PowerDecode: a PowerShell Script Decoder Dedicated to Malware Analysis

Giuseppe Mario ***Malandrone*** [1], *Giovanni* ***Virdis*** [2], *Giorgio* ***Giacinto***[3], *Davide* ***Maiorca*** [4]

[1] *Ufficio Sicurezza Informatica, Numera Sistemi e Informatica S.p.A*
[2] *Ufficio Sicurezza Informatica, Numera Sistemi e Informatica S.p.A*
[3] *Department of Electrical and Electronic Engineering, University of Cagliari*
[4] *Department of Electrical and Electronic Engineering, University of Cagliari*

Abstract

In recent years, PowerShell-based attacks have been widely employed to compromise systems' security. Attackers can easily hide such malicious scripts in file formats (e.g., Office document macros) that can be easily delivered via large-scale spam mail campaigns. Moreover, attackers employ obfuscation techniques that make the PowerShell code able to evade the most common anti-malware protections and perform unauthorized actions that will target the confidentiality, integrity and availability of an information system. In this paper, we present PowerDecode, an open-source module for the de-obfuscation and the analysis of PowerShell scripts. In particular, this module receives a script as an input and returns its obfuscated layers, its original de-obfuscated variant and a report about possible malicious activities. We tested PowerDecode on almost 3000 malicious scripts and the attained results showed significantly improved de-obfuscation performances in comparison to state-of-the-art systems. More specifically, PowerDecode was able to resolve multiple types of obfuscation and collect important information about attacks, such as malicious URLs and IP addresses contacted by malware. Finally, PowerDecode can be easily integrated in other malware analysis systems, and can represent a precious aid to identify malicious activities.

Keywords

PowerShell, Malware, Obfuscation

1. Introduction

Most important antimalware software companies, identified a large number of cyberattacks based on the exploitation of PowerShell features. These attacks employ a technique defined as "living off the land", which consist of exploiting a legitimate tool in the victim's operating system for malicious purposes. A reason why cybercriminals prefer this attack mode is essentially due to the ability of PowerShell to launch commands in a hidden way which load machine code instructions directly into

ITASEC, April 2021
EMAIL: gmalandrone@numera.it (G.Malandrone); giovanni.virdis@numera.it (G.Virdis); giacinto@unica.it (G.Giacinto); davide.maiorca@unica.it (D.Maiorca)

© 2020 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

memory or establish a connection to a remote server. PowerShell is a preferred attack vector also due to the supported scripting language, which can be easily obfuscated. Obfuscation is a widely used technique to circumvent the most common signature-based antimalware protections [14], making the malicious code difficult to detect. In 2016, the Symantec Blue Coat Malware Analysis Sandbox, analyzed 49127 PowerShell scripts and observed that 95.4% of these scripts were malicious, in addition, from 4782 samples analyzed manually, 111 different types of malware were identified. Based on statistic carried out by Symantec, the year 2016 saw a sudden increase in attacks based on PowerShell scripts. It was observed that attackers used to embed PowerShell scripts in Word file macros, and sent them as attachments in spam mails. The opening of the document by the victim should have run a PowerShell script in hidden mode, starting the attack [1].The years after 2016 saw a further increase in the use of PowerShell. In fact, according to the report published by McAfee Labs about the most widespread web threats in 2019, PowerShell, compared to the previous year, showed a 460% increase in use as an attack vector to compromise a remote system [2]. In the year 2020, due to the health emergency caused by COVID-19, the spread of PowerShell malware increased further. Indeed, as observed by McAfee in the report published in November 2020, the global impact of COVID-19 has prompted cybercriminals to adapt their cybercrime campaigns to attract victims with pandemic themes and exploit the realities of a workforce working for home and significant proliferation of Microsoft malicious attacks on Office documents pushed new PowerShell malware to rise 117% [3]. PowerShell-based attacks are still a complex issue, especially due to code obfuscation. In fact, to know the extent of these attacks, it is often necessary to perform code de-obfuscation and dynamic analysis. The current state of the art offers various open-source tools dedicated to this purpose [4], [5], [6], [7], however these tools, as will be shown, have some algorithmic flaws that do not always allow the correct analysis of the malware. PowerDecode aims to fill this gap. The implemented de-obfuscation algorithm based on an accurate model of obfuscated code, allowed to de-obfuscate and analyze a large number of scripts with which other pre-existing tools failed. The PowerDecode module is currently available as open-source software on GitHub [21], [22]. The rest of the paper is organized as follows: Section 2 provides a description of the main features of PowerShell including scripting language and malware concept. Section 3 provides a classification of the main types of obfuscation achievable on PowerShell. Section 4 provides an overview of the related work in the field. Section 5 describes the features of the proposed system PowerDecode. Section 6 discusses the results of evaluation. Section 7 closes the paper.

2. Background

PowerShell is an object-oriented command interpreter developed by Microsoft, and it is present on all Windows-based operating systems, starting from Windows XP. The shell is based on the .NET Common Language Runtime (CLR), and accepts and returns .NET objects [8]. PowerShell has been designed for the following purposes:

- File system management and configuration;
- Programming using scripting language;
- Management of registry keys.

In this section we give an overview of supported shell commands and we define the concept of PowerShell malware.

2.1. Cmdlets

Cmdlets are characteristic PowerShell commands, which allow for interactions between users and shells. Their syntactic structure follows specific nomenclature rules, as they are composed of a verb and a noun separated by a hyphen. PowerShell offers the possibility to invoke a cmdlet using an alias for easier typing. A set of aliases is defined as default setting, but users can also define new aliases to associate them with a given cmdlet or change the syntax of an existing alias. As PowerShell is an object-oriented programming language, it allows to treat cmdlets as methods that can receive as input

(or return) objects, and that can also be overridden. The most relevant cmdlets employed in the context of this work are showed on Table 8 in Appendix A.

2.2. PowerShell Malware

Although scripting-based languages are typically employed for benign purposes, they can also be exploited for malicious purposes. For this reason PowerShell, supports a script execution policy. As a default setting, the execution of scripts is disabled. Hence, if the user wants to run a script, he must explicitly enable its execution. However, this security setting has proved to be ineffective [16]. Various ways have been identified to execute scripts regardless of the lock imposed by the execution policy [9]. For this reason, attackers may easily execute PowerShell malwares [15]. We distinguish between two types of malicious attacks: file-based and file-less.

```
(new-object System.net.webclient).downloadfile(
'http://MaliciousUrl.com\malware.exe', 'file.exe');
Start-process 'file.exe'
```

Listing 1.1: An example of PowerShell file-based malware

Listing 1.1 shows an example of file-based malware. This code establishes a connection to a URL and downloads a payload (an executable malicious file). Then, it runs the downloaded payload.

```
$c = @"
[DllImport("kernel32.dll")] public static extern IntPtr VirtualAlloc(IntPtr w, uint x, uint y,
uint z);
[DllImport("kernel32.dll")] public static extern IntPtr CreateThread(IntPtr u, uint v, IntPtr w,
IntPtr x, uint y, IntPtr z);
[DllImport("msvcrt.dll")] public static extern IntPtr memset(IntPtr x, uint y, uint z);
[DllImport("kernel32.dll")] public static extern bool VirtualProtect(IntPtr lpAddress, uint
dwSize, uint flNewProtect, out uint lpflOldProtect);
"@
$o = Add-Type -memberDefinition $c -Name "Win32" -namespace Win32Functions -passthru
$x=$o::VirtualAlloc(0,0x1000,0x3000,0x04);
[Byte[]]$sc = 0xfc,0xe8,[truncated] 0xd5;
for ($i=0;$i -le ($sc.Length-1);$i++) {$o::memset([IntPtr]($x.ToInt32()+$i), $sc[$i], 1) | out-
null;}
$oldprotect = 0;
$here=$o::VirtualProtect($x, [UInt32]0x1000, [UInt32]0x20, [Ref]$oldprotect);
$z=$o::CreateThread(0,0,$x,0,0,0);
```

Listing 1.2: An example of PowerShell file-less malware

Listing 1.2 shows an example of file-less malware. This code first imports the *kernel32.dll* and *msvcrt.dll* libraries. Then, it declares a hexadecimal values array, which represents assembly instructions (shellcode). Finally, a thread is created within a PowerShell process and the shellcode is injected into this thread.

File-based malware requires the creation of a new file on the victim's storage device. This aspect makes such attacks easier to detect by anti-malware engines. In addition, contacted URLs might be recognized as malicious, by checking for their presence in a blacklist. Unlike the latter, file-less malware does not need to create new files, as the payload is embedded in the code in the form of hexadecimal instructions. All actions performed by file-less malware appear to be executed by the legitimate “Powershell.exe” process. However, over the years, anti-malware software companies have detected and analyzed numerous PowerShell attacks, obtaining relevant information to creating malware signatures with which it is possible to recognize even some file-less malware [10].

3. PowerShell Obfuscation

To evade the most common anti-malware protection measures, attackers usually employ several code obfuscation techniques that aim to make the code hard to understand both for the anti-malware programs and the human users. Formally, obfuscation can be defined as the alteration of the code syntax, which however keeps the semantics unchanged. Although there are infinite ways to obfuscate a given code, the applicable techniques, according to the taxonomy proposed by Bohannon [11], [12] can be classified into five different types:

- String-based: in this case, the code is manipulated as a string, applying related operations as concatenating, reordering, reversing or substring replacing. The resulting code, to be executed, must be evaluated by the Invoke-Expression cmdlet or "&" evaluation operator.
- Base64: it consists in the application of the base64 encoding standard. The resulting code, to be executed, must be passed as input to the shell preceded by the "powershell" function call and the flag "-e".
- Encoded: this obfuscation type is performed by converting each individual character into the matching character of a column on the ASCII table [13] or by applying a cryptographic algorithm. The resulting code, to be executed, must be evaluated by the Invoke-Expression cmdlet.
- Compressed: it consists of the application of a PowerShell supported data compression algorithm [8]. Resulting code, to be executed must be evaluated by the Invoke-Expression cmdlet.
- Randomization: it is a weak obfuscation form that consists of randomly inserting uppercase characters, space characters, or symbols not interpreted by the shell [17].

Table 9 in Appendix B provides an example for each obfuscation type described.
PowerShell scripting language allows to apply different obfuscation techniques recursively to the same script. In this way, the resulting code could contain multiple obfuscation layer, but only the first layer (last obfuscation type applied) can be seen. Listing 1.3, 1.4, 1.5, 1.6 on Appendix C, show an example of a multi-layer obfuscated script. There are several open-source customized obfuscators available on the Internet [11]. These tools are able to generate obfuscated code by changing its syntax randomly, often making it very difficult to understand the sequence of transformations performed on the original code.

4. Related Work

In the current state of the art there are different open-source tools dedicated to the de-obfuscation of PowerShell malwares. In this paper we mention PSDecode [6], [7] and PowerDrive [4], [5]. They both perform de-obfuscation using two different techniques:

- Invoke-Expression cmdlet overriding: as seen above, a wide variety of obfuscations rely on the dependency on the Invoke-Expression cmdlet. By overriding this cmdlet it is possible to force the script execution to return the string it was trying to convert into a statement.
- Regular expressions: this technique consists of assuming common patterns that occur in string obfuscation. These patterns are detected in the code and removed. In this way it is possible to reconstruct the original script.

However, these tools do not employ these techniques optimally, making it impossible in some cases to resolve certain types of obfuscation such as string-based format applied into multiple layers. These limitations will be discussed in detail in Section 6.

5. Introducing PowerDecode

PowerDecode is an innovative tool dedicated to de-obfuscate PowerShell scripts, which are typically obfuscated across multiple layers. Similarly to previously proposed tools it performs cmdlet overriding and regular expressions techniques. The PowerDecode de-obfuscation algorithm is based on an accurate model of obfuscation, ideally represented by a unary syntax tree. Due implicit knowledge of this data structure, PowerDecode is able to solve all obfuscations generable by Invoke-Obfuscation [11]. All result obtained following the analysis are saved on a text report file.

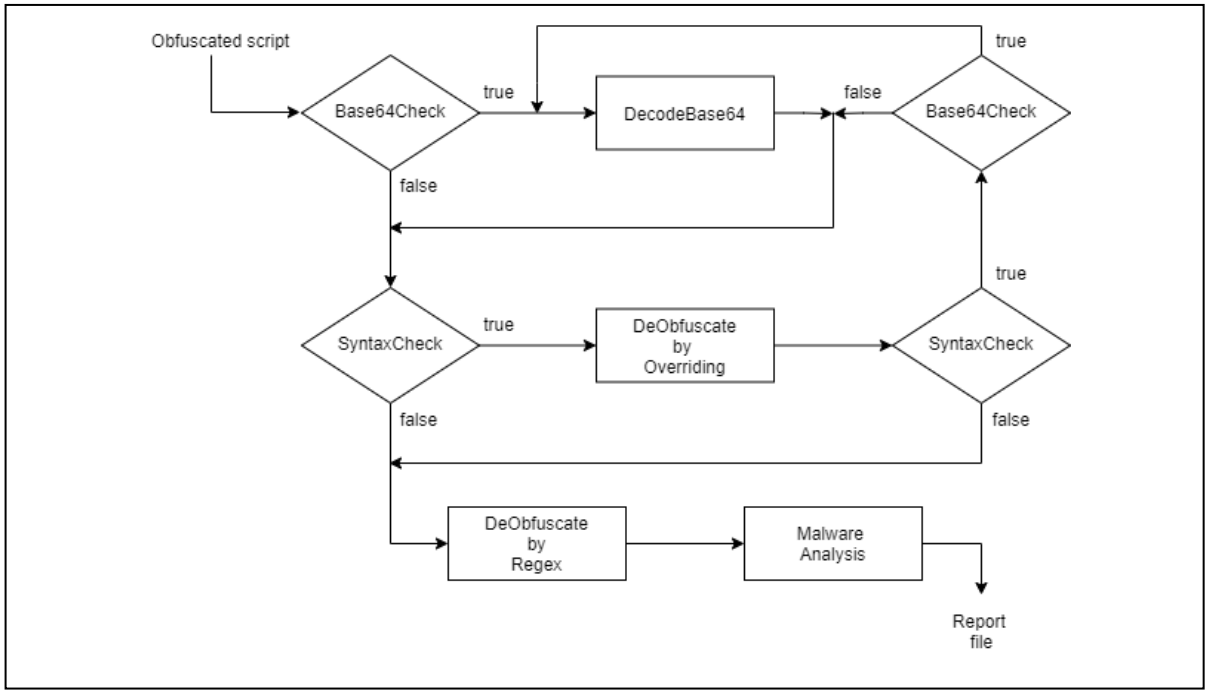

Figure 1: PowerDecode operation scheme

5.1. Operation Scheme

PowerDecode operation scheme is showed in Figure 1. The system receives as input a text file from which extract the code to de-obfuscate. The de-obfuscation process, takes place according the following algorithm:

1. *Base64Check*: if the code contains base64 encoding store this layer and go to the next step, otherwise skip to step 3;
2. *DecodeBase6*4: remove base64 encoding;
3. *SyntaxCheck*: if the syntax of the resulting code from the previous step is correct, store this layer and go to the next step, otherwise skip to the step 6;
4. *DeobfuscatebyOverriding*: remove the current obfuscation layer by cmdlet overriding;
5. *SyntaxCheck*: if the code syntax resulting from the previous step is correct, go back to step 1, otherwise go to the next step;
6. *DeobfuscatebyRegex*: consider the last stored layer and de-obfuscate it by applying regular expressions to remove obfuscation residuals. If the resulting code has changed, store this layer;

Finally, having the plaintext code available, and its obfuscation layers, the *MalwareAnalysis* stage of the PowerDecode algorithm performs the three following steps:

- Some specific patterns are applied to each stored layer, which will be identified by a label that represents the obfuscation type (string-based, base64, encoded, compressed). All layers with their respective label are written on the report file;
- If the code contains some URLs, the system extracts them and performs a connection to check the related HTTP response status code. In this way, active and offline URLs are distinguished and written on the report file;
- If malware injects shellcode into memory, related hexadecimal instructions are extracted and written on the report file.

5.2. Unary Syntax Tree Model

An obfuscated PowerShell script, in order to be executed, must respect the syntactic rules of the PowerShell scripting language, regardless of the tool with which it was generated.Consequently, to de-obfuscate the PowerShell code it is sufficient to rely essentially on the PowerShell framework. As seen in Section 3, a wide range of obfuscations achievable on PowerShell, are based on recomposing strings by an evaluation function. Hence, it is possible to generalize this dynamic through an obfuscation model. A generic script, containing multiple obfuscation layers, can be abstractly represented by a unary syntax tree composed of *N* nodes, where the *i-th* node of the tree corresponds to the *i-th* obfuscation layer for $i \in [\,1\,,\ N-1\,]$, i.e. a block of obfuscated code ($c_i\,, d_i$), argument of an evaluation function f_i. The last node (*Layer N*), corresponds to a code block (c_N), weakly obfuscated or not obfuscated, without any dependence on the evaluation function. This structure is showed in Figure 2.

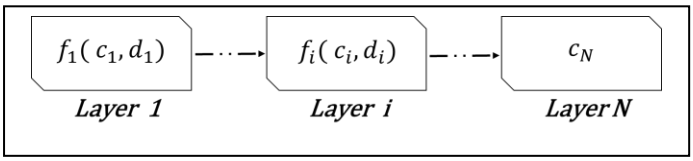

Figure 2: Unary syntax tree model

The evaluation function f_i it can take different forms depending on the obfuscation at i-th layer.
We distinguish between two major cases:

- *I-th* layer containing base64 obfuscation: the evaluation function f_i coincides with the "powershell" function call preceding encoded base64 string;
- *I-th* layer containing string-based, encoded or compressed obfuscation: the evaluation function f_i coincides with the "Invoke-Expression" cmdlet;

As a code string, the f_i could be also obfuscated using randomization or string-based format.
The code block ($c_i\,, d_i$) consists of the following parts:

- c_i : a sub-block of obfuscated code, containing unreadable data;
- d_i : a sub-block of code containing some information about the obfuscation technique applied in the current layer, necessary for the conversion of c_i into meaningful data, i.e, to reconstruct the next layer in runtime.

The obfuscated script execution takes place according to the following dynamic:

$$\begin{cases} f_{i+1}(c_{i+1}, d_{i+1}) = f_i(c_i\,, d_i) & ,\ 1 \le i \le N-1 \\ c_N = f_i(c_i, d_i) & ,\ i = N \end{cases}$$

Where $f_{i+1}(c_{i+1}, d_{i+1})$ is the obfuscated code at *layer i+1*. It coincides with the returned value from the execution of the code block $f_i(c_i, d_i)$ for $1 \leq i \leq N-1$.
If $i = N$, we obtain $f_N(c_N, d_N) = c_N$, corresponding to a code block at *layer N*, without any dependence on evaluation function. The execution of code c_N determines the execution of PowerShell commands contained on it.
To demonstrate the applicability of this model, let us consider the example shown in Listing 1.3, 1.4, 1.5, 1.6 of Appendix C.
The 1st obfuscation layer: $f_1(c_1, d_1)$ contains base64 encoding. The components of this layer are shown in Table 1.

Table 1
Unary syntax tree node components at 1st layer

Component	Syntax	Description
f_1	`Powershell`	Recursive call to PowerShell
c_1	`IAAoAE4ARQBXAC{...}AJwApAA==`	Base64 encoded string
d_1	`-e`	Encoding flag

The execution of the code $f_1(c_1, d_1)$ returns the code $f_2(c_2, d_2)$ corresponding to the 2nd obfuscation layer, containing compressed format. The components of this layer are shown in Table 2.

Table 2
Unary syntax tree node components at 2nd layer

Component	Syntax	Description
f_2	`& ( $enV:comsPEC[4,15,25]-Join'')`	Obfuscated Invoke-Expression cmdlet
c_2	`09BQqjavrTaprTaorTasrTarrTaurTaqrTatVdJNU1AvUtdRz09O BZKJRal6QKpYITcxpxzICADi1AqQTDEQB5ckFpXoqmtqKtQoqClo KKgUZ7j6+`	Compressed string
d_2	`,[io.COMprEssIoN.cOMpRESSIonMoDe]::DEcOmPREsS )| %{NEW-OBJECt SYsteM.iO.streamREAdER( $_ , [teXT.Encoding]::asCii)}).ReadTOeND()|`	Compression algorithm data

The execution of the code $f_2(c_2, d_2)$ returns the code $f_3(c_3, d_3)$ corresponding to the 3rd obfuscation layer, containing string-based format. The components of this layer are shown in Table 3.

Table 3
Unary syntax tree node components at 3rd layer

Component	Syntax	Description
f_3	`& ( $shELLid[1]+$sHeLlId[13]+'X')`	Obfuscated Invoke-Expression cmdlet
c_3	`('r','oce','are.','s malw','P','exe','s','Start-')`	Reordered string
d_3	`"{7}{4}{0}{1}{6}{3}{2}{5}"-f`	Data about the ordering of sub-strings

The executions of the code $f_3(c_3, d_3)$ returns the code $f_4(c_4, d_4) = c_4$ corresponding to the code in its original form, containing a command directly executable by the shell. The components of this layer are shown in Table 4.

Table 4
Unary syntax tree node components at 4th layer

Component	Syntax	Description
f_4	-	-
c_4	`Start-process malware.exe`	Code not dependent on evaluation function
d_4	-	-

5.3. De-Obfuscation Functions

The de-obfuscation strategy implemented by PowerDecode, relies on the proposed unary syntax tree model. To this end, the following three de-obfuscation methods are employed:

1. Base64 Layers Removal

Base64 encoding is detected by the function *Base64Check* applying regular expressions. In case of a match, the current code is passed as input to the *DecodeBase64* function, which removes the encoding using the appropriate method supported by language [8].

2. De-obfuscating Layers Containing Invoke-Expression Cmdlet

While the *SyntaxCheck* function returns "true" analyzing a given layer, if the code isn't base64 encoded, it is passed as input to the *DeobfuscatebyOverriding* function. Here, a local execution environment is allocated to run the code changing its semantics. The goal is to prevent the code from running normally and force it to return the actions it was trying to perform.
This is basically implemented by overriding the Invoke-Expression cmdlet. Precisely, the cmdlet is redefined to perform the same actions performed by the Write-Output cmdlet, i.e. evaluating syntactic constructs, recomposing strings without converting them to statements and finally writing the resulting code into a variable. In this way, if the obfuscated code contains a call to the Invoke-Expression cmdlet, it will return a string containing the instructions it should have executed, corresponding to the next layer.
After the last obfuscation layer is removed, the code is executed. To avoid malicious actions, further cmdlets are overridden. This strategy is also adopted to collect some information about actions attempted by malware and to remove some anti-debugging techniques performed. These overriding procedures are applied by redefining cmdlets functions, in such a way that the original behavior is erased and replaced with some instructions in order to intercept cmdlet calls and write related data on the report file. According to this logic, Start-Sleep, Add-Type, Start-Process, Stop-Process, New-Object, Invoke-Item cmdlets are overridden.
Cmdlet overriding technique was already employed on similar pre-existing tools [4], [6], however PowerDecode implements it in a different way, based on an implicit knowledge of the unary syntax tree. The main constraint imposed by this model, requires that obfuscation layers containing Invoke-Expression calls must be resolved by only cmdlet overriding technique, applied cyclically. Using other techniques such as replacing strings by regular expressions could result in information loss, making impossible to recover the original code.

3. Obfuscation Residual Removal

PowerDecode employs regular expressions as de-obfuscation technique just in the final stage of the de-obfuscation algorithm. According to the syntax tree model, the code c_N may contain some obfuscation residual (such as string concatenation "+" evaluated by the "&" operator).
The function *DeobfuscatebyRegex*, implementing a set of regular expressions [6], [7], performs the removal of these obfuscation symbols.

6. Experimental Evaluation

For the purpose of comparing the performance of PowerDecode with those attained by similar tools (PowerDrive and PSDecode) [5], [7], we employed a dataset of 2906 PowerShell malicious scripts extracted from macros embedded in malicious MS Office documents obtained from VirusTotal. The results of these tests are shown in Table 5.

Table 5
Comparision of performance

Analyzed scripts	De-obfuscated by PowerDrive	De-obfuscated by PSDecode	De-obfuscated by PowerDecode
2906	2139	875	2874

Based on the results obtained, in comparison to PowerDrive and PSDecode, PowerDecode, was able to resolve a wider range of obfuscations. In particular, the pre-existing tools showed the following limitations:

- PSDecode attempts to solve Invoke-Expression dependent obfuscation layers applying sequentially regular expressions and cmdlet overriding. This approach may generate errors when code is executed to return the next layer. PowerDecode algorithm, unlike this latter, applies regular expressions as a final stage, only after all Invoke-Expression dependent layers have been removed. In this way, PowerDecode solved successfully all Invoke-Expression dependent obfuscation layers.
- Similarly to PowerDrive, PowerDecode implements a base64 encoding recognizer. This feature made it possible to manage this encoding more efficiently. Conversely, PSDecode tries to immediately decode the script to verify if was base64 encoded. This strategy fails when the input text file has encoding other than UTF-8.
- PowerDrive applies a limited number of regular expressions, which do not allow to remove some recurring obfuscations. PowerDecode, unlike this latter applies the same set of regular expressions of PSDecode, wider than the previous one, which allows to match a large number of obfuscations patterns not dependent of Invoke-Expression, not solvable using cmdlet overriding technique.
- Cmdlet overriding technique, in order to remove a single obfuscation layer, requires code execution. Both PowerDrive and PSDecode perform this technique executing the code by recursive call to PowerShell. This approach returns an execution error when the obfuscated code contains the string-based reorder or reverse format. PowerDecode, unlike the others, performs cmdlet overriding, executing the code by Invoke-Expression cmdlet. This approach has proven effective for all of these obfuscation types.

Few scripts were not completely de-obfuscated as they contained obfuscation types not dependent on Invoke-Expression, like pieces of code stored into variables or string-based obfuscation variants not matched by regular expressions. This is due to the fact that these cases are not representable by the unary syntax tree model. However, no scripts that PowerDecode was unable to completely de-obfuscate have been de-obfuscated by the previous tools.
One feature that has proved to be important for statistical purposes is the obfuscation recognizer

implemented by PowerDecode. In particular, it made it possible to classify 6018 detected layers and to carry out a statistics on the most used obfuscation techniques. Table 6 shows the results of this statistics.

Table 6
Statistics on obfuscation techniques applied

Obfuscation type	Layers detected	Rate
String-based	3320	55,16%
Base64	1420	23,6%
Compressed	681	11,32%
Encoded	597	9,92%

Likewise, cmdlet overriding implemented by PowerDecode, allowed to intercept and record actions performed by malware sample. Table 7 shows the results of this analysis.

Table 7
Action performed by malicious scripts

Action	Attempts	Rate
Invoke-Item	594	20,4%
Start-Process	474	16,3%
Start-Sleep	24	0,8%

Most scripts analyzed were found to belong to the file-based category. Instead, only 30 scripts analyzed (1%), resulted belonging to the file-less type.

7. Discussion and Conclusions

In this work, we initially introduced the issue of PowerShell malware and obfuscation techniques employed to avoid threat detection. Subsequently, we presented the PowerDecode software project providing a detailed description of its operating logic.
The experimental evaluation highlighted the high performance of PowerDecode on de-obfuscating a wide number of PowerShell malware. Pre-existing tools encountered several difficulties in resolving some obfuscation types. For example, PSDecode was unable to solve base64 encoding efficiently and PowerDrive could not de-obfuscate several string-based layers. PowerDecode was designed following an accurate analysis of these drawbacks. At the same time the project combined the strengths of these tools.
An important advantage offered by PowerDecode, unlike other similar tools [19], is the simplicity on de-obfuscating complex syntactic constructs. In fact, thanks to the de-obfuscation algorithm based on unary syntax tree model, PowerDecode allows to solve obfuscation successfully, independently of the syntactic complexity of the code.
PowerDecode can be easily integrated with complementary tools dedicated to extraction of malicious macros from Office documents [18], which often tend to overlook the problem of PowerShell code obfuscation.
Although the current version of PowerDecode represents a valid malware analysis tool, it can still be improved in some features. One of these is file-less malware analysis. In these cases, in fact, the tool simply extracts the shellcode in the form of hexadecimal values. Hence, it is necessary to employ a disassembler to obtain useful information about malware [20].

8. References

[1] Symantec, “The increased use of PowerShell in attacks”, [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/increased-use-of-powershell-in-attacks-16-en.pdf

[2] McAfee, ”McAfee Labs Threats Report, August 2019”, [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-aug-2019.pdf

[3] McAfee, “McAfee Sees COVID-19-Themed Threats and PowerShell Malware Surge in Q2 2020”, [Online]. Available: https://ir.mcafee.com/node/6571/pdf

[4] D.Ugarte, D.Maiorca, F.Cara, G.Giacinto, “PowerDrive: Accurate De-Obfuscation and Analysis of PowerShell Malware”, *16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Springer, Gothenburg, Sweden, pagg 240-259, 2019.

[5] D.Ugarte, “PowerDrive”, [Online]. Available: https://github.com/denisugarte/PowerDrive

[6] R3MRUM, “From Emotet, PSDecode is born!”, [Online]. Available: https://r3mrum.wordpress.com/2017/12/15/from-emotet-psdecode-is-born/

[7] R3MRUM, “PSDecode”, [Online]. Available: https://github.com/R3MRUM/PSDecode

[8] Microsoft, “PowerShell Documentation”, [Online]. Available: https://docs.microsoft.com/en-us/powershell/

[9] McAfee, “Fileless Malware Execution with PowerShell Is Easier than You May Realize”, [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-fileless-malware-execution.pdf

[10] Chronicle Security, “VirusTotal”, [Online]. Available: https://www.virustotal.com

[11] D.Bohannon, “Invoke-Obfuscation”, [Online]. Available: https://github.com/danielbohannon/Invoke-Obfuscation

[12] D.Bohannon, “PowerShell Command Line Argument Obfuscation Techniques” [Online]. Available: https://nullcon.net/website/archives/pdf/goa-2017/invoke-obfuscation-nullcon-2017.pdf

[13] ASCII Table, [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/d/dd/ASCII-Table.svg

[14] A.Mujumdar, G. Masiwal e B. Meshram, “Analysis of Signature-Based and Behavior-Based Anti-Malware Approaches”, International Journal Of Advance Research, Ideas And Innovations In Technology,2019.

[15] M. Nelson, “Powershell-Payload-Excel-Delivery”, https://github.com/enigma0x3/Powershell-Payload-Excel-Delivery

[16] S. Pontiroli, R. Martinez, “The Tao of.NET and PowerShell Malware Analysis”, VB2015 — the 25th Virus Bulletin International Conference, 2015.

[17] D. Hendler, S. Kels e A. Rubin, “AMSI-Based Detection of Malicious PowerShell Code Using Contextual Embeddings”, ACM Asia Conference on Computer and Communications Security, 2020.

[18] C. Liu, B. Xia, M. Yu, Y. Liu, ”PSDEM: A Feasible De-Obfuscation Method for Malicious PowerShell Detection”, IEEE Symposium on Computers and Communications (ISCC), 2018.

[19] Z.Li, Q.Chen, C.Xiong, Y.Chen, T.Zhu, H.Yang, “Effective and Light-Weight Deobfuscation and SemanticAware Attack Detection for PowerShell Scripts”, ACM SIGSAC Conference on Computer and Communications Security,2019.

[20] M.Graeber, “PowerShellArsenal”, [Online]. Available: https://github.com/mattifestation/PowerShellArsenal

[21] G.Malandrone, “Studio e Sviluppo di un Rilevatore di Attacchi Avanzati Basati su PowerShell”, Master thesis, 2020.

[22] G.Malandrone, “PowerDecode”, [Online]. Available: https://github.com/Malandrone/PowerDecode

Appendix A: PowerShell Cmdlets

Table 8
Most relevant cmdlets

Cmdlet	Alias	Description
Invoke-Expression	iex	Evaluates or runs a specified string as a command
Write-Output	write, echo	Sends the specified object down the pipeline to the next command
Start-Process	saps, start	Starts one or more processes on the local computer
Stop-Process	spps, kill	Stops one or more running processes
Invoke-Item	ii	Performs the default action on the specified item
Start-Sleep	sleep	Suspends the activity in a script or session for the specified period of time
New-Object	-	Creates an instance of a Microsoft .NET Framework or COM object
Add-Type	-	Adds a Microsoft .NET Core class to a PowerShell session

Appendix B: PowerShell Obfuscation Types

Table 9
Most common PowerShell obfuscation types applied to “New-Object” string

Type	Subtype	Example
String-based	Concatenate	`‘Ne’+’w’+’-Ob’+’je’+’ct’`
	Reorder	`“{1}{0}{2}” -f ‘-Obj’,’New’,’ect’`
	Reverse	`“tcejbO-weN” [ (“tcejbO-weN”.length-1)..0] -join’’`
	Replace	`“fwjih-Object” -replace (‘fwjih’ , ‘New’)`
Base64	-	`TmV3LU9iamVjdAo=`
Encoded	Binary	`( (1001110, 1100101, 1110111 , 101101 ,1001111, 1100010, 1101010, 1100101,1100011 ,1110100) |foreach {( [convert]::toInt16( ( $_.toString() ) ,2)-as [char]) } ) -join''`
	Octal	`(( 116,145,167 , 55,117 , 142,152,145 , 143,164)|foreach {( [convert]::toInt16( ( $_.toString() ) ,8)-as [char]) } ) -join''`
	Decimal	`( ( 78 , 101 , 119, 45 ,79 , 98,106 , 101 ,99 , 116) |foreach {( [convert]::toInt16( ( $_.toString() ) ,10)-as [char]) } ) -join''`
	Hexadecimal	`(('4e' , '65' , '77' , '2d' , '4f','62', '6a', '65' , '63' ,'74')|foreach {( [convert]::toInt16( ( $_.toString() ) ,16)-as [char]) } ) -join''`
	Bxor	`( [char[]] (97 ,74 ,88, 2, 96,77 , 69 , 74 ,76,91)| foreach { [char]($_ -bxor"0x2F" ) } ) -join''`
	Secure string	`([runtime.interopservices.marshal]:: ([runtime.interopservices.marshal].getmembers()[2].name) .invoke( [runtime.interopservices.marshal]:: securestringtoglobalallocunicode( $('76492d1116743f0 […] wA=' | convertto-securestring -key (111..88)) )) )`
Compressed	Deflatestream	`( new-object io.compression.deflatestream( [system.io.memorystream][convert]::frombase64string( '80st1/VPykpNLgEA' ) ,[iO.COmpReSsION.COmPRessionModE]::DecOMPReSS ) |ForeACH-oBjecT{NeW-ObjecT SysTem.iO.stReAMrEaDER($_,[TExt.EnCODING]::AsCII ) } |fOREACh-OBJECt { $_.rEadTOeNd()} )`
Randomization	Up-low case	`nEW-OBJeCt`
	Ticking	`N`e`w-Object`
	Whitespacing	`New-Object`

Appendix C: Example of an Obfuscated PowerShell Script

This appendix shows a sample of a PowerShell script containing several obfuscation layers. The 1st layer showed in Listing 1.3, corresponds to the visible part of the code. It contains a base64 string which
encapsulates all underlying layers.

```
powershell -e
IAAoAE4ARQBXAC0ATwBCAEoARQBDAHQAIAAgAEkATwAuAGMATwBtAHAAUgBFAHMAUwBJAG8AbgAuAEQARQBmAGwAQQB0AGUAcw
B0AHIAZQBBAE0AKAAgAFsAaQBvAC4ATQBFAE0ATwBSAFkAcwB0AFIAZQBBAG0AXQAgAFsAUwB5AHMAVABlAE0ALgBDAE8AbgB2
AEUAcgB0AF0AOgA6AGYAcgBPAG0AYgBBAFMARQA2ADQAUwB0AHIASQBOAEcAKAAnADAAOQBCAFEAcQBqAGEAdgByAFQAYQBwAH
IAVABhAG8AcgBUAGEAcwByAFQAYQByAHIAVABhAHUAcgBUAGEAcQByAFQAYQB0AFYAZABKAE4AVQAxAEEAdgBVAHQAZABSAHoA
MAA5AE8AQgBaAEsASgBSAGEAbAA2AFEASwBwAFkASQBUAGMAeABwAHgAegBJAEMAQQBEAGkAMQBBAHEAUQBUAEQARQBRAEIANQ
BjAGsARgBwAFgAbwBxAG0AdABxAEsAdABRAG8AcQBDAGwAbwBLAEsAZwBVAFoANwBqADYAKwBHAFMAbQBSAEIAdgBHAGEAcQBz
AFUAZQA2AFQANgA1AEgAZwBDADIAYwBhAHgAMgB1AG8AUgA2AHAAbwBBACcAIAApACAALABbAGkAbwAuAEMATwBNAHAAcgBFAH
MAcwBJAG8ATgAuAGMATwBNAHAAUgBFAFMAUwBJAG8AbgBNAG8ARABlAF0AOgA6AEQARQBjAE8AbQBQAFIARQBzAFMAIAApAHwA
IAAlAHsATgBFAFcALQBPAEIASgBFAEMAdAAgACAAUwBZAHMAdABlAE0ALgBpAE8ALgBzAHQAcgBlAGEAbQBSAEUAQQBkAEUAUg
AoACAAJABfACAALAAgAFsAdABlAFgAVAAuAEUAbgBjAG8AZABpAG4AZwBdADoAOgBhAHMAQwBpAGkAKQB9ACkALgBSAGUAYQBk
AFQATwBlAE4ARAAoACkAfAAgACYAIAAoACAAJABlAG4AVgA6AGMAbwBtAHMAUABFAEMAWwA0ACwAMQA1ACwAMgA1AF0ALQBKAG
8AaQBuACcAJwApAA==
```

Listing 1.3: Base64 obfuscation on 1st layer

The 2nd layer showed in Listing 1.4 contains an obfuscated code in the compressed format which encapsulates all underlying layers.

```
   (NEW-OBJECt  IO.cOmpREsSIon.DEflAtestreAM( [io.MEMORYstReAm]
[SysTeM.COnvErt]::frOmbASE64StrING('09BQqjavrTaprTaorTasrTarrTaurTaqrTatVdJNU1AvUtdRz09OBZKJRal6QK
pYITcxpxzICADi1AqQTDEQB5ckFpXoqmtqKtQoqCloKKgUZ7j6+GSmRBvGaqsUe6T65HgC2cax2uoR6poA' )
,[io.COMprEssIoN.cOMpRESSIonMoDe]::DEcOmPREsS )| %{NEW-OBJECt  SYsteM.iO.streamREAdER( $_ ,
[teXT.Encoding]::asCii)}).ReadTOeND()| & ( $enV:comsPEC[4,15,25]-Join'')
```

Listing 1.4: Compressed obfuscation on 2nd layer

The 3rd layer showed in Listing 1.5 contains an obfuscated code in the string-based format which encapsulates the underlying layer.

```
(("{7}{4}{0}{1}{6}{3}{2}{5}"-f 'r','oce','are.','s malw','P','exe','s','Start-')) | & (
$shELLid[1]+$sHeLlId[13]+'X')
```

Listing 1.5: String-based obfuscation on 3rd layer

Removing the last obfuscation layer, the original code, showed in Listing 1.6, is recovered.

```
Start-Process malware.exe
```

Listing 1.6: Original not obfuscated code on 4th layer