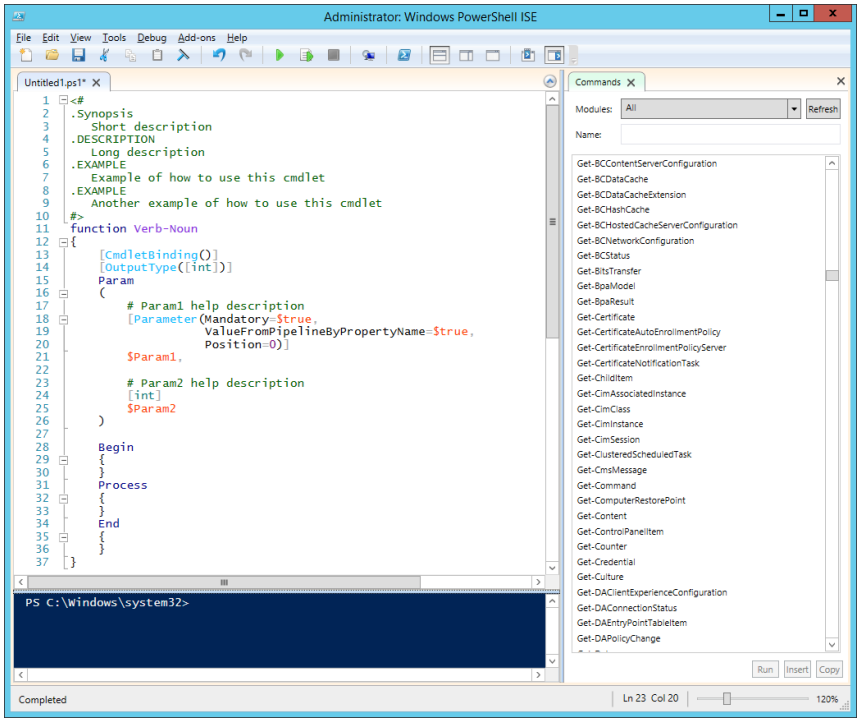


Figure 62. Hit Ctrl+J to view the snippets and select the first one.

This is the environment where you can start building cmdlets and Figure 62 provides a quick way to start with. PowerShell ISE ships with some default snippets. When you are on the script pane, you can hit Ctrl+J. All the available snippets will be displayed and you can choose one. Figure 62 shows the result of selecting the first snippet that appeared after hitting Ctrl+J and this is the basic structure of a self-defined cmdlet. It consists of the help text for the cmdlet *(lines 1-10)*, the cmdlet name *(line 11)* some parameters and 3 blocks *(line 28, 31, 34)* to run our code.

Another thing you might now about self-defined functions is that after creating a custom function, we have to write help for this function.

NOTE
You can get full information about writing help for your cmdlets by looking at the help file. Type: "Get-Help about_Comment_Based_Help".

NOTE
You can get full information about function by looking at the help file. Type: "Get-Help about_functions" or "Get-Help about_functions_advanced" or "Get-Help about_about_functions_advanced_methods" and so on.

3.5 Summary

In this section we have covered many different shapes of using PowerShell. Once we make sure that we have the latest and greatest version of PowerShell installed, we can start discovering PowerShell. Help System is our best friend. It helps us figuring out how to deal with any kind of task and moreover it helps us explore each cmdlet capabilities. PowerShell implements an object-oriented environment and this means that most of the cmdlets return objects as an output. Pipeline let us pass the output from one command as input to the next command. PowerShell feature of piping objects facilitates users and especially administrators. There are cmdlets of selecting, sorting, filtering, converting and exporting that can help us to deal with any kind of administrative task. Finally PowerShell Toolmaking is the term that can describe a way that PowerShell tools can be made. If you want to start toolmaking, PowerShell ISE is an environment that self-defined cmdlets, script and generally modules can be written.

Chapter 4. Security Features

Security plays two important roles in PowerShell. The first is the security of PowerShell itself. Scripting languages have long been a vehicle of email-based malware on Windows, so PowerShell's security features have been carefully designed to thwart this danger. The second role is the set of security-related tasks you are likely to encounter when working with you computer: script signing, credentials, events, just to name a few [29].

In this chapter we are going to clarify the initial windows powershell security setting. Subsequently we will be referred to the Security Module of PowerShell and some other security relative cmdlets. Finally, we are going to give some examples of using *Get-EventLog* and *Get-WinEvent* cmdlets for managing Windows Event Logs.

NOTE
You can get full information about execution policies by looking at the help file. Type: "Get-Help about_execution_policies".

4.1 Initial PowerShell Security Settings

By default, the shell will not run files with a PS1 file name extension when you double-click on them. That extension is associated with Notepad. In fact, by default, the shell will not run scripts at all because of a built-in feature called the Execution Policy, which describes the conditions under which a script will run. It is set to Restricted out of the box, which prohibits all scripts from running and enables the shell only for interactive use. The Execution Policy can be changed using the Set-ExecutionPolicy cmdlet.

NOTE
You can get the execution policy for the current session by looking at the help file. Type: "Get-Help Get-ExecutionPolicy".

NOTE
You can set the execution policy for the current session by looking at the help file. Type: "Get-Help Set-ExecutionPolicy".

4.2 PowerShell Security Module

Once we get past the basic concepts of execution policies, security in PowerShell has a PowerShell-oriented layer of security controls. In fact, the PowerShell security module is an intermediate stage before you can explore to help you as a system administrator before you step off into the depths of "raw" level of .NET code. In fact, on every PowerShell version 3.0 and later installations there is a module for security named Microsoft.PowerShell.Security.

You can get basic information about this module as shown in figure 63.

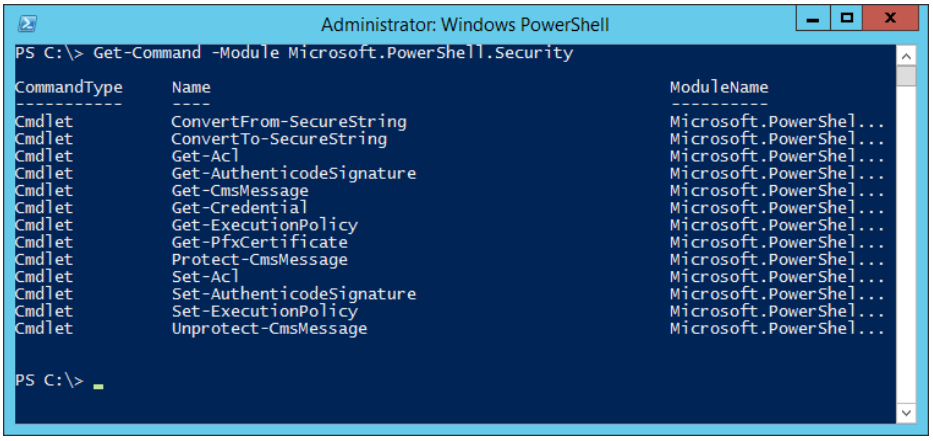


Figure 63. cmdlets that the Security Module contains.

As we can see in figure 63, although the pool of security cmdlets provided by Microsoft Security Module is fairly small, it offers support for a few key areas:

- Access Control List (ACL)
- Authenticode Signature
- Credential
- Execution Policy
- PfxCertificate
- SecureString
- CmsMessage

Once you begin to explore security concepts beyond this basic set of cmdlet you will need to become more steeped in .NET security concepts as well as the Microsoft's implementation of more universal security ideas. .NET has come a long way since its earliest incarnations to provide a very robust model for dealing with security as a whole. Yet, it is good to know, that, as an administrator, basic credential, provider and certificate management can be handled with PowerShell trivially thanks to the security module. [30]

4.3 Other PowerShell Security relative cmdlets

We have already given an example of using "Get-EventLog" cmdlet *(at 3.3.1.5 Using Help to deal with a task Paradigm)*. "Get-EventLog" cmdlet is part of the Microsoft.PowerShell.Management module and it can be used for getting the events of an event log, or a list of the event logs, on the local or remote computers [31].

"Get-EventLog" gets events only from classic event logs. There is another cmdlet that can help us getting the eventlog in newer versions of Windows *(Vista, Server 2008, and later versions)*. It is called "Get-WinEvent" and it has designed to replace the "Get-EventLog" cmdlet [32].

NOTE
You can get full information about accessing event logs via PowerShell by looking at the help file. Type: "Get-Help Get-EventLog" or "Get-Help Get-WinEvent" and read about syntax to use these cmdlets with the most efficient way.

4.4 Managing Windows Event Log via PowerShell

As we have already mentioned, PowerShell "Get-EventLog" and "Get-WinEvent" cmdlets let us query and work with event log data on a Windows system. [33] In this section we are going to provide some key uses of these cmdlets.

4.4.1 "Get-EventLog" cmdlet

We are going to work with "Get-EventLog" cmdlet and before we begin we cannot pass getting the help for this cmdlet to analyze the available parameters so we can use it with the more efficient way. Furthermore we are going to work with objects of type EventLogEntry and it is good to know what this type of object contains *(properties and methods)*. So we pipe to Get-Member to do this.

Figure 64 shows how we can determine which event logs exist on a system. We make use of the Get-EventLog parameter called List to achieve this.

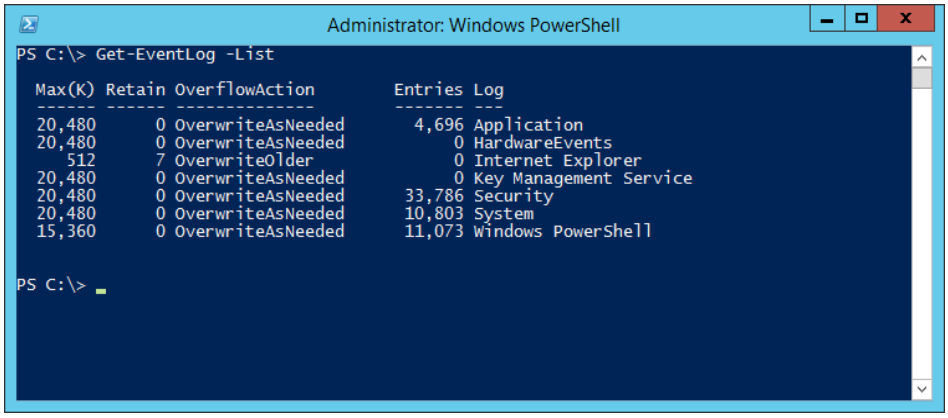


Figure 64. Get-EventLog to determine which even logs exist on a system.

Remember that we can use the pipeline to format or filter events coming.
Figure 65 shows the way to display to a table, only specific properties of the 10 most recent System events.

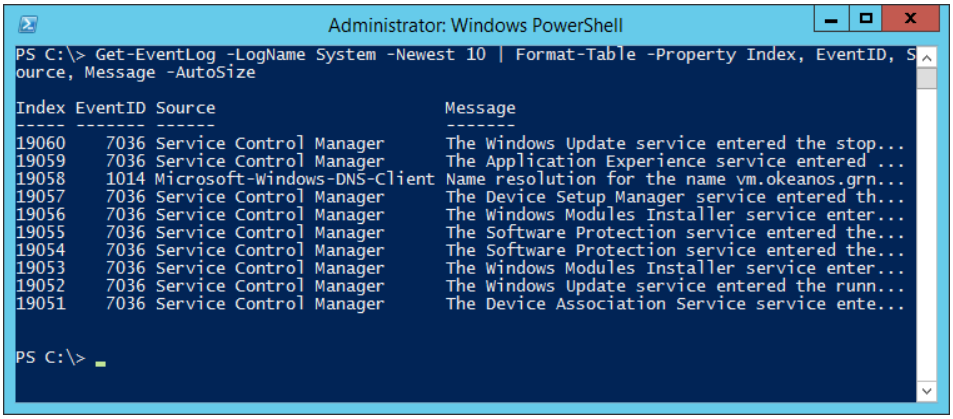


Figure 65. Get-EventLog to get the 10 most recent entries of the System event log combined with Format-Table.

Figure 66 provides a representation of piping objects to Where-Object cmdlet. Where-Object is going to grab all of the Application EventLogEntry objects that coming from the pipeline and it will display only these that mention the term "powershell"

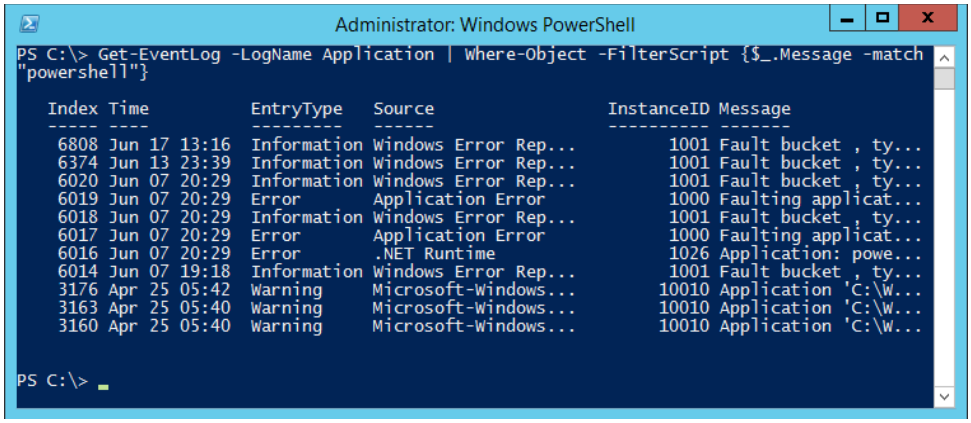


Figure 66. Searching the event log for entries that mention the term "powershell".

Two parameters of the Get-EventLog cmdlet we might be able to use are the *InstanceId* and *After*. Figure 67 provides a way to measure the failure logon events from the security event log that created after the month June.

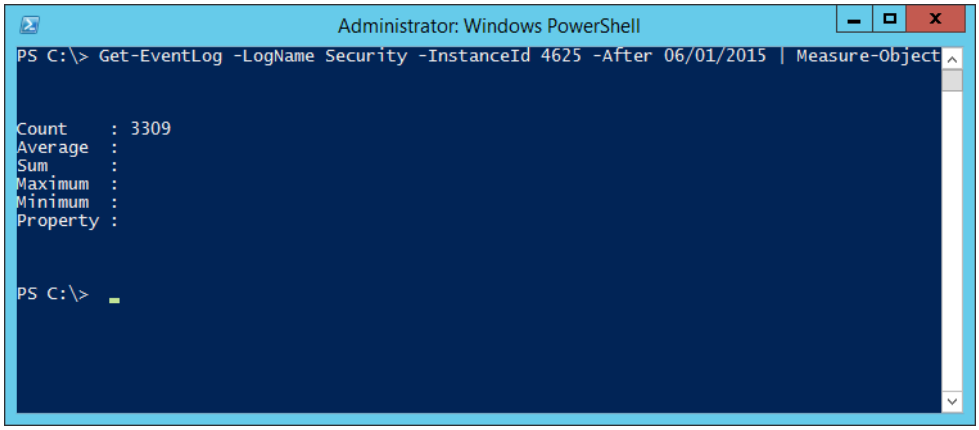


Figure 67. Measure events from Security Event Log created after a certain date.

Finally, another impressive result from managing event logs with PowerShell can be achieved. Consider that we want to display all the Security events grouped by a property. Searching at the help system, we realize that there is a cmdlet called "Group-Object" to help me group something. Figure 68 provides a way to use Get-EventLog combined with Group-Object and Sort-Object cmdlets. Firstly we get all of the Security Events that have been created after the month May. We send all these objects across the pipeline and Group-Objects will grouped them by the property EventId and then send them to be sorted and finally will be displayed, as shown in figure 68.

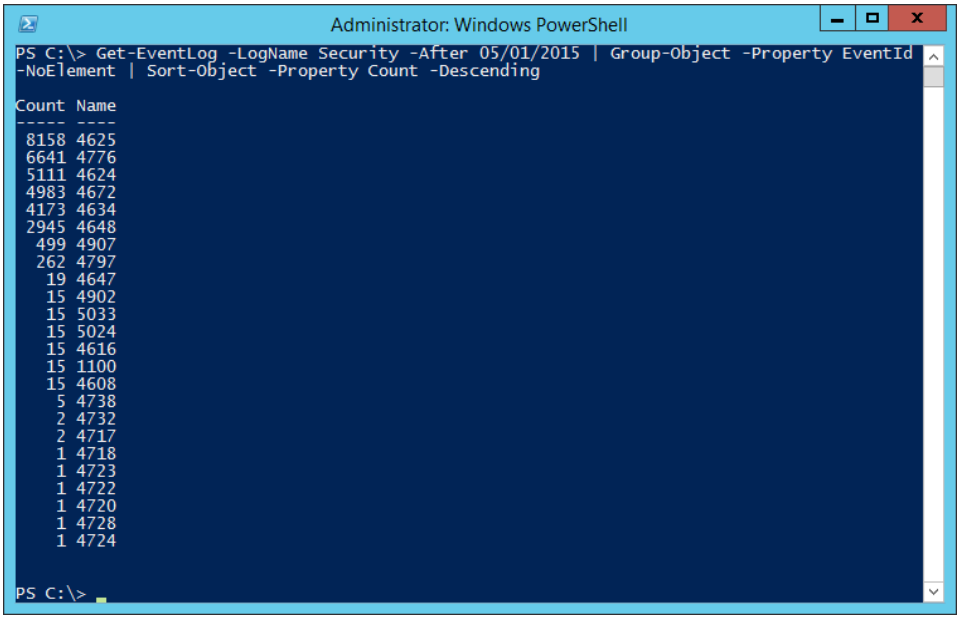


Figure 68. Get-EventLog combined with Group-Object and Sort-Object cmdlets.

4.4.2 "Get-WinEvent" cmdlet

We are going to work with "Get-WinEvent" cmdlet and before we begin we cannot pass getting the help for this cmdlet to analyze the available parameters so we can use it with the more efficient way. Furthermore we are going to work with objects of type EventLogRecord and it is good to know what this type of object contains *(properties and methods)*. So we pipe to Get-Member to do this.

Contrary to the Get-EventLog cmdlet, Get-WinEvent has a parameter called ListLog that we can use in order to receive the event logs appearing on a system. Figure 69 shows how to use ListLog parameter to achieve this. We can see that there are up to 296 different event logs that Get-WinEvent can access.

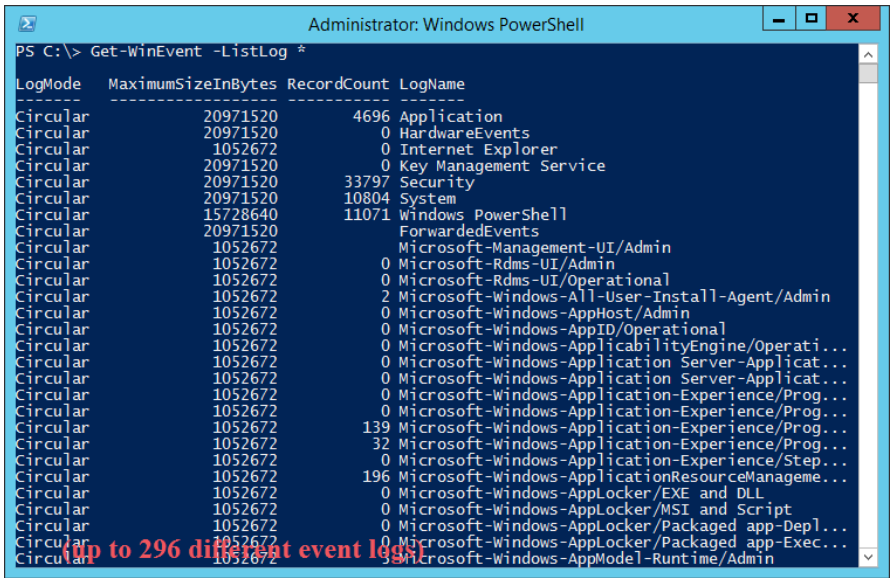


Figure 69. Get-WinEvent to determine which even logs exist on a system.

Figure 70 provides an example of how to use MaxEvents parameter of Get-WinEvent cmdlet. It takes only 15 events from the application event log and it send them to be displayed as a table.

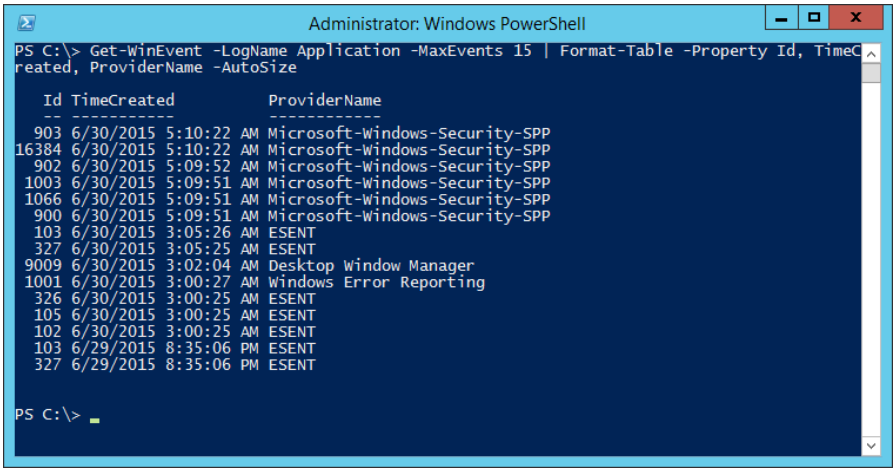


Figure 70. Get-WinEvent combined with Format-Table cmdlet.

Finally, another impressive result from using Get-WinEvent cmdlet and FilterHashtable parameter can be achieved. We are going to make a hash filter in order to get the first 1 event from the security event log that has id to be equals to 4625. After this we pipe to Format-List to display only selected properties as a list, as shown in figure 71.

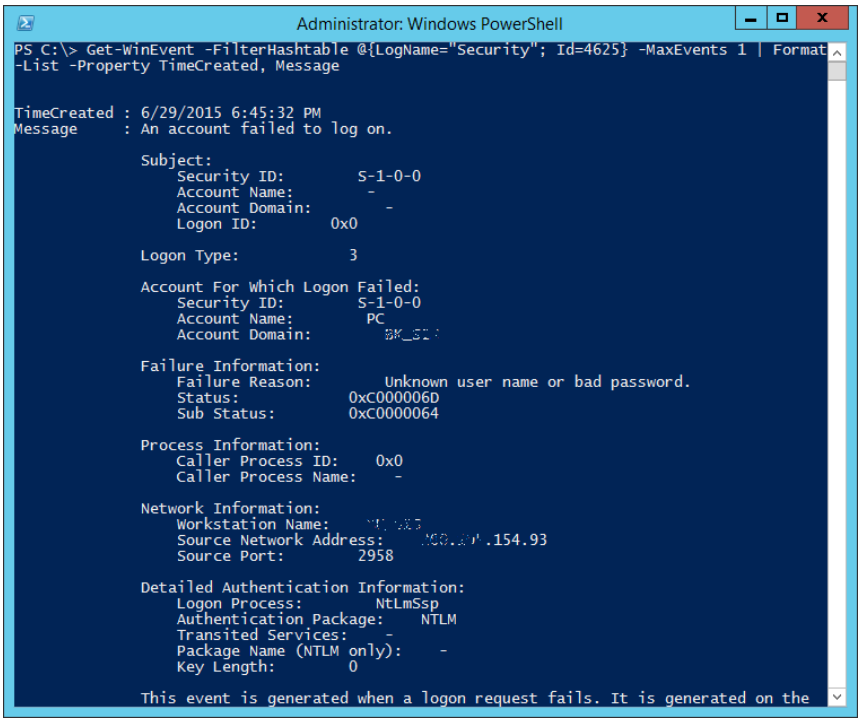


Figure 71. Get-WinEvent using FilterHashtable parameter.

4.5 Summary

Windows PowerShell enables you to control your script execution-policy. It also offers a Security Module where you can discover cmdlets that can help you dealing with security stuff. Finally "Get-EventLog" and "Get-WinEvent" cmdlet are introduced giving examples.

Chapter 5. HIDS with PowerShell

In this chapter, we are going to extend PowerShell capabilities by making modules and scripts that will implement a fully functional Host-Based Log Monitoring IDS. Firstly we will be describing the basic structure of the system and then all the code will be provided [37].

5.1 Overview of the implementation

5.1.1 Background of the components

Background of the components that has been used is described as follows:

- Windows Server 2012 R2 Server with Remote Desktop Connection enabled.
- SQL Server 2012 Express [35]
- PowerShell ISE

Way it works:

- Installing SQL Server 2012 Express [36]
- Creating a Database named "LogDB"
- Creating a new Table named "EVENTS" *(SQL query included)*
- Creating New Folder for PowerShell Modules
- Copy Modules in the folder we created
- Open a PowerShell console
- Importing the LogAnalysis Module
- Saving all of the available events in Database
- Auto Created Tables: DETAILS 4624, DETAILS 4625 etc.
- Schedule Automatic Database filling *(JobScheduler.ps1)*
- Creating a folder in the Desktop named "LogVisualization"
- Copy the LogVisualization.ps1 in the LogVisualization folder
- "Converting" the .ps1 script ton an executable file
- Running the executable to display the visualization

5.1.2 Overview of the components

The system consists of the following components:

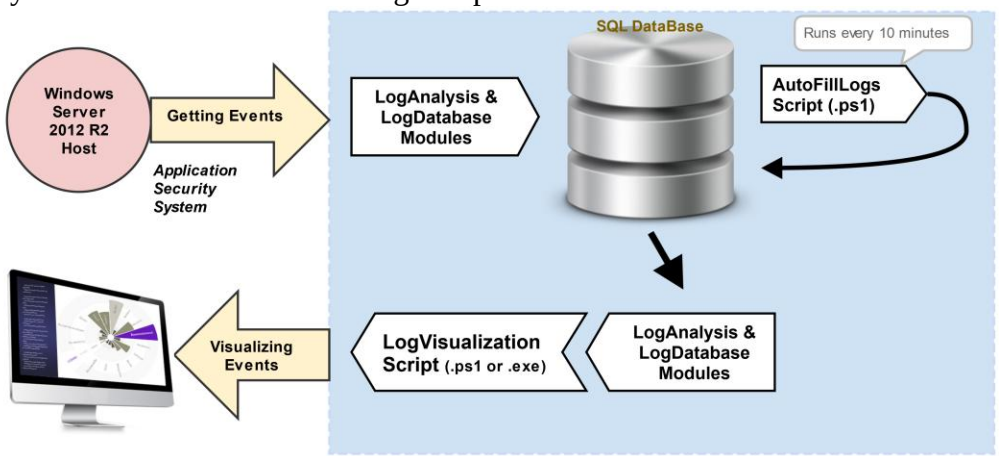


Figure 72. Components of the author's HIDS implementation.

Figure 72 provides a schema that can be described as follows:

- Getting events from the Windows Server Host machine *(data source)*.
- Events are being analyzed and stored in the database *(analysis engine)*.
- Script runs every ten minutes to ensure that the database is continually updated.
- Events are being analyzed and ready for visualization *(analysis engine)*.
- Visualizing events running *(response engine)*.

5.2 Custom PowerShell Modules and Scripts

5.2.1 Module: Log Analysis

```
$LogErrorLogPreference = 'c:\log-retries.txt'
$LogConnectionString =
        "server=localhost\SQLEXPRESS;database=LogDB;trusted_connection=True"

# Imports LogDatabase Module in order to be able to use Get-LogDatabaseData and Invoke-
LogDatabaseQuery cmdlets.
Import-Module LogDatabase


<#
 # ____________________________________________________________________________________________
 # ============================================================================================

  LogAnalysis Module contains 18 cmdlets. All these "LogAnalysis" cmdlets are divided in two 3
major groups.


   ## First Group ##
    # cmdlets that are used only within this module
      Get-DatabaseAvailableTableNames
      Set-LogEventInDatabase
      Set-TableAutoIncrementValue
      Clear-TableContentsFromDatabase
      Get-CaptionFromSId
      Get-LogonType
      Get-ImpersonationLevelExplanation
      Get-StatusExplanation
      Get-DatesUntilNow
      Get-TimeRangesForNames
      Get-TimeRangesForValues


   ## Second Group ##
    # cmdlets that are used only from the script: "LogVisualization.ps1"
      Get-TableRowNumber
      Get-LastEventDateFromDatabase
      Get-EventsOccured
      Get-HashTableForPieChart
      Get-HashTableForTimeLineChart
      Get-LogonIpAddresses
      Get-TableContents


   ## Third Group ##
    # cmdlets that are used only from the script: "ScheduleLogs.ps1"
      Get-LastStoredEvent


# ============================================================================================
# ____________________________________________________________________________________________

#>


# ================================================================
## First Group ## START
# cmdlets that are used only within this module

<#
.NAME
   Get-DatabaseAvailableTableNames

.SYNOPSIS
   Gets the available table names from database.

.SYNTAX
   Get-DatabaseAvailableTableNames

.DESCRIPTION
   The Get-DatabaseAvailableTableNames cmdlet gets the available table names from database.
   More specifically it will go out, send a query to the database and outputs a dataset that
```

```
   will contains values of type strings representing the names of the tables of the database.

   The Get-DatabaseAvailableTableNames is been used from the Set-LogEventInDatabase cmdlet.

.PARAMETERS
   None

.INPUTS
   None

.OUTPUTS
   [System.Data.DataSet]

.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 -------------------------

   PS C:\> Get-DatabaseAvailableTableNames

   This will return a dataset if strings that represent the names of the available tables of the
database.

#>
function Get-DatabaseAvailableTableNames
{
    [CmdletBinding()]
    Param()
    Process
    {
        Get-LogDatabaseData -connectionString $LogConnectionString `
                            -isSQLServer `
                            -query "SELECT TABLE_NAME
                                    FROM INFORMATION_SCHEMA.TABLES
                                    WHERE TABLE_TYPE = 'BASE TABLE' AND TABLE_CATALOG='LogDB'"
    }
}


<#
.NAME
   Set-LogEventInDatabase

.SYNOPSIS
   Sets the events in database.

.SYNTAX
   Set-LogEventInDatabase [[-EventLogRecordObject
<System.Diagnostics.Eventing.Reader.EventLogRecord[]>]]

.DESCRIPTION
   The Set-LogEventInDatabase cmdlet is basically the first cmdlet it has been written for the
LogAnalysis Module.
   It accepts objects of type: System.Diagnostics.Eventing.Reader.EventLogRecord and it works for
storing
   information of events to the Database.

   Basically Set-LogEventInDatabase is going to grab all of the events came from the pipeline and
insert them into the database.
   For each event that comes from the pipeline Set-LogEventInDatabase takes all of its properties
and inserts them into the EVENTS table, one event at a row.

   If a Security Event comes from the pipeline Set-LogEventInDatabase is able to create new tables
   (for events with Id 4624, 4625, 4907, 4672, 4634, 4648, 4797, 4776, 4735) if they do not exist.

.PARAMETERS
   -EventLogRecordObject <System.Diagnostics.Eventing.Reader.EventLogRecord[]>
    Gives to the cmdlet an array of objects to be set in database.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   [System.Diagnostics.Eventing.Reader.EventLogRecord]

   You can pipe EventLogRecord objects as input to this cmdlet.

.OUTPUTS
   None

.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 -------------------------

   PS C:\> Get-WinEvent -LogName Application, Security, System | Sort-Object -Property TimeCreated
| Set-LogEventInDatabase
```

```
   This command uses the Get-WinEvent cmdlet to get all of the Application, Security and System
events. It uses a pipeline operator (|)
   to send events to the Sort-Object cmdlet. Sort-Obejct command sort all these events by property
TimeCreated and it uses pipeline again
   to send events to the Set-LogEventInDatabase command. Set-LogEventInDatabase cmdlet accepts all
these eventlogrecord objects
   and sends the objects, one at a time, to be parsed and stored in the Database.

.EXAMPLE

   -------------------------- EXAMPLE 2 --------------------------


   PS C:\> Get-WinEvent -LogName Application, Security, System | Sort-Object -Property TimeCreated
| Where-Object -FilterScript {($_.Timecreated).Year -eq 2015} | Set-LogEventInDatabase

   This command uses the Get-WinEvent cmdlet to get all of the Application, Security and System
events. It uses a pipeline operator (|)
   to send events to the Sort-Object cmdlet. Sort-Obejct command sort all these events by property
TimeCreated and it uses pipeline again
   to send events to the Where-Object cmdlet. Where-Object is going to filter only events that
created in year 2015 and send the to the
   Set-LogEventInDatabase command. Set-LogEventInDatabase cmdlet accepts all these eventlogrecord
objects and sends the objects, one at a time,
   to be parsed and stored in the Database.

#>
function Set-LogEventInDatabase
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true)]
        [System.Diagnostics.Eventing.Reader.EventLogRecord[]]$EventLogRecordObject
    )
    Process
    {
        <# if you pass an eventlogrecord array as a parameter in Set-LogEventInDatabase cmdlet
           foreach loop will take this array and run the procedure foreach object of this array
           ---
           on the other hand if you pass eventlogrecord objects from the pipeline
           Set-LogEventInDatabase cmdlet will accept these objects one by one
           and foreach loop will be not used
        #>
        foreach ($ev in $EventLogObject){
            Write-Verbose "It will be stored $events.Count events from $log eventlog in database."

            # there was a problem by inserting in database nvarchar that contains "'" and ";"
characters
            # for this reason we replace char(') and char(;) with blank character to eliminate
problems
            [String]$messagestr = (($ev.Message -replace "'","") -replace ";","")
            # here the query is been made
            $query = "INSERT INTO EVENTS VALUES
                ('$($ev.Id)',
                 '$($ev.Version)',
                 '$($ev.Level)',
                 '$($ev.Task)',
                 '$($ev.Opcode)',
                 '$($ev.Keywords)',
                 '$($ev.RecordId)',
                 '$($ev.ProviderName)',
                 '$($ev.ProviderId)',
                 '$($ev.LogName)',
                 '$($ev.ProcessId)',
                 '$($ev.ThreadId)',
                 '$($ev.MachineName)',
                 '$($ev.TimeCreated)',
                 '$($ev.LevelDisplayName)',
                 '$($ev.OpcodeDisplayName)',
                 '$($ev.TaskDisplayName)',
                 '$($ev.KeywordsDisplayNames)',
                 '$messagestr')"

            Write-Verbose "Query will be: '$query'"
            # here the query is been invoked
            Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                    -isSQLServer `
                                    -query $query

            <# After inserting basic events information in the database,
             # Set-LogEventInDatabase will parse the message string of some critical security
events
             # and will create substantive tables with many critical information.
             #
             # Critical events that tranform into tables are:
             #  - DETAILS4624 : Successfull Logons
             #  - DETAILS4625 : Failure Logons
             #  - DETAILS4907 : Auditing settings on object were changed.
             #  - DETAILS4672 : Special privileges assigned to new logon.
             #  - DETAILS4634 : An account was logged off.
             #  - DETAILS4648 : A logon was attempted using explicit credentials.
```

```
            #  - DETAILS4797 : An attempt was made to query the existence of a blank password for
an account.
            #  - DETAILS4776 : The computer attempted to validate the credentials for an account.
            #  - DETAILS4735 : A security-enabled local group was changed.
            #>
           if ($ev.LogName -eq "Security") {
               $evMessage = $ev.Message.ToString()
               [String[]]$splitMessage = $evMessage -split "\r\n"
               $shortMessage = $splitMessage.Get(0)
               if ($ev.Id -eq 4624){
                   [String]$tableName = "DETAILS4624"
                   if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                       Write-Verbose "Table $tableName not found. It will be created."
                       Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                               -isSQLServer `
                                               -query "CREATE TABLE $tableName (
                                                       [Id] [bigint] IDENTITY(1,1) PRIMARY KEY
NOT NULL,
                                                       [LogName] [nvarchar](50) NULL,
                                                       [EventId] [int] NULL,
                                                       [EventRecordId] [int] NULL,
                                                       [LevelDisplayName] [nvarchar](max) NULL,
                                                       [Message] [nvarchar](max) NULL,
                                                       [TimeCreated] [nvarchar](max) NULL,
                                                       [ComputerName] [nvarchar](max) NULL,
                                                       [SID] [nvarchar](max) NULL,
                                                       [SIDCaption] [nvarchar](max) NULL,
                                                       [LogonType] [nvarchar](max) NULL,
                                                       [ImpersonationLeve1] [nvarchar](max) NULL,
                                                       [ImpersonationLevelExplanation]
[nvarchar](max) NULL,
                                                       [NewLogonSID] [nvarchar](max) NULL,
                                                       [NewLogonAccount] [nvarchar](max) NULL,
                                                       [CallerProccessId] [nvarchar](max) NULL,
                                                       [CallerProcessName] [nvarchar](max) NULL,
                                                       [SourceWorkStationName] [nvarchar](max)
NULL,
                                                       [SourceNetworkAddress] [nvarchar](max)
NULL,
                                                       [SourcePort] [nvarchar](max) NULL,
                                                       [LogonProcess] [nvarchar](max) NULL,
                                                       [AuthenticationPackage] [nvarchar](max)
NULL,
                                                        )"
                   }

                   $sid = $splitMessage.get(3).split(":").get(1).trimstart().trimend()

                   [String]$temp1 = $splitMessage.get(4).split(":").get(1).trimstart().trimend()
                   [String]$temp2 = $splitMessage.get(5).split(":").get(1).trimstart().trimend()
                   [String]$sidCaption = $temp2 + "\" + $temp1

                   [int]$logtype = $splitMessage.get(8).split(":").get(1).trimstart().trimend()
                   $logontype = Get-LogonType -LogonType $logtype

                   $impLevel =  $splitMessage.get(10).split(":").get(1).trimstart().trimend()
                   $impLevelExplanation = Get-ImpersonationLevelExplanation -ImpersonationLevel
$impLevel

                   $newLogonSid = $splitMessage.get(13).split(":").get(1).trimstart().trimend()

                   [string]$tempAcc1 =
$splitMessage.get(14).split(":").get(1).trimstart().trimend()
                   [string]$tempAcc2 =
$splitMessage.get(15).split(":").get(1).trimstart().trimend()

                   [string]$newLogonAcc = $tempAcc2 + "\" +$tempAcc1

                   [string]$callerProcessId =
$splitMessage.get(20).split(":").get(1).trimstart().trimend().ToString()

                   [string[]]$callerProcessNameTemp1 = $splitMessage.get(21).split("")
                   [int]$callerProcessNameTemp2 = $callerProcessNameTemp1.count-1
                   $callerProcessName =
$splitMessage.get(21).split("").get($callerProcessNameTemp2)

                   [string]$sourceWorkstationName =
$splitMessage.get(24).split(":").get(1).trimstart().trimend()
                   [String]$sourceNetworkAddress =
$splitMessage.get(25).split(":").get(1).trimstart().trimend()
                   [string]$sourcePort =
$splitMessage.get(26).split(":").get(1).trimstart().trimend()

                   [string]$logonProcess =
$splitMessage.get(29).split(":").get(1).trimstart().trimend()
                   [string]$authenticationPackage =
$splitMessage.get(30).split(":").get(1).trimstart().trimend()

                   $query = "INSERT INTO $tableName VALUES
                          ('$($ev.LogName)',
                           '$($ev.Id)',
                           '$($ev.RecordId)',
                           '$($ev.LevelDisplayName)',
                           '$shortMessage',
```

```
                        '$($ev.TimeCreated)',
                        '$($ev.MachineName)',
                        '$sid',
                        '$sidCaption',
                        '$logontype',
                        '$impLevel',
                        '$impLevelExplanation',
                        '$newLogonSid',
                        '$newLogonAcc',
                        '$callerProcessId',
                        '$callerProcessName',
                        '$sourceWorkstationName',
                        '$sourceNetworkAddress',
                        '$sourcePort',
                        '$logonProcess',
                        '$authenticationPackage')"

                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: '$query'"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                } elseif ($ev.Id -eq 4625) {
                    [String]$tableName = "DETAILS4625"
                    if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                        Write-Verbose "Table $tableName not found. It will be created."
                        Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                                -isSQLServer `
                                                -query "CREATE TABLE DETAILS4625 (
                                                        [Id] [bigint] IDENTITY(1,1) NOT NULL,
                                                        [LogName] [nvarchar](50) NULL,
                                                        [EventId] [int] NULL,
                                                         [EventRecordId] [int] NULL,
                                                        [LevelDisplayName] [nvarchar](max) NULL,
                                                         [Message] [nvarchar](max) NULL,
                                                        [TimeCreated] [nvarchar](max) NULL,
                                                        [ComputerName] [nvarchar](max) NULL,
                                                         [SID] [nvarchar](max) NULL,
                                                        [SIDCaption] [nvarchar](max) NULL,
                                                        [LogonType] [nvarchar](max) NULL,
                                                        [NewLogonSID] [nvarchar](max) NULL,
                                                        [NewLogonAccount] [nvarchar](max) NULL,
                                                        [FailureReason] [nvarchar](max) NULL,
                                                        [Status] [nvarchar](max) NULL,
                                                        [SubStatus] [nvarchar](max) NULL,
                                                        [CallerProccessId] [nvarchar](max) NULL,
                                                        [CallerProcessName] [nvarchar](max) NULL,
                                                        [SourceWorkStationName] [nvarchar](max)
NULL,
                                                        [SourceNetworkAddress] [nvarchar](max)
NULL,
                                                        [SourcePort] [nvarchar](max) NULL,
                                                        [LogonProcess] [nvarchar](max) NULL,
                                                        [AuthenticationPackage] [nvarchar](max)
NULL,
                                                        CONSTRAINT PK_DETAILS4625 PRIMARY KEY
CLUSTERED (Id ASC)
                                                        ON [PRIMARY]
                                                         )"
                    }

                    $sid =
$splitMessage.get(3).split("").get($splitMessage.get(3).split("").count-1)

                    [String]$temp1 =
$splitMessage.get(4).split("").get($splitMessage.get(4).split("").count-1)
                    [String]$temp2 =
$splitMessage.get(5).split("").get($splitMessage.get(5).split("").Count-1)
                    [String]$sidCaption = $temp2 + "\" + $temp1

                    [int]$logtype =
$splitMessage.get(8).split("").get($splitMessage.get(8).split("").Count-1)
                    $logontype = Get-LogonType -LogonType $logtype

                    $newLogonSid =
$splitMessage.get(11).split("").get($splitMessage.get(11).split("").Count-1)

                    [string]$tempAcc1 =
$splitMessage.get(12).split("").get($splitMessage.get(12).split("").Count-1)
                    [string]$tempAcc2 =
$splitMessage.get(13).split("").get($splitMessage.get(13).split("").Count-1)

                    [string]$newLogonAcc = $tempAcc2 + "\" +$tempAcc1
                    #[string]$logonId =
$splitMessage.get(16).split(":").get(1).trimstart().trimend()

                    [String]$failureReason =
$splitMessage.Get(16).Split(":").Get(1).TrimStart().TrimEnd()

                    [string]$statusCode =
$splitMessage.Get(17).Split("").Get($splitMessage.get(17).split("").count-1)
                    [string]$status = Get-StatusExplanation -Status $statusCode
```

```
                    [string]$subCode =
$splitMessage.Get(18).Split("").Get($splitMessage.get(18).split("").count-1)
                    [string]$subStatus = Get-StatusExplanation -Status $subCode

                    [string]$callerProcessId =
$splitMessage.get(21).split("").Get($splitMessage.get(21).split("").count-1)

                    [string[]]$callerProcessNameTemp1 = $splitMessage.get(22).split("")

                    [String]$callerProcessName =
$splitMessage.get(22).split("").Get($splitMessage.get(22).split("").count-1)

                    [string]$sourceWrkStName =
$splitMessage.Get(25).Split("").Get($splitMessage.Get(25).Split("").Count-1)
                    [string]$sourceNtwAd =
$splitMessage.Get(26).Split("").Get($splitMessage.Get(26).Split("").Count-1)
                    [string]$sourcePrt =
$splitMessage.Get(27).Split("").Get($splitMessage.Get(27).Split("").Count-1)

                    [string]$logonProcess =
$splitMessage.get(30).split(":").Get(1).TrimStart().TrimEnd()
                    [string]$authenticationPackage =
$splitMessage.get(31).split("").Get($splitMessage.Get(31).Split("").Count-1)

                    $query = "INSERT INTO DETAILS4625 VALUES
                           ('$($ev.LogName)',
                            '$($ev.Id)',
                            '$($ev.RecordId)',
                            '$($ev.LevelDisplayName)',
                            '$shortMessage',
                            '$($ev.TimeCreated)',
                            '$($ev.MachineName)',
                            '$sid',
                            '$sidCaption',
                            '$logontype',
                            '$newLogonSid',
                            '$newLogonAcc',
                            '$failureReason',
                            '$status',
                            '$subStatus',
                            '$callerProcessId',
                            '$callerProcessName',
                            '$sourceWrkStName',
                            '$sourceNtwAd',
                            '$sourcePrt',
                            '$logonProcess',
                            '$authenticationPackage')"

                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: '$query'"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                } elseif ($ev.Id -eq 4907) {
                    [String]$tableName = "DETAILS4907"
                    if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                        Write-Verbose "Table $tableName not found. It will be created."
                        Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                                -isSQLServer `
                                                -query "CREATE TABLE $tableName (
                                                        [Id] [bigint] IDENTITY(1,1) NOT NULL,
                                                        [LogName] [nvarchar](50) NULL,
                                                        [EventId] [int] NULL,
                                                        [EventRecordId] [int] NULL,
                                                        [LevelDisplayName] [nvarchar](max) NULL,
                                                        [Message] [nvarchar](max) NULL,
                                                        [TimeCreated] [nvarchar](max) NULL,
                                                        [ComputerName] [nvarchar](max) NULL,
                                                        [SID] [nvarchar](max) NULL,
                                                        [SIDCaption] [nvarchar](max) NULL,
                                                        [CallerProcessId] [nvarchar](max) NULL,
                                                        [CallerProcessName] [nvarchar](max) NULL,
                                                        [ObjectServer] [nvarchar](max) NULL,
                                                        [ObjectType] [nvarchar](max) NULL,
                                                        [ObjectName] [nvarchar](max) NULL,
                                                        [HandleId] [nvarchar](max) NULL,
                                                        [OriginalSecurityDescriptor]
[nvarchar](max) NULL,
                                                        [NewSecurityDescriptor] [nvarchar](max)
NULL,
                                                        CONSTRAINT PK_DETAILS4907 PRIMARY KEY
CLUSTERED (Id ASC)
                                                        ON [PRIMARY]
                                                         )"
                    }

                    $sid =
$splitMessage.get(3).split("").get($splitMessage.get(3).split("").count-1)

                    [String]$temp1 =
$splitMessage.get(4).split("").get($splitMessage.get(4).split("").count-1)
                    [String]$temp2 =
$splitMessage.get(5).split("").get($splitMessage.get(5).split("").Count-1)
```

```
                    [String]$sidCaption = $temp2 + "\" + $temp1

                    [string]$objectServer =
$splitMessage.get(9).split("").get($splitMessage.get(9).split("").Count-1)
                    [string]$objectType =
$splitMessage.get(10).split("").get($splitMessage.get(10).split("").Count-1)

                    [string]$objectName =
$splitMessage.get(11).split("").get($splitMessage.get(11).split("").Count-1)
                    [string]$handleId =
$splitMessage.get(12).split("").get($splitMessage.get(12).split("").Count-1)

                    [string]$callerProcessId =
$splitMessage.get(15).split(":").Get(1).TrimStart().TrimEnd()

                    [String]$callerProcessNameTemp = $splitMessage.Get(16) -split ("Process
Name:")

                    [String]$callerProcessName = $callerProcessNameTemp.TrimStart()


                    [string]$originalSecurityDescriptor =
$splitMessage.Get(19).Split("").get($splitMessage.Get(19).split("").Count-1)
                    [string]$newSecurityDescriptor =
$splitMessage.Get(20).Split("").get($splitMessage.Get(20).split("").Count-1)

                    $query = "INSERT INTO $tableName VALUES
                           ('$($ev.LogName)',
                            '$($ev.Id)',
                            '$($ev.RecordId)',
                            '$($ev.LevelDisplayName)',
                            '$shortMessage',
                            '$($ev.TimeCreated)',
                            '$($ev.MachineName)',
                            '$sid',
                            '$sidCaption',
                            '$callerProcessId',
                            '$callerProcessName',
                            '$objectServer',
                            '$objectType',
                            '$objectName',
                            '$handleId',
                            '$originalSecurityDescriptor',
                            '$newSecurityDescriptor')"

                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: $query"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                } elseif ($ev.Id -eq 4672) {
                    [String]$tableName = "DETAILS4672"
                    if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                        Write-Verbose "Table $tableName not found. It will be created."
                        Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                                -isSQLServer `
                                                -query "CREATE TABLE $tableName (
                                                        [Id] [bigint] IDENTITY(1,1) NOT NULL,
                                                        [LogName] [nvarchar](50) NULL,
                                                        [EventId] [int] NULL,
                                                         [EventRecordId] [int] NULL,
                                                        [LevelDisplayName] [nvarchar](max) NULL,
                                                         [Message] [nvarchar](max) NULL,
                                                        [TimeCreated] [nvarchar](max) NULL,
                                                        [ComputerName] [nvarchar](max) NULL,
                                                         [SID] [nvarchar](max) NULL,
                                                        [SIDCaption] [nvarchar](max) NULL,
                                                        CONSTRAINT PK_DETAILS4672 PRIMARY KEY
CLUSTERED (Id ASC)
                                                        ON [PRIMARY]
                                                         )"
                    }

                    $sid =
$splitMessage.get(3).split("").get($splitMessage.get(3).split("").count-1)

                    [String]$temp1 =
$splitMessage.get(4).split("").get($splitMessage.get(4).split("").count-1)
                    [String]$temp2 =
$splitMessage.get(5).split("").get($splitMessage.get(5).split("").Count-1)
                    [String]$sidCaption = $temp2 + "\" + $temp1


                    $query = "INSERT INTO $tableName VALUES
                           ('$($ev.LogName)',
                            '$($ev.Id)',
                            '$($ev.RecordId)',
                            '$($ev.LevelDisplayName)',
                            '$shortMessage',
                            '$($ev.TimeCreated)',
                            '$($ev.MachineName)',
                            '$sid',
                            '$sidCaption')"
```

```
                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: $query"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                } elseif ($ev.Id -eq 4634) {
                    [String]$tableName = "DETAILS4634"
                    if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                        Write-Verbose "Table $tableName not found. It will be created."
                        Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                                -isSQLServer `
                                                -query "CREATE TABLE $tableName (
                                                        [Id] [bigint] IDENTITY(1,1) NOT NULL,
                                                        [LogName] [nvarchar](50) NULL,
                                                        [EventId] [int] NULL,
                                                         [EventRecordId] [int] NULL,
                                                        [LevelDisplayName] [nvarchar](max) NULL,
                                                         [Message] [nvarchar](max) NULL,
                                                        [TimeCreated] [nvarchar](max) NULL,
                                                        [ComputerName] [nvarchar](max) NULL,
                                                         [SID] [nvarchar](max) NULL,
                                                        [SIDCaption] [nvarchar](max) NULL,
                                                        [LogonType] [nvarchar](max) NULL,
                                                        CONSTRAINT PK_DETAILS4634 PRIMARY KEY
CLUSTERED (Id ASC)
                                                        ON [PRIMARY]
                                                         )"
                    }

                    $sid =
$splitMessage.get(3).split("").get($splitMessage.get(3).split("").count-1)

                    [String]$temp1 =
$splitMessage.get(4).split("").get($splitMessage.get(4).split("").count-1)
                    [String]$temp2 =
$splitMessage.get(5).split("").get($splitMessage.get(5).split("").Count-1)
                    [String]$sidCaption = $temp2 + "\" + $temp1

                    [int]$logtype =
$splitMessage.get(8).split("").get($splitMessage.get(8).split("").Count-1)
                    $logontype = Get-LogonType -LogonType $logtype

                    $query = "INSERT INTO $tableName VALUES
                           ('$($ev.LogName)',
                            '$($ev.Id)',
                            '$($ev.RecordId)',
                            '$($ev.LevelDisplayName)',
                            '$shortMessage',
                            '$($ev.TimeCreated)',
                            '$($ev.MachineName)',
                            '$sid',
                            '$sidCaption',
                            '$logontype')"

                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: $query"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                } elseif ($ev.Id -eq 4648) {
                    [String]$tableName = "DETAILS4648"
                    if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                        Write-Verbose "Table $tableName not found. It will be created."
                        Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                                -isSQLServer `
                                                -query "CREATE TABLE $tableName (
                                                        [Id] [bigint] IDENTITY(1,1) NOT NULL,
                                                        [LogName] [nvarchar](50) NULL,
                                                         [EventId] [int] NULL,
                                                          [EventRecordId] [int] NULL,
                                                        [LevelDisplayName] [nvarchar](max) NULL,
                                                         [Message] [nvarchar](max) NULL,
                                                        [TimeCreated] [nvarchar](max) NULL,
                                                        [ComputerName] [nvarchar](max) NULL,
                                                         [SID] [nvarchar](max) NULL,
                                                        [SIDCaption] [nvarchar](max) NULL,
                                                        [LogonAttempAccount] [nvarchar](max) NULL,
                                                        [TargetServerName] [nvarchar](max) NULL,
                                                        [TargetServerInfo] [nvarchar](max) NULL,
                                                        [ProcessId] [nvarchar](max) NULL,
                                                        [ProcessName] [nvarchar](max) NULL,
                                                        [NetworkAddress] [nvarchar](max) NULL,
                                                        [NetworkPort] [nvarchar](max) NULL,
                                                        CONSTRAINT PK_DETAILS4648 PRIMARY KEY
CLUSTERED (Id ASC)
                                                        ON [PRIMARY]
                                                         )"
                    }

                    $sid =
$splitMessage.get(3).split("").get($splitMessage.get(3).split("").count-1)
```

```
                    [String]$temp1 =
$splitMessage.get(4).split("").get($splitMessage.get(4).split("").count-1)
                    [String]$temp2 =
$splitMessage.get(5).split("").get($splitMessage.get(5).split("").Count-1)
                    [String]$sidCaption = $temp2 + "\" + $temp1

                    [String]$tempAc1 =
$splitMessage.get(10).split("").get($splitMessage.get(10).split("").count-1)
                    [String]$tempAc2 =
$splitMessage.get(11).split("").get($splitMessage.get(11).split("").Count-1)
                    [String]$logonAttempAccount = $tempAc2 + "\" + $tempAc1

                    [String]$targetServerName =
$splitMessage.get(15).split("").get($splitMessage.get(15).split("").Count-1)

                    [String]$targetServerInfo =
$splitMessage.get(16).split("").get($splitMessage.get(16).split("").Count-1)

                    [String]$processId =
$splitMessage.get(19).split("").get($splitMessage.get(19).split("").Count-1)

                    [String]$processName =
$splitMessage.get(20).split("").get($splitMessage.get(20).split("").Count-1)

                    [String]$networkAddress =
$splitMessage.get(23).split("").get($splitMessage.get(23).split("").Count-1)

                    [String]$networkPort =
$splitMessage.get(24).split("").get($splitMessage.get(24).split("").Count-1)

                    $query = "INSERT INTO $tableName VALUES
                           ('$($ev.LogName)',
                            '$($ev.Id)',
                            '$($ev.RecordId)',
                            '$($ev.LevelDisplayName)',
                            '$shortMessage',
                            '$($ev.TimeCreated)',
                            '$($ev.MachineName)',
                            '$sid',
                            '$sidCaption',
                            '$logonAttempAccount',
                            '$targetServerName',
                            '$targetServerInfo',
                            '$processId',
                            '$processName',
                            '$networkAddress',
                            '$networkPort')"

                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: $query"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                }  elseif ($ev.Id -eq 4797) {
                    [String]$tableName = "DETAILS4797"
                    if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                        Write-Verbose "Table $tableName not found. It will be created."
                        Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                                -isSQLServer `
                                                -query "CREATE TABLE $tableName (
                                                        [Id] [bigint] IDENTITY(1,1) NOT NULL,
                                                        [LogName] [nvarchar](50) NULL,
                                                        [EventId] [int] NULL,
                                                         [EventRecordId] [int] NULL,
                                                        [LevelDisplayName] [nvarchar](max) NULL,
                                                         [Message] [nvarchar](max) NULL,
                                                        [TimeCreated] [nvarchar](max) NULL,
                                                        [ComputerName] [nvarchar](max) NULL,
                                                         [SID] [nvarchar](max) NULL,
                                                        [SIDCaption] [nvarchar](max) NULL,
                                                        [CallerWorkStation] [nvarchar](max) NULL,
                                                        [TargetAccount] [nvarchar](max) NULL,
                                                        CONSTRAINT PK_DETAILS4797 PRIMARY KEY
CLUSTERED (Id ASC)
                                                        ON [PRIMARY]
                                                         )"
                    }

                    $sid =
$splitMessage.get(3).split("").get($splitMessage.get(3).split("").count-1)

                    $callerWorkstation =
$splitMessage.get(9).split("").get($splitMessage.get(9).split("").count-1)

                    [String]$temp1 =
$splitMessage.get(4).split("").get($splitMessage.get(4).split("").count-1)
                    [String]$temp2 =
$splitMessage.get(5).split("").get($splitMessage.get(5).split("").Count-1)
                    [String]$sidCaption = $temp2 + "\" + $temp1

                    [String]$tempAc1 =
$splitMessage.get(10).split("").get($splitMessage.get(10).split("").count-1)
```

```
                    [String]$tempAc2 =
$splitMessage.get(11).split("").get($splitMessage.get(11).split("").Count-1)
                    [String]$targetAccount = $tempAc2 + "\" + $tempAc1

                    $query = "INSERT INTO $tableName VALUES
                           ('$($ev.LogName)',
                            '$($ev.Id)',
                            '$($ev.RecordId)',
                            '$($ev.LevelDisplayName)',
                            '$shortMessage',
                            '$($ev.TimeCreated)',
                            '$($ev.MachineName)',
                            '$sid',
                            '$sidCaption',
                            '$callerWorkstation',
                            '$targetAccount')"

                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: $query"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                } elseif ($ev.Id -eq 4776) {
                    [String]$tableName = "DETAILS4776"
                    if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                        Write-Verbose "Table $tableName not found. It will be created."
                        Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                                -isSQLServer `
                                                -query "CREATE TABLE $tableName (
                                                        [Id] [bigint] IDENTITY(1,1) NOT NULL,
                                                        [LogName] [nvarchar](50) NULL,
                                                        [EventId] [int] NULL,
                                                         [EventRecordId] [int] NULL,
                                                        [LevelDisplayName] [nvarchar](max) NULL,
                                                         [Message] [nvarchar](max) NULL,
                                                        [TimeCreated] [nvarchar](max) NULL,
                                                        [ComputerName] [nvarchar](max) NULL,
                                                         [AuthenticationPackage] [nvarchar](max)
NULL,
                                                        [LogonAccount] [nvarchar](max) NULL,
                                                        [SourceWorkstation] [nvarchar](max) NULL,
                                                        [ErrorCode] [nvarchar](max) NULL,
                                                        CONSTRAINT PK_DETAILS4776 PRIMARY KEY
CLUSTERED (Id ASC)
                                                        ON [PRIMARY]
                                                         )"
                    }

                    [string]$authenticationPackage =
$splitMessage.get(2).split("").get($splitMessage.get(2).split("").count-1)
                    [string]$logonAccount =
$splitMessage.get(3).split("").get($splitMessage.get(3).split("").count-1)
                    [string]$sourceWorkstation =
$splitMessage.get(4).split("").get($splitMessage.get(4).split("").count-1)
                    [string]$errorCode =
$splitMessage.get(5).split("").get($splitMessage.get(5).split("").count-1)

                    $query = "INSERT INTO $tableName VALUES
                           ('$($ev.LogName)',
                            '$($ev.Id)',
                            '$($ev.RecordId)',
                            '$($ev.LevelDisplayName)',
                            '$shortMessage',
                            '$($ev.TimeCreated)',
                            '$($ev.MachineName)',
                            '$authenticationPackage',
                            '$logonAccount',
                            '$sourceWorkstation',
                            '$errorCode')"

                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: $query"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                }  elseif ($ev.Id -eq 4735) {
                    [String]$tableName = "DETAILS4735"
                    if (!((Get-DatabaseAvailableTableNames).table_name).contains($tableName)){
                        Write-Verbose "Table $tableName not found. It will be created."
                        Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                                -isSQLServer `
                                                -query "CREATE TABLE $tableName (
                                                        [Id] [bigint] IDENTITY(1,1) NOT NULL,
                                                        [LogName] [nvarchar](50) NULL,
                                                        [EventId] [int] NULL,
                                                         [EventRecordId] [int] NULL,
                                                        [LevelDisplayName] [nvarchar](max) NULL,
                                                         [Message] [nvarchar](max) NULL,
                                                        [TimeCreated] [nvarchar](max) NULL,
                                                        [ComputerName] [nvarchar](max) NULL,
                                                         [SID] [nvarchar](max) NULL,
                                                        [SIDCaption] [nvarchar](max) NULL,
```

```
                                                    [GroupSID] [nvarchar](max) NULL,
                                                    [GroupCaption] [nvarchar](max) NULL,
                                                    [ChangedAccountName] [nvarchar](max) NULL,
                                                    [ChangesHistory] [nvarchar](max) NULL,
                                                    CONSTRAINT PK_DETAILS4735 PRIMARY KEY
CLUSTERED (Id ASC)
                                                    ON [PRIMARY]
                                                     )"
                    }

                    [string]$sid =
$splitMessage.get(3).split("").get($splitMessage.get(3).split("").count-1)

                    [String]$temp1 =
$splitMessage.get(4).split("").get($splitMessage.get(4).split("").count-1)
                    [String]$temp2 =
$splitMessage.get(5).split("").get($splitMessage.get(5).split("").Count-1)
                    [String]$sidCaption = $temp2 + "\" + $temp1

                    [string]$groupSid =
$splitMessage.get(9).split("").get($splitMessage.get(9).split("").count-1)

                    [String]$tempGr1 =
$splitMessage.get(10).split("").get($splitMessage.get(10).split("").count-1)
                    [String]$tempGr2 =
$splitMessage.get(11).split("").get($splitMessage.get(11).split("").Count-1)
                    [String]$groupCaption = $tempGr2 + "\" + $tempGr1

                    [string]$changedAccountName =
$splitMessage.get(14).split("").get($splitMessage.get(14).split("").count-1)
                    [string]$changesHistory =
$splitMessage.get(15).split("").get($splitMessage.get(15).split("").count-1)

                    $query = "INSERT INTO $tableName VALUES
                           ('$($ev.LogName)',
                            '$($ev.Id)',
                            '$($ev.RecordId)',
                            '$($ev.LevelDisplayName)',
                            '$shortMessage',
                            '$($ev.TimeCreated)',
                            '$($ev.MachineName)',
                            '$sid',
                            '$sidCaption',
                            '$groupSid',
                            '$groupCaption',
                            '$changedAccountName',
                            '$changesHistory')"

                    Write-Verbose "oh found security event $($ev.LogName)"
                    Write-Verbose "Query will be: $query"

                    Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                            -isSQLServer `
                                            -query $query
                }
            }
        }
    }
}


<#
.NAME
   Set-TableAutoIncrementValue

.SYNOPSIS
   Sets the auto increment value of a table to be zero.

.SYNTAX
   Set-TableAutoIncrementValue

.DESCRIPTION
   All the tables in database created to automatically generate a unique number when a new record
is inserted into a table.
   Set-TableAutoIncrementValue cmdlet is used from Clear-TableContentsFromDatabase cmdlet.

.PARAMETERS
   -Table <String[]>
    Gives to the cmdlet a string value that represents a table name.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

.OUTPUTS
   None

.NOTES
```

```
.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Set-TableAutoIncrementValue -Table EVENTS

.EXAMPLE

   -------------------------- EXAMPLE 2 --------------------------

   PS C:\> Set-TableAutoIncrementValue -Table EVENTS, DETAILS4624

#>
function Set-TableAutoIncrementValue
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [String[]]$Table
    )
    Begin
    {
        $Value=0
    }
    Process
    {
        foreach ($ta in $Table){
                $query = "DBCC CHECKIDENT ('$ta',reseed,$Value)"
                Write-Verbose "Query from 'Set-TableAutoIncrementValue cmdlet' will be: '$query'"
                Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                        -isSQLServer
                                        -query $query
        }
    }
}


<#
.NAME
   Clear-TableContentsFromDatabase

.SYNOPSIS
   It removes all the contents from any table of the database.

.SYNTAX

.DESCRIPTION
   This cmdlet Clear-TableContentsFromDatabase helps you interact with the LogDatabase
   and erase the contents of a specific table. You can pass multible tables at once. See examples.

.PARAMETERS
   -Table <String[]>
    Gives to the cmdlet a string value that represents a table name.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

.OUTPUTS
   None

.NOTES

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Clear-TableContentsFromDatabase -Table EVENTS

.EXAMPLE

   -------------------------- EXAMPLE 2 --------------------------

   PS C:\> Clear-TableContentsFromDatabase -Table EVENTS, DETAILS4624

#>
function Clear-TableContentsFromDatabase
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [String[]]$Table
```

```
    )
    Process
    {
        foreach ($ta in $Table) {
            $query = "DELETE FROM $ta"
            Write-Verbose "Query will be '$query'"
            Write-Verbose "Deleted Records FOR '$ta' Table:"
            Invoke-LogDatabaseQuery -connection $LogConnectionString `
                                    -isSQLServer `
                                    -query $query

            Set-TableAutoIncrementValue -Table: $ta
        }
    }
}


<#
.NAME
   Get-LogonType

.SYNOPSIS
   Gets the explanation of a logon type.

.SYNTAX
   Get-LogonType [[-LogonType] <Int32>]

.DESCRIPTION
   For some security events there is a logon type within the message of the event.
   This logon type is represented by a number. This cmdlet takes as a parameter
   the Logon Type as an integer and returns a string with what this integer means.

.PARAMETERS
   -LogonType <Int32>
    Gives to the cmdlet an integer value that represents a logon type.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

   You cannot pipe input to this cmdlet.

.OUTPUTS
   [System.String]

.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Get-LogonType -LogonType 8

#>
function Get-LogonType
{
    [CmdletBinding()]
    [OutputType([String])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [int]$LogonType
    )
    Process
    {
        switch ($LogonType){
            2{"2: Interactive (logon at keyboard and screen of system)";break}
            3{"3: Network (i.e. connection to shared folder on this computer from elsewhere on
network)";break}
            4{"4: Batch (i.e. scheduled task)";break}
            5{"5: Service (Service startup)";break}
            7{"7: Unlock (i.e. unnattended workstation with password protected screen
saver)";break}
            8{"8: NetworkCleartext (Logon with credentials sent in the clear text. Most often
indicates a logon to IIS with basic authentication)";break}
            9{"9: NewCredentials such as with RunAs or mapping a network drive with alternate
credentials.  This logon type does not seem to show up in any events.";break}
            10{"10: RemoteInteractive (Terminal Services, Remote Desktop or Remote
Assistance)";break}
            11{"11: CachedInteractive (logon with cached domain credentials such as when logging
on to a laptop when away from the network)";break}
            default {"Logon Type could not be determined."}
        }
    }
}
```

```
<#
.NAME
   Get-ImpersonationLevelExplanation

.SYNOPSIS
   Gets the explanation of an impersonation level.

.SYNTAX
   Get-ImpersonationLevelExplanation [[-ImpersonationLevel] <String>]

.DESCRIPTION
   For security events with id 4624 there is an impersonation level within the message of the
event.
   This impersonation level is represented by a string. This cmdlet takes as a parameter
   the impersonation level as a string and returns a string with what this impersonation level
means.

.PARAMETERS
   -ImpersonationLevel <String>
    Gives to the cmdlet a string value that represents an impersonation level.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

   You cannot pipe input to this cmdlet.

.OUTPUTS
   [System.String]

.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Get-ImpersonationLevelExplanation -ImpersonationLevel Anonymous

#>
function Get-ImpersonationLevelExplanation
{
    [CmdletBinding()]
    [OutputType([String])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [String]$ImpersonationLevel
    )
    Process
    {
        switch ($ImpersonationLevel) {
            "Anonymous"{"Anonymous COM impersonation level that hides the identity of the caller.
Calls to WMI may fail with this impersonation level.";break}
            "Default"{"Default impersonation.";break}
            "Delegate"{"Delegate-level COM impersonation level that allows objects to permit other
objects to use the credentials of the caller. This level, which will work with WMI calls but may
constitute an unnecessary security risk, is supported only under Windows 2000."; break}
            "Identify"{"Identify-level COM impersonation level that allows objects to query the
credentials of the caller. Calls to WMI may fail with this impersonation level."; break}
            "Impersonation"{"Impersonate-level COM impersonation level that allows objects to use
the credentials of the caller. This is the recommended impersonation level for WMI calls."; break}
             default {"Impersonation Level could not be determined."}
        }
    }
}


<#
.NAME
   Get-StatusExplanation

.SYNOPSIS
   Gets the explanation of an event status.

.SYNTAX
   Get-StatusExplanation [[-Status] <String>]

.DESCRIPTION
   For security events with id 4625 there is an status within the message of the event.
   This status is represented by a hexadecimal number. This cmdlet takes as a parameter
   the status as a string and returns a string with what this status means.

.PARAMETERS
   -Status <String>
    Gives to the cmdlet a string value that represents an event status.
```

```
    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

   You cannot pipe input to this cmdlet.

.OUTPUTS
   [System.String]

.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Get-StatusExplanation -Status 0xC0000064

#>
function Get-StatusExplanation
{
    [CmdletBinding()]
    [OutputType([String])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [string]$Status
    )
    Process
    {
        switch ($Status){
            "0xC0000064"{"User name does not exist.";break}
            "0xC000006A"{"User name is correct but the password is wrong.";break}
            "0xC0000234"{"User is currently locked out.";break}
            "0xC0000072"{"Account is currently disabled.";break}
            "0xC000006F"{"User tried to logon outside his day of week or time of day
restrictions.";break}
            "0xC0000070"{"Workstation restriction.";break}
            "0xC0000193"{"Account expiration.";break}
            "0xC0000071"{"Expired password.";break}
            "0xC0000133"{"Clocks between DC and other computer too far out of sync.";break}
            "0xC0000224"{"User is required to change password at next logon.";break}
            "0xC0000225"{"Evidently a bug in Windows and not a risk.";break}
            "0xc000015b"{"The user has not been granted the requested logon type (aka logon right)
at this machine.";break}
            default{$Status}
        }
    }
}


<#
.NAME
   Get-DatesUntilNow

.SYNOPSIS
   Gets an array of datetime object from a specified date until now.

.SYNTAX
   Get-DatesUntilNow [-DateTime <DateTime>] [-Reverse <switch>]

.DESCRIPTION
   This cmdlet is been created in order to work with Get-TimeRangesForNames and Get-
TimeRangesForValues.

.PARAMETERS
   -DateTime <DateTime>
    Gives to the cmdlet a datetime value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

   -Reverse <switch>
    Determines if the array will be reversed or not.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None
```

```
   You cannot pipe input to this cmdlet.

.OUTPUTS
   [System.Collections.ArrayList]
    with DateTime objects

.NOTES

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Get-DatesUntilNow -DateTime (Get-Date).AddDays(-20)

   This example will output the following:

   Tuesday, June 30, 2015 4:50:22 PM
   Monday, June 29, 2015 4:50:22 PM
   Sunday, June 28, 2015 4:50:22 PM
   Saturday, June 27, 2015 4:50:22 PM
   Friday, June 26, 2015 4:50:22 PM
   Thursday, June 25, 2015 4:50:22 PM
   Wednesday, June 24, 2015 4:50:22 PM
   Tuesday, June 23, 2015 4:50:22 PM
   Monday, June 22, 2015 4:50:22 PM
   Sunday, June 21, 2015 4:50:22 PM
   Saturday, June 20, 2015 4:50:22 PM
   Friday, June 19, 2015 4:50:22 PM
   Thursday, June 18, 2015 4:50:22 PM
   Wednesday, June 17, 2015 4:50:22 PM
   Tuesday, June 16, 2015 4:50:22 PM
   Monday, June 15, 2015 4:50:22 PM
   Sunday, June 14, 2015 4:50:22 PM
   Saturday, June 13, 2015 4:50:22 PM
   Friday, June 12, 2015 4:50:22 PM
   Thursday, June 11, 2015 4:50:22 PM
   Wednesday, June 10, 2015 4:50:22 PM

.EXAMPLE

   -------------------------- EXAMPLE 2 --------------------------

   PS C:\> [System.DateTime]$date = "06/01/2015"
   PS C:\> Get-DatesUntilNow -DateTime $date -Reverse

   This example will output the following:

   Monday, June 1, 2015 4:48:43 PM
   Tuesday, June 2, 2015 4:48:43 PM
   Wednesday, June 3, 2015 4:48:43 PM
   Thursday, June 4, 2015 4:48:43 PM
   Friday, June 5, 2015 4:48:43 PM
   Saturday, June 6, 2015 4:48:43 PM
   Sunday, June 7, 2015 4:48:43 PM
   Monday, June 8, 2015 4:48:43 PM
   Tuesday, June 9, 2015 4:48:43 PM
   Wednesday, June 10, 2015 4:48:43 PM
   Thursday, June 11, 2015 4:48:43 PM
   Friday, June 12, 2015 4:48:43 PM
   Saturday, June 13, 2015 4:48:43 PM
   Sunday, June 14, 2015 4:48:43 PM
   Monday, June 15, 2015 4:48:43 PM
   Tuesday, June 16, 2015 4:48:43 PM
   Wednesday, June 17, 2015 4:48:43 PM
   Thursday, June 18, 2015 4:48:43 PM
   Friday, June 19, 2015 4:48:43 PM
   Saturday, June 20, 2015 4:48:43 PM
   Sunday, June 21, 2015 4:48:43 PM
   Monday, June 22, 2015 4:48:43 PM
   Tuesday, June 23, 2015 4:48:43 PM
   Wednesday, June 24, 2015 4:48:43 PM
   Thursday, June 25, 2015 4:48:43 PM
   Friday, June 26, 2015 4:48:43 PM
   Saturday, June 27, 2015 4:48:43 PM
   Sunday, June 28, 2015 4:48:43 PM
   Monday, June 29, 2015 4:48:43 PM
   Tuesday, June 30, 2015 4:48:43

#>
function Get-DatesUntilNow
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [System.DateTime]$DateTime,
        [switch]$Reverse
    )
    Begin
    {
        $DatesArray = New-Object System.Collections.ArrayList
```

```
        $co = 0
    }
    Process
    {
        $timeSpan = (New-TimeSpan -Start (Get-Date) -End $DateTime).Days*-1

        for ($in =0; $in -le $timeSpan; $in++ ) {
            $forDay = (Get-Date).AddDays(-$in)
            $ArrayListAddition = $DatesArray.Add($forDay)
            $co = $co-1
        }

        if ($Reverse){
            $DatesArray.Reverse()
        }
    }
    End
    {
        return $DatesArray
    }
}


<#
.NAME
   Get-TimeRangesForNames

.SYNOPSIS
   Gets an array of time ranges until the current date.

.SYNTAX
   Get-TimeRangesForNames [-DateTime <DateTime>]

.DESCRIPTION
   This cmdlet is used by the Get-HashTableForTimeLineChart cmdlet.
   Charts in Windows Forms can have as data source a hash table.
   Every record on a hash table basically consists of name and value.
   In order to make time line charts we had to specify names and values.
   With this way Get-TimeRangesForNames will provide the names that our hash table will contain.

   In addition, Get-TimeRangesForNames works as follows:
    - If we want to get time ranges from a date that abstains from the current date less than 7
days:
      time ranges will be hourly separated (see example 1)
    - If we want to get time ranges from a date that abstains from the current date more than 7
days but less than 30 days:
      time ranges will be daily separated (see example 2)
    - If we want to get time ranges from a date that abstains from the current date more than 30
days:
      time ranges will be weekly separated (see example 3)

.PARAMETERS
   -DateTime <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false


.INPUTS
   None

.OUTPUTS
   [System.Collections.ArrayList]
    with string value objects
.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 -------------------------

   PS C:\> [System.DateTime]$date = "06/29/2015"
   PS C:\> Get-TimeRangesForNames -DateTime $date

   This example will output the following:

   29_Jun_00-01
   29_Jun_01-02
   29_Jun_02-03
   29_Jun_03-04
   29_Jun_04-05
   29_Jun_05-06
   29_Jun_06-07
   29_Jun_07-08
   29_Jun_08-09
   29_Jun_09-10
   29_Jun_10-11
   29_Jun_11-12
   29_Jun_12-13
   29_Jun_13-14
```

```
    29_Jun_14-15
    29_Jun_15-16
    29_Jun_16-17
    29_Jun_17-18
    29_Jun_18-19
    29_Jun_19-20
    29_Jun_20-21
    29_Jun_21-22
    29_Jun_22-23
    29_Jun_23-00
    30_Jun_00-01
    30_Jun_01-02
    30_Jun_02-03
    30_Jun_03-04
    30_Jun_04-05
    30_Jun_05-06
    30_Jun_06-07
    30_Jun_07-08
    30_Jun_08-09
    30_Jun_09-10
    30_Jun_10-11
    30_Jun_11-12
    30_Jun_12-13
    30_Jun_13-14
    30_Jun_14-15
    30_Jun_15-16
    30_Jun_16-17
    30_Jun_17-18
    30_Jun_18-19
    30_Jun_19-20
    30_Jun_20-21
    30_Jun_21-22
    30_Jun_22-23
    30_Jun_23-00

.EXAMPLE

    -------------------------- EXAMPLE 2 --------------------------

    PS C:\> [System.DateTime]$date = "06/01/2015"
    PS C:\> Get-TimeRangesForNames -DateTime $date

    This example will output the following:

    01_Jun
    02_Jun
    03_Jun
    04_Jun
    05_Jun
    06_Jun
    07_Jun
    08_Jun
    09_Jun
    10_Jun
    11_Jun
    12_Jun
    13_Jun
    14_Jun
    15_Jun
    16_Jun
    17_Jun
    18_Jun
    19_Jun
    20_Jun
    21_Jun
    22_Jun
    23_Jun
    24_Jun
    25_Jun
    26_Jun
    27_Jun
    28_Jun
    29_Jun
    30_Jun

.EXAMPLE

    -------------------------- EXAMPLE 3 --------------------------

    PS C:\> [System.DateTime]$date = "05/01/2015"
    PS C:\> Get-TimeRangesForNames -DateTime $date

    This example will output the following:

    01_May-02_May
    03_May-09_May
    10_May-16_May
    17_May-23_May
    24_May-30_May
    31_May-06_Jun
    07_Jun-13_Jun
    14_Jun-20_Jun
    21_Jun-27_Jun
    28_Jun-04_Jul
```

```
#>
function Get-TimeRangesForNames
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [string]$DateTime
    )

     Begin
    {
        $namesArray = New-Object System.Collections.ArrayList
        # converts the string value came from the parameter to a datetime object
        $DateToWorkWith = [System.DateTime]$DateTime
        # by creating a timespan object we can find how many days the datetime that came as input
abstains from the current date
        $timeSpan = New-TimeSpan -Start $DateToWorkWith -End (get-date)
    }
    Process
    {
        # if the received date abstains from the current date less that 7 days (number 7)
        # hour time ranges will be constructed
        if ($timeSpan.Days -lt 7){
            # for each day makes ranges of hour
            for ($i = 0 ; ($i -le $timeSpan.Days); $i++){
                for ($j = 0; $j -le 23; $j++){
                    switch ($j){
                    0{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_00-01";break}
                    1{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_01-02";break}
                    2{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_02-03";break}
                    3{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_03-04";break}
                    4{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_04-05";break}
                    5{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_05-06";break}
                    6{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_06-07";break}
                    7{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_07-08";break}
                    8{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_08-09";break}
                    9{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_09-10";break}
                    10{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_10-11";break}
                    11{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_11-12";break}
                    12{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_12-13";break}
                    13{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_13-14";break}
                    14{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_14-15";break}
                    15{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_15-16";break}
                    16{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_16-17";break}
                    17{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_17-18";break}
                    18{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_18-19";break}
                    19{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_19-20";break}
                    20{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_20-21";break}
                    21{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_21-22";break}
                    22{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_22-23";break}
                    23{ $temp = ($DateToWorkWith.AddDays($i).ToString("dd_MMM"))+"_23-00";break}
                    }
                    $addition = $namesArray.Add($temp)
                }
            }

        # if the received date abstains from the current date more that 7 but less than 30 days
        # day time ranges will be constructed
        } elseif ($timeSpan.Days -gt 7 -and $timeSpan.Days -le 30){
            for ($i=0; $DateToWorkWith.AddDays($i).date -le (Get-Date).date; $i++){
                $temp = $DateToWorkWith.AddDays($i).ToString("dd_MMM")
                $addition = $namesArray.Add($temp)
            }

        # if the received date abstains from the current date more that 30 days
        # week time ranges will be constructed
        } elseif ($timeSpan.Days -gt 30){
            if ($DateToWorkWith.DayOfWeek.value__ -ne 0){
                switch ($DateToWorkWith.DayOfWeek.value__){
                1 {
                    $temp = $DateToWorkWith.ToString("dd_MMM")+"-
"+$DateToWorkWith.AddDays(5).ToString("dd_MMM")
                    $next = $DateToWorkWith.AddDays(6)
                ;break}
                2 {
                    $temp = $DateToWorkWith.ToString("dd_MMM")+"-
"+$DateToWorkWith.AddDays(4).ToString("dd_MMM")
                    $next = $DateToWorkWith.AddDays(5)
                ;break}
                3 {
                    $temp = $DateToWorkWith.ToString("dd_MMM")+"-
"+$DateToWorkWith.AddDays(3).ToString("dd_MMM")
                    $next = $DateToWorkWith.AddDays(4)
                ;break}
                4 {
                    $temp = $DateToWorkWith.ToString("dd_MMM")+"-
"+$DateToWorkWith.AddDays(2).ToString("dd_MMM")
                    $next = $DateToWorkWith.AddDays(3)
                ;break}
```

```
                5 {
                    $temp = $DateToWorkWith.ToString("dd_MMM")+"-
"+$DateToWorkWith.AddDays(1).ToString("dd_MMM")
                    $next = $DateToWorkWith.AddDays(2)
                ;break}
                6 {
                    $temp = $DateToWorkWith.ToString("dd_MMM")+"-
"+$DateToWorkWith.AddDays(5).ToString("dd_MMM")
                    $next = $DateToWorkWith.AddDays(1)
                ;break}

                }
                $addition = $namesArray.Add($temp)
            } else {
                $next = $DateToWorkWith
            }

            [System.Collections.ArrayList]$dates = (Get-DatesUntilNow -DateTime $next)
            $dates.reverse()

            foreach ($date in $dates){

                if ($date.DayOfWeek.value__ -eq 0){
                    $temp = $date.ToString("dd_MMM")+"-"+$date.AddDays(6).ToString("dd_MMM")
                    $addition = $namesArray.Add($temp)
                }
            }
        }
    }
    End
    {
        # finally it outputs a System.Collections.ArrayList
        Write-Output $namesArray
    }

}


<#
.NAME
   Get-TimeRangesForValues

.SYNOPSIS
   Gets an array of time ranges until the current date to be used as SQL Queries.

.SYNTAX
   Get-TimeRangesForValues [-DateTime <DateTime>]

.DESCRIPTION
   This cmdlet is used by the Get-HashTableForTimeLineChart cmdlet.
   Charts in Windows Forms can have as data source a hash table.
   Every record on a hash table basically consists of name and value.
   In order to make time line charts we had to specify names and values.
   With this way Get-TimeRangesForValues will provide the values that our hash table will contain.

   In addition, Get-TimeRangesForValues works as follows:
    - If we want to get time ranges from a date that abstains from the current date less than 7
days:
      time ranges will be hourly separated (see example 1)
    - If we want to get time ranges from a date that abstains from the current date more than 7
days but less than 30 days:
      time ranges will be daily separated (see example 2)
    - If we want to get time ranges from a date that abstains from the current date more than 30
days:
      time ranges will be weekly separated (see example 3)

.PARAMETERS
   -DateTime <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false


.INPUTS
   None

.OUTPUTS
   [System.Collections.ArrayList]
    with string value objects
.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> [System.DateTime]$date = "06/29/2015"
   PS C:\> Get-TimeRangesForValues -DateTime $date

   This example will output the following:
```

```
    '06/29/2015 00:00:00' AND '06/29/2015 00:59:59'
    '06/29/2015 01:00:00' AND '06/29/2015 01:59:59'
    '06/29/2015 02:00:00' AND '06/29/2015 02:59:59'
    '06/29/2015 03:00:00' AND '06/29/2015 03:59:59'
    '06/29/2015 04:00:00' AND '06/29/2015 04:59:59'
    '06/29/2015 05:00:00' AND '06/29/2015 05:59:59'
    '06/29/2015 06:00:00' AND '06/29/2015 06:59:59'
    '06/29/2015 07:00:00' AND '06/29/2015 07:59:59'
    '06/29/2015 08:00:00' AND '06/29/2015 08:59:59'
    '06/29/2015 09:00:00' AND '06/29/2015 09:59:59'
    '06/29/2015 10:00:00' AND '06/29/2015 10:59:59'
    '06/29/2015 11:00:00' AND '06/29/2015 11:59:59'
    '06/29/2015 12:00:00' AND '06/29/2015 12:59:59'
    '06/29/2015 13:00:00' AND '06/29/2015 13:59:59'
    '06/29/2015 14:00:00' AND '06/29/2015 14:59:59'
    '06/29/2015 15:00:00' AND '06/29/2015 15:59:59'
    '06/29/2015 16:00:00' AND '06/29/2015 16:59:59'
    '06/29/2015 17:00:00' AND '06/29/2015 17:59:59'
    '06/29/2015 18:00:00' AND '06/29/2015 18:59:59'
    '06/29/2015 19:00:00' AND '06/29/2015 19:59:59'
    '06/29/2015 20:00:00' AND '06/29/2015 20:59:59'
    '06/29/2015 21:00:00' AND '06/29/2015 21:59:59'
    '06/29/2015 22:00:00' AND '06/29/2015 22:59:59'
    '06/29/2015 23:00:00' AND '06/29/2015 23:59:59'
    '06/30/2015 00:00:00' AND '06/30/2015 00:59:59'
    '06/30/2015 01:00:00' AND '06/30/2015 01:59:59'
    '06/30/2015 02:00:00' AND '06/30/2015 02:59:59'
    '06/30/2015 03:00:00' AND '06/30/2015 03:59:59'
    '06/30/2015 04:00:00' AND '06/30/2015 04:59:59'
    '06/30/2015 05:00:00' AND '06/30/2015 05:59:59'
    '06/30/2015 06:00:00' AND '06/30/2015 06:59:59'
    '06/30/2015 07:00:00' AND '06/30/2015 07:59:59'
    '06/30/2015 08:00:00' AND '06/30/2015 08:59:59'
    '06/30/2015 09:00:00' AND '06/30/2015 09:59:59'
    '06/30/2015 10:00:00' AND '06/30/2015 10:59:59'
    '06/30/2015 11:00:00' AND '06/30/2015 11:59:59'
    '06/30/2015 12:00:00' AND '06/30/2015 12:59:59'
    '06/30/2015 13:00:00' AND '06/30/2015 13:59:59'
    '06/30/2015 14:00:00' AND '06/30/2015 14:59:59'
    '06/30/2015 15:00:00' AND '06/30/2015 15:59:59'
    '06/30/2015 16:00:00' AND '06/30/2015 16:59:59'
    '06/30/2015 17:00:00' AND '06/30/2015 17:59:59'
    '06/30/2015 18:00:00' AND '06/30/2015 18:59:59'
    '06/30/2015 19:00:00' AND '06/30/2015 19:59:59'
    '06/30/2015 20:00:00' AND '06/30/2015 20:59:59'
    '06/30/2015 21:00:00' AND '06/30/2015 21:59:59'
    '06/30/2015 22:00:00' AND '06/30/2015 22:59:59'
    '06/30/2015 23:00:00' AND '06/30/2015 23:59:59'

.EXAMPLE

    -------------------------- EXAMPLE 2 --------------------------

    PS C:\> [System.DateTime]$date = "06/01/2015"
    PS C:\> Get-TimeRangesForValues -DateTime $date

    This example will output the following:

    '06/01/2015 00:00:00' AND '06/01/2015 23:59:59'
    '06/02/2015 00:00:00' AND '06/02/2015 23:59:59'
    '06/03/2015 00:00:00' AND '06/03/2015 23:59:59'
    '06/04/2015 00:00:00' AND '06/04/2015 23:59:59'
    '06/05/2015 00:00:00' AND '06/05/2015 23:59:59'
    '06/06/2015 00:00:00' AND '06/06/2015 23:59:59'
    '06/07/2015 00:00:00' AND '06/07/2015 23:59:59'
    '06/08/2015 00:00:00' AND '06/08/2015 23:59:59'
    '06/09/2015 00:00:00' AND '06/09/2015 23:59:59'
    '06/10/2015 00:00:00' AND '06/10/2015 23:59:59'
    '06/11/2015 00:00:00' AND '06/11/2015 23:59:59'
    '06/12/2015 00:00:00' AND '06/12/2015 23:59:59'
    '06/13/2015 00:00:00' AND '06/13/2015 23:59:59'
    '06/14/2015 00:00:00' AND '06/14/2015 23:59:59'
    '06/15/2015 00:00:00' AND '06/15/2015 23:59:59'
    '06/16/2015 00:00:00' AND '06/16/2015 23:59:59'
    '06/17/2015 00:00:00' AND '06/17/2015 23:59:59'
    '06/18/2015 00:00:00' AND '06/18/2015 23:59:59'
    '06/19/2015 00:00:00' AND '06/19/2015 23:59:59'
    '06/20/2015 00:00:00' AND '06/20/2015 23:59:59'
    '06/21/2015 00:00:00' AND '06/21/2015 23:59:59'
    '06/22/2015 00:00:00' AND '06/22/2015 23:59:59'
    '06/23/2015 00:00:00' AND '06/23/2015 23:59:59'
    '06/24/2015 00:00:00' AND '06/24/2015 23:59:59'
    '06/25/2015 00:00:00' AND '06/25/2015 23:59:59'
    '06/26/2015 00:00:00' AND '06/26/2015 23:59:59'
    '06/27/2015 00:00:00' AND '06/27/2015 23:59:59'
    '06/28/2015 00:00:00' AND '06/28/2015 23:59:59'
    '06/29/2015 00:00:00' AND '06/29/2015 23:59:59'
    '06/30/2015 00:00:00' AND '06/30/2015 23:59:59'

.EXAMPLE

    -------------------------- EXAMPLE 3 --------------------------
```

```
    PS C:\> [System.DateTime]$date = "05/01/2015"
    PS C:\> Get-TimeRangesForValues -DateTime $date

    This example will output the following:

    '05/01/2015 00:00:00' AND '05/02/2015 23:59:59'
    '05/03/2015 00:00:00' AND '05/09/2015 23:59:59'
    '05/10/2015 00:00:00' AND '05/16/2015 23:59:59'
    '05/17/2015 00:00:00' AND '05/23/2015 23:59:59'
    '05/24/2015 00:00:00' AND '05/30/2015 23:59:59'
    '05/31/2015 00:00:00' AND '06/06/2015 23:59:59'
    '06/07/2015 00:00:00' AND '06/13/2015 23:59:59'
    '06/14/2015 00:00:00' AND '06/20/2015 23:59:59'
    '06/21/2015 00:00:00' AND '06/27/2015 23:59:59'
    '06/28/2015 00:00:00' AND '07/04/2015 23:59:59'

#>
function Get-TimeRangesForValues
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [string]$DateTime
    )

    Begin
    {
        $datesArray = New-Object System.Collections.ArrayList
        # converts the string value came from the parameter to a datetime object
        $DateToWorkWith = [System.DateTime]$DateTime
        # by creating a timespan object we can find how many days the datetime that came as input
abstains from the current date
        $timeSpan = New-TimeSpan -Start $DateToWorkWith -End (get-date)
    }
    Process
    {
        # if the received date abstains from the current date less that 7 days (number 7)
        # hour time ranges will be constructed
        if ($timeSpan.Days -lt 7){
            # for each day makes ranges of hour
            # ready to be used as sql queries
            for ($i = 0 ; ($i -le $timeSpan.Days); $i++){
                for ($j = 0; $j -le 23; $j++){
                    switch ($j){
                    0{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
00:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 00:59:59")+"'";break}
                    1{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
01:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 01:59:59")+"'";break}
                    2{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
02:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 02:59:59")+"'";break}
                    3{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
03:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 03:59:59")+"'";break}
                    4{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
04:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 04:59:59")+"'";break}
                    5{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
05:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 05:59:59")+"'";break}
                    6{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
06:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 06:59:59")+"'";break}
                    7{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
07:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 07:59:59")+"'";break}
                    8{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
08:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 08:59:59")+"'";break}
                    9{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
09:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 09:59:59")+"'";break}
                    10{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
10:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 10:59:59")+"'";break}
                    11{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
11:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 11:59:59")+"'";break}
                    12{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
12:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 12:59:59")+"'";break}
                    13{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
13:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 13:59:59")+"'";break}
                    14{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
14:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 14:59:59")+"'";break}
                    15{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
15:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 15:59:59")+"'";break}
                    16{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
16:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 16:59:59")+"'";break}
                    17{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
17:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 17:59:59")+"'";break}
                    18{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
18:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 18:59:59")+"'";break}
                    19{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
19:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 19:59:59")+"'";break}
                    20{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
20:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 20:59:59")+"'";break}
                    21{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
21:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 21:59:59")+"'";break}
                    22{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
22:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 22:59:59")+"'";break}
```

```
                    23{ $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy
23:00:00")+"'"+" AND "+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 23:59:59")+"'";break}
                    }
                    $addition = $datesArray.Add($temp)
                }
            }
        # if the received date abstains from the current date more that 7 but less than 30 days
        # day time ranges will be constructed
        # ready to be used as sql queries
        } elseif ($timeSpan.Days -gt 7 -and $timeSpan.Days -le 30){

             for ($i=0; $DateToWorkWith.AddDays($i).date -le (Get-Date).date; $i++){
                $temp = "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 00:00:00")+"'"+" AND
"+ "'"+$DateToWorkWith.AddDays($i).ToString("MM/dd/yyyy 23:59:59")+"'"
                $addition = $datesArray.Add($temp)
            }
        # if the received date abstains from the current date more that 30 days
        # week time ranges will be constructed
        # ready to be used as sql queries
        } elseif ($timeSpan.Days -gt 30){
            if ($DateToWorkWith.DayOfWeek.value__ -ne 0){
                switch ($DateToWorkWith.DayOfWeek.value__){
                1 {
                    $temp = "'"+$DateToWorkWith.ToString("MM/dd/yyyy 00:00:00")+"'"+" AND "+
"'"+$DateToWorkWith.AddDays(5).ToString("MM/dd/yyyy 23:59:59")+"'"
                    $next = $DateToWorkWith.AddDays(6)
                ;break}
                2 {
                    $temp = "'"+$DateToWorkWith.ToString("MM/dd/yyyy 00:00:00")+"'"+" AND "+
"'"+$DateToWorkWith.AddDays(4).ToString("MM/dd/yyyy 23:59:59")+"'"
                    $next = $DateToWorkWith.AddDays(5)
                ;break}
                3 {
                    $temp = "'"+$DateToWorkWith.ToString("MM/dd/yyyy 00:00:00")+"'"+" AND "+
"'"+$DateToWorkWith.AddDays(3).ToString("MM/dd/yyyy 23:59:59")+"'"
                    $next = $DateToWorkWith.AddDays(4)
                ;break}
                4 {
                    $temp = "'"+$DateToWorkWith.ToString("MM/dd/yyyy 00:00:00")+"'"+" AND "+
"'"+$DateToWorkWith.AddDays(2).ToString("MM/dd/yyyy 23:59:59")+"'"
                    $next = $DateToWorkWith.AddDays(3)
                ;break}
                5 {
                    $temp = "'"+$DateToWorkWith.ToString("MM/dd/yyyy 00:00:00")+"'"+" AND "+
"'"+$DateToWorkWith.AddDays(1).ToString("MM/dd/yyyy 23:59:59")+"'"
                    $next = $DateToWorkWith.AddDays(2)
                ;break}
                6 {
                    $temp = "'"+$DateToWorkWith.ToString("MM/dd/yyyy 00:00:00")+"'"+" AND "+
"'"+$DateToWorkWith.ToString("MM/dd/yyyy 23:59:59")+"'"
                    $next = $DateToWorkWith.AddDays(1)
                ;break}

                }
                $addition = $datesArray.Add($temp)
            } else {
                $next = $DateToWorkWith
            }
            [System.Collections.ArrayList]$dates = (Get-DatesUntilNow -DateTime $next)
            $dates.reverse()
            foreach ($date in $dates){
                if ($date.DayOfWeek.value__ -eq 0){
                    $temp = "'"+$date.ToString("MM/dd/yyyy 00:00:00")+"'"+" AND "+
"'"+$date.AddDays(6).ToString("MM/dd/yyyy 23:59:59")+"'"
                    $addition = $datesArray.Add($temp)
                }
            }
        }
    }
    End
    {
        # finally it outputs a System.Collections.ArrayList
        Write-Output $datesArray
    }
}

## First Group ## END
# cmdlets that are used only within this module
# ================================================================


# ================================================================
## Second Group ## START
# cmdlets that are used only from the script: "LogVisualization.ps1"

<#
.NAME
   Get-TableRowNumber

.SYNOPSIS
   Get a number that represents rows of a table in database.

.SYNTAX
```

```
   Get-TableRowNumber [-Table <String[]>] [-After <String>]

.DESCRIPTION
   This cmdlet is used by the first window of the "LogVisualization.ps1" in order to inform the
user
   about how many event record exist in the database.

.PARAMETERS
   -Table <String[]>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

   -After <String>
    Gives to the cmdlet a string value.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

.OUTPUTS
   An integer value.

.NOTES

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Get-TableRowNumber -Table Events -After "05/01/2015"
   45049

.EXAMPLE

   -------------------------- EXAMPLE 2 --------------------------

   PS C:\> Get-TableRowNumber -Table Events -After "06/26/2015"
   4641

#>
function Get-TableRowNumber
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [String[]]$Table,
        [String]$After
    )
    Process
    {
        foreach ($ta in $Table){
            if ($After -eq 0){
                [int]$number = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                -isSQLServer `
                                -query "SELECT COUNT(*) from $ta").item(0)

                Write-Output $number
            } else {
                [int]$number = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                -isSQLServer `
                                -query "SELECT COUNT(*) AS Count from $ta
                                        WHERE TimeCreated >= '$After'").Count

                Write-Output $number
            }
        }
    }
}


<#
.NAME
   Get-LastEventDateFromDatabase

.SYNOPSIS
   Gets the date of the last event of the database.

.SYNTAX
   Get-LastEventDateFromDatabase [-Table <String>]

.DESCRIPTION
```

```
   This cmdlet makes an sql query to receive an String value.
   This string is the output of the cmdlet and represents the timecreated column value
   of the last record found in the database. In other words it finds the oldest record
   of a table and returns the timecreated column value.

.PARAMETERS
   -Table <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
  None

.OUTPUTS
  A string value.

.NOTES

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Get-LastEventDateFromDatabase events

   This example will output the following:

   01/30/2015 12:49:38

#>
function Get-LastEventDateFromDatabase
{
    [CmdletBinding()]
    #[OutputType([String])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [string]$Table
    )
    Process
    {
        [string]$eve = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                            -isSQLServer
                                            -query "SELECT TOP 1 TimeCreated from
$Table").TimeCreated
        # Write-Output
        $eve
    }
}


<#
.NAME
   Get-EventsOccured

.SYNOPSIS
   Get the number of the events that have been occured.

.SYNTAX
   Get-EventsOccured [-Table <String>] [-After <String>] [-LogName <String>] [-SecurityType
<String>]

.DESCRIPTION
   This cmdlet provides an integer value to be displayed in the EventsOccured TextField of the
LogVisualization script.

.PARAMETERS
   -Table <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

   -After <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

    -LogName <String>
    Gives to the cmdlet a string value.
```

```
    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

    -SecurityType <String>
    Gives to the cmdlet a string value.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

.OUTPUTS
   An integer value.

.NOTES

.EXAMPLE

   -------------------------- EXAMPLE 1 -------------------------

   PS C:\> Get-EventsOccured -Table Events -After "06/25/2015"

   This example will output the following:

   5718

.EXAMPLE

   -------------------------- EXAMPLE 2 -------------------------

   PS C:\> Get-EventsOccured -Table Events -After "06/25/2015" -LogName Application

   This example will output the following:

   238

.EXAMPLE

   -------------------------- EXAMPLE 3 -------------------------

   PS C:\> Get-EventsOccured -Table Events -After "06/25/2015" -LogName Security -SecurityType
Failure

   This example will output the following:

   388

#>
function Get-EventsOccured
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [string]$Table,
        [Parameter(Mandatory=$true)]
        [string]$After,
        [string]$LogName,
        [string]$SecurityType
    )
    Process
    {
        if ($LogName.Equals("")){
            $query = "SELECT COUNT(*) AS Count FROM $Table
                      WHERE TimeCreated >= '$After'"
        } elseif ($LogName -ne ""){
            if ($LogName -ne "Security"){
                $query = "SELECT COUNT(*) AS Count FROM $Table
                          WHERE LogName = '$LogName'
                          AND TimeCreated >= '$After'"
            } elseif ($LogName -eq "Security"){
                if ($SecurityType -eq "Failure"){
                    $query = "SELECT COUNT(*) AS Count FROM $Table
                              WHERE LogName = '$LogName' AND EventId = 4625
                              AND TimeCreated >= '$After'"

                } elseif ($SecurityType -eq "Success"){
                    $query = "SELECT COUNT(*) AS Count FROM $Table
                              WHERE LogName = '$LogName' AND EventId = 4624
                              AND TimeCreated >= '$After'"

                } elseif ($SecurityType -eq ""){
                    $query = "SELECT COUNT(*) AS Count FROM $Table
                              WHERE LogName = '$LogName'
```

```
                                AND TimeCreated >= '$After'"
                    }
                }
            }
        }
    }
    End
    {
        # finally has the query and communicates with the database
        $result = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                       -isSQLServer `
                                       -query $query).Count
        Write-Output $result
    }
}


<#
.NAME
   Get-HashTableForPieChart

.SYNOPSIS
   Gets a hash table to be used for pie chart.

.SYNTAX
   Get-HashTableForPieChart [[-Table <String>] [-After <String>]]

   Get-HashTableForPieChart [[-Table <String>] [-After <String>] [-LogName <String>]]

.DESCRIPTION
   This cmdlet is used from the LogVisualization script in order to display the pie charts.
   It implements simple group by queries to retrieve data from database.

.PARAMETERS
   -Table <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

   -After <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

   -LogName <String>
    Gives to the cmdlet a string value.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

.OUTPUTS
   [System.Collections.Hashtable]

.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Get-HashTableForPieChart -Table EVENTS -After "06/01/2015"

   This example will output the following:

   Name                           Value
   ----                           -----
   Application                    2003
   System                         4646
   Security                       16968


.EXAMPLE

   -------------------------- EXAMPLE 2 --------------------------

   PS C:\> Get-HashTableForPieChart -Table EVENTS -After "06/01/2015" -LogName Security

   This example will output the following:

   Name                           Value
   ----                           -----
```

```
   4722                           1
   4648                           1899
   4647                           13
   4776                           2554
   4902                           9
   4728                           1
   4616                           7
   4672                           3085
   4720                           1
   4624                           3124
   4634                           2568
   4907                           165
   1100                           9
   5024                           9
   4717                           1
   4738                           5
   4732                           2
   4723                           1
   4797                           136
   4608                           9
   4625                           3359
   4724                           1
   5033                           9

#>
function Get-HashTableForPieChart
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$false,
                   Position=0)]
        [string]$Table,
        [Parameter(Mandatory=$true)]
        [string]$After,
        [string]$LogName
    )
    Begin
    {
        [System.Collections.Hashtable]$hashTable = [ordered]@{}
    }
    Process
    {
        # if a logname has not been specified information will be grouped by logname
        if ($LogName.Equals("")){
            $query = "SELECT LogName AS Name, COUNT(*) AS Count FROM $Table
                      WHERE TimeCreated >= '$After'
                      GROUP BY LogName
                      ORDER BY Count DESC"
        # if a logname has been specified information will be grouped by eventid
        } elseif ($LogName -ne ""){
            $query = "SELECT EventId AS Name, COUNT(*) AS Count FROM $Table
                          WHERE LogName = '$LogName'
                          AND TimeCreated >= '$After'
                          GROUP BY EventId
                          ORDER BY Count DESC"
        }
    }
    End
    {
        # get the query and now communicates with the database
        $result = Get-LogDatabaseData -connectionString $LogConnectionString `
                                  -isSQLServer `
                                  -query $query

        foreach ($res in $result){
            $hashTable.Add(($res.Name).tostring(),$res.Count)
        }
        Write-Output $hashTable
    }
}


<#
.NAME
   Get-HashTableForTimeLineChart

.SYNOPSIS
   Gets a hash table to be used for timeline chart.

.SYNTAX
   Get-HashTableForPieChart [[-Table <String>] [-After <String>]]

   Get-HashTableForPieChart [[-Table <String>] [-After <String>] [-LogName <String>]]

   Get-HashTableForPieChart [[-Table <String>] [-After <String>] [-LogName <String>] [-
SecurityType <String>]]

.DESCRIPTION
   This cmdlet is used from the LogVisualization script in order to display the timeline charts.
   It implements simple group by queries to retrieve data from database.

.PARAMETERS
```

```
    -Table <String>
     Gives to the cmdlet a string value.

     Required?                    true
     Position?                    1
     Default value
     Accept pipeline input?       true
     Accept wildcard characters?  false

    -After <String>
     Gives to the cmdlet a string value.

     Required?                    true
     Position?                    1
     Default value
     Accept pipeline input?       true
     Accept wildcard characters?  false

    -LogName <String>
     Gives to the cmdlet a string value.

     Required?                    false
     Position?                    1
     Default value
     Accept pipeline input?       true
     Accept wildcard characters?  false

    -SecurityType <String>
     Gives to the cmdlet a string value.

     Required?                    false
     Position?                    1
     Default value
     Accept pipeline input?       true
     Accept wildcard characters?  false

.INPUTS
    None

.OUTPUTS
    [System.Collections.Hashtable]

.NOTES
    None

.EXAMPLE

    -------------------------- EXAMPLE 1 --------------------------

    PS C:\> Get-HashTableForTimeLineChart -Table EVENTS -After "06/01/2015"

    This example will output the following:

    Name                           Value
    ----                           -----
    01_Jun                         285
    02_Jun                         356
    03_Jun                         131
    04_Jun                         229
    05_Jun                         659
    06_Jun                         662
    07_Jun                         550
    08_Jun                         265
    09_Jun                         84
    10_Jun                         315
    11_Jun                         662
    12_Jun                         208
    13_Jun                         520
    14_Jun                         953
    15_Jun                         1111
    16_Jun                         1127
    17_Jun                         1075
    18_Jun                         1038
    19_Jun                         536
    20_Jun                         1032
    21_Jun                         1505
    22_Jun                         998
    23_Jun                         1139
    24_Jun                         1926
    25_Jun                         1071
    26_Jun                         1068
    27_Jun                         1036
    28_Jun                         839
    29_Jun                         900
    30_Jun                         1059
    01_Jul                         283


.EXAMPLE

    -------------------------- EXAMPLE 2 --------------------------

    PS C:\> Get-HashTableForTimeLineChart -Table EVENTS -After "02/01/2015" -LogName Security -
SecurityType Failure
```

```
   This example will output the following:

   Name                           Value
   ----                           -----
   01_Feb-07_Feb                  0
   08_Feb-14_Feb                  0
   15_Feb-21_Feb                  0
   22_Feb-28_Feb                  0
   01_Mar-07_Mar                  0
   08_Mar-14_Mar                  0
   15_Mar-21_Mar                  0
   22_Mar-28_Mar                  0
   29_Mar-04_Apr                  0
   05_Apr-11_Apr                  0
   12_Apr-18_Apr                  0
   19_Apr-25_Apr                  177
   26_Apr-02_May                  1501
   03_May-09_May                  310
   10_May-16_May                  350
   17_May-23_May                  3914
   24_May-30_May                  202
   31_May-06_Jun                  674
   07_Jun-13_Jun                  299
   14_Jun-20_Jun                  1085
   21_Jun-27_Jun                  1220
   28_Jun-04_Jul                  82

#>
function Get-HashTableForTimeLineChart
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$false,
                   ValueFromPipelineByPropertyName=$false,
                   Position=0)]
        [string]$Table,
        [Parameter(Mandatory=$true)]
        [string]$After,
        [string]$LogName,
        [string]$SecurityType
    )
    Begin
    {
        $hashTable = [ordered]@{}
        $DateToWorkWith = [System.DateTime]$After
        $timeSpan = New-TimeSpan -Start $DateToWorkWith -End (get-date)
    }
    Process
    {
         # gets the appropriate time ranges and then simply makes the SQL queries
        $timeRangesForNames = Get-TimeRangesForNames -DateTime $After
        $timeRangesForValues = Get-TimeRangesForValues -DateTime $After

        if ($LogName -eq ""){
            $counter = 0
            foreach ($timeRange in $timeRangesForValues){
                $query = "SELECT COUNT(*) AS Count FROM $Table
                          WHERE (TimeCreated BETWEEN $timeRange)"

                $a = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                          -isSQLServer `
                                          -query $query).Count

                $addition = $hashTable.Add($timeRangesForNames.get($counter),$a)
                $counter++
            }
        } elseif ($LogName -ne ""){
            if ($LogName -ne "Security"){
                $counter = 0
                foreach ($timeRange in $timeRangesForValues){
                    $query = "SELECT COUNT(*) AS Count FROM $Table
                              WHERE LogName = '$LogName'
                              AND (TimeCreated BETWEEN $timeRange)"
                    $a = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                              -isSQLServer `
                                              -query $query).Count
                    $addition = $hashTable.Add($timeRangesForNames.get($counter),$a)
                    $counter++
                }
            } elseif ($LogName -eq "Security"){
                if ($SecurityType -eq ""){
                    $counter = 0
                    foreach ($timeRange in $timeRangesForValues){
                        $query = "SELECT COUNT(*) AS Count FROM $Table
                                  WHERE LogName = '$LogName'
                                  AND (TimeCreated BETWEEN $timeRange)"
                        $a = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                                  -isSQLServer `
                                                  -query $query).Count
                        $addition = $hashTable.Add($timeRangesForNames.get($counter),$a)
                        $counter++
                    }
```

```
                } elseif ($SecurityType -eq "Failure"){
                    $counter = 0
                    foreach ($timeRange in $timeRangesForValues){
                        $query = "SELECT COUNT(*) AS Count FROM $Table
                                  WHERE LogName = '$LogName' AND EventId = 4625
                                  AND (TimeCreated BETWEEN $timeRange)"
                        $a = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                                  -isSQLServer `
                                                  -query $query).Count
                        $addition = $hashTable.Add($timeRangesForNames.get($counter),$a)
                        $counter++
                    }
                } elseif ($SecurityType -eq "Success"){
                    $counter = 0
                    foreach ($timeRange in $timeRangesForValues){
                        $query = "SELECT COUNT(*) AS Count FROM $Table
                                  WHERE LogName = '$LogName' AND EventId = 4624
                                  AND (TimeCreated BETWEEN $timeRange)"
                        $a = (Get-LogDatabaseData -connectionString $LogConnectionString `
                                                  -isSQLServer `
                                                  -query $query).Count
                        $addition = $hashTable.Add($timeRangesForNames.get($counter),$a)
                        $counter++
                    }
                }
            }
        }
     }
    End
    {
        Write-Output $hashTable
    }
}


<#
.NAME
   Get-LogonIpAddresses

.SYNOPSIS
   Gets the ip addresses for some kinds of logon types.

.SYNTAX
   Get-LogonIpAddresses [-LogonType <String>] [-After <String>]

.DESCRIPTION
   This cmdlet is used from the LogVisualization script in order to display the Additional
Information Panel.
   It implements simple group by queries to retrieve data from database.

.PARAMETERS
   -LogonType <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

   -After <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
   None

.OUTPUTS
   [System.Collections.Specialized.OrderedDictionary]
.NOTES

.EXAMPLE

   -------------------------- EXAMPLE 1 --------------------------

   PS C:\> Get-LogonIpAddresses -LogonType Failure -After "06/25/2015"

   This example will output the following:

   Name                           Value
   ----                           -----
   74.7.195.38                    173
   91.218.160.70                  173
   213.132.228.252                47
   103.55.130.162                 30
   83.212.116.67                  6
   -                              2
   64.4.124.174                   2
   66.240.236.119                 1
```

```
   200.105.154.93                  1
   83.14.193.236                   1

#>
function Get-LogonIpAddresses
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [string]$LogonType,
        [Parameter(Mandatory=$true)]
        [string]$After
    )
    Begin
    {
        $hashTable = [ordered]@{}
    }
    Process
    {
        if ($LogonType -eq "Failure"){
            # from table4625 : failure logon events
            $query = "SELECT SourceNetworkAddress, COUNT(*) AS Count FROM DETAILS4625
                      WHERE TimeCreated >= '$After'
                      GROUP BY SourceNetworkAddress
                      ORDER BY Count Desc"
        } elseif($LogonType -eq "Success"){
            # from table4624 : successful logon events
            $query = "SELECT SourceNetworkAddress, COUNT(*) AS Count FROM DETAILS4624
                      WHERE TimeCreated >= '$After'
                      GROUP BY SourceNetworkAddress
                      ORDER BY Count Desc"
        } elseif ($LogonType -eq "Explicit"){
            # from table4648 : successful logon using explicit credentials events
            $query = "SELECT NetworkAddress, COUNT(*) AS Count FROM DETAILS4648
                      WHERE TimeCreated >= '$After'
                      GROUP BY NetworkAddress
                      ORDER BY Count Desc"
        }
    }
    End
    {
        # get the query and now communicates with the database
        $result = Get-LogDatabaseData -connectionString $LogConnectionString `
                                      -isSQLServer `
                                      -query $query

        if ($LogonType -eq "Failure"){
            foreach ($re in $result){
                $hashTable.Add($re.SourceNetworkAddress, $re.Count)
            }
        } elseif($LogonType -eq "Success"){
            foreach ($re in $result){
                $hashTable.Add($re.SourceNetworkAddress, $re.Count)
            }
        } elseif ($LogonType -eq "Explicit"){
            foreach ($re in $result){
                $hashTable.Add($re.NetworkAddress, $re.Count)
            }
        }
        Write-Output $hashTable
    }
}


<#
.NAME
   Get-TableContents

.SYNOPSIS
   Gets contents from database.

.SYNTAX
   Get-TableContents [-query [String]]

.DESCRIPTION
   This cmdlet is used from the LogVisualization script in order to interact with Detection
Actions Panel.
   It implements any query will be retrieved from the user.

.PARAMETERS
   -query <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
```

```
   None

.OUTPUTS
   [Selected.System.Data.DataRow]

.NOTES
   None

#>
function Get-TableContents
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [string]$query
    )

    Process
    {
        $result = Get-LogDatabaseData -connectionString $LogConnectionString `
                                      -isSQLServer `
                                      -query ($query)
    }
    End
    {
        Write-Output ($result.getenumerator() | select *)
    }
}

## Second Group ## END
# cmdlets that are used only from the script: "LogVisualization.ps1"
# ===================================================================


# ===================================================================
## Third Group ## START
# cmdlets that are used only from the script: "ScheduleLogs.ps1"

<#
.NAME
   Get-LastStoredEvent

.SYNOPSIS
   Gets the last stored event from database.

.SYNTAX
   Get-LastStoredEvent [-LogName <String>]

.DESCRIPTION
   This cmdlet is used from the ScheduleLogs script
   in order to get information about the last stored event in database.

.PARAMETERS
   -LogName <String>
    Gives to the cmdlet a string value.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       true
    Accept wildcard characters?  false

.INPUTS
  None

.OUTPUTS
   [System.String[]]

.NOTES
   None

.EXAMPLE

   -------------------------- EXAMPLE 1 -------------------------

   PS C:\> Get-LastStoredEvent -LogName Security

   This example will output the following:

   Security
   57285
   07/01/2015 06:20:00


#>
function Get-LastStoredEvent
{
    [CmdletBinding()]
    [OutputType([String[]])]
```

```
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [String]$LogName
    )
    Process
    {
        $a = Get-LogDatabaseData -connectionString $LogConnectionString `
                                 -isSQLServer `
                                 -query "SELECT TOP 1 * from EVENTS
                                         WHERE LogName = '$LogName'
                                         ORDER BY EventRecordId DESC"

        $b= $a.LogName
        $c = [string]$a.eventrecordid
        $d = $a.timeCreated
        [String[]]$out= $b,$c, $d
        Write-Output $out
    }
}

## Third Group ## END
# cmdlets that are used only from the script: "ScheduleLogs.ps1"
# ===================================================================


Export-ModuleMember -Variable MOLErrorLogPreference
Export-ModuleMember -Function Get-DatabaseAvailableTableNames,
                              Set-LogEventInDatabase,
                              Set-TableAutoIncrementValue,
                              Clear-TableContentsFromDatabase,
                              Get-LogonType,
                              Get-ImpersonationLevelExplanation,
                              Get-StatusExplanation,
                              Get-DatesUntilNow,
                              Get-TimeRangesForNames,
                              Get-TimeRangesForValues,
                              Get-TableRowNumber,
                              Get-LastEventDateFromDatabase,
                              Get-EventsOccured,
                              Get-HashTableForPieChart,
                              Get-HashTableForTimeLineChart,
                              Get-LogonIpAddresses,
                              Get-TableContents,
                              Get-LastStoredEvent
```

5.2.2 Module: Log Database

```
<#
.NAME
   Get-LogDatabaseData

.Synopsis
   Queries information from the database.

.DESCRIPTION
    Get-LogDatabaseData is to be used when you want to query information from the
database.

.PARAMETERS
    -ConnectionString<String>
        Tells PowerShell how to find the database server, what database to connect to,
and how to authenticate.
        You can find more connection string examples at:
"http://connectionstrings.com"

        Required?                    false
        Position?                    named
        Default value                Local computer
        Accept pipeline input?       True (ByPropertyName)
        Accept wildcard characters?  false

    -isSQLServer<Switch>
        Include this switch when your connection string points to a Microsoft SQL
Server.
        Omit this string for all other database server types, and PowerShell will use
        OleDB instead. You'll need to make sure your connection string is OleDB
compatible
        and that you're installed the necessary OleDB drivers to access your databse.
        That can be MySQL, Access, Oracle, or whatever you like.
```

```
    -Query<String>
        This is the actual SQL language query that you want to run.
        This module isn't going to dive into detail on that language; we assume you
know it already.
        If you'd like to learn more about the SQL language, there are numerous books
and videos on the subject.
.NOTES
    Get-LogDatabaseData will retrieve data and place it into the pipeline.
    Within the pipeline, you'get objects with properties that correspond to the
columns of the database.
    We're not going to dive into further detail on how the two database functions:
    (Get-LogDatabaseData & Invoke-LogDatabaseQuery) operate internally.
    These functions internally utilize the .NET Frameworkm and so for this module
they're out of scope.
    The functions do, however, provide a nice wrapper around .NET, so that you can
access databases
    without having to mess around with the raw .NET Framework stuff.

#>
function Get-LogDatabaseData
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        [String]$connectionString,
        [String]$query,
        [switch]$isSQLServer
    )
    if ($isSQLServer) {
        Write-Verbose 'in SQL Server mode'
        $connection = New-Object -TypeName `
            System.Data.SqlClient.SqlConnection
    } else {
        Write-Verbose 'in OleDB mode'
        $connection = New-Object -TypeName `
            System.Data.OleDb.OleDbConnection
    }

    $connection.ConnectionString = $connectionString
    $command = $connection.CreateCommand()
    $command.CommandText = $query

    if ($isSQLServer) {
        $adapter = New-Object -TypeName System.Data.SqlClient.SqlDataAdapter $command
    } else {
        $adapter = New-Object -TypeName System.Data.OleDb.OleDbDataAdapter $command
    }

    $dataset = New-Object -TypeName System.Data.DataSet
    #I put in var a to prevent to return an int value
    $a = $adapter.Fill($dataset)
    $connection.Close()
}

<#
.Name
   Invoke-LogDatabaseQuery

.Synopsis
   Make changes on the database.

.DESCRIPTION
   Invoke-LogDatabaseQuery is for when you want to make changes on the database.
   You can add, remove or change data.

.PARAMETERS
    -ConnectionString<String>
        Tells PowerShell how to find the database server, what database to connect to,
and how to authenticate.
        You can find more connection string examples at:
"http://connectionstrings.com"

        Required?                    false
        Position?                    named
        Default value                Local computer
        Accept pipeline input?       True (ByPropertyName)
        Accept wildcard characters?  false

    -isSQLServer<Switch>
        Include this switch when your connection string points to a Microsoft SQL
Server.
```

```
        Omit this string for all other database server types, and PowerShell will use
        OleDB instead. You'll need to make sure your connection string is OleDB
compatible
        and that you're installed the necessary OleDB drivers to access your databse.
        That can be MySQL, Access, Oracle, or whatever you like.

    -Query<String>
        This is the actual SQL language query that you want to run.
        This module isn't going to dive into detail on that language; we assume you
know it already.
        If you'd like to learn more about the SQL language, there are numerous books
and videos on the subject.

.NOTES
    Invoke-LogDatabaseQuery doesn't write anything to the pipeline; it just runs your
query.
    It also declares support for the -WhatIf and -Confirm parameterns via its
SupportsShouldProcess attribute.

#>
function Invoke-LogDatabaseQuery
{
    [CmdletBinding(SupportsShouldProcess=$true,
                   ConfirmImpact='Low')]
    Param
    (
        [string]$connectionString,
        [string]$query,
        [switch]$isSQLServer
    )
    if ($isSQLServer) {
        Write-Verbose 'in SQL Server mode'
        $connection = New-Object -TypeName System.Data.SqlClient.SqlConnection
    } else {
        Write-Verbose 'in OleDB mode'
        $connection = New-Object -TypeName System.Data.OleDb.OleDbConnection
    }
    $connection.ConnectionString = $connectionString
    $command = $connection.CreateCommand()
    $command.CommandText = $query
    if ($pscmdlet.shouldprocess($query)) {
        $connection.Open()
        Write-Verbose $query
        # ExecuteNonQuery: Executes a Transact-SQL statement against the connection
and returns the number of rows affected.
        $returnValue = $command.ExecuteNonQuery()
        $connection.close()
    }
}
```

5.2.3 Script: Schedule Logs

```
<# ScheduleLogs script runs every ten minutes to ensure that the database is updated
with new events #>
Import-Module LogAnalysis

$array = New-Object System.Collections.ArrayList

<# SYSTEM #>

#here it takes information (logname, eventrecordid, timecreated) of the last record
stored in events table of database
# and stores these information in a string array
[string[]]$lastSystemEvent = Get-LastStoredEvent -LogName System

# it takes index (record id) of last system event stored in database
[int]$lastSystemEventRecordId= $lastSystemEvent.get(1)

# it takes last 50 system events and stores it to eventlogrecord array
[System.Diagnostics.Eventing.Reader.EventLogRecord[]]$SysEvents = Get-WinEvent -
LogName System -MaxEvents 500

# this foreach
foreach ($ev in $SysEvents){
    if($ev.RecordId -ne $lastSystemEventRecordId){
        $a= $array.Add($ev)
    } else {
        break
    }
}
```

```
<# APPLICATION #>

#here it takes information (logname, eventrecordid, timecreated) of the last record
stored in events table of database
# and stores these information in a string array
[string[]]$lastApplicationEvent = Get-LastStoredEvent -LogName Application


# it takes index (or record id) of last system event stored in database
[int]$lastApplicationEventRecordId= $lastApplicationEvent.get(1)


# it takes last 50 application events and stores it to eventlogrecord array
[System.Diagnostics.Eventing.Reader.EventLogRecord[]]$ApEvents = Get-WinEvent -LogName
Application -MaxEvents 500

# this foreach
foreach ($ev in $ApEvents){
    if($ev.RecordId -ne $lastApplicationEventRecordId){
        $a = $array.Add($ev)
    } else {
        break
    }
}


<# SECURITY #>

#here it takes information (logname, eventrecordid, timecreated) of the last record
stored in events table of database
# and stores these information in a string array
[string[]]$lastSecurityEvent = Get-LastStoredEvent -LogName Security


# it takes index (or record id) of last system event stored in database
[int]$lastSecurityEventRecordId= $lastSecurityEvent.get(1)

# it takes last 50 security events and stores it to eventlogrecord array
[System.Diagnostics.Eventing.Reader.EventLogRecord[]]$SecEvents = Get-WinEvent -
LogName Security -MaxEvents 500

# this foreach
foreach ($ev in $SecEvents){
    if($ev.RecordId -ne $lastSecurityEventRecordId){
        $a= $array.Add($ev)
    } else {
        break
    }
}

# finally it has been created an array with eventrecord (we can see this piping the
array to gm)
# we sort this array by TimeCreated property and we send all new events to database
$array | Sort-Object -Property timecreated | Set-LogEventInDatabase
```

5.2.4 Script: Log Scheduler

```
<# This script created in order to create a Scheduled Job for filling the
DataBase every 10 minutes.
   Actually every 10 minutes calls ScheduleLogs.ps1 and this is doing the
job.
   When you will run the script you will be prompt for credential.
   After this you can go to Task Scheduler from Administrative Tools to
check if the job appears there.  #>


# Change these three variables to whatever you want
$jobname = "Automate Log Database Filling"

# Here is where your ScheduleLogs.ps1 script exists.
$script =
"C:\Users\Administrador\Documents\WindowsPowerShell\Modules\LogAnalysis\Sche
duleLogs.ps1"
$repeat = (New-TimeSpan -Minutes 10)
```

```
# The script below will run as the specified user (you will be prompted for
credentials)
# and is set to be elevated to use the highest privileges.
# In addition, the task will run every 10 minutes or however long specified
in $repeat.
$scriptblock = [scriptblock]::Create($script)
#$trigger = New-JobTrigger -AtStartup -RepeatIndefinitely -
RepetitionInterval $repeat
$trigger = New-JobTrigger -Once -At (Get-Date).Date -RepeatIndefinitely -
RepetitionInterval $repeat
$msg = "Enter the username and password that will run the task"
$credential = $Host.UI.PromptForCredential("Task username and
password",$msg,"$env:userdomain\$env:username",$env:userdomain)


$options = New-ScheduledJobOption -RunElevated -ContinueIfGoingOnBattery -
StartIfOnBattery -HideInTaskScheduler
Register-ScheduledJob -Name $jobname -ScriptBlock $scriptblock -Trigger
$trigger -ScheduledJobOption $options -Credential $credential

#after this go to task scheduler to check if the job appears there
```

5.2.5 Script: Log Visualization

```
$LogErrorLogPreference = 'c:\log-retries.txt'
$LogConnectionString =
        "server=localhost\SQLEXPRESS;database=LogDB;trusted_connection=True"

Import-Module LogAnalysis


# load the appropriate assemblies
[void][Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
[void][Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms.DataVisualizati
on")
Add-Type -AssemblyName System.Windows.Forms

function Get-DatesUntilNow
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        [System.DateTime]$DateTime
    )
    Begin
    {
        $DatesArray = New-Object System.Collections.ArrayList
        [int]$co = 0
    }
    Process
    {
        while ((Get-Date).AddDays($co) -ge $DateTime) {
            $forDay = (Get-Date).AddDays($co)
            $ArrayListAddition = $DatesArray.Add($forDay)
            $co = $co-1
        }
        $DatesArray.Reverse()
    }
    End
    {
        Write-Output $DatesArray
    }
}
```

```
function Get-LogVisualization
{
   [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$false,
                   ValueFromPipelineByPropertyName=$false,
                   Position=0)]
        [string]$AfterDate,
        $Type

    )
    Begin
    {
        Add-Type -AssemblyName System.Windows.Forms
        $StartForm = New-Object system.Windows.Forms.Form

        $StartForm.Text = "Preparing events"
        $StartForm.Width = 450
        $StartForm.Height = 180
        $StartForm.MaximizeBox = $False
        $StartForm.StartPosition = 'CenterScreen'
        $StartForm.FormBorderStyle = [System.Windows.Forms.FormBorderStyle]::Fixed3D
        $StartForm.BackColor = [System.Drawing.Color]::WhiteSmoke

        $Status = New-Object System.Windows.Forms.Label
        $Status.Size = '400, 30'
        $Status.Location = '5,20'
        $Status.Text = "Ready?"
        $Status.Font = New-Object System.Drawing.Font("Calibri",15)
        $StartForm.Controls.Add($Status)

        $ProcButton = New-Object System.Windows.Forms.Button
        $ProcButton.Text = "Yes"
        $ProcButton.Font = New-Object System.Drawing.Font("Calibri",15)
        $ProcButton.size = '60,40'
        $ProcButton.Location = '195,85'

        $StartForm.Controls.Add($ProcButton)


        $ProcButton.Add_Click({


            $ProcButton.Enabled = $false

            # proetoimasia dedomenwn gia to textbox EVENTS OCCURED
            $Status.Text = "Preparing events..."
            $StartForm.Refresh()

            $Status.Text = "Preparing all events..."
            $StartForm.Refresh()

            $Status.Text = "Counting events..."
            $StartForm.Refresh()
            $Global:AllDataEventsOccured = Get-EventsOccured -Table EVENTS -After
$AfterDate
            $Global:AppDataEventsOccured = Get-EventsOccured -Table EVENTS -After
$AfterDate -LogName Application
            $Global:SecDataEventsOccured = Get-EventsOccured -Table EVENTS -After
$AfterDate -LogName Security
            $Global:SecSuccDataEventsOccured = Get-EventsOccured -Table EVENTS -After
$AfterDate -LogName Security -SecurityType Success
            $Global:SecFailDataEventsOccured = Get-EventsOccured -Table EVENTS -After
$AfterDate -LogName Security -SecurityType Failure
            $Global:SysDataEventsOccured = Get-EventsOccured -Table EVENTS -After
$AfterDate -LogName System

            $Status.Text = "Preparing pie charts: All Events..."
            $StartForm.Refresh()
            # proetoimasia dedomenwn gia textarea EVENTS GROUP PIES

            $Global:AllDataGroupByLogName = Get-HashTableForPieChart -Table EVENTS -
After $AfterDate
            $Status.Text = "Preparing pie charts: Application..."
            $StartForm.Refresh()
            $Global:AppDataGroupByEventId = Get-HashTableForPieChart -Table EVENTS -
After $AfterDate -LogName Application
            $Status.Text = "Preparing pie charts: Security..."
            $StartForm.Refresh()
            $Global:SecDataGroupByEventId = Get-HashTableForPieChart -Table EVENTS -
After $AfterDate -LogName Security
```

```
            $Status.Text = "Preparing pie charts: System..."
            $StartForm.Refresh()
            $Global:SysDataGroupByEventId = Get-HashTableForPieChart -Table EVENTS -
After $AfterDate -LogName System



            # proetoimasia dedomenwn gia textarea EVENTS GROUP TIMELINES
            $Status.Text = "Preparing time line charts: All Events..."
            $StartForm.Refresh()

            $Global:AllDataTimeLine = Get-HashTableForTimeLineChart -Table EVENTS -
After $AfterDate
            $Status.Text = "Preparing time line charts: Application..."
            $StartForm.Refresh()
            $Global:AppDataTimeLine = Get-HashTableForTimeLineChart -Table EVENTS -
After $AfterDate -LogName Application
            $Status.Text = "Preparing time line charts: Security..."
            $StartForm.Refresh()
            $Global:SecDataTimeLine = Get-HashTableForTimeLineChart -Table EVENTS -
After $AfterDate -LogName Security
            $Status.Text = "Preparing time line charts: Failure Logons..."
            $StartForm.Refresh()
            $Global:SecSuccDataTimeLine = Get-HashTableForTimeLineChart -Table EVENTS
-After $AfterDate -LogName Security -SecurityType Success
            $Status.Text = "Preparing time line charts: Successful Logons..."
            $StartForm.Refresh()
            $Global:SecFailDataTimeLine = Get-HashTableForTimeLineChart -Table EVENTS
-After $AfterDate -LogName Security -SecurityType Failure
            $Status.Text = "Preparing time line charts: System..."
            $StartForm.Refresh()
            $Global:SysDataTimeLine = Get-HashTableForTimeLineChart -Table EVENTS -
After $AfterDate -LogName System

            $Status.Text = "Preparing IP Addresses information: Failure Logons..."
            $StartForm.Refresh()
            $Global:FailureLogonData = Get-LogonIpAddresses -LogonType Failure -After
$AfterDate

            $Status.Text = "Preparing IP Addresses information: Successful Logons..."
            $StartForm.Refresh()
            $Global:SuccessLogonData = Get-LogonIpAddresses -LogonType Success -After
$AfterDate

            $Status.Text = "Preparing IP Addresses information: Successful Logons
using explicit credentials..."
            $StartForm.Refresh()
            $Global:ExplicitLogonData = Get-LogonIpAddresses -LogonType Explicit -
After $AfterDate



            $Status.Text = "Preparing events: Done..."
            $StartForm.Refresh()
            $close = $StartForm.close()

        })

        $show = $StartForm.ShowDialog()
         <# Preparing all information to be ready for visualization #>

    }
    Process
    {
        $Form = New-Object system.Windows.Forms.Form

        $Form.Text = "Intrusion Detection Results"
        $Form.Width = 1200
        $Form.Height = 700
        $Form.MaximizeBox = $False
        $Form.StartPosition = 'CenterScreen'
        $Form.FormBorderStyle = [System.Windows.Forms.FormBorderStyle]::Fixed3D
        $Form.BackColor = [System.Drawing.Color]::CornflowerBlue

        # Set the font of the text to be used within the form
        $Font = New-Object System.Drawing.Font("Calibri",15)

        $Form.Font = $Font

        $panel1 = New-Object System.Windows.Forms.TabPage
        $panel1.Size = '1191,667'
        #$panel1.Location = '40,22'
```

```
$panel1.TabIndex = 0
$panel1.Text = "Data Visualization"
$panel1.BackColor = [System.Drawing.Color]::WhiteSmoke

$panel2 = New-Object System.Windows.Forms.TabPage
$panel2.Size = '1191,667'
#$panel2.Location = '40,22'
$panel2.TabIndex = 1
$panel2.Text = "Additional Information"
$panel2.BackColor = [System.Drawing.Color]::WhiteSmoke

$panel3 = New-Object System.Windows.Forms.TabPage
$panel3.Size = '1191,667'
#$panel2.Location = '40,22'
$panel3.TabIndex = 2
$panel3.Text = "Detection Actions"
$panel3.BackColor = [System.Drawing.Color]::WhiteSmoke

$tab_control = new-object System.Windows.Forms.TabControl
$tab_control.Controls.Add($panel1)
$tab_control.Controls.Add($panel2)
$tab_control.Controls.Add($panel3)
$tab_control.Size = '1191,667'
$tab_control.TabIndex = 0

$Form.Controls.Add($tab_control)

$TypeOfViewGroupBox = New-Object System.Windows.Forms.GroupBox
$TypeOfViewGroupBox.Location = '15,10'
$TypeOfViewGroupBox.Size = '180,160'
$TypeOfViewGroupBox.Text = "Type of view:"

$HistoryRadioButton = New-Object System.Windows.Forms.RadioButton
$HistoryRadioButton.Location = '20,30'
$HistoryRadioButton.Size = '100,30'
$HistoryRadioButton.Text = "History"
#$HistoryRadioButton.Checked = $true
$HistoryRadioButton.Name = "history"

$IntradayRadioButton = New-Object System.Windows.Forms.RadioButton
$IntradayRadioButton.Location = '20,60'
$IntradayRadioButton.Size = '100,30'
$IntradayRadioButton.Text = "Intraday"
$IntradayRadioButton.Name = "intraday"

$CustomRangeRadioButton = New-Object System.Windows.Forms.RadioButton
$CustomRangeRadioButton.Location = '20,90'
$CustomRangeRadioButton.Size = '145,30'
$CustomRangeRadioButton.Text = "Custom Range"
$CustomRangeRadioButton.Name = "custom"

# Add the GroupBox controls
$TypeOfViewGroupBox.Controls.Add($HistoryRadioButton)
$TypeOfViewGroupBox.Controls.Add($IntradayRadioButton)
$TypeOfViewGroupBox.Controls.Add($CustomRangeRadioButton)
$TypeOfViewGroupBox.Enabled = $false

#to TypeOfViewGroupBox topotheteitai sth forma
$panel1.Controls.Add($TypeOfViewGroupBox)

# analoga me to poios typos optikopoihshs irthe epilegei to antistoixo koumpi
# kai thetei ta ypoloipa anenerga
if ($Type.Equals("History")) {
    $HistoryRadioButton.Checked = $true
    $TypeOfViewGroupBox.Enabled = $false
} elseif ($Type.Equals("Intraday")) {
    $IntradayRadioButton.Checked = $true
    $TypeOfViewGroupBox.Enabled = $false
} elseif ($Type.Equals("Custom")) {
    $CustomRangeRadioButton.Checked = $true
    $TypeOfViewGroupBox.Enabled = $false
}


# Create a group that will contain Type of Chart radio buttons
$TypeOfChartGroupBox = New-Object System.Windows.Forms.GroupBox
$TypeOfChartGroupBox.Location = '230,10'
$TypeOfChartGroupBox.Size = '200,120'
$TypeOfChartGroupBox.Text = "Select type of chart:"

# Creating the collection of chart radio buttons
$PieRadioButton = New-Object System.Windows.Forms.RadioButton
```

```
        $PieRadioButton.Location = '40,35'
        $PieRadioButton.Size = '100,30'
        $PieRadioButton.Text = "Pie"
        $PieRadioButton.Checked = $true
        $PieRadioButton.Name = "pie"

        $TimeLineRadioButton = New-Object System.Windows.Forms.RadioButton
        $TimeLineRadioButton.Location = '40,70'
        $TimeLineRadioButton.Size = '100,30'
        $TimeLineRadioButton.Text = "Timeline"
        $TimeLineRadioButton.Name = "timeline"

        # Add the GroupBox controls
        $TypeOfChartGroupBox.Controls.Add($PieRadioButton)
        $TypeOfChartGroupBox.Controls.Add($TimeLineRadioButton)

        #to TypeOfViewGroupBox topotheteitai sth forma
        $panel1.Controls.Add($TypeOfChartGroupBox)

        $LogNameGroupBox = New-Object System.Windows.Forms.GroupBox
        $LogNameGroupBox.Location = '670,10'
        $LogNameGroupBox.Size = '200,135'
        $LogNameGroupBox.Text = "Select LogName:"

        $AllEventsRadioButton = New-Object System.Windows.Forms.RadioButton
        $AllEventsRadioButton.Location = '10,25'
        $AllEventsRadioButton.Size = '130,20'
        $AllEventsRadioButton.Text = "All events"
        $AllEventsRadioButton.Checked = $true
        $AllEventsRadioButton.Name = "allEvents"

        $LogNameGroupBox.Controls.Add($AllEventsRadioButton)


        $ApplicationEventsRadioButton = New-Object System.Windows.Forms.RadioButton
        $ApplicationEventsRadioButton.Location = '10,52'
        $ApplicationEventsRadioButton.Size = '185,23'
        $ApplicationEventsRadioButton.Text = "Application events"
        #$ApplicationEventsRadioButton.Checked = $true
        $ApplicationEventsRadioButton.Name = "appEvents"

        $SecurityEventsRadioButton = New-Object System.Windows.Forms.RadioButton
        $SecurityEventsRadioButton.Location = '10,81'
        $SecurityEventsRadioButton.Size = '185,23'
        $SecurityEventsRadioButton.Text = "Security events"
        #$ApplicationEventsRadioButton.Checked = $true
        $SecurityEventsRadioButton.Name = "secEvents"


        $SystemEventsRadioButton = New-Object System.Windows.Forms.RadioButton
        $SystemEventsRadioButton.Location = '10,109'
        $SystemEventsRadioButton.Size = '185,23'
        $SystemEventsRadioButton.Text = "System events"
        #$ApplicationEventsRadioButton.Checked = $true
        $SystemEventsRadioButton.Name = "sysEvents"

        $LogNameGroupBox.Controls.Add($AllEventsRadioButton)
        $LogNameGroupBox.Controls.Add($ApplicationEventsRadioButton)
        $LogNameGroupBox.Controls.Add($SecurityEventsRadioButton)
        $LogNameGroupBox.Controls.Add($SystemEventsRadioButton)

        $panel1.Controls.Add($LogNameGroupBox)


        # Label for the available machines combobox
        $MachinesLabel = New-Object system.windows.forms.label
        $MachinesLabel.Text = "Select Machine:"
        $MachinesLabel.Size = '140,20'
        $MachinesLabel.Location = '450,20'

        $panel1.Controls.Add($MachinesLabel)

        # Create a comboBox list that will contain Available Machines
        $MachinesComboBox = New-Object System.Windows.Forms.ComboBox
        $MachinesComboBox.Location = '450,50'
        $MachinesComboBox.Size = '200,120'
        #$MachinesComboBox.Text = "Select machine:"
        $MachinesComboBox.DropDownStyle =
[System.Windows.Forms.ComboBoxStyle]::DropDownList


        [String[]]$machines = $env:COMPUTERNAME
```

```
        foreach ($machine in $machines){
            # putting in variable to prevent from output to console
            $m = $MachinesComboBox.Items.Add($machine)
        }

        #to TypeOfViewGroupBox topotheteitai sth forma
        $panel1.Controls.Add($MachinesComboBox)
        # sets 1st value of $machines as the default value of the combobox
        $MachinesComboBox.SelectedIndex = 0
        # Creating the collection of type of chart radio buttons
        #$TypeOfChartGroupBox = New-Object System.Windows.Forms.GroupBox


        # Label for the available security events combobox
        $SecurityEventsLabel = New-Object system.windows.forms.label
        $SecurityEventsLabel.Text = "Select Security Event:"
        $SecurityEventsLabel.Size = '190,30'
        $SecurityEventsLabel.Location = '900,20'

        $panel1.Controls.Add($SecurityEventsLabel)


        # Create a comboBox list that will contain Available Security Events
        $SecEventsComboBox = New-Object System.Windows.Forms.ComboBox
        $SecEventsComboBox.Location = '900,50'
        $SecEventsComboBox.Size = '200,120'
        #$MachinesComboBox.Text = "Select machine:"
        $SecEventsComboBox.DropDownStyle =
[System.Windows.Forms.ComboBoxStyle]::DropDownList
        $SecEventsComboBox.Enabled = $false

        [String[]]$SecEvents = "All Security Events","Logon Failure","Logon Success"

        foreach ($ev in $SecEvents){
            # putting in variable to prevent from output to console
            $m = $SecEventsComboBox.Items.Add($ev)
        }

        #to TypeOfViewGroupBox topotheteitai sth forma
        $panel1.Controls.Add($SecEventsComboBox)
        # sets 1st value of $machines as the default value of the combobox
        $SecEventsComboBox.SelectedIndex = 0

        $SaveChartButton = New-Object System.Windows.Forms.Button
        $SaveChartButton.Location = '981,100'
        $SaveChartButton.Size = '120,30'
        $SaveChartButton.Text = "Save Chart"
        $panel1.Controls.Add($SaveChartButton)


        # Creating the collection of type of chart radio button
        # Label for the available machines combobox
        $EventsOccuredLabel = New-Object system.windows.forms.label
        $EventsOccuredLabel.Text = "Total Event Log
Events Found:"
        $EventsOccuredLabel.Size = '240,60'
        $EventsOccuredLabel.Location = '16,195'
        $EventsOccuredLabel.Font = New-object System.Drawing.Font('Calibri', 18,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        $panel1.Controls.Add($EventsOccuredLabel)


        $EventsOccuredTextBox = New-Object -TypeName 'System.Windows.Forms.TextBox'
        $EventsOccuredTextBox.Size = '100,40'
        $EventsOccuredTextBox.Location = '20,255'
        $EventsOccuredTextBox.ReadOnly = $true

        $EventsOccuredTextBox.Text = $Global:AllDataEventsOccured | Out-String


        $panel1.Controls.Add($EventsOccuredTextBox)

        $EventsGroupTextBox = New-Object System.Windows.Forms.TextBox

         # add textbox
        $EventsGroupTextBox.Multiline = $true
        $EventsGroupTextBox.Height = 300
        $EventsGroupTextBox.Width = 290
        $EventsGroupTextBox.Location ='20,310'
        $EventsGroupTextBox.BorderStyle = [System.Windows.Forms.BorderStyle]::Fixed3D
```

```
        $EventsGroupTextBox.ScrollBars = [System.Windows.Forms.ScrollBars]::Both
        $EventsGroupTextBox.ReadOnly = $true
        $EventsGroupTextBox.Font = New-Object System.Drawing.Font("Times New
Roman",12)

        $EventsGroupTextBox.Text = $Global:AllDataGroupByLogName.GetEnumerator()  |
                                   select Name, Value |
                                   Sort-Object -Property Value -Descending  |
                                   Out-String -Width 40


        #$EventsGroupTextBox.Text = $AllDataGroupByLogName | sort count -Descending |
Out-String

        $panel1.Controls.Add($EventsGroupTextBox)

        $Chart = New-Object System.Windows.Forms.DataVisualization.Charting.Chart
        $Chart.Width = 850
        $Chart.Height = 450
        $Chart.Left = 320
        $Chart.Top = 160

        # create a chartarea to draw on and add to chart
        $ChartArea = New-Object
System.Windows.Forms.DataVisualization.Charting.ChartArea
        $ChartArea.AxisX.Interval = 1
        $ChartArea.AxisY.Title = "Number of events"
        $ChartArea.AxisY.TitleFont =  New-Object System.Drawing.Font("Calibri",15)
        $ChartArea.AxisX.Title = "Date or Time range"
        $ChartArea.AxisX.TitleFont =  New-Object System.Drawing.Font("Calibri",15)
        $Chart.MaximumSize.Width = 20

        #$ChartArea.AxisX.ScrollBar = $true

        $Chart.ChartAreas.Add($ChartArea)


        $panel1.controls.add($Chart)

        [void]$Chart.Series.Add("Data")

        $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
        $Chart.Series["Data"].BorderWidth = 3

        $Chart.Series["Data"].Points.DataBindXY($Global:AllDataGroupByLogName.Keys,
$Global:AllDataGroupByLogName.Values)

        # set chart type
        $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Pie
        #$Form.Activate()

        <# PANEL 2 Components #>


        $FailureLogonLabel = New-Object System.Windows.Forms.Label
        $FailureLogonLabel.Text = "Failure Logons: IP Addresses:"
        $FailureLogonLabel.Size = '150,60'
        $FailureLogonLabel.Location = '60,30'
        $FailureLogonLabel.Font = New-object System.Drawing.Font('Calibri', 16,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        $panel2.Controls.Add($FailureLogonLabel)


        $FailureLogonGroupTextBox = New-Object System.Windows.Forms.TextBox

         # add textbox
        $FailureLogonGroupTextBox.Multiline = $true
        $FailureLogonGroupTextBox.Height = 300
        $FailureLogonGroupTextBox.Width = 300
        $FailureLogonGroupTextBox.Location ='60,100'
        $FailureLogonGroupTextBox.BorderStyle =
[System.Windows.Forms.BorderStyle]::Fixed3D
        $FailureLogonGroupTextBox.ScrollBars = [System.Windows.Forms.ScrollBars]::Both
        $FailureLogonGroupTextBox.ReadOnly = $true
        $FailureLogonGroupTextBox.Font = New-Object System.Drawing.Font("Times New
Roman",16)

        $panel2.Controls.Add($FailureLogonGroupTextBox)
```

```
        $FailureLogonGroupTextBox.Text = $Global:FailureLogonData.getenumerator() |
                                                select Name, Value |
                                                Sort-Object -Property Value -
Descending  |
                                                Out-String -Width 30


        $SuccessfulLogonLabel = New-Object System.Windows.Forms.Label
        $SuccessfulLogonLabel.Text = "Successful Logons: IP Addresses:"
        $SuccessfulLogonLabel.Size = '180,60'
        $SuccessfulLogonLabel.Location = '400,30'
        $SuccessfulLogonLabel.Font = New-object System.Drawing.Font('Calibri', 16,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        $panel2.Controls.Add($SuccessfulLogonLabel)


        $SuccessfulLogonGroupTextBox = New-Object System.Windows.Forms.TextBox

         # add textbox
        $SuccessfulLogonGroupTextBox.Multiline = $true
        $SuccessfulLogonGroupTextBox.Height = 300
        $SuccessfulLogonGroupTextBox.Width = 300
        $SuccessfulLogonGroupTextBox.Location ='400,100'
        $SuccessfulLogonGroupTextBox.BorderStyle =
[System.Windows.Forms.BorderStyle]::Fixed3D
        $SuccessfulLogonGroupTextBox.ScrollBars =
[System.Windows.Forms.ScrollBars]::Both
        $SuccessfulLogonGroupTextBox.ReadOnly = $true
        $SuccessfulLogonGroupTextBox.Font = New-Object System.Drawing.Font("Times New
Roman",16)

        $panel2.Controls.Add($SuccessfulLogonGroupTextBox)


        $SuccessfulLogonGroupTextBox.Text = $Global:SuccessLogonData.getenumerator() |
                                                select Name, Value |
                                                Sort-Object -Property Value -
Descending  |
                                                Out-String -Width 30


        $ExplicitLogonLabel = New-Object System.Windows.Forms.Label
        $ExplicitLogonLabel.Text = "Explicit Credentials Logons: IP Addresses:"
        $ExplicitLogonLabel.Size = '270,60'
        $ExplicitLogonLabel.Location = '750,30'
        $ExplicitLogonLabel.Font = New-object System.Drawing.Font('Calibri', 16,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        $panel2.Controls.Add($ExplicitLogonLabel)


        $ExplicitLogonGroupTextBox = New-Object System.Windows.Forms.TextBox

         # add textbox
        $ExplicitLogonGroupTextBox.Multiline = $true
        $ExplicitLogonGroupTextBox.Height = 300
        $ExplicitLogonGroupTextBox.Width = 300
        $ExplicitLogonGroupTextBox.Location ='750,100'
        $ExplicitLogonGroupTextBox.BorderStyle =
[System.Windows.Forms.BorderStyle]::Fixed3D
        $ExplicitLogonGroupTextBox.ScrollBars =
[System.Windows.Forms.ScrollBars]::Both
        $ExplicitLogonGroupTextBox.ReadOnly = $true
        $ExplicitLogonGroupTextBox.Font = New-Object System.Drawing.Font("Times New
Roman",16)

        $panel2.Controls.Add($ExplicitLogonGroupTextBox)


        $ExplicitLogonGroupTextBox.Text = $Global:ExplicitLogonData.getenumerator() |
                                                select Name, Value |
                                                Sort-Object -Property Value -
Descending  |
                                                Out-String -Width 30
```

```
        <# PANEL 3 #>


        $SQLQueryTextBox = New-Object System.Windows.Forms.TextBox
        $SQLQueryTextBox.Text = 'put your sql query here'
        $SQLQueryTextBox.Name = 'SQLQueryTextBox'
        $SQLQueryTextBox.TabIndex = 0
        $SQLQueryTextBox.Size = '1090,200'
        $SQLQueryTextBox.Location = '45,20'
        $SQLQueryTextBox.DataBindings.DefaultDataSourceUpdateMode = 0

        $panel3.Controls.Add($SQLQueryTextBox)


        $GetTableButton = New-Object System.Windows.Forms.Button

        $GetTableButton.UseVisualStyleBackColor = $True
        $GetTableButton.Text = 'Get Table'
        $GetTableButton.DataBindings.DefaultDataSourceUpdateMode = 0
        $GetTableButton.TabIndex = 1
        $GetTableButton.Name = 'GetTableButton'
        $GetTableButton.Size = '180,30'
        $GetTableButton.Location = '45,65'

        $panel3.Controls.Add($GetTableButton)

        $GetTableButton.add_Click({
            get-tablecontents -query $SQLQueryTextBox.Text | Out-GridView

        })



        $SaveTableToCsvButton = New-Object System.Windows.Forms.Button

        $SaveTableToCsvButton.UseVisualStyleBackColor = $True
        $SaveTableToCsvButton.Text = 'Save Table to CSV file'
        $SaveTableToCsvButton.DataBindings.DefaultDataSourceUpdateMode = 0
        $SaveTableToCsvButton.TabIndex = 1
        $SaveTableToCsvButton.Name = 'SaveTableToCsvButton'
        $SaveTableToCsvButton.Size = '200,30'
        $SaveTableToCsvButton.Location = '45,105'
        $panel3.Controls.Add($SaveTableToCsvButton)


        $SaveTableToCsvButton.add_Click({


             # For EXE USE THIS
            [switch]$pathExists = Test-Path -Path ((Get-
Location).Path+"\exportedData")
            #FOR EXE USE THIS
            $path = (Get-Location).Path+"\exportedData"
            # elegxei an o fakelos exportedData yparxei!
            # an den yparxei ton dhmiourgei kai vazei ekei mesa tis ta csv kai ta html
            if (!$pathExists){
                New-Item -ItemType Directory -Force -Path $path
            }
          <#

             # FOR SCRIPT USE THIS
            [switch]$pathExists = Test-Path -Path $PSScriptRoot\exportedData
            $path = $PSScriptRoot+"\exportedData"

            #write-host $path
            # elegxei an o fakelos chartsImages yparxei!
            # an den yparxei ton dhmiourgei kai vazei ekei mesa tis eikones
            if (!$pathExists){
                New-Item -ItemType Directory -Force -Path $PSScriptRoot\exportedData
            }

           #>
            [string]$fileName = (get-date).ToString("dd-MMM-yyyy-HH-mm-ss_")

            $fileName += ((((((($SQLQueryTextBox.Text.Replace("
","_")).Replace("*","All")).

Replace(">","GT")).Replace("<","LT")).
                                                    Replace(":","-")).Replace("/","-
")).Replace("\","-")

            if (($fileName.Length+$path.Length) -gt 255){
```

```
                $fileName = (get-date).ToString("dd-MMM-yyyy-HH-mm-ss_")
                write-host $fileName  $path
                $fileName += "bigQuery"
            }
            #

            get-tablecontents -query $SQLQueryTextBox.Text | export-csv -Path
("$path\$fileName.csv")
            write-host "File Saved as: ""$path\$fileName.csv"

        })

        $SaveTableToHtmButton = New-Object System.Windows.Forms.Button

        $SaveTableToHtmButton.UseVisualStyleBackColor = $True
        $SaveTableToHtmButton.Text = 'Save Table to HTM file'
        $SaveTableToHtmButton.DataBindings.DefaultDataSourceUpdateMode = 0
        $SaveTableToHtmButton.TabIndex = 1
        $SaveTableToHtmButton.Name = 'SaveTableToCsvButton'
        $SaveTableToHtmButton.Size = '200,30'
        $SaveTableToHtmButton.Location = '45,150'
        $panel3.Controls.Add($SaveTableToHtmButton)


        $SaveTableToHtmButton.add_Click({


             # For EXE USE THIS
            [switch]$pathExists = Test-Path -Path ((Get-
Location).Path+"\exportedData")
            #FOR EXE USE THIS
            $path = (Get-Location).Path+"\exportedData"
            # elegxei an o fakelos exportedData yparxei!
            # an den yparxei ton dhmiourgei kai vazei ekei mesa tis ta csv kai ta html
            if (!$pathExists){
                New-Item -ItemType Directory -Force -Path $path
            }

           <#
             # FOR SCRIPT USE THIS
            [switch]$pathExists = Test-Path -Path $PSScriptRoot\exportedData
            $path = $PSScriptRoot+"\exportedData"

            #write-host $path
            # elegxei an o fakelos chartsImages yparxei!
            # an den yparxei ton dhmiourgei kai vazei ekei mesa tis eikones
            if (!$pathExists){
                New-Item -ItemType Directory -Force -Path $PSScriptRoot\exportedData
            }
           #>
            [string]$fileName = (get-date).ToString("dd-MMM-yyyy-HH-mm-ss_")

            $fileName += ((((((($SQLQueryTextBox.Text.Replace("
","_")).Replace("*","All")).

Replace(">","GT")).Replace("<","LT")).
                                                    Replace(":","-")).Replace("/","-
")).Replace("\","-")
            #write-host "$path\$fileName.csv"

            if (($fileName.Length+$path.Length) -gt 255){
                $fileName = (get-date).ToString("dd-MMM-yyyy-HH-mm-ss_")
                write-host $fileName  $path
                $fileName += "bigQuery"
            }
            get-tablecontents -query $SQLQueryTextBox.Text | ConvertTo-Html | Out-File
-FilePath ("$path\$fileName.htm")
            write-host "File Saved as: ""$path\$fileName.htm"

        })



        <######################>
        <# Event Handlers Start #>

        $AllEventsRadioButton.Add_Click({
            $SecEventsComboBox.Enabled = $false
            $SecEventsComboBox.SelectedIndex = 0
            $EventsOccuredTextBox.Text = $Global:AllDataEventsOccured | Out-String
            #$EventsOccuredTextBox.Text = $AppData.count | Out-String
            if ($PieRadioButton.Checked){
```

```
                $EventsGroupTextBox.Text =
$Global:AllDataGroupByLogName.GetEnumerator()   |
                                                select Name, Value |
                                                Sort-Object -Property Value -
Descending  |
                                                Out-String -Width 30

$Chart.Series["Data"].Points.DataBindXY($Global:AllDataGroupByLogName.Keys,
$Global:AllDataGroupByLogName.Values)
                $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Pie
            } elseif($TimeLineRadioButton.Checked){
                $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
                $EventsGroupTextBox.Text = $Global:AllDataTimeLine.getenumerator() |
                                                                    select Name, Value
|
                                                                     Out-String -Width
35
                $Chart.Series["Data"].Points.DataBindXY($Global:AllDataTimeLine.Keys,
$Global:AllDataTimeLine.Values)
                $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
            }

        })


        $ApplicationEventsRadioButton.Add_Click({
            $SecEventsComboBox.Enabled = $false
            $SecEventsComboBox.SelectedIndex = 0
            $EventsOccuredTextBox.Text = $Global:AppDataEventsOccured | Out-String
            if ($PieRadioButton.Checked){
                $EventsGroupTextBox.Text =
$Global:AppDataGroupByEventId.GetEnumerator()   |
                                    select Name, Value |
                                    Sort-Object -Property Value -Descending   |
                                    Out-String -Width 20


$Chart.Series["Data"].Points.DataBindXY($Global:AppDataGroupByEventId.Keys,
$Global:AppDataGroupByEventId.Values)
                $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Pie
            } elseif($TimeLineRadioButton.Checked){
                $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
                $EventsGroupTextBox.Text = $Global:AppDataTimeLine.getenumerator() |
                                                                    select Name, Value
|
                                                                     Out-String -Width
35
                $Chart.Series["Data"].Points.DataBindXY($Global:AppDataTimeLine.Keys,
$Global:AppDataTimeLine.Values)
                $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
            }
        })


        $SecurityEventsRadioButton.Add_Click({
            $SecEventsComboBox.Enabled = $true

            switch ($SecEventsComboBox.SelectedItem) {
                "All Security Events"{
                    $PieRadioButton.Enabled = $true
                    $EventsOccuredTextBox.Text = $Global:SecDataEventsOccured | Out-
String
                    if ($PieRadioButton.Checked){
                    #edw den mpainei pote ousiastika

                       $EventsGroupTextBox.Text =
$Global:SecDataGroupByEventId.GetEnumerator()   |
                                    select Name, Value |
                                    Sort-Object -Property Value -Descending   |
                                    Out-String -Width 20

$Chart.Series["Data"].Points.DataBindXY($Global:SecDataGroupByEventId.Keys,
$Global:SecDataGroupByEventId.Values)
                        $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Pie
```

```
                    } elseif($TimeLineRadioButton.Checked){
                        $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
                        $EventsGroupTextBox.Text =
$Global:SecDataTimeLine.GetEnumerator() | select name,value | Out-String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecDataTimeLine.Keys,
$Global:SecDataTimeLine.Values)
                        $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    };break
                }
                "Logon Failure"{
                    $PieRadioButton.Enabled = $false
                    $PieRadioButton.Checked = $false
                    $TimeLineRadioButton.Checked = $true
                    $EventsOccuredTextBox.Text = $Global:SecFailDataEventsOccured |
Out-String

                    $Chart.Series["Data"].Color =
[System.Drawing.Color]::PaleVioletRed
                    $EventsGroupTextBox.Text =
$Global:SecFailDataTimeLine.GetEnumerator() |
                                                                    select name,value | Out-
String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecFailDataTimeLine.Keys,
$Global:SecFailDataTimeLine.Values)
                    $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    ;break
                }
                "Logon Success"{
                    $PieRadioButton.Enabled = $false
                    $PieRadioButton.Checked = $false
                    $TimeLineRadioButton.Checked = $true
                    $EventsOccuredTextBox.Text = $Global:SecSuccDataEventsOccured |
Out-String

                    $Chart.Series["Data"].Color = [System.Drawing.Color]::YellowGreen
                    $EventsGroupTextBox.Text =
$Global:SecSuccDataTimeLine.GetEnumerator() | select name,value | Out-String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecSuccDataTimeLine.Keys,
$Global:SecSuccDataTimeLine.Values)
                    $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    ;break
                }
            }
        })


        $SystemEventsRadioButton.Add_Click({
            $SecEventsComboBox.Enabled = $false
            $SecEventsComboBox.SelectedIndex = 0
            $EventsOccuredTextBox.Text = $Global:SysDataEventsOccured | Out-String
            if ($PieRadioButton.Checked){
                $EventsGroupTextBox.Text =
$Global:SysDataGroupByEventId.GetEnumerator()  |
                                    select Name, Value |
                                    Sort-Object -Property Value -Descending  |
                                    Out-String -Width 20

$Chart.Series["Data"].Points.DataBindXY($Global:SysDataGroupByEventId.Keys,
$Global:SysDataGroupByEventId.Values)
                $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Pie
            } elseif($TimeLineRadioButton.Checked){
                $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
                $EventsGroupTextBox.Text = $Global:SysDataTimeLine.getenumerator() |
                                                                    select Name, Value
|
                                                                     Out-String -Width
35
                $Chart.Series["Data"].Points.DataBindXY($Global:SysDataTimeLine.Keys,
$Global:SysDataTimeLine.Values)
                $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
            }
        })
```

```
        $PieRadioButton.add_click({
            if($AllEventsRadioButton.Checked){
                $EventsOccuredTextBox.Text = $Global:AllDataEventsOccured | Out-String
                $EventsGroupTextBox.Text =
$Global:AllDataGroupByLogName.GetEnumerator()  |
                                    select Name, Value |
                                    Sort-Object -Property Value -Descending  |
                                    Out-String -Width 30

$Chart.Series["Data"].Points.DataBindXY($Global:AllDataGroupByLogName.Keys,
$Global:AllDataGroupByLogName.Values)
            } elseif ($ApplicationEventsRadioButton.Checked){
                $EventsOccuredTextBox.Text = $Global:AppDataEventsOccured | Out-String
                $EventsGroupTextBox.Text =
$Global:AppDataGroupByEventId.GetEnumerator()  |
                                    select Name, Value |
                                    Sort-Object -Property Value -Descending  |
                                    Out-String -Width 20


$Chart.Series["Data"].Points.DataBindXY($Global:AppDataGroupByEventId.Keys,
$Global:AppDataGroupByEventId.Values)
            } elseif ($SecurityEventsRadioButton.Checked){
                switch ($SecEventsComboBox.SelectedItem) {
                    "All Security Events"{
                        $EventsOccuredTextBox.Text = $Global:SecDataEventsOccured |
Out-String
                        $EventsGroupTextBox.Text =
$Global:SecDataGroupByEventId.GetEnumerator()  |
                                    select Name, Value |
                                    Sort-Object -Property Value -Descending  |
                                    Out-String -Width 20


$Chart.Series["Data"].Points.DataBindXY($Global:SecDataGroupByEventId.Keys,
$Global:SecDataGroupByEventId.Values)
                    }
                }
            } elseif ($SystemEventsRadioButton.Checked){
                $EventsOccuredTextBox.Text = $Global:SysDataEventsOccured | Out-String
                $EventsGroupTextBox.Text =
$Global:SysDataGroupByEventId.GetEnumerator()  |
                                    select Name, Value |
                                    Sort-Object -Property Value -Descending  |
                                    Out-String -Width 20


$Chart.Series["Data"].Points.DataBindXY($Global:SysDataGroupByEventId.Keys,
$Global:SysDataGroupByEventId.Values)
            }


            # set chart type
            #$Chart.Series["Data"].Points.DataBindXY($AllPieData.Keys,
$AllPieData.Values)
            $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Pie
        })

        #$timelinePressedCounter=0
        $TimeLineRadioButton.Add_Click({

            if($AllEventsRadioButton.Checked){

                $EventsOccuredTextBox.Text = $Global:AllDataEventsOccured | Out-String
                $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
                $EventsGroupTextBox.Text = $Global:AllDataTimeLine.getenumerator() |
                                                                    select Name, Value
|
                                                                     Out-String -Width
35
                $Chart.Series["Data"].Points.DataBindXY($Global:AllDataTimeLine.Keys,
$Global:AllDataTimeLine.Values)


            } elseif ($ApplicationEventsRadioButton.Checked){
                $EventsOccuredTextBox.Text = $Global:AppDataEventsOccured | Out-String
                $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
                $EventsGroupTextBox.Text = $Global:AppDataTimeLine.GetEnumerator() |
select name,value | Out-String -Width 35
```

```
                $Chart.Series["Data"].Points.DataBindXY($Global:AppDataTimeLine.Keys,
$Global:AppDataTimeLine.Values)
            } elseif ($SecurityEventsRadioButton.Checked){
                switch ($SecEventsComboBox.SelectedItem) {
                    "All Security Events"{
                        $EventsOccuredTextBox.Text = $Global:SecDataEventsOccured |
Out-String
                        $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan

                        $EventsGroupTextBox.Text =
$Global:SecDataTimeLine.GetEnumerator() | select name,value | Out-String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecDataTimeLine.Keys,
$Global:SecDataTimeLine.Values)
                        #$Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    }
                    "Logon Failure"{
                        $EventsOccuredTextBox.Text = $Global:SecFailDataEventsOccured
| Out-String
                        $Chart.Series["Data"].Color =
[System.Drawing.Color]::PaleVioletRed

                        $EventsGroupTextBox.Text =
$Global:SecFailDataTimeLine.GetEnumerator() | select name,value | Out-String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecFailDataTimeLine.Keys,
$Global:SecFailDataTimeLine.Values)
                        #$Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    }
                    "Logon Success"{
                        $EventsOccuredTextBox.Text = $Global:SecSuccDataEventsOccured
| Out-String
                        $Chart.Series["Data"].Color =
[System.Drawing.Color]::YellowGreen

                        $EventsGroupTextBox.Text =
$Global:SecSuccDataTimeLine.GetEnumerator() | select name,value | Out-String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecSuccDataTimeLine.Keys,
$Global:SecSuccDataTimeLine.Values)
                        #$Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    }
                }
            } elseif ($SystemEventsRadioButton.Checked){
                $EventsOccuredTextBox.Text = $Global:SysDataEventsOccured | Out-String
                $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
                $EventsGroupTextBox.Text = $Global:SysDataTimeLine.GetEnumerator() |
select name,value | Out-String -Width 35

                $Chart.Series["Data"].Points.DataBindXY($Global:SysDataTimeLine.Keys,
$Global:SysDataTimeLine.Values)
            }

            $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
        })



        # event handler for SecEventsComboBox
        $SecEventsComboBox.Add_SelectedIndexChanged({
            switch ($SecEventsComboBox.SelectedItem) {
                "All Security Events"{
                    $PieRadioButton.Enabled = $true
                    $EventsOccuredTextBox.Text = $Global:SecDataEventsOccured | Out-
String
                    if ($PieRadioButton.Checked){
                    #edw den mpainei pote ousiastika

                       $EventsGroupTextBox.Text =
$Global:SecDataGroupByEventId.GetEnumerator()  |
                                    select Name, Value |
                                    Sort-Object -Property Value -Descending  |
                                    Out-String -Width 20

$Chart.Series["Data"].Points.DataBindXY($Global:SecDataGroupByEventId.Keys,
$Global:SecDataGroupByEventId.Values)
                        $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Pie
```

```
                    } elseif($TimeLineRadioButton.Checked){
                        $Chart.Series["Data"].Color = [System.Drawing.Color]::DarkCyan
                        $EventsGroupTextBox.Text =
$Global:SecDataTimeLine.GetEnumerator() | select name,value | Out-String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecDataTimeLine.Keys,
$Global:SecDataTimeLine.Values)
                        $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    };break
                }
                "Logon Failure"{
                    $PieRadioButton.Enabled = $false
                    $PieRadioButton.Checked = $false
                    $TimeLineRadioButton.Checked = $true
                    $EventsOccuredTextBox.Text = $Global:SecFailDataEventsOccured |
Out-String

                    $Chart.Series["Data"].Color =
[System.Drawing.Color]::PaleVioletRed
                    $EventsGroupTextBox.Text =
$Global:SecFailDataTimeLine.GetEnumerator() |
                                                                    select name,value | Out-
String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecFailDataTimeLine.Keys,
$Global:SecFailDataTimeLine.Values)
                    $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    ;break
                }
                "Logon Success"{
                    $PieRadioButton.Enabled = $false
                    $PieRadioButton.Checked = $false
                    $TimeLineRadioButton.Checked = $true
                    $EventsOccuredTextBox.Text = $Global:SecSuccDataEventsOccured |
Out-String

                    $Chart.Series["Data"].Color = [System.Drawing.Color]::YellowGreen
                    $EventsGroupTextBox.Text =
$Global:SecSuccDataTimeLine.GetEnumerator() | select name,value | Out-String -Width 35

$Chart.Series["Data"].Points.DataBindXY($Global:SecSuccDataTimeLine.Keys,
$Global:SecSuccDataTimeLine.Values)
                    $Chart.Series["Data"].ChartType =
[System.Windows.Forms.DataVisualization.Charting.SeriesChartType]::Line
                    ;break
                }
            }
        })


        $SaveChartButton.Add_Click({
            # arxika h onomasia ksekina me stigmiotypo wras sth morfi 14-Jun-2015-21-
02-55
            [string]$fileName = (get-date).ToString("dd-MMM-yyyy-HH-mm-ss_")
            $fileName += $Type
            if ($PieRadioButton.Checked){
                if ($AllEventsRadioButton.Checked){
                    $fileName += "AllEventsPie"
                } elseif($ApplicationEventsRadioButton.Checked){
                    $fileName += "ApplicationEventsPie"
                } elseif($SecurityEventsRadioButton.Checked){
                     $fileName += "SecurityEventsPie"
                } elseif($SystemEventsRadioButton){
                     $fileName += "SystemEventsPie"
                }
            } elseif ($TimeLineRadioButton.Checked){
                if ($AllEventsRadioButton.Checked){
                    $fileName += "AllEventsTimeLine"
                } elseif($ApplicationEventsRadioButton.Checked){
                    $fileName += "ApplicationEventsTimeLine"
                } elseif($SecurityEventsRadioButton.Checked){
                    if ($SecEventsComboBox.SelectedIndex -eq 0){
                        $fileName += "AllSecurityEventsTimeLine"
                    } elseif($SecEventsComboBox.SelectedIndex -eq 1){
                        $fileName += "FailureLogonSecurityEventsTimeLine"
                    } elseif($SecEventsComboBox.SelectedIndex -eq 2){
                        $fileName += "SuccessLogonSecurityEventsTimeLine"
                    }
                } elseif($SystemEventsRadioButton){
```

```
                    $fileName += "SystemEventsTimeLine"
                }
            }


             # For EXE USE THIS
            [switch]$pathExists = Test-Path -Path ((Get-
Location).Path+"\chartsImages")
            #FOR EXE USE THIS
            $path = (Get-Location).Path+"\chartsImages"
            # elegxei an o fakelos exportedData yparxei!
            # an den yparxei ton dhmiourgei kai vazei ekei mesa tis ta csv kai ta html
            if (!$pathExists){
                New-Item -ItemType Directory -Force -Path $path
            }

           <#
            # For Script USE THIS
            [switch]$pathExists = Test-Path -Path $PSScriptRoot\chartsImages

            $path = $PSScriptRoot+"\chartsImages"
            # elegxei an o fakelos chartsImages yparxei!
            # an den yparxei ton dhmiourgei kai vazei ekei mesa tis eikones
            if (!$pathExists){
                New-Item -ItemType Directory -Force -Path $PSScriptRoot\chartsImages
            }
            #>

            $Chart.Width = 2000
            $Chart.Height = 1000
            # $PSScriptRoot: This is an automatic variable set to the current
file's/module's directory
            $Chart.SaveImage("$path\$fileName.png","png")
            $Chart.Width = 850
            $Chart.Height = 450
        })

        <# Event Handlers End #>
        <######################>




        # Get the results from the button click
        $dialogResult = $Form.ShowDialog()
        #$dialogResult
    }
}



function Get-PreparedForVisualization
{
    [CmdletBinding()]
    Param
    (
    )
    Process
    {
        # this is to start with a window that will inform the user
        # for what it will happen
        # data have to be loaded
        $PreparingDataForm = New-Object System.Windows.Forms.Form

        $PreparingDataForm.Text = "Preparing Data"
        $PreparingDataForm.Width = 500
        $PreparingDataForm.Height = 470
        $PreparingDataForm.MaximizeBox = $False
        $PreparingDataForm.StartPosition = 'CenterScreen'
        $PreparingDataForm.FormBorderStyle =
[System.Windows.Forms.FormBorderStyle]::Fixed3D
        $PreparingDataForm.BackColor = [System.Drawing.Color]::LightCyan

        $EventsFoundLabel = New-Object system.windows.forms.label
        $EventsFoundLabel.Text = "Total Event Log Events Found:"
        $EventsFoundLabel.Size = '312,30'
        $EventsFoundLabel.Location = '25,90'
        $EventsFoundLabel.Font = New-object System.Drawing.Font('Calibri', 18,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)
```

```
        $PreparingDataForm.Controls.Add($EventsFoundLabel)

        $EventsFoundTextBox = New-Object -TypeName 'System.Windows.Forms.TextBox'
        $EventsFoundTextBox.Size = '100,20'
        $EventsFoundTextBox.Location = '340,90'
        $EventsFoundTextBox.Font = New-Object System.Drawing.Font("Times New
Roman",16)
        $EventsFoundTextBox.ReadOnly = $true

        $EventsFoundTextBox.Text = Get-TableRowNumber -Table events | Out-String

        $PreparingDataForm.Controls.Add($EventsFoundTextBox)

        $InformationLabel = New-Object system.windows.forms.label
        $InformationLabel.Text = "Utilizing Windows PowerShell"
        $InformationLabel.Size = '450,30'
        $InformationLabel.Location = '115,20'
        $InformationLabel.ForeColor = [System.Drawing.Color]::DimGray
        $InformationLabel.Font = New-object System.Drawing.Font('Calibri', 15,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        $PreparingDataForm.Controls.Add($InformationLabel)

        $InformationLabel2 = New-Object system.windows.forms.label
        $InformationLabel2.Text = "for Host-based IDS Log Monitoring"
        $InformationLabel2.Size = '450,30'
        $InformationLabel2.Location = '100,50'
        $InformationLabel2.ForeColor = [System.Drawing.Color]::DimGray
        $InformationLabel2.Font = New-object System.Drawing.Font('Calibri', 15,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

         $PreparingDataForm.Controls.Add($InformationLabel2)

        $ProceedLabel = New-Object system.windows.forms.label
        $ProceedLabel.Text = "Proceed by choosing a time range."
        $ProceedLabel.Size = '400,40'
        $ProceedLabel.Location = '40,145'
        $ProceedLabel.Font = New-object System.Drawing.Font('Calibri', 12,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        $PreparingDataForm.Controls.Add($ProceedLabel)

        # Add Button
        $ProceedButton = New-Object System.Windows.Forms.Button
        $ProceedButton.Location = New-Object System.Drawing.Size(165,350)
        $ProceedButton.Font = New-object System.Drawing.Font('Calibri', 12,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)
        $ProceedButton.BackColor = [System.Drawing.Color]::Gray
        $ProceedButton.ForeColor = [System.Drawing.Color]::Black
        $ProceedButton.Size = New-Object System.Drawing.Size(150,53)
        $ProceedButton.Text = "Proceed To
Visualization"


        $PreparingDataForm.Controls.Add($ProceedButton)


        $RangeGroupBox = New-Object System.Windows.Forms.GroupBox
        $RangeGroupBox.Location = '30,187'
        $RangeGroupBox.Size = '200,120'
        $RangeGroupBox.Text = "Select time range:"
        $RangeGroupBox.Font = New-object System.Drawing.Font('Calibri', 14,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        $HistoryRangeRadioButton = New-Object System.Windows.Forms.RadioButton
        $HistoryRangeRadioButton.Location = '40,25'
        $HistoryRangeRadioButton.Size = '100,30'
        $HistoryRangeRadioButton.Text = "History"
        $HistoryRangeRadioButton.Checked = $true
        $HistoryRangeRadioButton.Name = "history"
        $HistoryRangeRadioButton.Font = New-object System.Drawing.Font('Calibri', 12,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)



        $IntradayRangeRadioButton = New-Object System.Windows.Forms.RadioButton
        $IntradayRangeRadioButton.Location = '40,53'
        $IntradayRangeRadioButton.Size = '100,30'
        $IntradayRangeRadioButton.Text = "Intraday"
        $IntradayRangeRadioButton.Name = "intraday"
        $IntradayRangeRadioButton.Font = New-object System.Drawing.Font('Calibri', 12,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)
```

```
        $CustomRangeRadioButton = New-Object System.Windows.Forms.RadioButton
        $CustomRangeRadioButton.Location = '40,82'
        $CustomRangeRadioButton.Size = '130,30'
        $CustomRangeRadioButton.Text = "Custom Range"
        $CustomRangeRadioButton.Name = "custom"
        $CustomRangeRadioButton.Font = New-object System.Drawing.Font('Calibri', 12,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)


        # Add the GroupBox controls
        $RangeGroupBox.Controls.Add($HistoryRangeRadioButton)
        $RangeGroupBox.Controls.Add($IntradayRangeRadioButton)
        $RangeGroupBox.Controls.Add($CustomRangeRadioButton)


        #to TypeOfViewGroupBox topotheteitai sth forma
        $PreparingDataForm.Controls.Add($RangeGroupBox)

        # Create a comboBox list that will contain Available Security Events

        # Label for the available machines combobox
        $AfterDateLabel = New-Object system.windows.forms.label
        $AfterDateLabel.Text = "After Date:"
        $AfterDateLabel.Size = '100,20'
        $AfterDateLabel.Location = '260,190'
        $AfterDateLabel.Font = New-object System.Drawing.Font('Calibri', 12,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        $PreparingDataForm.Controls.Add($AfterDateLabel)

        $CustomRangeListBox = New-Object System.Windows.Forms.ListBox
        $CustomRangeListBox.Location = '260,210'
        $CustomRangeListBox.Size = '150,120'

        $CustomRangeListBox.Enabled = $false


        $CustomRangeListBox.Font = New-object System.Drawing.Font('Calibri', 10,
[System.Drawing.FontStyle]::Bold, [System.Drawing.GraphicsUnit]::Point,0)

        #$CustomRangeComboBox.DropDownStyle =
[System.Windows.Forms.ComboBoxStyle]::DropDown

        $PreparingDataForm.Controls.Add($CustomRangeListBox)


        [System.DateTime]$LastEventDate = Get-LastEventDateFromDatabase -Table events

        $DatesArray = New-Object System.Collections.ArrayList

        [int]$Global:c=0
        while ((Get-Date).AddDays($c) -ge $LastEventDate) {
            $forDay = (Get-Date).AddDays($c)
            $ArrayListAddition = $DatesArray.Add($forDay.ToString("MM/dd/yyyy"))

            $c=$c-1
            #$d = (Get-Date).AddDays($c)

        }

        if (($LastEventDate.Day).Equals((Get-Date).AddDays($c).Day)) {
            #after this adds and the last event date
            $ArrayListAddition =
$DatesArray.Add($LastEventDate.ToString("MM/dd/yyyy"))
            #$DatesArray.Count
        }


        foreach ($day in $DatesArray){
            # putting in variable to prevent from output to console
            $m = $CustomRangeListBox.Items.Add($day)
        }


        #to TypeOfViewGroupBox topotheteitai sth forma
        $PreparingDataForm.Controls.Add($CustomRangeListBox)
        # sets 1st value of $machines as the default value of the combobox
        #$CustomRangeListBox.SelectedIndex = 0
        # Creating the collection of type of chart radio button
```

```
        $System_Drawing_Size = New-Object System.Drawing.Size
        $System_Drawing_Size.Width = 460
        $System_Drawing_Size.Height = 30

        $EventsArray = New-Object System.Collections.ArrayList

        <# Event Handlers Start #>
        <######################>

        $CustomRangeRadioButton.Add_Click({
            $CustomRangeListBox.Enabled = $true
        })
        $HistoryRangeRadioButton.Add_Click({
            $CustomRangeListBox.Enabled = $false
            $str = Get-TableRowNumber -Table events
            $EventsFoundTextBox.Text = $str

        })
        $IntradayRangeRadioButton.Add_Click({
            $CustomRangeListBox.Enabled = $false
            $str = Get-TableRowNumber -Table events -After (Get-Date -Format
"MM/dd/yyyy 00:00:00").ToString()
            $EventsFoundTextBox.Text = $str
        })
        $CustomRangeListBox.Add_Click({
            [string]$tempDate = $CustomRangeListBox.SelectedItems + " 00:00:00"
            $str = Get-TableRowNumber -Table events -After $tempDate
            $EventsFoundTextBox.Text = $str | Out-String
        })
        $Global:Type = ""
        $i=0

         #Add Button event
        $ProceedButton.Add_Click({

            if(!($EventsFoundTextBox.Text.Equals("0"))){

                $ProceedButton.Enabled = $false
                $CustomRangeRadioButton.Enabled = $false
                $HistoryRangeRadioButton.Enabled = $false
                $IntradayRangeRadioButton.Enabled = $false
                $CustomRangeListBox.Enabled = $false

                $PreparingDataForm.Close()

            } else {
                Write-Host "Zero Events Found. Nothing can be vizualized"
            }

        })

        <# Event Handlers End #>
        <######################>
        $a = $PreparingDataForm.ShowDialog()

    }
    End
    {
        # otan o xrhsths pathsei to koumpi procceed to visualization mia apo tis
parakatw times pernaei san parametros sto epomeno parathyro
        # analoga me ti epithimei na optikopoihsei (istoriko intraday h custom) h
parametros einai mia hmeromhnia
        #
        #$Global:Type = ""
        if ($HistoryRangeRadioButton.Checked) {
            $Global:Type =  "History"
            $afterDate =  Get-LastEventDateFromDatabase -Table EVENTS
        } elseif ($IntradayRangeRadioButton.Checked){
            $Global:Type =  "Intraday"
            $afterDate =  (get-date).date.ToString("MM/dd/yyyy HH:mm:ss")
        } else {
            $Global:Type =  "Custom"
            $afterDate = [string]$CustomRangeListBox.SelectedItems + " 00:00:00"
        }
        Write-Output $afterDate
    }
}


Get-LogVisualization -AfterDate (Get-PreparedForVisualization) -Type $Global:Type
```

5.3 Summary

My implementation consists of:

- Having installed SQL Server 2012 Express [36]
- LogAnalysis module
- LogDatabase module
- LogScheduler script
- ScheduleLogs script
- LogVisualization script

Figure 72 provides a schema that can be described as follows:

- Getting events from the Windows Server Host machine *(data source)*.
- Events are being analyzed and stored in the database *(analysis engine)*.
- Script runs every ten minutes to ensure that the database is continually updated.
- Events are being analyzed and ready for visualization *(analysis engine)*.
- Visualizing events running *(response engine)*.

Chapter 6. Implementation Results

In this chapter we are going to provide all different kinds of results we get from our *PowerShell HIDS Log Monitoring* tool.

Once we make sure that we have all the system installed and configured properly, we begin with testing it. Figure 73 shows our desktop. There is a folder called *LogVisualization* and we can click on it to view its contents.

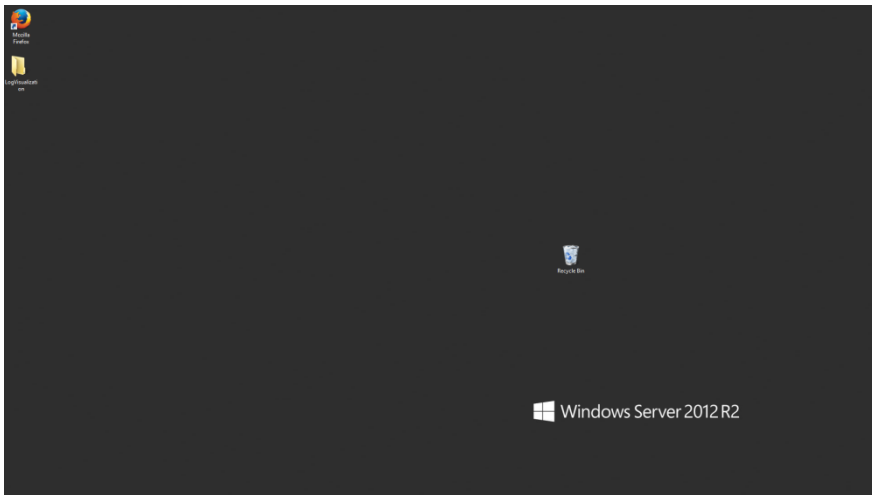


Figure 73. Desktop contains LogVisualization folder.

Within LogVisualization there is an executable file, as shown in figure 74. We can double click on it in order to run and open the program.

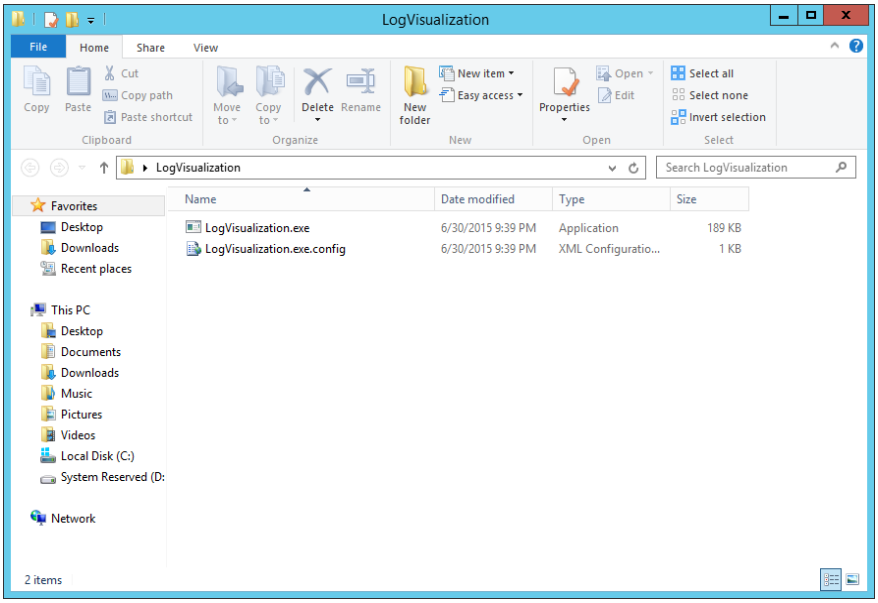


Figure 74. LogVisualization folder contains an executable to run the program.

Before we proceed with describing each of the capabilities of our tool, we have to make a brief introduction about how we transformed our *.ps1* script to an executable file and what the LogVisualization.ps1 contains.

The idea was to make a system simple and easy to use and run. Without worry about how to run a *.ps1* script, users can double click on the executable and get the same results. The procedure to convert PowerShell scripts into EXE files was provided by Microsoft TechNet Gallery, and more specifically from Ingo Karstein [34].

LogVisualization.ps1 consists of 3 functions and two of these implement GUI forms:

- Get-PreparedForVisualization
  - Provides the main welcome form
- Get-LogVisualization
  - Provides two forms, *one for the verification of the visualization and one for the visualization.*
- Get-DatesUntilNow

After running the executable a window is displayed, as shown in figure 75. This is the main GUI form and it enables us to choose one of the available use cases.

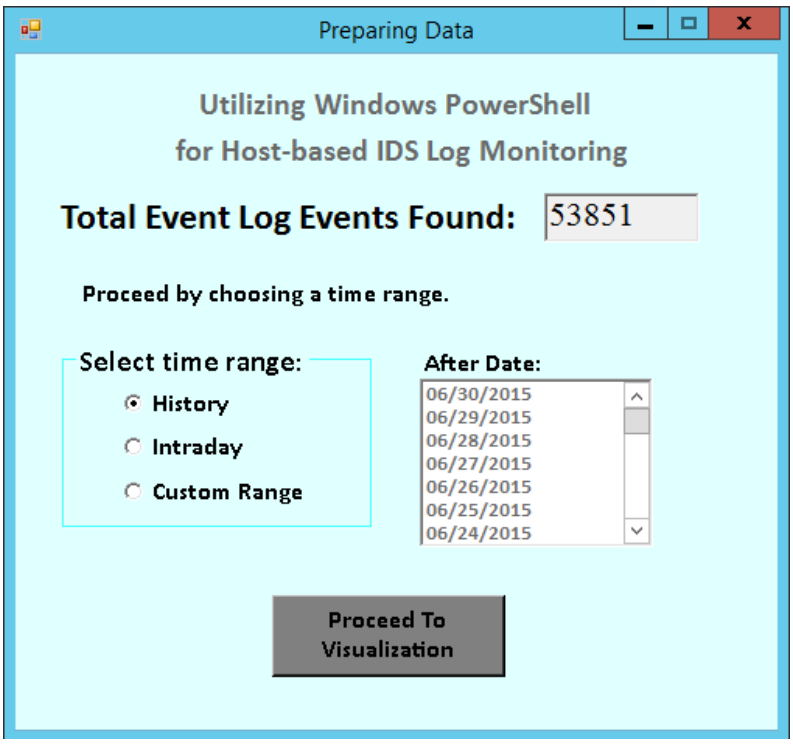


Figure 75. Main panel of the tool (History).

We can see at this window that there is a textbox showing how many events found in database, a group of radiobuttons that enable us move to another use case, a listbox for choosing a custom range to visualize and finally a button for proceeding to the visualization.

Figures 75, 76, and 77 show the available use cases. We are going to describe each of these separately in the following pages.

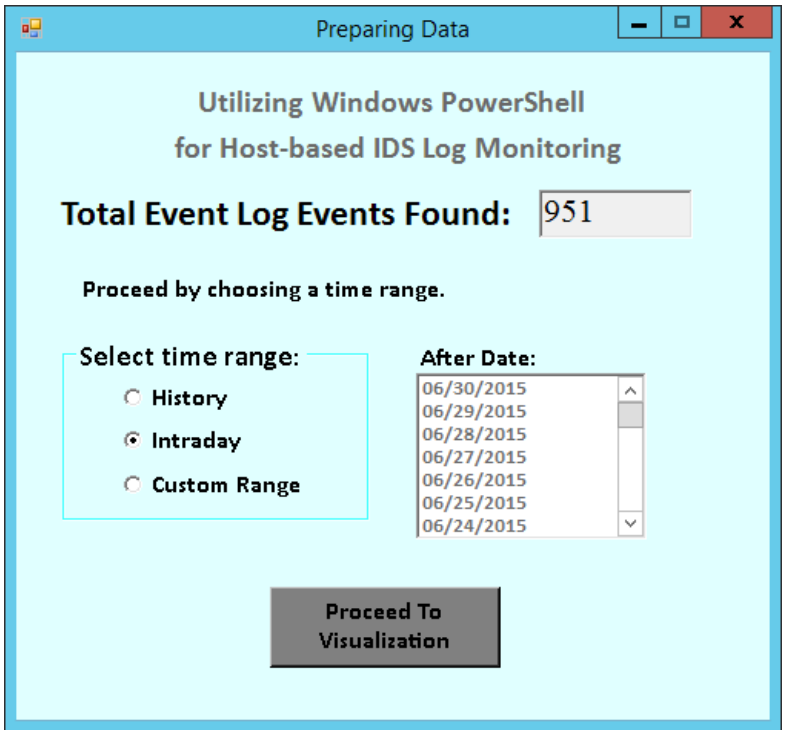


Figure 76. Main panel of the tool (Intraday).

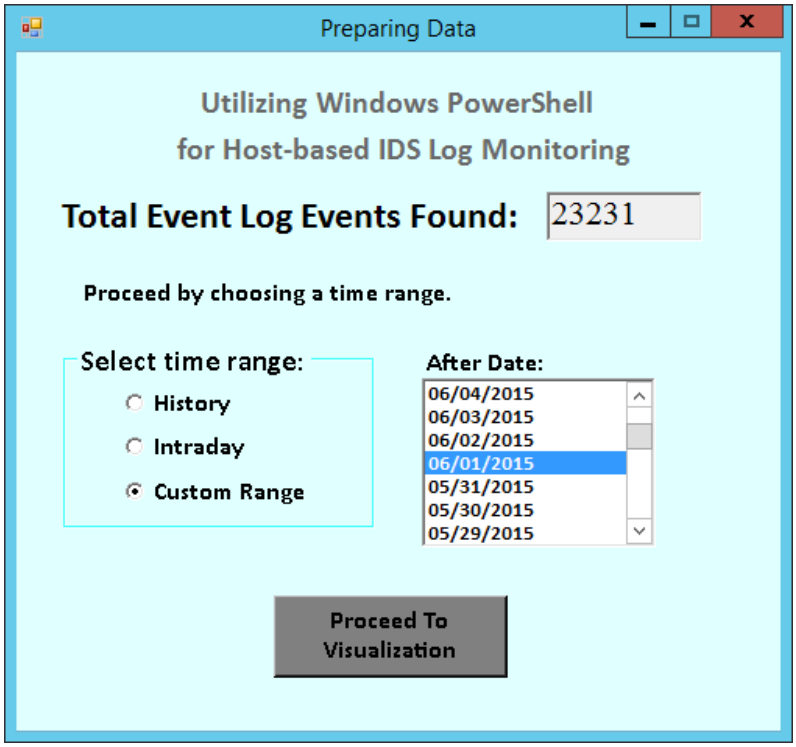


Figure 77. Main panel of the tool (Custom).

In the following 3 sections we are going to provide examples of how our tool works. If we hit *proceed to visualization* the second form will be displayed, as shown in figure 78. The purpose of this window is to inform the user about what is going on in the background.

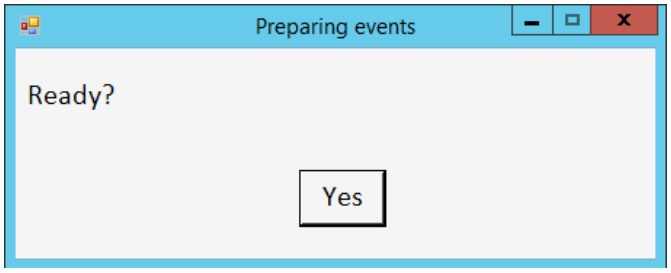


Figure 78. Preparing events panel.

6.1 History use case

Consider that we have chosen the History RadioButton. This will look at the database, find and analyze all of the available data and finally the data will be visualized, as shown in figure 79.

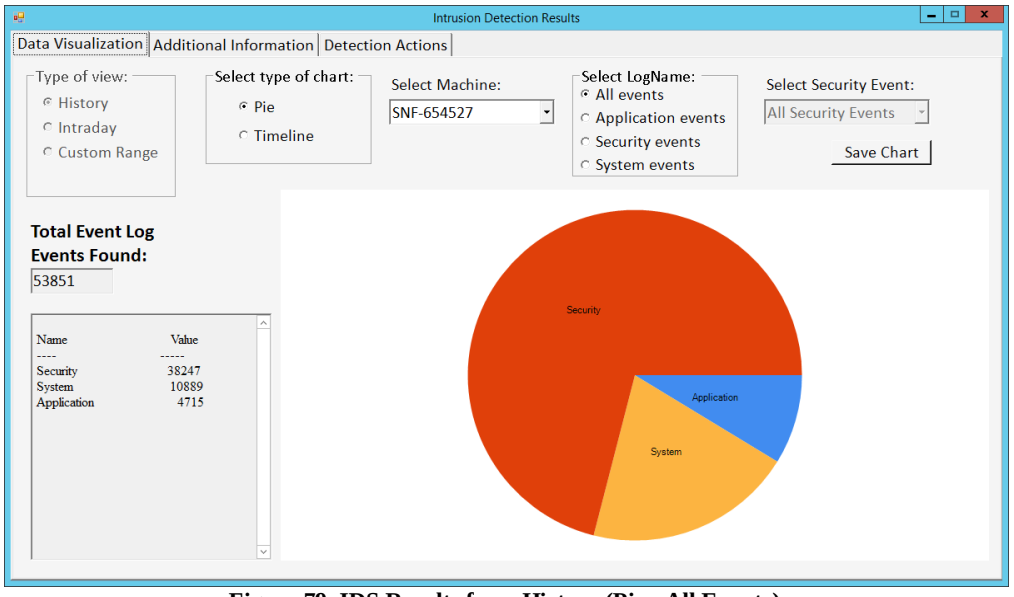


Figure 79. IDS Results from History (Pie - All Events).

Intrusion Detection Result window basically consists of three panels. The first panel we are going to describe is the Data Visualization panel. When we are on this panel, we can navigate from different types of charts and event logs. There is also a text box and a group text box with information about the charts.

Figure 79 shows the initial state of the *Intrusion Detection Result* window. We can observe that the Pie chart and All events radiobuttons are selected. This means that the chart displays all events found in the database in a pie. Group Text box informs us that there are currently 38247 Security, 10889 System and 4715 Application events in the database. In fact, when you are in All events Pie Group Textbox will display all events found in the database grouped by event log.

We can navigate to Application events radio button. Now, the chart displays only application events found in the database in a pie. Group Textbox now contains only Application events grouped by eventid, as shown in figure 80.

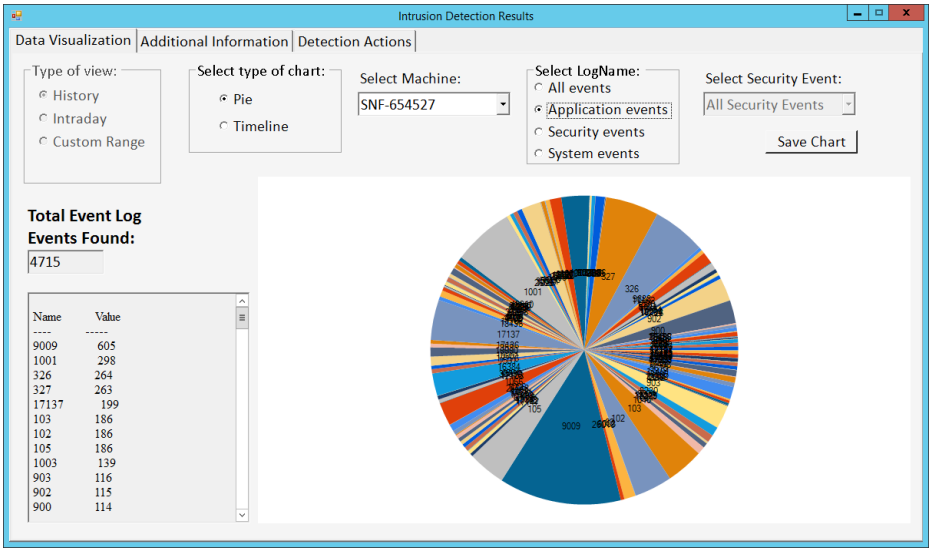


Figure 80. IDS Results from History (Pie - Application Events).

Figure 81 displays the *Intrusion Detection Result* window by choosing Security events RadioButton. The chart displays only security events found in the database in a pie. Group Textbox now contains only Security events grouped by eventid.

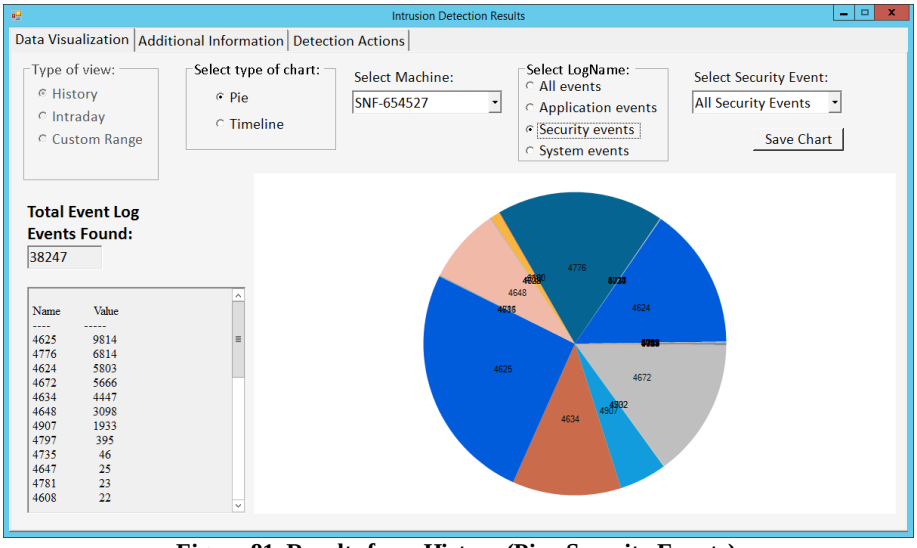


Figure 81. Results from History (Pie - Security Events).

Finally, we navigate to System events radio button and the chart only displays System events found in the database in a pie, as shown in figure 82. Group Textbox now contains only System events grouped by eventid.

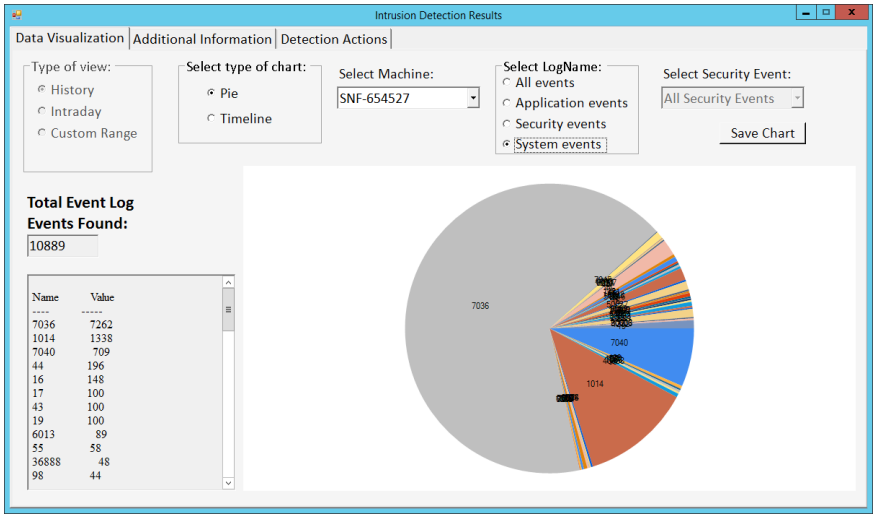


Figure 82. Results from History (Pie - System Events).

All of the available event log radiobuttons has been described. Now we are going to navigate to the Timeline radiobutton. We choose the Timeline and All event radio buttons and this will display a line chart, as shown in figure 83.

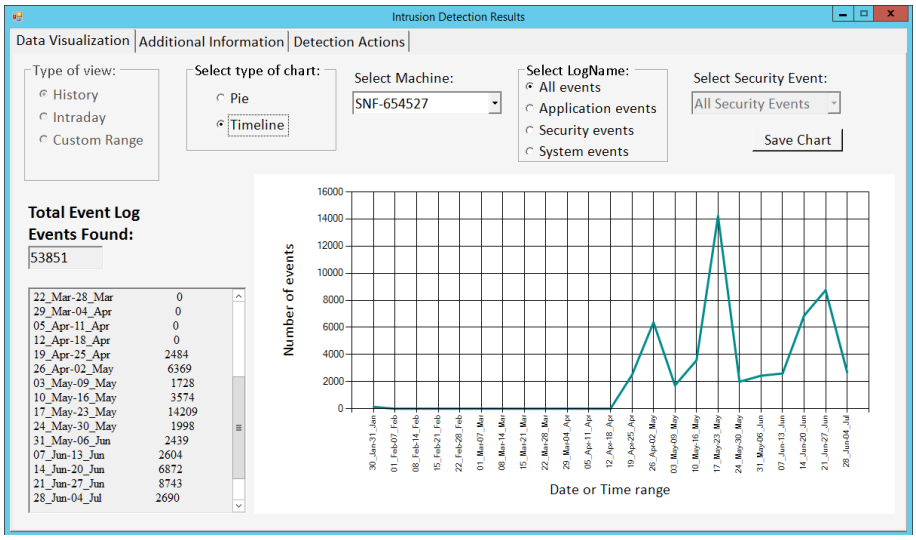


Figure 83. Results from History (Timeline - All events).

Timeline charts in the program contain a number of events on the vertical axis, and some time ranges on the horizontal axis. Time ranges may vary depending on the time distance we want to visualize. The implementation of the time ranges works as follows:

- If we want to get time ranges from a date that abstains from the current date less than 7 days, time ranges will be hourly separated.
- If we want to get time ranges from a date that abstains from the current date more than 7 days but less than 30 days, time ranges will be daily separated.
- If we want to get time ranges from a date that abstains from the current date more than 30 days, time ranges will be weekly separated *(as in figure 83)*.

Group Textbox now displays the time ranges and the number of events occurred.

We navigate to Application events radio button and we receive similar results, as shown in figure 84.

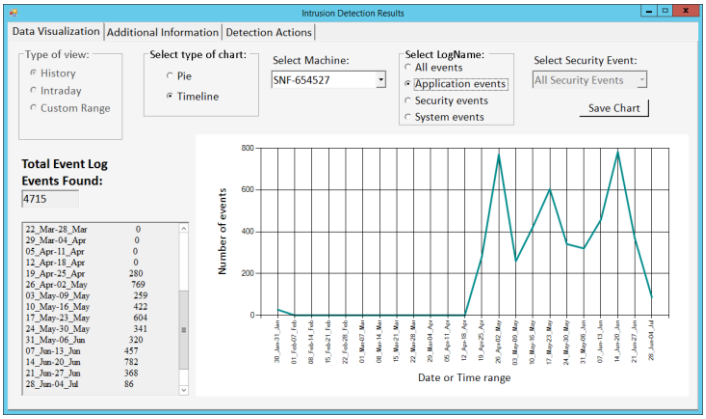


Figure 84. Results from History (Timeline - Application events).

We navigate to Security events radio button and we receive similar results, as shown in figure 85.

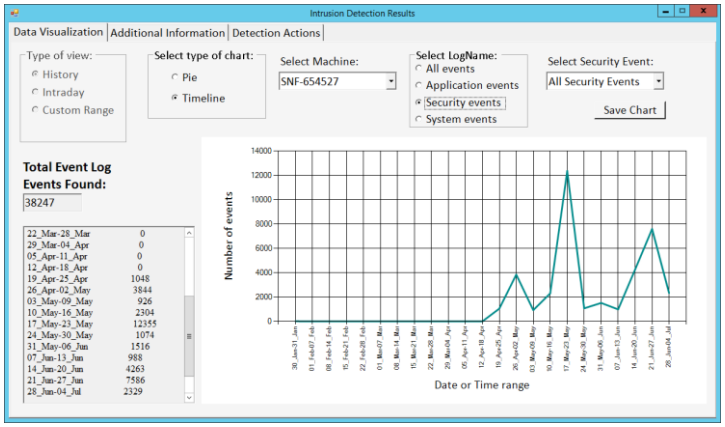


Figure 85. Results from History (Timeline - Security events).

Working with security timeline charts, we can detect incidents. Figure 86 displays the timeline chart for logon failure events. We can realize that from 17 May to 23 May detected almost 4000 failure attempts of gaining access of our system.

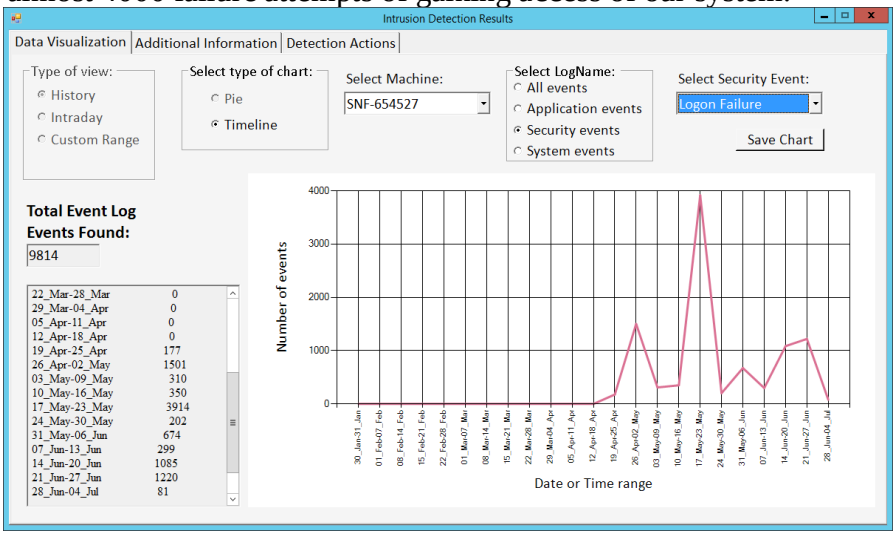


Figure 86. Results from History (Timeline - Logon Failure events).

We can receive similar crucial information by navigating to Log Success events, as shown in figure 87.

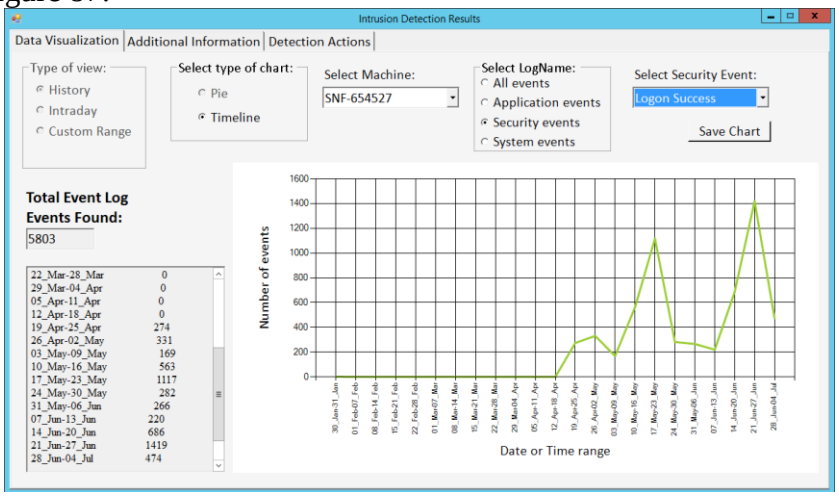


Figure 87. Results from History (Timeline – Logon Success events).

Finally, figure 88 displays the system events in a timeline.

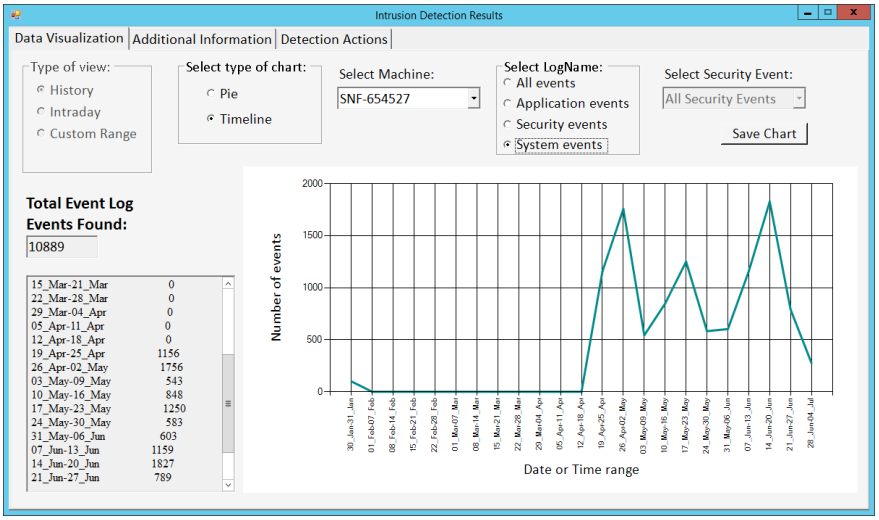


Figure 88. Results from History (Timeline - System events).

Observing the previous figures, we realize that there is a button called "Save Chart". This button converts and exports the current chart to an image file. Consider we have selected the Logon Failure radiobutton as a line *(figure 86)*. After hitting "Save Chart" a new folder will be created *(if it does not already exist)* in the folder that contain our executable file, as shown in figure 89.

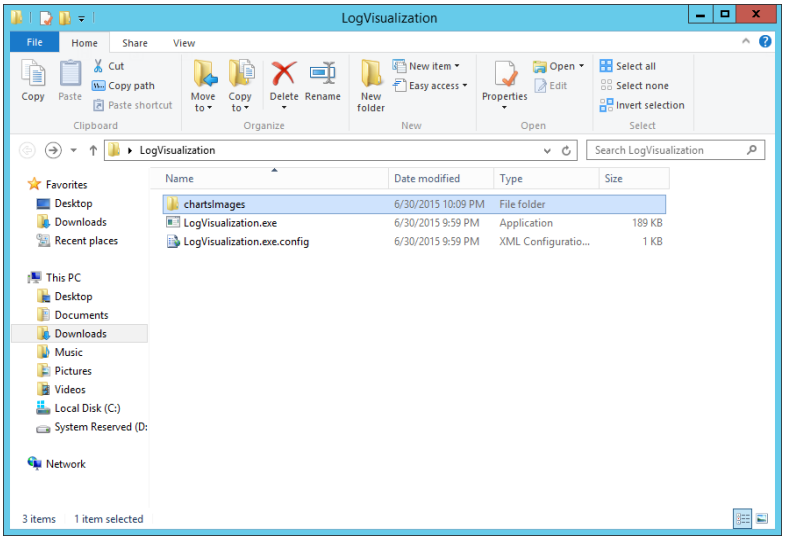


Figure 89. Automatically creates a folder for images.

Finally, the chartImages folder, will contain our selected chart with a unique and particular file name, as shown in figure 90.

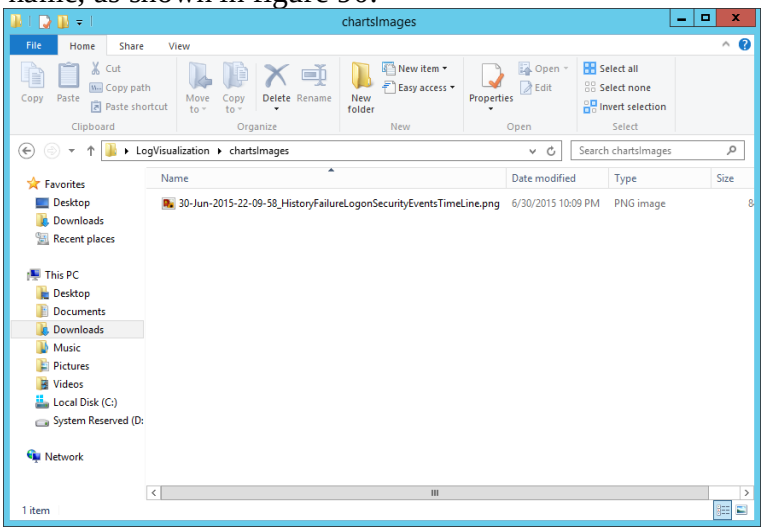


Figure 90. Export history failure logon timeline chart to image.

We can double click on the image file to display the image, as shown in figure 91.

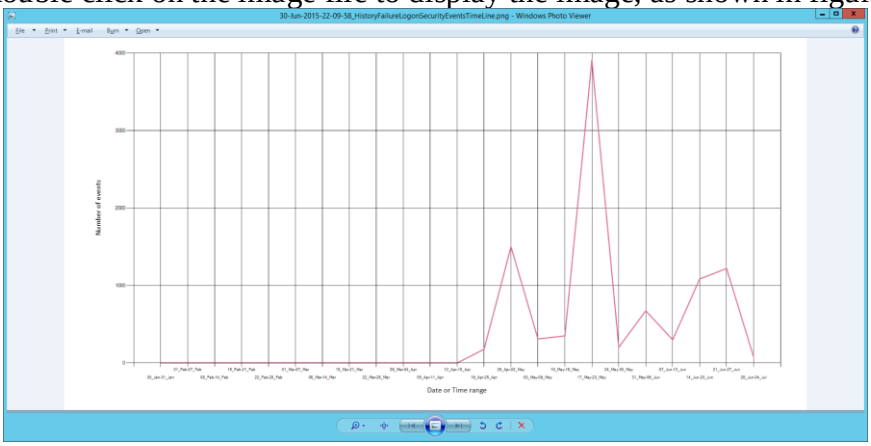


Figure 91. Exported timeline chart example.

The previous figures described the contents of the Data Visualization panel. We, now, navigate to Additional Information panel, as shown in figure 92.

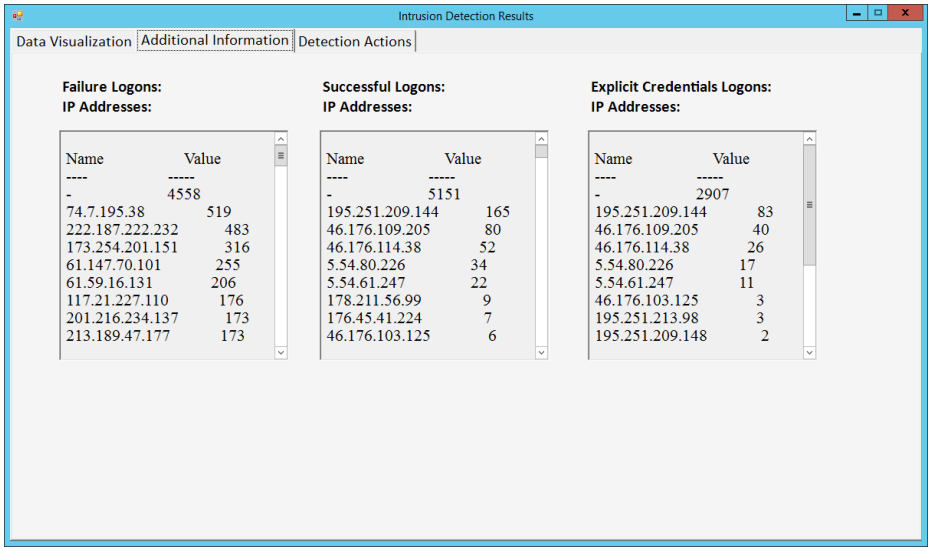


Figure 92. History - Additional Information panel

Figure 92 shows the Additional Information panel. This panel contain three different group text boxes where we view some critical information. At the first text box, we can view the IP addresses that attempted to gain access to our system but they failed. At the second text box, we can view the IP addresses that attempted to gain access to our system and they succeeded. Finally at the third text box, we can view the IP addresses that gained access to our system using explicit credentials.

The visualization results of the History use Case completed. In order to visualize Intraday events we are going to close and run again the program.

6.2 Intraday use case

Consider that we have chosen the Intraday RadioButton, as shown in figure 76. This will look at the database, find and analyze only the events that occurred the current date. Figure 93 provides the initial state of the IDS window with events occurred the current date.

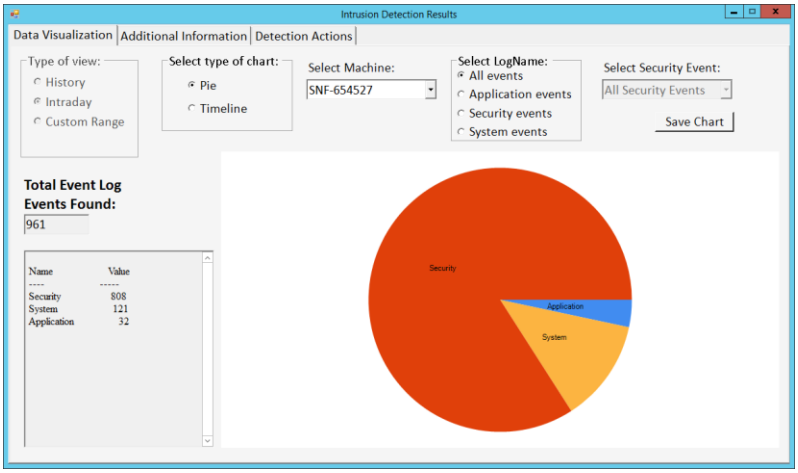


Figure 93. IDS Results from Intraday (Pie - All Events).

Figure 94 displays the Intraday Security Timeline results. We can observe that the chart displays time ranges per hour. We take a closer look at the Group Textbox and we realize that the highest point of the chart displays that between six and seven o'clock at 30 June, occurred 116 security events.

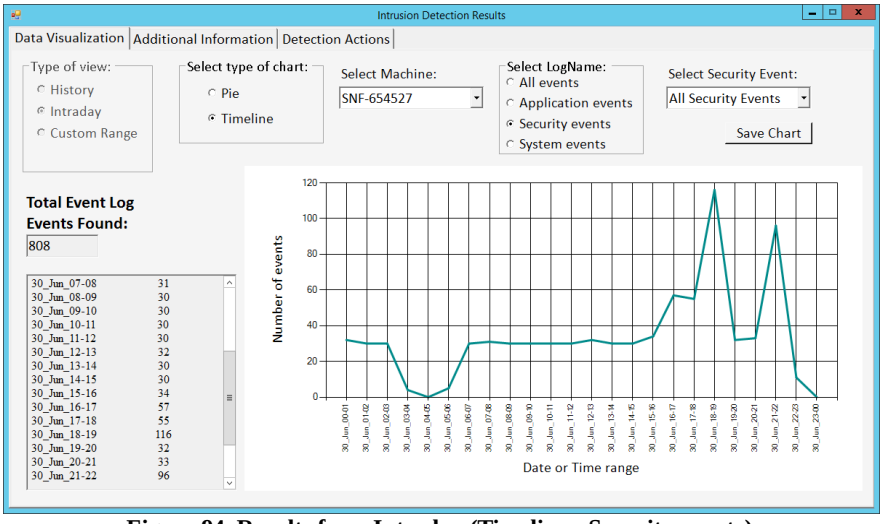


Figure 94. Results from Intraday (Timeline - Security events).

As we have already mentioned, working with security timeline charts, we can easily detect failure logon incidents. Figure 95 displays the timeline chart for logon failure events occurred the current date. We can realize that between 9 and 10 pm, 47 failed attempts of gaining access on our system occurred.

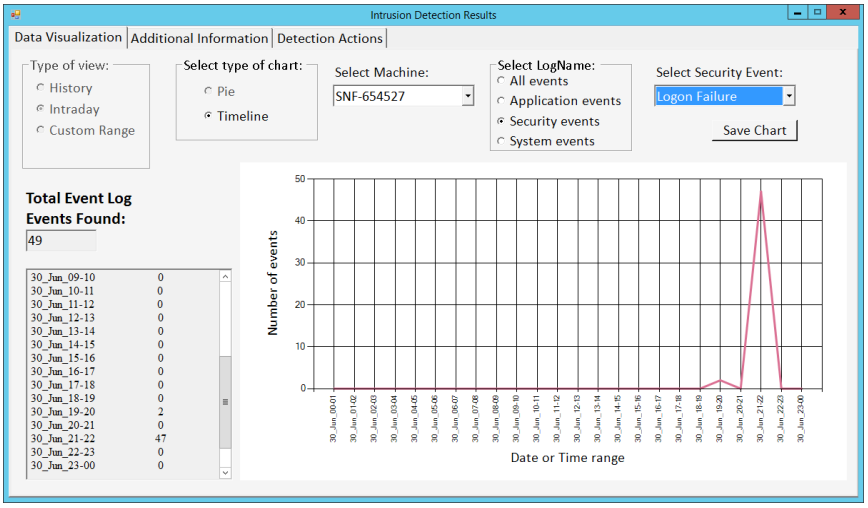


Figure 95. Results from Intraday (Timeline – Logon Failure events).

Navigating to Additional Information panel, we obtain critical information. In fact, we can see that these 47 failure logon events occurred from a specific IP address, as shown in figure 96.

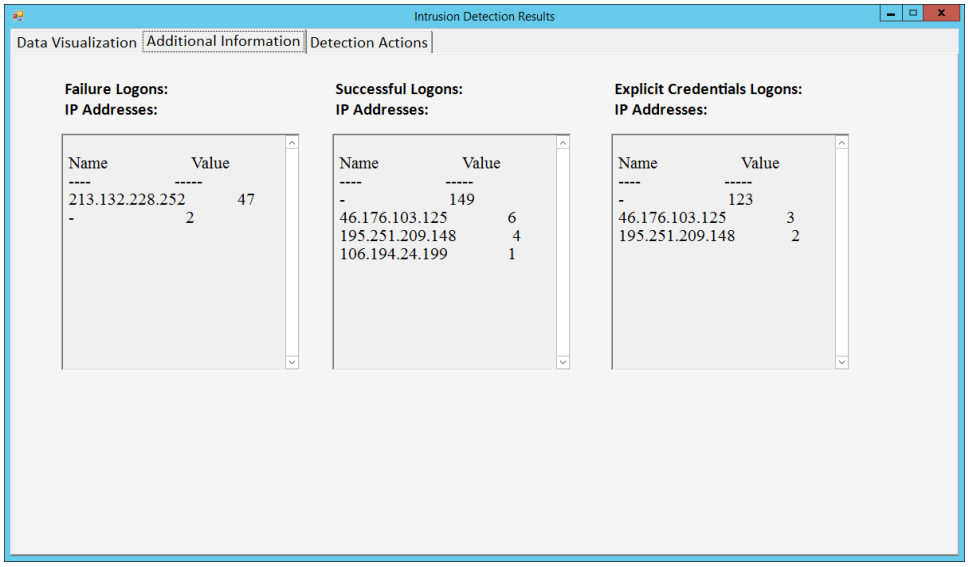


Figure 96. Intraday - Additional Information panel.

An example of the Intraday visualization completed. In order to visualize Custom Range events we are going to close and run again the program.

6.3 Custom Range use case

Consider that we have chosen the Custom Range RadioButton, as shown in figure 77. This is going to look at the database, find and analyze only the events that occurred after 06/01/2015. Figure 97 provides the initial state of the IDS window with events occurred after 06/01/2015. The chart displays all events found after the specific date in a pie. Group Text box displays all events found after the specific date grouped by event log.

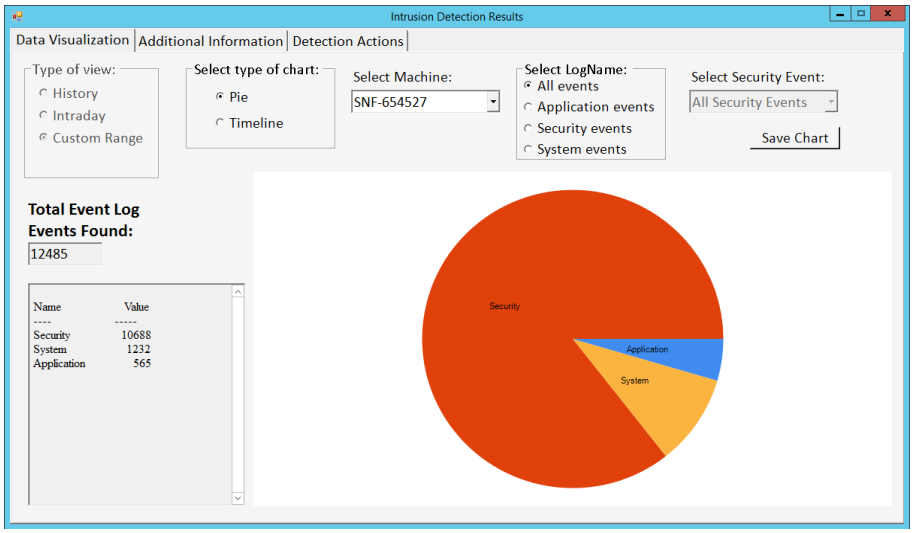


Figure 97. IDS Results from Custom Range (Pie - All Events).

We can navigate to Security events RadioButton to display only security events found in a pie, as shown in figure 98. Group Text box now contains only Security events that occurred after 06/01/2015 grouped by eventid.

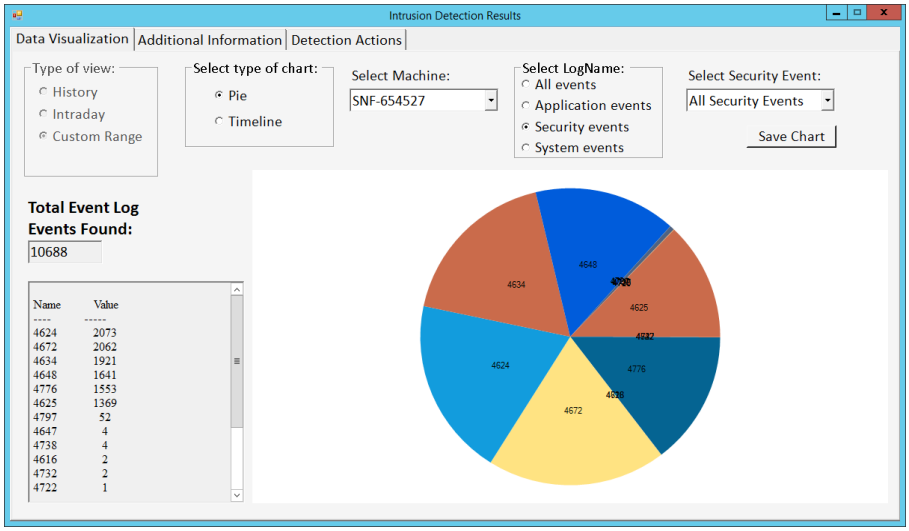


Figure 98. Results from Custom Range (Pie - Security Events).

We can navigate to System and Application RadioButton to display events as a pie. Now we are going to navigate to the Timeline RadioButton. We choose the Timeline and Security Logon Failure events and this will display a line chart, as shown in figure 99.

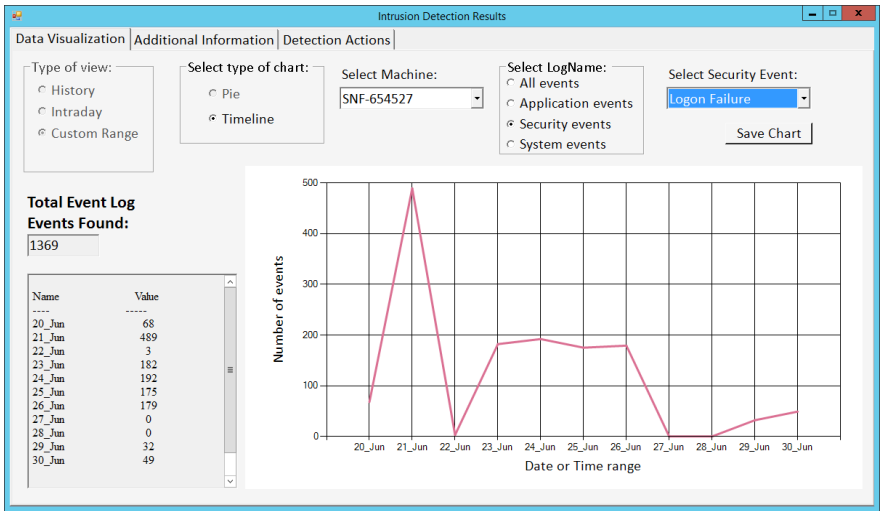


Figure 99. Results from Custom Range (Timeline - Security Logon Failure Events).

All we can see in figure 99 is that we have time ranges per day, and at 21 of June, 489 failed attempts of gaining access on our system occurred. We are going to save the current chart to an image file. After hitting "Save Chart", a new image file appears in the chartImages folder, as shown in figure 100.

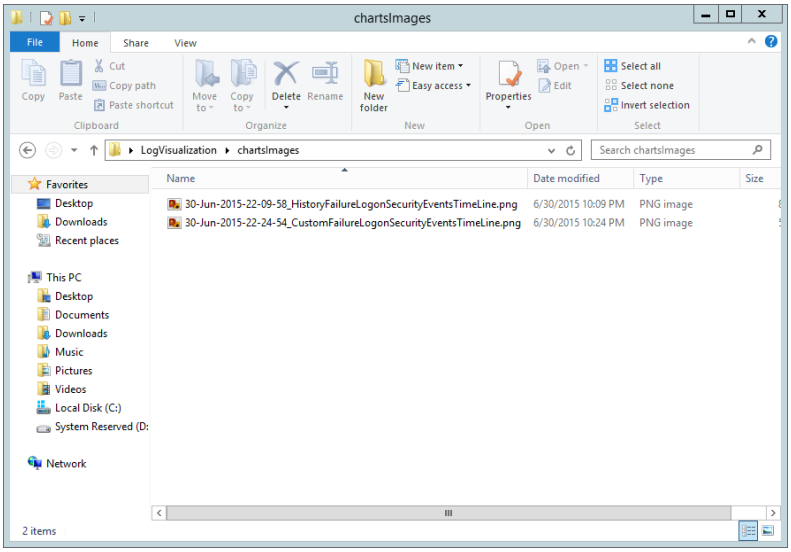


Figure 100. Export custom range failure logon timeline chart to image.

Navigating to Additional Information panel, we obtain critical information. In fact, we can see that 483 of 489 failure logon events occurred from a specific IP address as shown in figure 101.

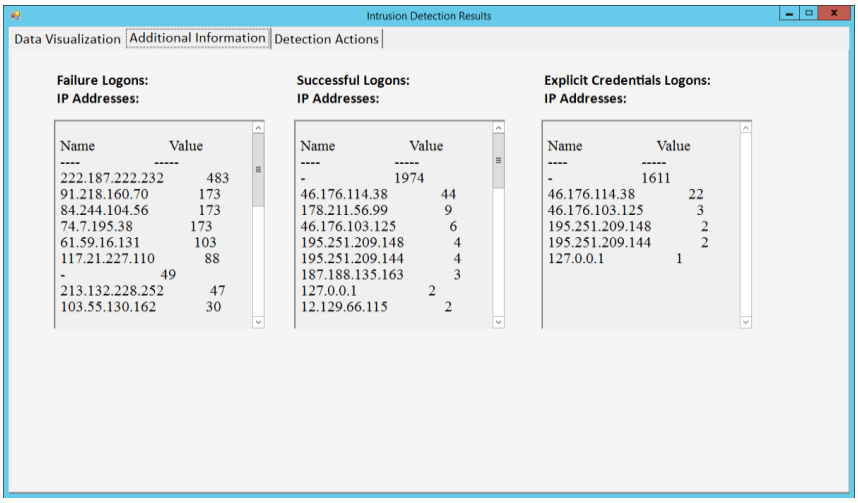


Figure 101. Custom Range - Additional Information panel.

6.4 Detection Actions

Detection Actions is the third panel of the IDS window. This panel let us do some administration and reporting. It provides a text area where we can type our SQL query and communicate with the database in real-time. In addition, it contains three buttons which enable us to get the table of our query, export the table to Csv file or to Htm file. Figure 102 shows the initial state of the Detection Actions panel.

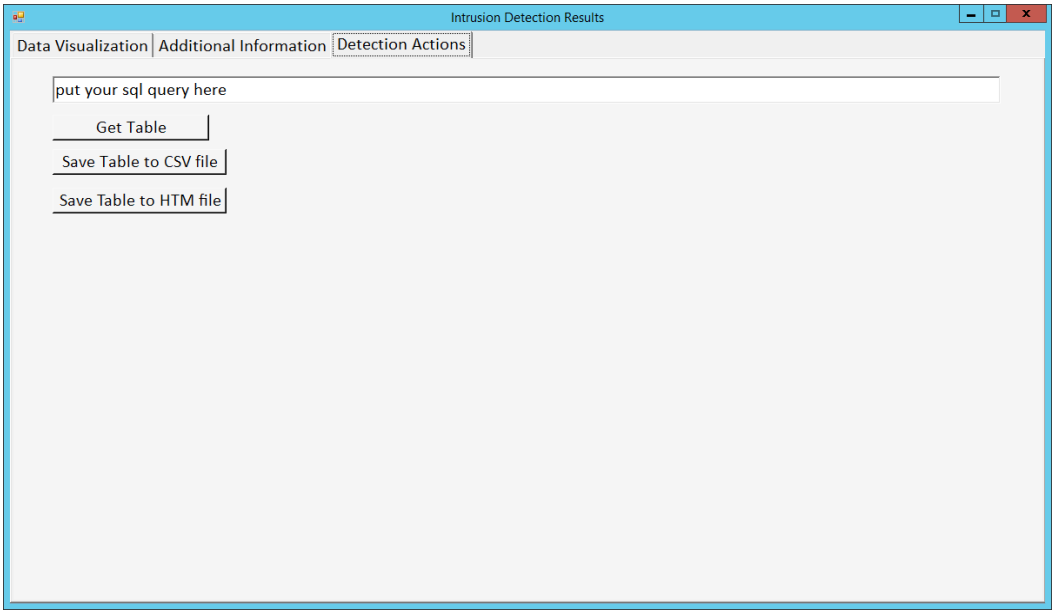


Figure 102. Detection Actions panel - initial state.

We proceed by typing a simple SQL query to get the latest 10 records from the table events, ordered by timecreated column descending, as shown in figure 103. We have constructed our query and we can now get the table, by hitting Get Table.

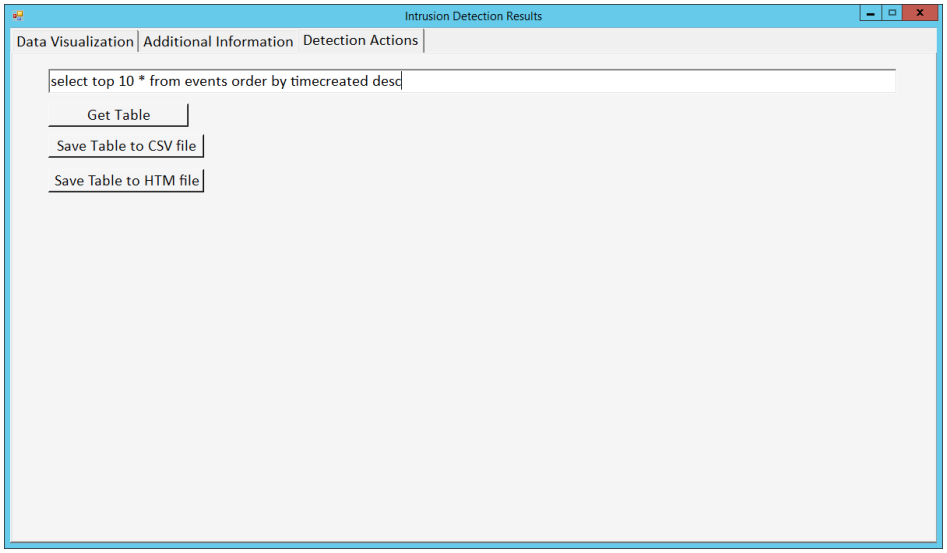


Figure 103. Detection Actions panel - simple query.

PowerShell is going to communicate with the database, receive a dataset of information and display a window with all information as a GridView panel, as shown in figure 104. All these with one line of code.

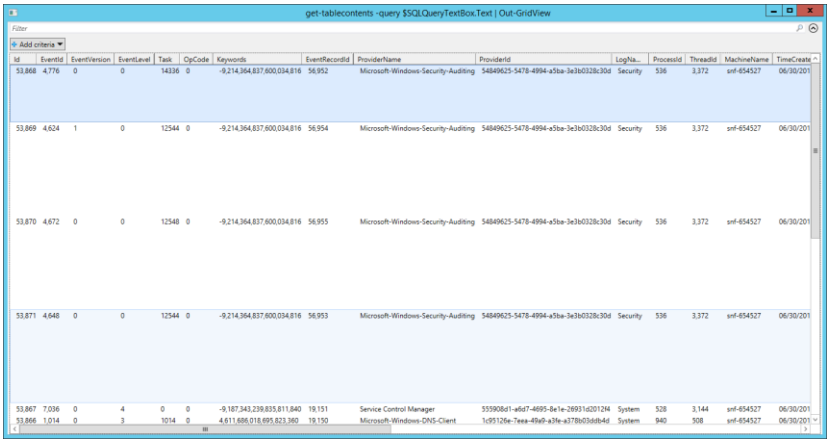


Figure 104. Getting simple query table using Out-GridView cmdlet.

Taking advantage of the ConvertTo and Export cmdlets that PowerShell provides, we can move a step forward and save all the data in a Csv or in an Htm file.

By hitting the Save Table to CSV file button, the program creates a new folder *(if it does not already exist)* for saving all of these files, as shown in figure 105.

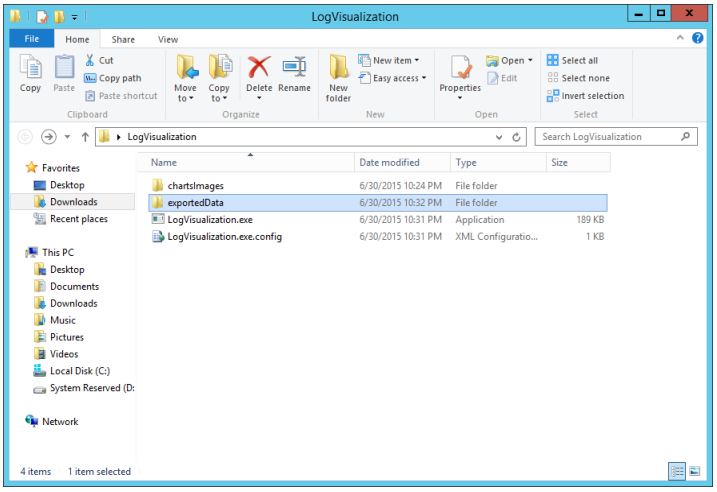


Figure 105. Automatically creates folder for exportedData.

Finally, the exportedData folder will contain the Csv file that we exported, as shown in figure 106.

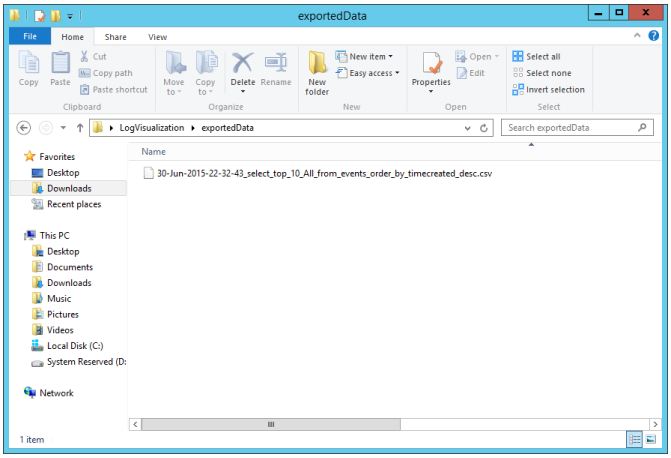


Figure 106. Export simple table to Csv.

As we can realize, each file we export, has a unique name because title is been generated to contain a snapshot of the current date and time. Furthermore, it consist of the snapshot followed by the query that generated this file.

Finally, by hitting the Save Table to HTM file button, the program is going to export the table data as an htm file within the exportedData folder, as shown in figure 107.

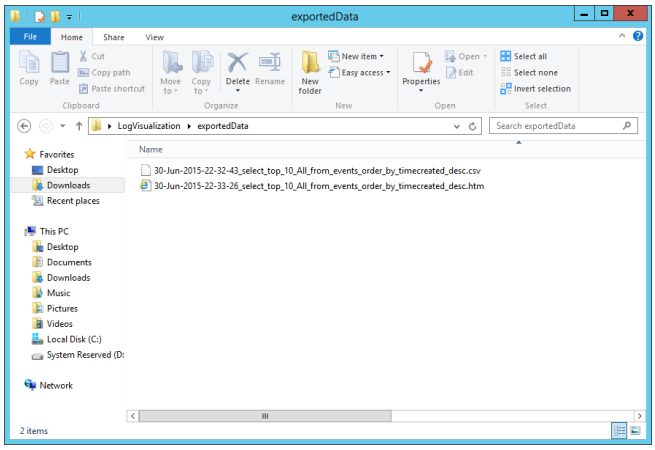


Figure 107. Export simple table to Htm.

6.5 Detecting & Reporting Example

In this section we are going to provide an example of detecting malicious incidents and making reports to be used as evidence.

As we have already observed in figure 99, numerous logon failure events was detected in specific date. Figure 108 shows again information for logon failure events occurred after 20 of June emphasizing on the 21 of June.

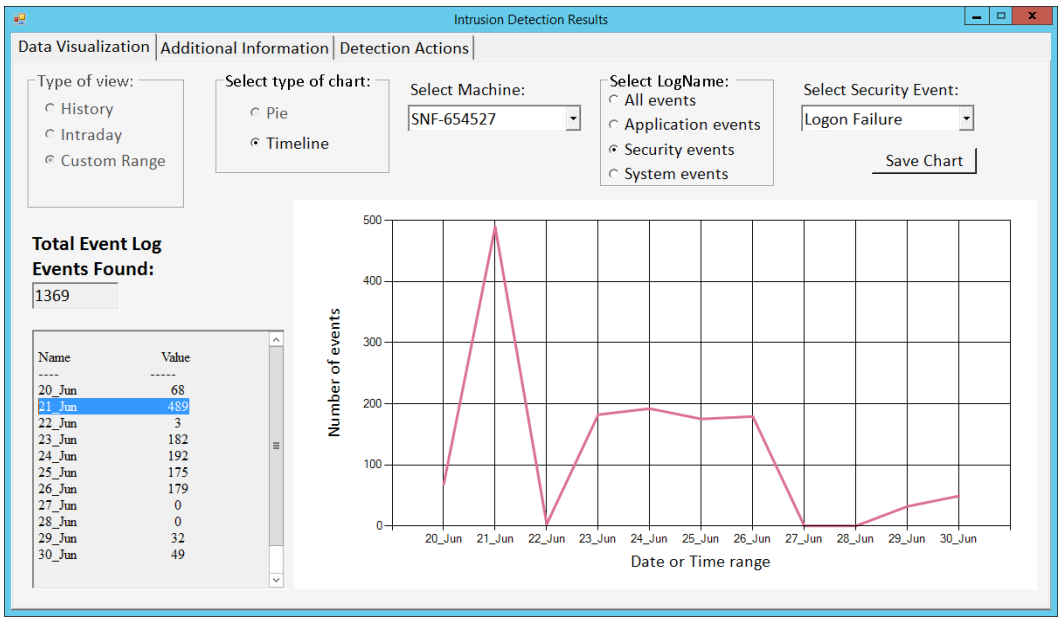


Figure 108. Numerous logon failure events in 21 of June.

As administrators, or security analysts, we should make reports that prove an anomalous detection. To achieve this we are going to construct a query to get the information we want. Figure 109 provides an example of using the Detection Actions panel, in order to export data to Csv or Htm file.

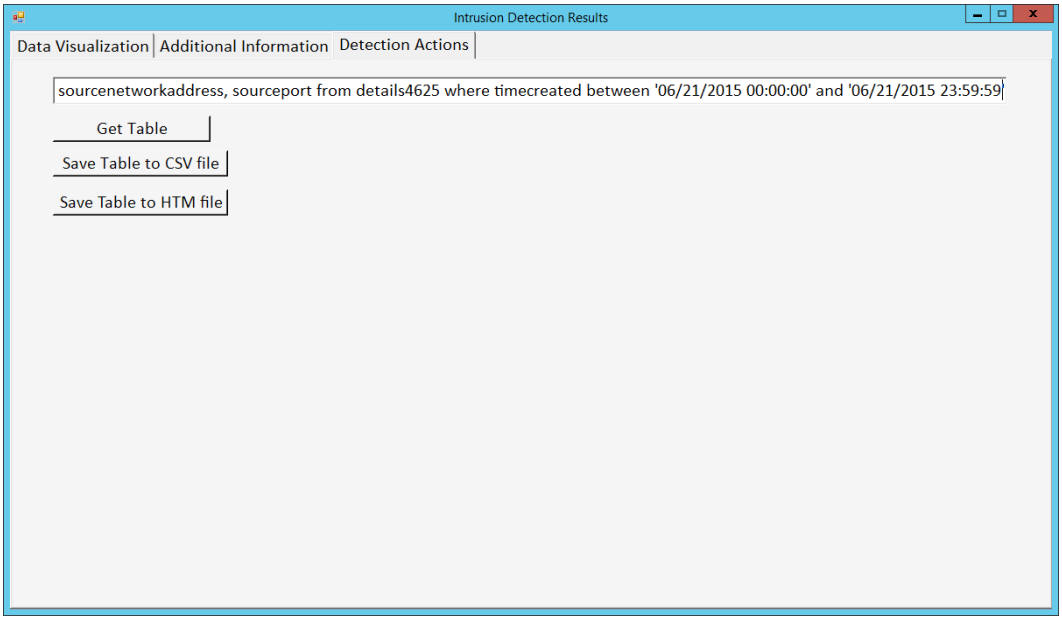


Figure 109. SQL query to get events that occurred on a specific date.

The SQL query that we constructed in figure 109, is been displayed as a table by hitting Get Table button, as shown in figure 110.

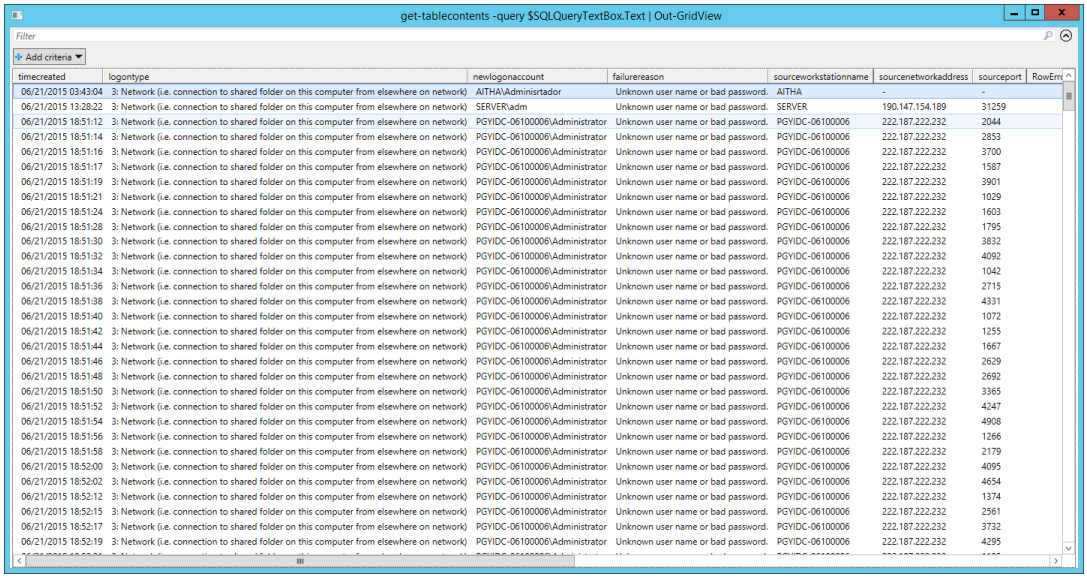


Figure 110. Getting a more complex query table using Out-GridView cmdlet.

As we can see in figure 110, in 21 of June, a particular IP address tried to break into our system every more than 400 times. In particular, after 6 o'clock pm it was attacking our system, every 2 seconds.

As we have detected some imminent threats we move forward to export all these information to Csv and Htm files. To do this we hit the Save Table to CSV file and the Save Table to HTM file buttons and we can see the files exported in the exportedData folder, as shown in figure 111.

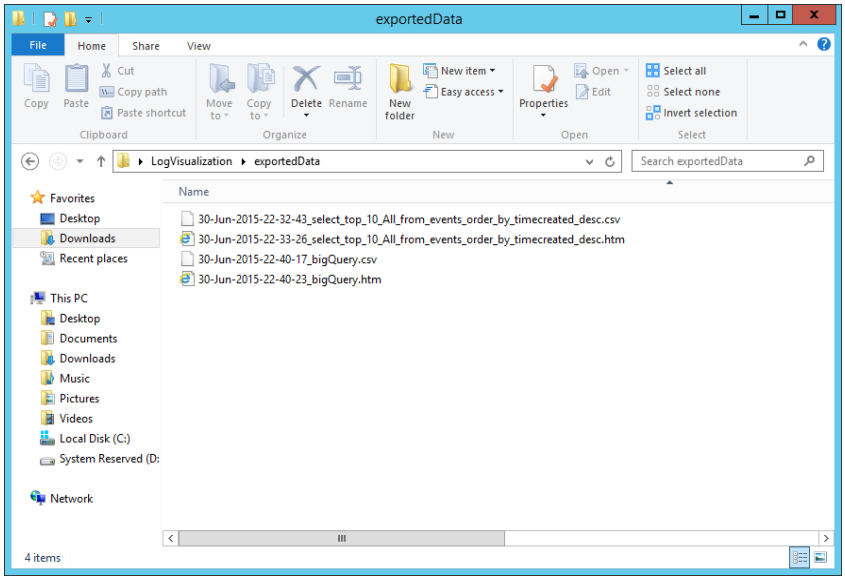


Figure 111. Export more complex table to CSV and HTM.

As we have already mentioned, the exported files take an automatically generated unique name. It consists of a snapshot of the current date and time followed by our SQL query. When the SQL query contains more than 256 characters it is being replaced by the text "bigQuery" and it needs custom modification.

After we made our reports we double click on the htm file and a browser automatically opens to display the file contents, as shown in figure 112. This file can be used as evidence.

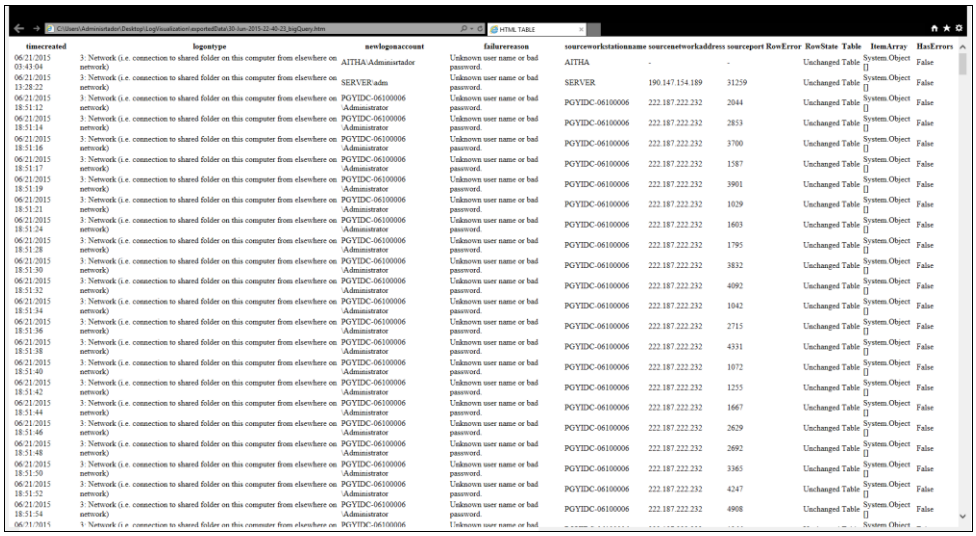


Figure 112. HTML Report to be used as an evidence.

6.6 Summary

Utilizing Windows PowerShell for Host-based IDS Log Monitoring achieved. A fully automated IDS system has developed. Users, or Advanced Users can run the executable of the LogVisualization.ps1 script to run the visualization procedure. They can select between three concepts:

- **History visualization** which analyzes all the data in the database.
- **Intraday visualization** which analyzes the data of the current date.
- **Custom Range** visualization which analyzes the data after a specific date that the user is concerned about.

Furthermore, the *Intrusion Detection Results* window consists of three panels:

- Data Visualization panel, where you can view timeline or pie charts.
- Additional Information panel, where you can view some critical information.
- Detection Actions panel, where you can interact with the database to get tables or exporting tables to a Csv or Htm file.

Chapter 7. Conclusions & Future Work

Windows Powershell is a tool installed by default in most versions of Windows OS. It is designed by Microsoft for purposes of system management and administration. PowerShell is the environment where you can think about what you want, you type it and you get it. It is easy to learn it and you can use it to deal with any kind of task.

PowerShell is a powerful tool where:

- You can deal with any kind of task.
    - As many times as you will be requested to.
        - Reducing repetitiveness by writing and using scripts and modules.
            - Giving always best results.
                - With one tool in use.

It was designed as an administrative language however it has tremendous capability in regards to Security scripting and monitoring.

Historically intrusion analysts have depended on tools for the identification and interpretation of this type of information, and there are lots of tools out there that will do this type of monitoring/analysis. They all are expensive, and for a small to medium sized shop that needs to monitor this information, Microsoft has provided the tool to monitor and extrapolate this type of information and it is called Windows PowerShell.

For future development this thesis would propose the following:

- Parsing more critical security events.
- Monitoring User Activities *(e.g. analyzing shell commands)*.
- Analyzing Execution of System Programs *(e.g. analyzing system calls)*.
- Extend capabilities to generate alerts, too.
- Compatible to work with Domain Controller *(getting events from multiple computers)*.
- Compatible to work with Linux System Hosts.
- Parameterize *Additional Actions* panel to show the available database tables and to constructs the query without the need of typing.
- Utilizing NoSQL DBMS technologies *(e.g. MongoDB)* for storing and analyzing event log data.

References

1. Michael J. Weeks: Intrusion Analysis Using Windows PowerShell. Abstract and Introduction. Copyright SANS Institute (2014) 1-2. Paper is available online at: [http://www.sans.org/reading-room/whitepapers/detection/intrusion-analysis-windows-powershell-34585]
2. Jack G. Albright: The basics of an IT Security Policy. What is an IT Security Policy and Summary. Copyright SANS Institute (2002) 3. Paper is available online at: [http://www.giac.org/paper/gsec/1863/basics-security-policy/103278]
3. Honan B. BH consulting: Layered Security, Protecting your data in today's threat landscape. InformationWeek.Com whitepaper (2011) 6. Paper is available online at: [http://www.informationweek.com/pdf_whitepapers/approved/1320416107_Tripwire_Layered_Security_white_paper.pdf]
4. Berge M., Ernst & Young: Security Resources, What is Intrusion Detection? Sans Institute (2011) Article is available online at: [http://www.sans.org/security-resources/idfaq/what_is_id.php]
5. Gupta S.: Logging and Monitoring to detect network intrusions and compliance violations in the environment. InfoSec Reading Room, Copyright SANS Institute (2012) 2. Paper is available online at: [http://www.sans.org/reading-room/whitepapers/logging/logging-monitoring-detect-network-intrusions-compliance-violations-environment-33985]
6. Mclaren C.: Layered Security – Inside and Out, Intrusion Detection Systems (IDS). Copyright SANS Institute (2003) 7. Paper is available online at: [http://www.giac.org/paper/gsec/2599/layered-security/104465]
7. J. P. Planquart: Application of neural networks to intrusion detection. Reading Room, Copyright SANS Institute 2-4. Paper is available online at: [http://www.sans.org/reading-room/whitepapers/detection/application-neural-networks-intrusion-detection-336]
8. R. G. Bace, "Intrusion Detection", Macmillan Technical Publishing (2000)
9. Microsoft.com-TechNet Library: Scripting with Windows PowerShell (2014), available online at: [https://technet.microsoft.com/en-us/library/bb978526.aspx]
10. Jones D., Hicks J,.: Learn Windows PowerShell 3 in a month of lunches. 2nd edn. Installing Windows PowerShell. Manning Shelter Island. Printed in the United States of America, (2013) 6-7. Book is available at: [http://www.szemtsov.net/books/learn_windows_powershell_3_in_a_month_of_lunches_2nd_edition.pdf]
11. Snover J., Helmick J.: Getting Started with PowerShell 3.0 Jump Start, Microsoft Virtual Academy Training Courses. Don't fear the shell epd. (2013) 07:00 – 13:00, video is available at: [https://www.microsoftvirtualacademy.com/en-US/training-courses/getting-started-with-powershell-3-0-jump-start-8276]
12. Jones D., Hicks J,.: Learn Windows PowerShell 3 in a month of lunches. 2nd edn. Why you can't afford to Ignore PowerShell, Life with PowerShell. Manning Shelter Island. Printed in the United States of America, (2013) 2
13. Wikipedia.com: Windows PowerShell (2015) Versions, available on-line at: [https://en.wikipedia.org/wiki/Windows_PowerShell].
14. Jones D., Hicks J,.: Learn Windows PowerShell 3 in a month of lunches. 2nd edn. Choose you weapon. Manning Shelter Island. Printed in the United States of

America, (2013) 9-11. Book is available at: [http://www.szemtsov.net/books/learn_windows_powershell_3_in_a_month_of_lunches_2nd_edition.pdf]

15. Microsoft.com-Download Center: Windows Management Framework 4.0. available for download (including instructions) at: [https://www.microsoft.com/en-us/download/details.aspx?id=40855] If the link has been expired or changed, you can search for it in the Download Center of Microsoft, here: [https://www.microsoft.com/en-us/download/default.aspx]
16. Snover J., Helmick J.: Getting Started with PowerShell 3.0 Jump Start, Microsoft Virtual Academy Training Courses. The help system epd., The help system, Get-Help, Command Syntax, (2013) 00:17 – 54:00, video is available at: [https://www.microsoftvirtualacademy.com/en-US/training-courses/getting-started-with-powershell-3-0-jump-start-8276]
17. Jones D., Hicks J,.: Learn Windows PowerShell 3 in a month of lunches. 2nd edn. Aliases: nicknames for commands. Manning Shelter Island. Printed in the United States of America, (2013) 39-40. Book is available at: [http://www.szemtsov.net/books/learn_windows_powershell_3_in_a_month_of_lunches_2nd_edition.pdf]
18. Jones D., Hicks J,.: Learn Windows PowerShell 3 in a month of lunches. 2nd edn. What are providers. Manning Shelter Island. Printed in the United States of America, (2013) 49-50. Book is available at: [http://www.szemtsov.net/books/learn_windows_powershell_3_in_a_month_of_lunches_2nd_edition.pdf]
19. Windows PowerShell Help System, Conceptual help topic: about_modules. You get is by typing: "Get-Help about_modules"
20. Snover J., Helmick J.: Getting Started with PowerShell 3.0 Jump Start, Microsoft Virtual Academy Training Courses. The pipeline epd., (2013) 00:17 – 27:00, video is available at: [https://www.microsoftvirtualacademy.com/en-US/training-courses/getting-started-with-powershell-3-0-jump-start-8276]
21. Jones D., Hicks J,.: Learn Windows PowerShell 3 in a month of lunches. 2nd edn. What are objects. Manning Shelter Island. Printed in the United States of America, (2013) 85-92. Book is available at: [http://www.szemtsov.net/books/learn_windows_powershell_3_in_a_month_of_lunches_2nd_edition.pdf]
22. Jones D., Hicks J,.: Learn Windows PowerShell 3 in a month of lunches. 2nd edn. Comparison operators, Filter Left. Manning Shelter Island. Printed in the United States of America, (2013) 135. Book is available at: [http://www.szemtsov.net/books/learn_windows_powershell_3_in_a_month_of_lunches_2nd_edition.pdf]
23. Windows PowerShell Help System, Conceptual help topic: about_wmi. You get is by typing: "Get-Help about_wmi"
24. Snover J., Helmick J.: Getting Started with PowerShell 3.0 Jump Start, Microsoft Virtual Academy Training Courses. The pipeline epd., (2013) 05:00 – 07:00, video is available at: [https://www.microsoftvirtualacademy.com/en-US/training-courses/getting-started-with-powershell-3-0-jump-start-8276]
25. Jones D., Hicks J,.: Learn Windows PowerShell 3 in a month of lunches. 2nd edn. The pipeline: connecting commands, Using cmdlets that modify the system.

Manning Shelter Island. Printed in the United States of America, (2013) 70. Book is available at: [http://www.szemtsov.net/books/learn_windows_powershell_3_in_a_month_of_lunches_2nd_edition.pdf]
26. Jones D., Hicks J,.: Learn PowerShell Toolmaking in a month of lunches. What is Toolmaking, Manning Shelter Island. Printed in the United States of America, (2013) 3-4.
27. Windows PowerShell Help System, Conceptual help topic: about_windows_powershell_ise. You get is by typing: "Get-Help about_windows_powershell_ise"
28. Windows PowerShell Help System, Conceptual help topic: about_functions. You get is by typing: "Get-Help about_functions"
29. Holmes L.: Windows PowerShell Cookbook. 3 edn. Security and Script Signing. O'Reilly Media, Printed in the United States of America, (2012) 515
30. Johnson M.: PowerShell Security – On the box PowerShell Security. (2012). Article is available online at: [http://www.poshsec.com/2012/10/on-the-box-powershell-security]
31. Windows PowerShell Help System, Command help topic: "Get-EventLog". You get is by typing: "Get-Help Get-EventLog".
32. Windows PowerShell Help System, Command help topic, Notes: "Get-WinEvent". You get is by typing: "Get-Help Get-WinEvent -Full".
33. Holmes L.: Windows PowerShell Cookbook. 3 edn. Security and Script Signing. O'Reilly Media, Printed in the United States of America, (2012) 633
34. Microsoft.com – TechNet Gallery, P2EXE: Convert PowerShell Scripts to EXE Files (2015). Script Available online at: [https://gallery.technet.microsoft.com/PS2EXE-Convert-PowerShell-9e4e07f1]
35. Microsoft.com-Download Center: Microsoft SQL Server 2012 Express. available for download (including instructions) at: [https://www.microsoft.com/en-us/download/details.aspx?id=29062]
36. Jones D., Hicks J,.: Learn PowerShell Toolmaking in a month of lunches. Adding database access, Manning Shelter Island. Printed in the United States of America, (2013) 144-145.
37. GitHub.com – Utilizing Windows PowerShell for Host-Based IDS Log Monitoring: GreekIT Code Repository. Available online at: [https://github.com/greekit/PowerShell]
38. List of powershell books. Available online at: [http://www.hofferle.com/list-of-powershell-books/]