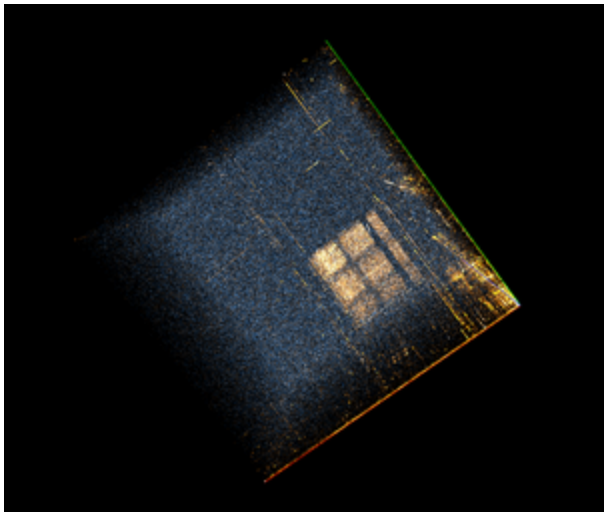
Nanocore & CypherIT

malwareindepth.com/defeating-nanocore-and-cypherit

April 4, 2020

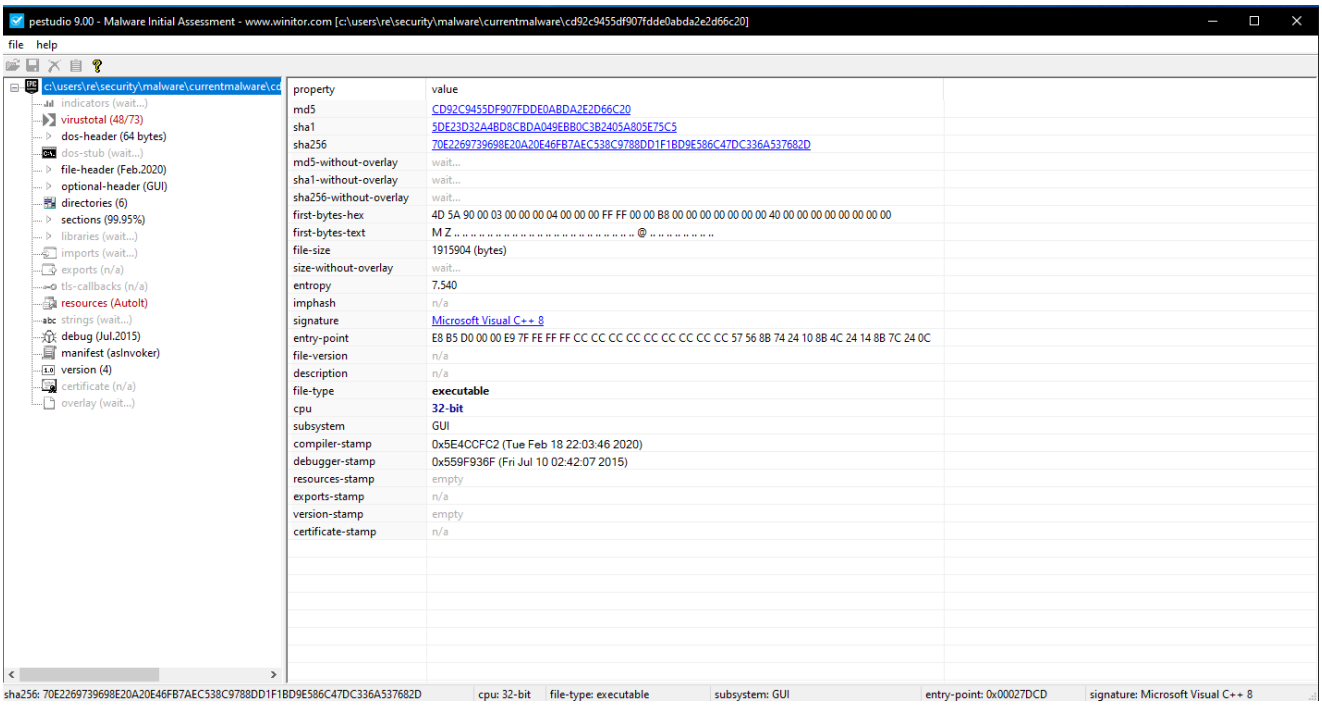
2 years ago



Hello everyone! Its been a while since I've posted. There's been some changes in my life that have distracted me from my malware temporarily. One of those updates is a career change. I will officially be working as a security researcher and in preparation of that I felt that I needed to keep my reverse engineering skills sharp. So I went to any.runs malware trends page, and randomly picked a sample. I ended up picking a Nanocore sample to analyze. Nanocore has been around for many years and is one of the simpler and cheaper malware familieis out there but I never had the availability during work to look at it. Since I generally focus on targeted malware, I knew this was going to be a good change of pace. The sample can be found here if you wish to follow along.

Technical Analysis

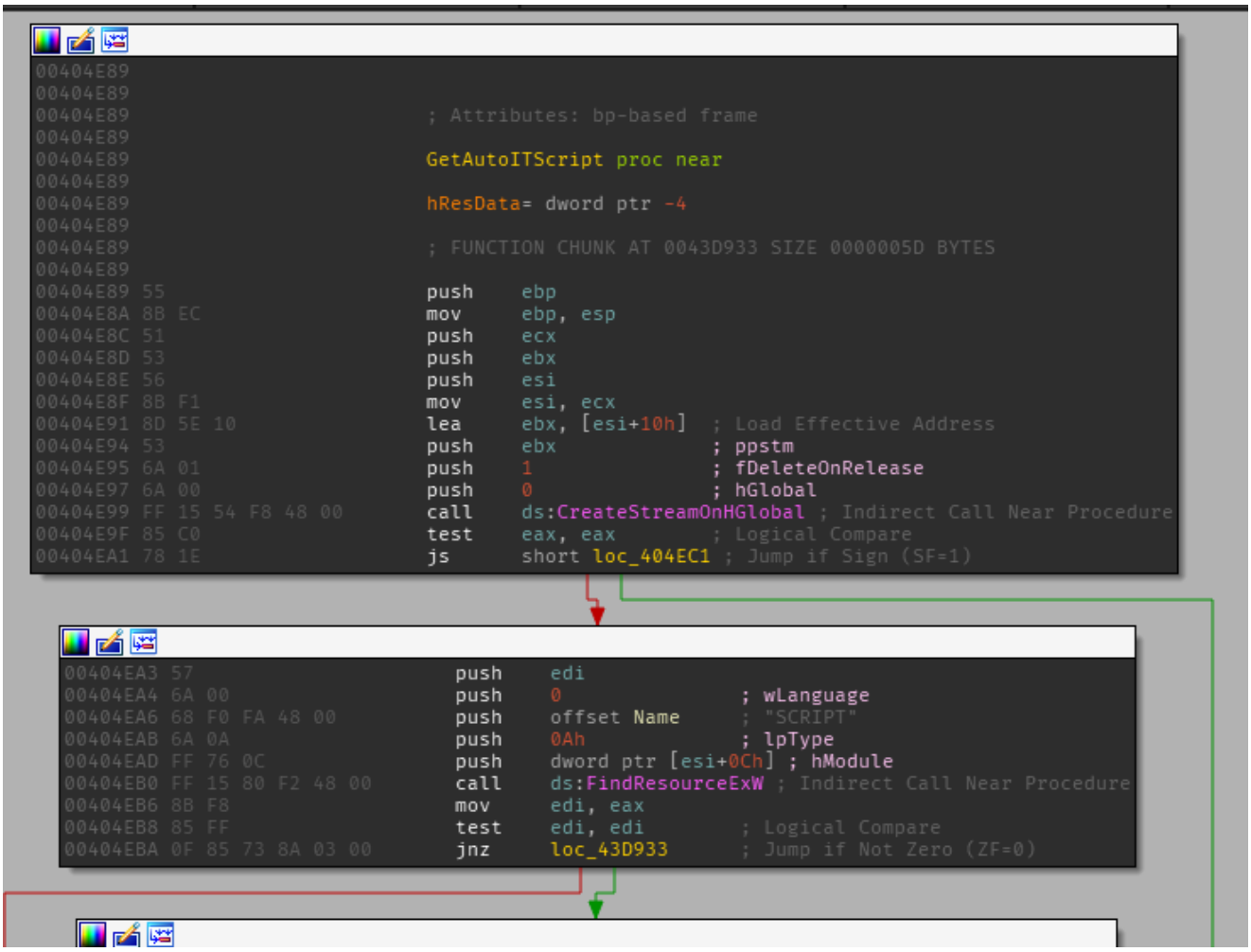
First step as usual, is opening the sample in PE studio for a quick triage.



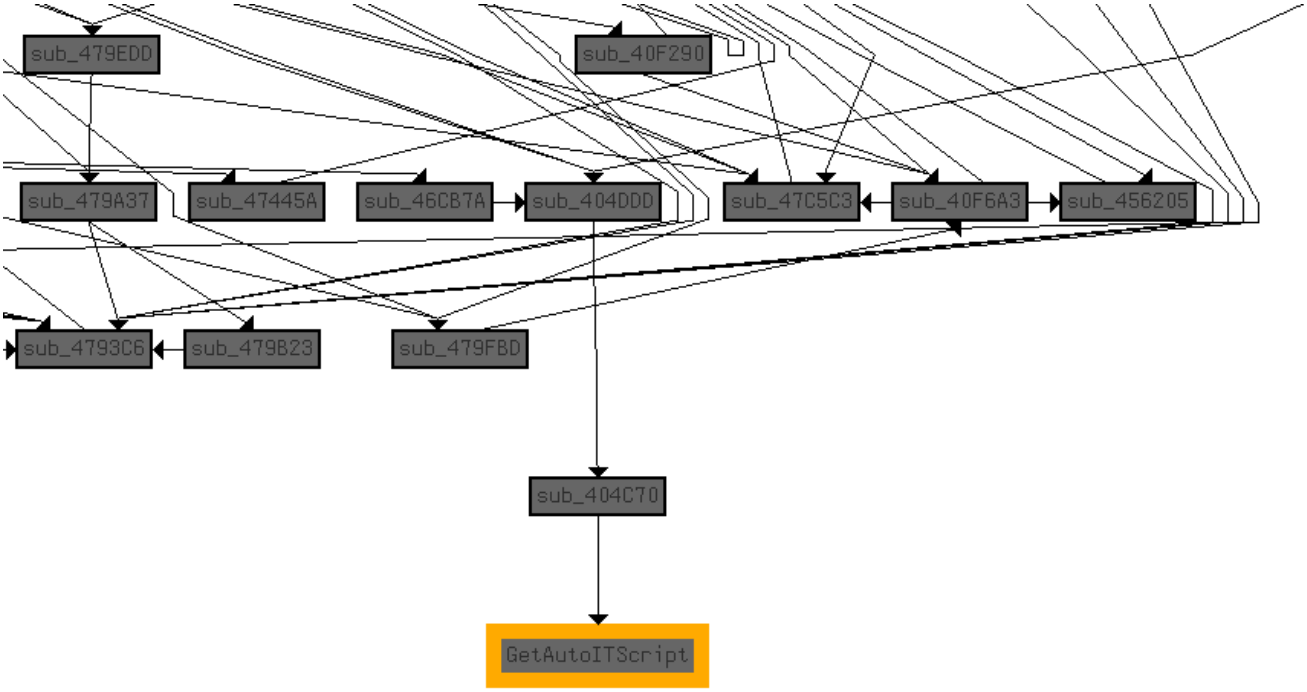
From the output here you can see its a Cpp application with a rather high entropy of 7.5. So there is definitely some encrypted or compressed content here. You can also see that there is an embedded resource within the application. Immediately the AutoIT caught my eye as that's not something I have dealt with before.

type (8)	name	file-offset (32)	signature	non-standard	size (1092673 byt...	file-ratio (57.03%)	md5	entropy	language (2)	first-bytes-hex	first-bytes-text
icon	1	0x000C76E0	icon	-	296	0.02 %	D6F27BF763EB666AF934477958ACF362	3.664	English-Un...	28 00 00 00 10 00 00 00 20 00 00 00 01 ...	(... ... ...
icon	2	0x000C7808	icon	-	296	0.02 %	78F30E363A0499F530D057B4D639D36E	2.059	English-Un...	28 00 00 00 10 00 00 00 20 00 00 00 01 ...	(... ... ...
icon	3	0x000C7930	icon	-	296	0.02 %	AD424F5F5D5FF4460343686C61E4F75E	2.255	English-Un...	28 00 00 00 10 00 00 00 20 00 00 00 01 ...	(... ... ...
icon	4	0x000C7A58	icon	-	744	0.04 %	C601B60AFD3317FF5B5DC586058630DC	2.660	English-Un...	28 00 00 00 20 00 00 00 40 00 00 00 01 ...	(... ... ... @ ... ...
icon	5	0x000C7D40	icon	-	296	0.02 %	EA374DDA5B84CACDE5120306D69BFAA9	2.634	English-Un...	28 00 00 00 10 00 00 00 20 00 00 00 01 ...	(... ... ...
icon	6	0x000C7E68	icon	-	3752	0.20 %	995951209C9E285CF0E0332D25B9B312	3.873	English-Un...	28 00 00 00 30 00 00 00 60 00 00 00 01 ...	(... ... 0 ... ... ` ... ...
icon	7	0x000C8D10	icon	-	2216	0.12 %	620B6F8E941E1C2D7EC5934D6D319F22	4.598	English-Un...	28 00 00 00 20 00 00 00 40 00 00 00 01 ...	(... ... ... @ ... ...
icon	8	0x000C95B8	icon	-	1384	0.07 %	7D5125C1700741CDBBC05496CF7234C0	4.981	English-Un...	28 00 00 00 10 00 00 00 20 00 00 00 01 ...	(... ... ...
icon	9	0x000C9B20	icon	-	13794	0.72 %	46C1E16E876119E4E626ECC309F31C51	7.911	English-Un...	89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 4...	.. P N G ... ... I H D R
icon	10	0x000CD104	icon	-	16936	0.88 %	B44E8D7588364EE607955B088747BED6	2.606	English-Un...	28 00 00 00 40 00 00 00 80 00 00 00 01 ...	(... ... @ ... ...
icon	11	0x000D132C	icon	-	9640	0.50 %	A96A94AD1A1E33F64FF2D3E167FB9467	2.812	English-Un...	28 00 00 00 30 00 00 00 60 00 00 00 01 ...	(... ... 0 ... ... ` ... ...
icon	12	0x000D38D4	icon	-	6760	0.35 %	12606923495BC5C6FD46405AD90824D7	2.764	English-Un...	28 00 00 00 28 00 00 00 50 00 00 00 01 ...	(... ... (... ... P ... ...
icon	13	0x000D533C	icon	-	4264	0.22 %	0A3E9865544865F0BE6A88C996708960	3.027	English-Un...	28 00 00 00 20 00 00 00 40 00 00 00 01 ...	(... ... ... @ ... ...
icon	14	0x000D63E4	icon	-	2440	0.13 %	2CD4C617697ADF4B60620E0FBDAA92C1	3.526	English-Un...	28 00 00 00 18 00 00 00 30 00 00 00 01 ...	(... ... ... 0 ... ...
icon	15	0x000D6D6C	icon	-	1720	0.09 %	F3B1004C9D3E9BF9B2226D162F9827A9	3.426	English-Un...	28 00 00 00 14 00 00 00 28 00 00 00 01 ...	(... ... ... (... ... .
icon	16	0x000D7424	icon	-	1128	0.06 %	FF40094E77198173CF772BD64BEE9D3D	3.884	English-Un...	28 00 00 00 10 00 00 00 20 00 00 00 01 ...	(... ... ...
menu	166	0x000D788C	menu	-	80	0.00 %	8140596AB00B98A11C13E6977D2D0977	2.683	English-Un...	00 00 00 00 90 00 43 00 6F 00 6E 00 74 ...	... ... C .. o .. n .. t .. e ..
dialog	1000	0x000D78DC	dialog	-	252	0.01 %	08E5FDCBCB2AB21352C8FC0E05B07DDB	3.040	English-Un...	01 00 FF FF 00 00 00 00 00 00 04 00 4C ...	... ... L ... ..
string-table	7	0x000D79D8	string-table	-	1428	0.07 %	D1F824F98742295A66A25225701DD6D8	3.347	English-Un...	00 00 00 00 00 00 00 00 00 00 09 00 28 ...	... ... (.. P ..
string-table	8	0x000D7F6C	string-table	-	1674	0.09 %	5BEAEEBDA5346956E395FAD21661F382	3.282	English-Un...	30 00 49 00 6E 00 63 00 6F 00 72 00 72 ...	0 .. I .. n .. c .. o .. r .. r .. e ..
string-table	9	0x000D85F8	string-table	-	1168	0.06 %	6B12D17C7622D8215889A2288098FA15	3.288	English-Un...	30 00 45 00 78 00 70 00 65 00 63 00 74 ...	0 .. E .. x .. p .. e .. c .. t .. e ..
string-table	10	0x000D8A88	string-table	-	1532	0.08 %	949955DBDA367F0C790615F7817E7886	3.284	English-Un...	1A 00 49 00 6E 00 76 00 61 00 6C 00 69 ...	.. I .. n .. v .. a .. l .. i .. d ..
string-table	11	0x000D9084	string-table	-	1628	0.08 %	9BC568A6176F738FFB3109E53235B579	3.263	English-Un...	3E 00 22 00 53 00 65 00 6C 00 65 00 63 ...	> .. " .. S .. e .. l .. e .. c .. t ..
string-table	12	0x000D96E0	string-table	-	1126	0.06 %	899B8766AEA5F88410C72162092575 4B	3.258	English-Un...	48 00 43 00 61 00 6E 00 20 00 70 00 61 ...	H .. C .. a .. n .. .. p .. a .. s ..
string-table	313	0x000D9B48	string-table	-	344	0.02 %	193A9143563395AD14C6DAC83D32E2AD	3.086	English-Un...	00 00 00 00 00 00 00 00 00 00 00 00 00 ...	... ...
rcdata	SCRIPT	0x000D9CA0	AutoIt	-	1016004	53.03 %	34BFCA25137C6259B566B0667BB6EFD8	8.000	neutral	A3 48 4B BE 98 6C 4A A9 99 4C 53 0A ...	.. H K ... l J ... L S ... .. H }
icon-group	99	0x001D1D64	icon-group	-	188	0.01 %	3F201C77CE09BE7C1CEC40109DAFD19B	3.120	English-Un...	00 00 01 00 0D 00 20 20 10 00 01 00 04 ...	... ...
icon-group	162	0x001D1E20	icon-group	-	20	0.00 %	7A9605CB416B1A091D889B9D9F37EC66	2.023	English-Un...	00 00 01 00 01 00 10 10 10 00 01 00 04 ...	... ... (..
icon-group	164	0x001D1E34	icon-group	-	20	0.00 %	F64C60B749269FCF6659C450DDA98486	1.843	English-Un...	00 00 01 00 01 00 10 10 10 00 01 00 04 ...	... ... (..
icon-group	169	0x001D1E48	icon-group	-	20	0.00 %	60F05E3B8EA9E18928923BDBCC112277	2.023	English-Un...	00 00 01 00 01 00 10 10 10 00 01 00 04 ...	... ... (..
version	1	0x001D1E5C	version	-	220	0.01 %	410F594F3AD95B1DD20E08E299B97C51	2.779	English-Un...	DC 00 34 00 00 00 56 00 53 00 5F 00 56 ...	... 4 ... ... V .. S .. _ .. V .. E ..

Even more suspicious is that its almost 53% of the file, and a maximum entropy value of 8. Seeing the large resource immediately leads me to look for resource related calls such as LockResource, SizeOfResource, LoadResource etc.



FindResource is only called within this function so if we assume that the AutoIT script is part of the malware, this function becomes increasingly important. This function will load the resource make some calls and load the resource data within [ebp+var_4].



Looking at the call graph shows this is a leaf node for the call graph, which can potentially mean that execution will continue outside of the scope of this application or all the information for this chain of calls was acquired. Looking at the parent function it opens a file passed as an argument.

```
; Attributes: bp-based frame

; int __stdcall LoadLibraryGetAndValidateAuoITScript(LPCWSTR lpLibFileName, int)
LoadLibraryGetAndValidateAuoITScrip proc near

var_14= byte ptr -14h
var_4= byte ptr -4
lpLibFileName= dword ptr  8
arg_4= dword ptr  0Ch

; FUNCTION CHUNK AT 0043D8E6 SIZE 0000004D BYTES

push    ebp
mov     ebp, esp
sub     esp, 18h        ; Integer Subtraction
push    esi
mov     esi, ecx
lea     ecx, [ebp+var_4] ; Load Effective Address
push    edi
FF FF   call    sub_404BB5      ; Call Procedure
48 00   push    offset aRb      ; "rb"
        push    [ebp+lpLibFileName] ; wchar_t *
02 00   call    __wfopen        ; Call Procedure
        mov     [esi], eax
        pop     ecx
        pop     ecx
        test    eax, eax        ; Logical Compare
8A 03 00 jz     loc_43D8E6      ; if unable to open file in read binary mode
```

Looking at calls to this function, there are references to various AutoIT strings.

```
v2 = 0;
sub_407667(&v7);
v8 = &off_48F968;
v9 = 0;
v10 = 0;
v11 = 0;
v12 = 0;
sub_403D31(a1);
*(_BYTE *)a2 = v12;
sub_404706(&lpLibFileName);
sub_407DE1(&v18, L">>>AUTOIT NO CMDEXECUTE<<<");
v15 = 0;
v16 = 0;
v17 = 0;
if ( !LoadLibraryGetAndValidateAuoITScript((FILE **)&v13, lpLibFileName, 1) )
{
  if ( sub_46955B((FILE **)&v13, v18, (int)&a1, &a2) )
  {
    sub_404E4A((FILE **)&v13);
  }
  else
  {
    free(a1);
    sub_404E4A((FILE **)&v13);
    v2 = 1;
  }
}
sub_405904((int)&v18);
sub_408047(&lpLibFileName);
sub_40928A(&v7);
sub_407DE1(&v18, L"CMDLINERAW");
sub_4084C0(&v18, &v14, 1, 0);
sub_405904((int)&v18);
v21 = (wchar_t *)256;
```

Jumping to the Main function it calls sub_403B3A which has a anti-debugger check. It calls IsDebuggerPresent and if it is, opens a message box and the process terminates

```
void __stdcall sub_403B3A(wchar_t *a1)
{
  int v1; // ecx
  WCHAR *v2; // eax
  const WCHAR *v3; // ST0C_4
  const WCHAR *v4; // ST08_4
  HWND v5; // eax
  WCHAR v6; // [esp+8h] [ebp-2002Ch]
  WCHAR Buffer; // [esp+10008h] [ebp-1002Ch]
  LPCWSTR lpFile; // [esp+20008h] [ebp-2Ch]
  LPCWSTR lpParameters; // [esp+20018h] [ebp-1Ch]
  LPWSTR FilePart; // [esp+20028h] [ebp-Ch]
  int v11; // [esp+2002Dh] [ebp-7h]

  sub_407667(&lpFile);
  FilePart = 0;
  *(_WORD *)((char *)&v11 + 1) = 0;
  GetCurrentDirectoryW(0x7FFFu, &Buffer);
  sub_403766(a1, (int)&v11);
  if ( IsDebuggerPresent() )
  {
    MessageBoxA(0, "This is a third-party compiled AutoIt script.", caption, 0x10u);
    goto LABEL_14;
  }
```

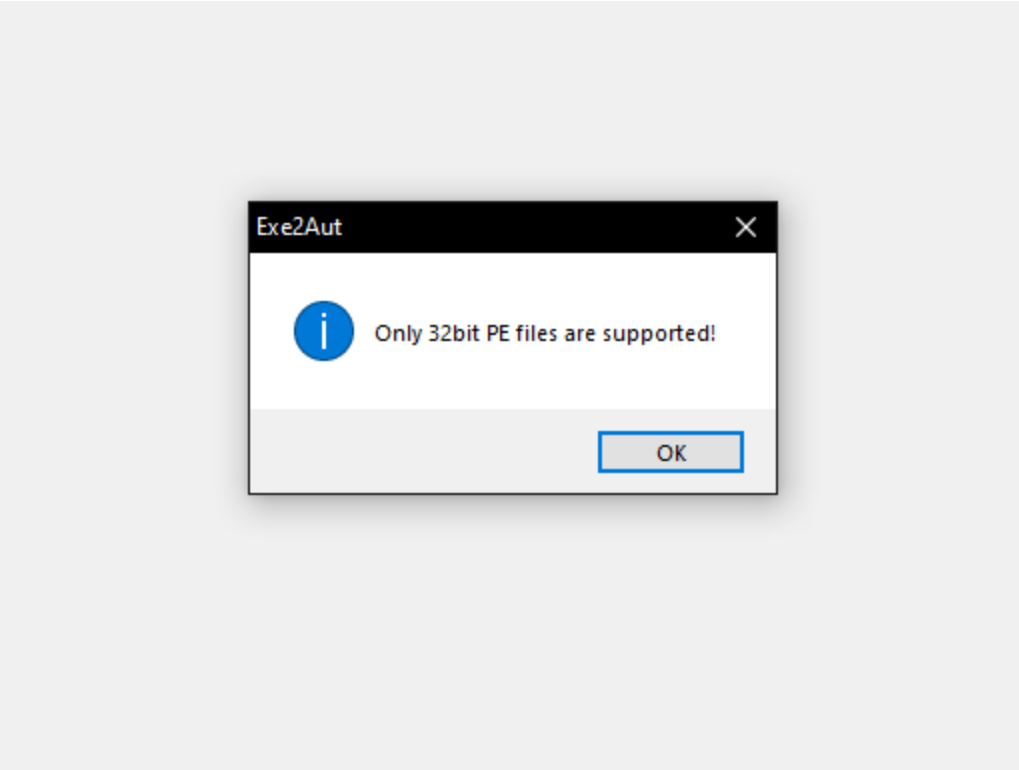
Following sub_408667, eventually the resource will be loaded from memory, and compared against a the compiled AutoIT header

```
7                 db    0
8 ; char AutoITCompiledHeader[8]
8 AutoITCompiledHeader db 0A3h            ; DATA XREF: LoadAndValidateAutoITScript:loc_404C93↑r
9                 db  48h ; H
A                 db  4Bh ; K
B                 db 0BEh ; ¾
C                 db  98h ; ˜
D                 db  6Ch ; l
E                 db  4Ah ; J
F                 db 0A9h ; ©             ; AU3 Compiled Header
0 ; char AutoITCompiledHeader_0[16]
0 AutoITCompiledHeader_0 db 99h           ; DATA XREF: LoadAndValidateAutoITScript+2E↑r
1                 db  4Ch ; L
2                 db  53h ; S
3                 db  0Ah
4                 db  86h ; †
5                 db 0D6h ; Ö
6                 db  48h ; H
7                 db  7Dh ; }
8                 db    0
9                 db    0
A                 db    0
B                 db    0
C                 db    0
D                 db    0
E                 db    0
```

Execution only continues if the header is correct, so we can assume it's going to load an AutoIT script. This coupled with the fact that it quits if you try to debug the executable, I'm comfortable in assuming this executable is going to load and run the compiled AutoIT script from its resource section.

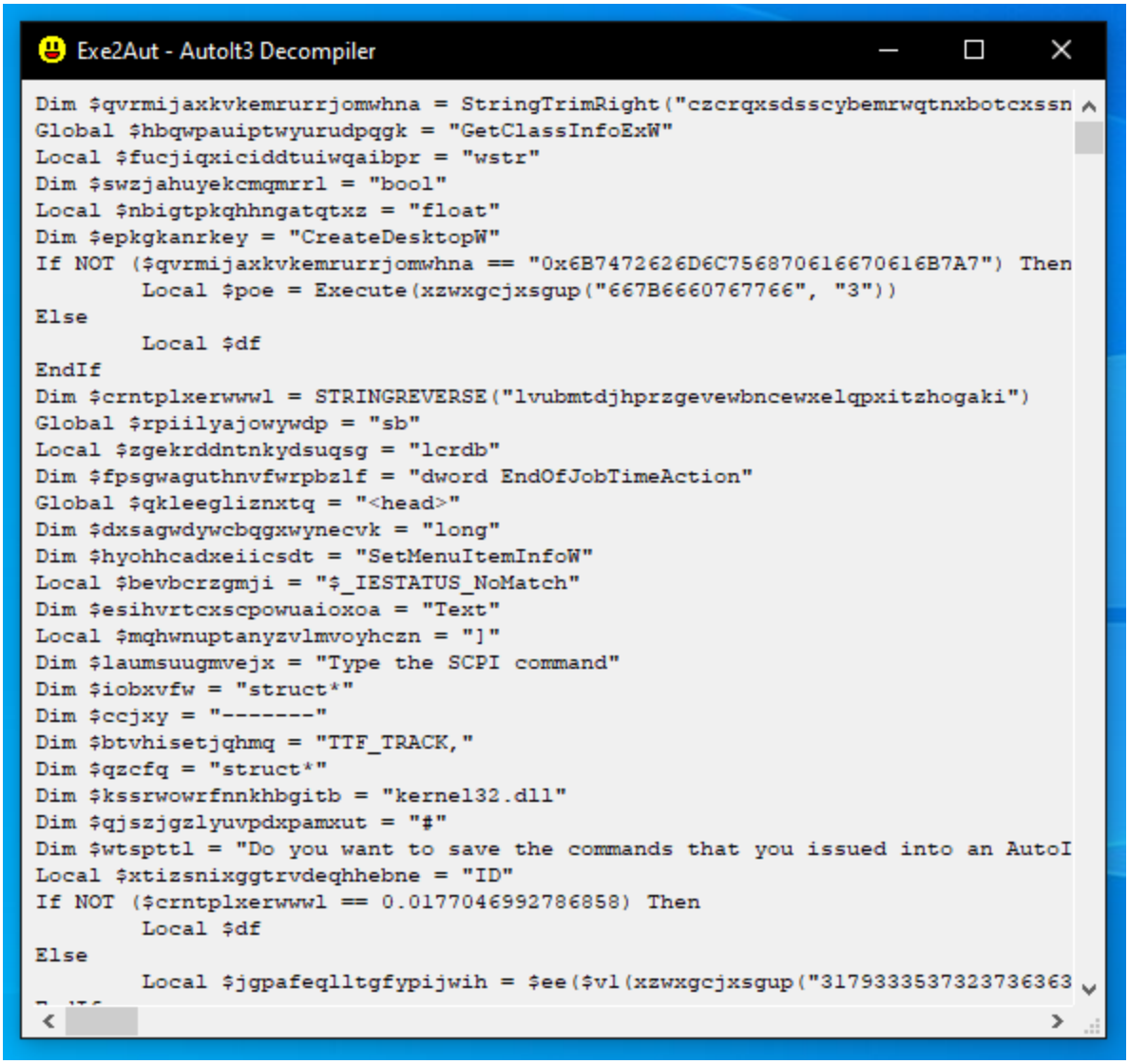
AutoIT Script

Now that we know the binary file we have been looking at is just a runtime environment for the AutoIT script resource we can take a look at the script itself. Extracting the resource with Resource Hacker and throwing it in a hex editor shows that it's a compiled script. Now there are a couple tools out there used to decompile AutoIT scripts. There is Exe2Aut which is what I went with to handle this compiled script. Although running this script through the application gave the following error...



Aut2Exe Error

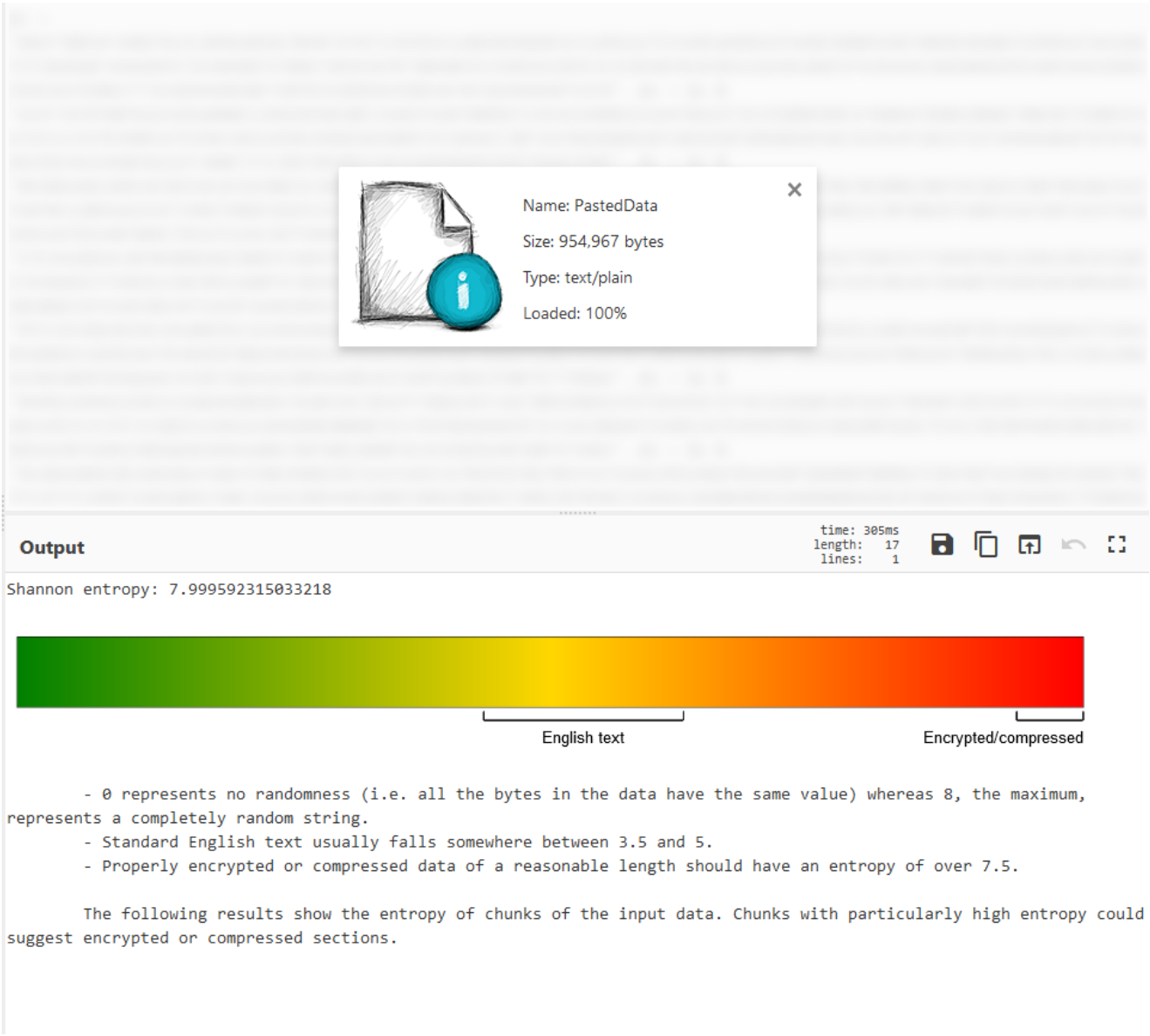
Googling around for this I found Hexacorn's post about this exact issue! Following his post we append our compiled script to the 32 bit stub and we get a valid decompilation of the script!



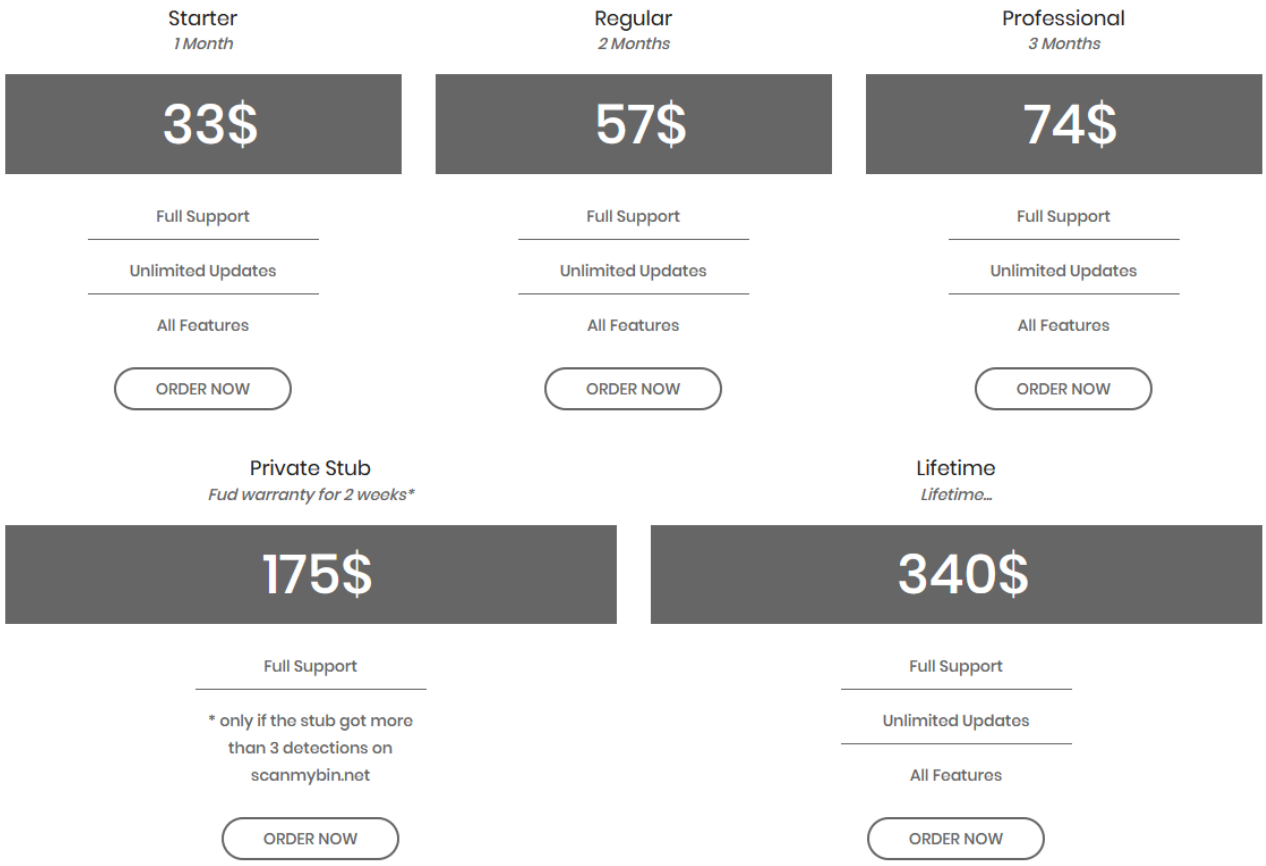
Copying the contents to a new file in VSCode and giving it a look over immediately shows something interesting. This script is 10901 lines long. The majority of the file looks like the following.

```
Dim $qvrmijaxkvkemrurrjomwhna = StringTrimRight("czcrqxsdsscybemrwqtnxbotcxssnzstrlodqtxowlckisow", 34)
Global $hbqwpauiptwyurudpqgk = "GetClassInfoExW"
Local $fucjiqxiciddtuiwqaibpr = "wstr"
Dim $swzjahuyekcmqmrrl = "bool"
Local $nbigtpkqhhngatqtxz = "float"
Dim $epkgkanrkey = "CreateDesktopW"
If NOT ($qvrmijaxkvkemrurrjomwhna == "0x6B7472626D6C7568706166706 16B7A7") Then
    Local $poe = Execute(xzwxgcjxsgup("6678666076 7766", "3"))
Else
    Local $df
EndIf
Dim $crntplxerwwwl = STRINGREVERSE("lvubmtdjhprzgevewbncewxelqpxitzhogaki")
Global $rpiilyajowywdp = "sb"
Local $zgekrddntnkydsuqsg = "lcrdb"
Dim $fpsgwaguthnvfwrpbzlf = "dword EndOfJobTimeAction"
Global $qkleegliznxtq = "<head>"
Dim $dxsagwdywcbqgxwynecvk = "long"
Dim $hyohhcadxeiicsdt = "SetMenuItemInfoW"
Local $bevbcrzgmji = "$_IESTATUS_NoMatch"
Dim $esihvrtcxscpowuaioxoa = "Text"
Local $mqhwnuptanyzvlmvoyhczn = "]"
Dim $laumsuugmvejx = "Type the SCPI command"
Dim $iobxvfw = "struct*"
Dim $ccjxy = "-------"
Dim $btvhisetjqhmq = "TTF_TRACK,"
Dim $qzcfq = "struct*"
Dim $kssrwowrfnnkhbgitb = "kernel32.dll"
Dim $qjszjgzlyuvpdxpamxut = "#"
Dim $wtspttl = "Do you want to save the commands that you issued into an AutoIt3 script?"
Local $xtizsnixggtrvdeqhhebne = "ID"
If NOT ($crntplxerwwwl == 0.0177046992786858) Then
    Local $df
Else
    Local $jgpafeqlltgfypijwih = $ee($v1(xzwxgcjxsgup("3179333537323736363137353633373537333743374237353632363437363743363037383636363937423742363436393637
EndIf
Dim $vgqqgrmfkr = Tan(57)
Dim $jmvjajexolcmekjjskcd = "</table>"
Global $qotohdxakgnupruiwr = "i"
Dim $efhuains = "int"
Global $nufvhvxuqk = "hwnd"
If NOT ($vgqqgrmfkr == 0) Then
    Opt(xzwxgcjxsgup("5076657D4D676B6A4C6D6061", "4"), xzwxgcjxsgup("30", "1"))
Else
    $poe($rei(xzwxgcjxsgup("3179343136333747373237343632363235323742374736323734333935313530363436353747353836353431353835353338", "1")))
EndIf

Func tzmiibdcwx($aaa, $bbb)
    Dim $rpusxcfme = Tan(9)
    Local $tgvkigifhumzb = "StringConstants.au3"
    Global $kulzqegppqyldsh = "dword"
    Local $cqzgcmxlizgyafsqjsajv = "long"
    Global $urhxcpmtn = "Mask"
    Global $oerzcev = "char Text["
    Local $ptsnipscanqtmlezjt = "handle"
    Local $ugpcfbahqttvc = "CloseEventLog"
    Local $ogxakfkl = "int"
    Global $vcuctwgvm = "GdipGetSolidFillColor"
    If NOT ($rpusxcfme == "xkpsiflneyxcdaief") Then
        Local $wzneciaftzmxvngecerfajkujgwyjzn = $poe(xzwxgcjxsgup("62696E617279746F737472696E67", xzwxgcjxsgup("34", "4")))
    Else
        Local $dvsdmjhkwwhodlxzfxln = $ee($v1(xzwxgcjxsgup("327A30363431353A343A3536343B3447353B35353533353A35353430343B3543353334313441353534403 43B344034
    EndIf
    Dim $cnrujuxyjkpuu = StringRight("danwvehrlervjcmcvznqxxghigqosenfd", 31)
    Dim $vehnvffvqzzwgccdd = "Size"
    Local $vqowhvfhzpekffqyd = "FIND"
    Global $ondxyndnsrjfdj = "38837D1400740431C0EB05B80100000021C075198B6C243C837D1400740431C0"
    Dim $lgzzywavoqbmdaorx = "_Worksheet"
    Dim $qxussoigyhvlli = "dwmapi.dll"
    Local $mqrdckwcboqungimsz = "bool"
    Local $hjntwhupavi = "int"
    Global $lqjnnyprzrmkb = "ByteRecv"
    Global $iexkegdjfcbmatioovw = "handle"
    If NOT ($cnrujuxyjkpuu == -3.27370380042812) Then
        Local $qhccowtetrhfrjskqnagr = $poe($wzneciaftzmxvngecerfajkujgwyjzn(xzwxgcjxsgup("347C3034313133373231333631343336324232323D3247323130303 23D333
    Else
        Local $gsxxovmoulwnwljshorrprzooz = $ee($v1(xzwxgcjxsgup("3078323437313731363737333631363836353737363836383639364437413739364337313639364537333736
    EndIf
    Local $orqldofxceehqoekmzvzpvztenerxkzsacumdho = $poe($wzneciaftzmxvngecerfajkujgwyjzn(xzwxgcjxsgup("347C36303335323C3237323732423333333303231333033363
    Dim $manspjvhlfh = STRINGREVERSE("0x6E6A6B7270767073")
    Dim $pgemahkwcrg = "DestroyIcon"
    Global $gqckwdksoxibuxuvshfgzp = "uint"
```

At the end of the file there is a large data blob that spans 3500 lines just on its own. Generally this means it's some sort of payload. Loading this data blob into CyberChef shows that it is most likely either compressed or encrypted. This rules simpler techniques such as XOR encryption.



With this information I knew I'd have to give the script a good hard look. After some googling about AutoIT crypters I came across CypherIT. CypherIT is a AutoIT crypter that is sold at 5 separate tiers. the first tier is 33$ for 1 month, 57$ for 2 months and 74$ for 3 months, 175$ for FUD for 2 weeks and finally a 340$ lifetime model.



Interestingly enough they even have a discord server that users can join for troubleshooting and getting updates on new versions.

Going back to the script.... After the large data blob is finished being initialized, it is passed to a function called skpekamgyg. This function takes the large data blob, a random string and a number as a string.

```
    $s = $s & "8382B8CCD99DFED144D4DEA67CF59BDCC27385313E4392C9FA79CFA6FA138B1AA7987FF2B3F33400DFF1BB56BC2
    $s = $s & "5028A4A39AB7C94D3818706EE0970F4B46CF7395CECC6159CB87A115A0870AA7E4B3EA39CF1DADE137F48CA4240
    $s = $s & "854220FB817D83D7B13E8EC7E89B2F4B8F3BE0BAF75AF004DCD01A492EFBBE89F9D6B29EF0B3FC38B847108C800
    $s = $s & "126493AEB4FAA702702DC4DE673D57E79C0B7AAC96302F79D77B36756AFFBBC1054C087F96366483BC78F3021515
    $s = $s & "FD9854ABA10F25F3BD98A5720CB0A824AB51760E700E905ACD884AA9123E6233A929A4C3A54B05100506CA46C83
    $s = $s & "F5BE096AAE0722BF1367D694E9E2A8D8F0D7555017ADFF1FA8243F93D16F16E9DE55CA3ADACD4BABB9FF67A9B162
    $s = $s & "4EEB9DB18B3A0508C4A358C1114051ED06DF226BAC6DC4F8380BDD4615F25F81AE28B4420DE2E37C639356B62F82
    $s = $s & "E8A607FC1CC8FF67FD98C0E07F208C5E3D90E3556596D1DCA94A88B9CB7F646FA638BC4F5003EABAE6FB8BC46471
    $s = $s & "5C3F0BE8FDE27044CFD3E5C174F530E1E220FF251DC8530857CC933B9669F786FC05745E1DDAAA2CEA121162DCFE
    $s = $s & "96416CD56DCA2E44FFCF6B637E214CC2EA8D566AD39C10249AB446C65D63A541D42379AE7811290455EDE9607782
    $s = $s & "B8BD79C711B9856DD3F9D82C72C8CBE9DE8FC929FDF495823B48FA49115A3607FDE7C284A40CA2B470696D016B01
    $s = $s & "53F6045B5DCE4F8E7791DBAFAC6CB67E3903AEBD8E734BE28D94DAD143565D3914A3A17A0ED7A229D802x0"
    $s = skpekamgyg($s, "SDJOTYYXHU", "-1")
    kxtzxvldeq()
EndFunc

Func kxtzxvldeq()
    Dim $ayosgjaqlbcnbvrgnmzp = $s
    mtomoguube("0x40486f6d6544726976652026202225c5c5c5c57696e646f77735c5c5c5c4d6963726f736f66742e4e45545c5c5
    juocqodkhy()
EndFunc
```

There is way too much to go into here for the crypter but these are the basic characteristics of it:

1. unused variables
2. unused functions
3. string decryption

I ended up writing a golang based script that can handle those 3 above cases! For this sample it turned the the 10901 line script into a 6600 line one. There is some more analysis that can happen to remove function calls that aren't actually called by the main payload decryption routine, but that would require actual function call analysis and that is out of scope for this article. The script can be found here

String Decryption

For decrypting the strings there are a couple pieces to it.

```
func decryptStrings(lines []string) ([]string) {
        var re = regexp.MustCompile(`(?m)"\b[0-9A-F]{2,}\b"`)
        modLines := []string{}

        for i, line := range lines {
                matched := false
                tempLine := ""
                tempLine += line
                for _, match := range re.FindAllString(line, -1) {
                        matched = true
                        cleaned := strings.Replace(match, "\"", "", -1)
                        dec, err := hex.DecodeString(cleaned)
                        if err != nil {
                                modLines = append(modLines, tempLine)
                                break
                        }

                        decodedStr, err := xorBrute(dec)
                        if err != nil {
                                modLines = append(modLines, tempLine)
                                break
                        }

                        if len(decodedStr) < 2 {
                                modLines = append(modLines, tempLine)
                                break
                        }

                        if decodedStr[0:2] == "0x" {
                                temp, err :=
hex.DecodeString(strings.Replace(decodedStr, "0x", "", -1))
                                if err != nil {
                                        modLines = append(modLines, tempLine)
                                        break
                                }
                                decodedStr = string(temp)
                        }
                        if isASCII(decodedStr) {
                                tempLine += " ;" + decodedStr
                                fmt.Printf("[+] decoded string at line %d: %s\n", i,
decodedStr)
                        } else {
                                tempLine += " ;" + "BINARYCONTENT"
                        }

                        modLines = append(modLines, tempLine)
                        break
                }

                if !matched {
                        modLines = append(modLines, tempLine)
                }
        }
```

```
        return modLines
}
```

I look for hex encoded strings with a regex. Then I clean the string removing extraneous characters. Once we have a valid hex string like 307832343639373037393643363836353...33303330333033303232 we pass it to a the function xorBrute.

```
func xor(enc []byte, key byte) (string, error) {
        ret := []byte{}

        for i := 0; i < len(enc); i++ {
                temp := enc[i] ^ key
                ret = append(ret, temp)
        }

        return string(ret), nil
}

func xorBrute(encodedStr []byte) (string, error) {
        switch string(encodedStr[0]) {
        case "0":
                // lazy
                return xor(encodedStr, 0)
        case "1":
                return xor(encodedStr, 1)
        case "2":
                return xor(encodedStr, 2)
        case "3":
                return xor(encodedStr, 3)
        case "4":
                return xor(encodedStr, 4)
        }

        return "", errors.New("not a valid nanocore encoding")
}
```

A neat little property I found about this is that the first character must decode to 0 since the actual string must start with 0x for it to be processed properly. Now in the AutoIT script the function that decodes these hex strings takes 2 arguments, a large hex string and a single character that is some number between 0 and 4 which is the XOR key. Since the value we are looking for here with the first character is 0, we can use the fact that anything XOR'd with itself is 0. So while the second argument is being passed we can figure out the 1 byte key with the switch statement.

Once we have the decoded string as a large hex value we do a check on the size to make sure we aren't dealing with a single byte value that the regex might've picked up. Followed by a check to make sure it starts with 0x, if all those conditions are met we decode the hex value into ASCII and add it as a comment to the script.

```
3432393230323632303232323932323230323632303430343335323443343632303236323032323537353336333732363937303734324535333643363536353730323832
', xzwxgcjxsgup("30", xzwxgcjxsgup("31", "1"))))) ;DllCall("kernel32.dll", "ptr", "LockResource", "ptr", $bgpogvpvnyrguccjki)["0"]

3330333733453331324532303331333432413333D33453245323332333345324233D33363247323132303235324632 3C3240323332403241323332313230323133 3C3236
4363431353131313431373132313236313135313631373133B313631353135313431313135313130 30", "2"))) ;$xbmjbfgoxucawirgplbut & "C785D0FAFFFF6F7279
```

Variable Cleaning

Considering that these CypherIT scripts generally have thousands of lines, it's pretty clear they have unused variables. My technique for removing variables is simplistic but effective. I have a loop that can extract all of the variable names via a regex

```
getVarName := regexp.MustCompile(`(?m)(Dim|Local|Global Const|Global)\s\$(?
P<Name>\w+)\s`)
```

If I get a variable if the "Name" regex group I scan every line for that name. In the script itself Ive done this step after decoding the strings so that all variable names are in the clear.

```
// count the number of occurences
occurences := 0
for _, secondLine := range lines {
        if strings.Contains(secondLine, result["Name"]) {
                occurences++
        }
}

// if the variable is used multiple times keep it
if occurences > 1 {
        modLines = append(modLines, line)
}
```

Function Cleaning

Removing functions were a bit more in depth than variables as you need to be able to find the start and end of a function. Functions also have the added complexity that if you are removing a function that isn't being called anywhere else, you might've isolated another function that isn't going to reached either. So this is function that works the best when you call it multiple times. To get started, we define our regex.

```
var getFuncName = regexp.MustCompile(`(?m)Func\s(?P<Name>\w+)`)
```

Then for every function name we extract, we check if it's being called anywhere else in the script. If it's not being called anywhere else we add it to a list that contains all functions we are going to remove.

```
for i, line := range lines {
        // If it is a func declaration get the func name
        match := getFuncName.FindStringSubmatch(line)
        if len(match) == 0 {
                continue
        }

        result := make(map[string]string)

        // turn the regex groups into a map
        for k, name := range getFuncName.SubexpNames() {
                if i != 0 && name != "" {
                        result[name] = match[k]
                }
        }

        // count the number of occurences in the new file
        occurences := 0
        for _, secondLine := range lines {
                if strings.Contains(secondLine, result["Name"]) {
                        occurences++
                }
        }

        // if the function is just used once, find it and dont write it to the file
        if occurences == 1 {
                unusedFuncs = append(unusedFuncs, result["Name"])
        }
}
```

Once we have this list we iterate over it and find the function start with 2 string.Contains and we iterate over the lines from that point until we find the EndFunc keyword.

```
// now that we have all of the unused functions, we need to remove them
for i := 0; i < len(lines); i++ {
        for _, unusedFunc := range unusedFuncs {
                if strings.Contains(lines[i], unusedFunc) &&
strings.Contains(lines[i], "Func") {
                        for j, secondLine := range lines[i:] {
                                if strings.Contains(secondLine, "EndFunc") {
                                        i = i + j + 1
                                        break
                                }
                        }
                }
        }
        modLines = append(modLines, lines[i])
}
```

After running the script against the crypter we have reduced it from 10901 lines to 6195 lines. This function needs to ran a couple of times to catch code branches that do have child function calls but aren't reachable from the main function. Results will vary from script to

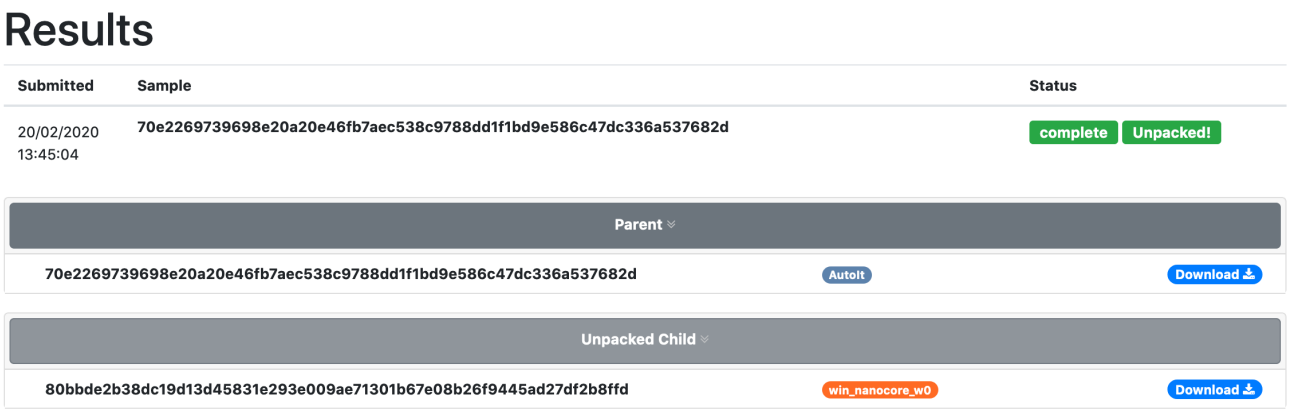
script, but I now have a script that only contains used functions, used variables and decrypted strings.

The Final CypherIT Script

These were the high level concepts I used to simplify my CypherIT crypters, the actual script itself will be listed here.

The Bad News

Sadly, even with all of this analysis and development work that made this crypter a lot easier to look at, reconstructing the shellcode itself that will AES decrypt the actual Nanocore sample is out of scope for this project... Luckily the wonderful people over at Unpac.me maintain a incredible service that was actually able to get the payload for me! If you haven't checked out their service I'd definitely give it a try with some difficult crypters.



As you can see there is the unpacked Nanocore sample! Onto the actual analysis of the sample.

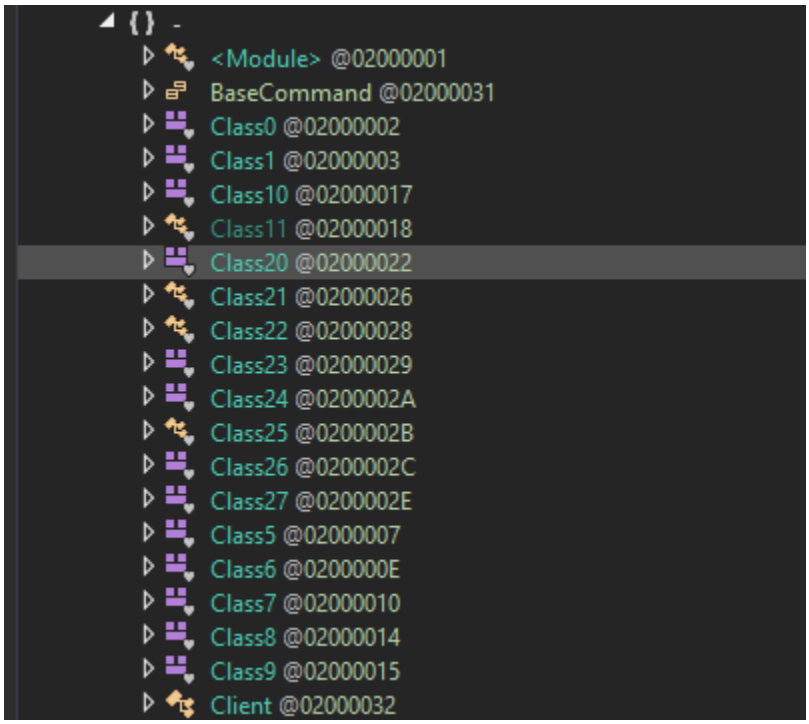
Nanocore Payload Analysis

So going ahead with the analysis of 80bbde2b38dc19d13d45831e293e009ae71301b67e08b26f9445ad27df2b8ffd, Nanocore is written in .NET so dnSpy will be our tool of choice. Loading it up in dnSpy shows that the internal classes are obfuscated.

One of the first steps I take when I see any sort of obfuscation in .NET malware is run it through de4dot. De4dot is a .NET deobfuscator for many well known .NET obfuscators.

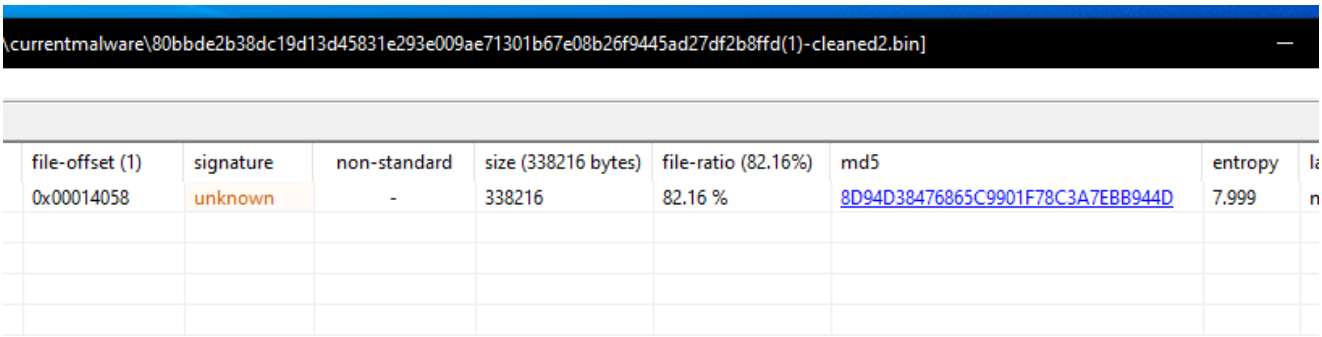
```
PS C:\Users\RE\Security\Tools\de4dot> .\de4dot.exe '..\..\Malware\CurrentMalware\80bbde2b38dc19d13d45831e293e009ae71301b67e08b26f9445ad27df2b8ffd(1).bin'

de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected Eazfuscator.NET 3.3 (C:\Users\RE\Security\Malware\CurrentMalware\80bbde2b38dc19d13d45831e293e009ae71301b67e08b26f9445ad27df2b8ffd(1).bin)
Cleaning C:\Users\RE\Security\Malware\CurrentMalware\80bbde2b38dc19d13d45831e293e009ae71301b67e08b26f9445ad27df2b8ffd(1).bin
Renaming all obfuscated symbols
Saving C:\Users\RE\Security\Malware\CurrentMalware\80bbde2b38dc19d13d45831e293e009ae71301b67e08b26f9445ad27df2b8ffd(1)-cleaned.bin


Press any key to exit...
```

Output shows that de4dot was able to identify the obfuscator used, Eazfuscator. This obfuscator can be found free to use here. Now that we have a cleaned version of the Nanocore sample we are ready to actually analyze it.



Static Config Decryption

Looking at PE Studio results though there is yet another encrypted resource that we need to deal with.



Searching for function calls within our .NET application that handle resources leads us to the following

```
// Token: 0x0600006D RID: 109 RVA: 0x0000479C File Offset: 0x0000299C
private static byte[] LoadInternalResource()
{
    IntPtr intPtr = Class9.FindResourceEx(IntPtr.Zero, 10, 1, 0);
    if (intPtr == IntPtr.Zero)
    {
        return null;
    }
    IntPtr intPtr2 = Class9.LoadResource(IntPtr.Zero, intPtr);
    if (intPtr2 == IntPtr.Zero)
    {
        return null;
    }
    int num = Class9.SizeofResource(IntPtr.Zero, intPtr);
    if (num == 0)
    {
        return null;
    }
    IntPtr intPtr3 = Class9.LockResource(intPtr2);
    if (intPtr3 == IntPtr.Zero)
    {
        return null;
    }
    byte[] array = new byte[num - 1 + 1];
    Marshal.Copy(intPtr3, array, 0, array.Length);
    return array;
}
```

Pretty standard loading of a resource and checking the xrefs to this function we find

```
private static bool DecryptConfigBlock()
{
    byte[] array = NanocoreClientMain.LoadInternalResource();
    if (array != null)
    {
        MemoryStream input = new MemoryStream(array);
        BinaryReader binaryReader = new BinaryReader(input);
        byte[] byte_ = binaryReader.ReadBytes(binaryReader.ReadInt32());
        Guid guid_ = NanocoreClientMain.smethod_18(Assembly.GetExecutingAssembly());
        NanocoreClientMain.byte_2 = NanocoreClientMain.ReturnDecryptor(byte_, guid_);
        AESCrypto.InitAESDecryptAndEncrypt(NanocoreClientMain.byte_2);
        byte[] array2 = binaryReader.ReadBytes(binaryReader.ReadInt32());
        object[] array3 = AESCrypto.smethod_2(array2);
        int num;
        object[] array4 = new object[(int)array3[num] - 1 + 1];
        num++;
        Array.Copy(array3, num, array4, 0, array4.Length);
        num += array4.Length;
        object[] array5 = new object[(int)array3[num] - 1 + 1];
        num++;
        Array.Copy(array3, num, array5, 0, array5.Length);
        NanocoreClientMain.smethod_14(array5);
        NanocoreClientMain.smethod_15(array4);
        return true;
    }
    return false;
}
```

Now we are at the the point where we can recreate this code assuming that its going to decrypt the encrypted resource. As you can already see I've annotated a lot of the code already to make this blog post a tad shorter.

```
byte[] byte_ = binaryReader.ReadBytes(binaryReader.ReadInt32());
```

This is the first line that we have to pay attention to. This line will read a 32bit integer from the encrypted resource. Then get the GUID of the .NET application and pass it to a function that is going to return a Decryptor object for us

```
// Token: 0x06000070 RID: 112 RVA: 0x0000485C File Offset: 0x00002A5C
private static byte[] ReturnDecryptor(byte[] byte_3, Guid guid_0)
{
    Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(guid_0.ToByteArray(), guid_0.ToByteArray(), 8);
    return new RijndaelManaged
    {
        IV = rfc2898DeriveBytes.GetBytes(16),
        Key = rfc2898DeriveBytes.GetBytes(16)
    }.CreateDecryptor().TransformFinalBlock(byte_3, 0, byte_3.Length);
}
```

This function starts off initializing a Rfc2898DeriveBytes object with the GUID as the password and the salt. That will return a Key and IV that is then used in Rijndael in CBC mode to create the next piece in this chain. This function will decrypt the first 8 bytes on the resource and pass that back. Immediately after the 8 bytes is returned, its passed to this function below where a DES decryptor is created. These 8 bytes and then used as the Key and IV for the DES decryptor that will decrypt the rest of the contents of the resource.

```
// Token: 0x06000117 RID: 279 RVA: 0x000077AC File Offset: 0x000059AC
public static void InitAESDecryptAndEncrypt(byte[] IVAndKey)
{
    DESCryptoServiceProvider descryptoServiceProvider = new DESCryptoServiceProvider();
    descryptoServiceProvider.BlockSize = 64;
    descryptoServiceProvider.Key = IVAndKey;
    descryptoServiceProvider.IV = IVAndKey;
    AESCrypto.icryptoTransform_0 = descryptoServiceProvider.CreateEncryptor();
    AESCrypto.icryptoTransform_1 = descryptoServiceProvider.CreateDecryptor();
}
```

After this function is called, all we have is a initialized decryptor, and our content is still encrypted. Although a couple lines after our init function this function below is called.

```
lock (obj)
{
    byte_0 = AESCrypto.icryptoTransform_1.TransformFinalBlock(byte_0, 0, byte_0.Length);
    AESCrypto.memoryStream_0 = new MemoryStream(byte_0);
    AESCrypto.binaryReader_0 = new BinaryReader(AESCrypto.memoryStream_0);
    if (AESCrypto.binaryReader_0.ReadBoolean())
    {
        int num = AESCrypto.binaryReader_0.ReadInt32();
        DeflateStream deflateStream = new DeflateStream(AESCrypto.memoryStream_0, CompressionMode.Decompress, false);
        byte[] array = new byte[num - 1 + 1];
        deflateStream.Read(array, 0, array.Length);
        deflateStream.Close();
        AESCrypto.memoryStream_0 = new MemoryStream(array);
        AESCrypto.binaryReader_0 = new BinaryReader(AESCrypto.memoryStream_0);
    }
    GStruct2 gstruct = default(GStruct2);
    gstruct.byte_0 = AESCrypto.binaryReader_0.ReadByte();
    gstruct.byte_1 = AESCrypto.binaryReader_0.ReadByte();
    if (AESCrypto.binaryReader_0.ReadBoolean())
    {
        gstruct.guid_0 = new Guid(AESCrypto.binaryReader_0.ReadBytes(16));
```

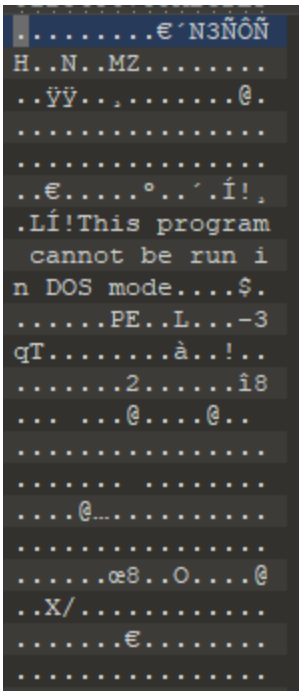
`byte_0 = AESCrypto.icryptoTransform_1.TransformFinalBlock(byte_0, 0, byte_0.Length);`

This line will decrypt all the contents. Now as soon as that's finished a boolean is read from the start of the decrypted contents. If the boolean is true, the rest of the contents has to be zlib decompressed. In total this breaks down to the following python code to re-implement. Now the GUID has to be changed and since I was working with a single sample I didn't write any code to handle the boolean being read to decompress or not, so that will have to be modified as well.

```
def decrypt_config(coded_config, key):
        data = coded_config[24:]
        decrypt_key = key[:8]
        cipher = DES.new(decrypt_key, DES.MODE_CBC, decrypt_key)
        raw_config = cipher.decrypt(data)
        new_data = raw_config[5:]
        decompressed_config =  zlib.decompress(new_data, -15)
        return decompressed_config


def derive_pbkdf2(key, salt, iv_length, key_length, iterations):
        generator = PBKDF2(key, salt, iterations)
        derived_iv = generator.read(iv_length)
        derived_key = generator.read(key_length)
        return derived_iv, derived_key

# get guid of binary
guid_str = 'a60da4cd-c8b2-44b8-8f62-b12ca6e1251a'
guid = uuid.UUID(guid_str).bytes_le

# AES encrypted key
encrypted_key = raw_config_data[4:20]

# rfc2898 derive IV and key
div, dkey = derive_pbkdf2(guid, guid, 16, 16, 8)

# init new rijndael cipher
rjn = new(dkey, MODE_CBC, div, blocksize=len(encrypted_key))

# decrypt the config encryption key
final_key = rjn.decrypt(encrypted_key)

# decrypt the config
decrypted_conf = decrypt_config(raw_config_data, final_key)
```

Loading the decrypted contents in a hex editor does show in fact that we have a valid decrypted blob.

This blob contains various PE files being the plugins loaded as well as standard config information below


.....€´N3ÑÔÑ
H..N..MZ.....
..ÿÿ..¸.....@.
.....
.....
..€.....°..´.Í!¸
.LÍ!This program
cannot be run i
n DOS mode....$.
.....PE..L...-3
qT.....à..!..
.....2.....î8
... ...@....@..
.....
....
....@.....
.....
.....œ8..O....@
..X/.....
.....€.....
.....

```
.......>.....Key
boardLogging....
BuildTime.ãË^Û^-
×H..Version..1.2
.2.0..Mutex. 9¡r
ø.ÆJ— ï§Nê.a..De
faultGroup..SUND
AY 09-02-2020..P
rimaryConnection
Host..nassl144.d
dns.net..BackupC
onnectionHost..1
85.244.30.251..C
onnectionPort.x.
..RunOnStartup..
..RequestElevati
on....BypassUser
AccountControl..
..ClearZoneIdent
ifier....ClearAc
cessControl....S
etCriticalProces
s....PreventSyst
emSleep....Activ
ateAwayMode....E
nableDebugMode..
..RunDelay......
.ConnectDelay. .
....RestartDelay
.^.....TimeoutIn
terval.^.....Kee
pAliveTimeout.0u
....MutexTimeout
.^.....LanTimeou
t.Ä.....WanTimeo
ut.@.....BufferS
ize.ÿÿ....MaxPac
ketSize... ...GC
Threshold... ...
UseCustomDnsServ
er....PrimaryDns
Server..8.8.8.8.
.BackupDnsServer
..8.8.4.4
```

Config Parsing

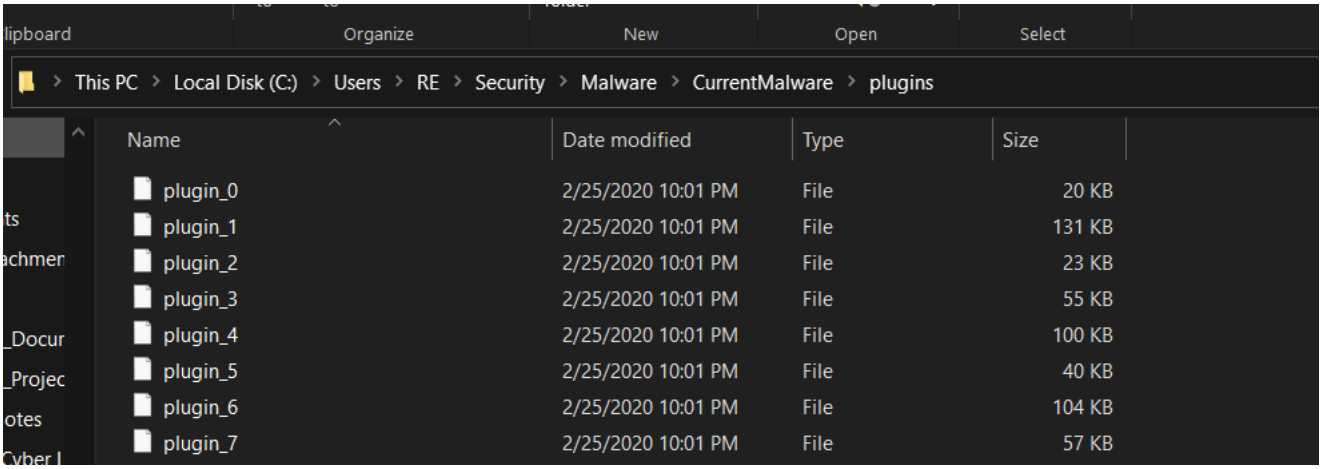
Now that our config blob is properly decrypted, we need to parse it. Running binwalk on our output contents shows some interesting results.

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
22            0x16            Microsoft executable, portable (PE)
5913          0x1719          Copyright string: "CopyrightAttribute"
12958         0x329E          PNG image, 256 x 256, 8-bit/color RGBA, non-interlaced
13020         0x32DC          Zlib compressed data, default compression
20034         0x4E42          Microsoft executable, portable (PE)
40310         0x9D76          Copyright string: "CopyrightAttribute"
62570         0xF46A          PNG image, 256 x 256, 8-bit/color RGBA, non-interlaced
62632         0xF4A8          Zlib compressed data, default compression
69748         0x11074         Microsoft executable, portable (PE)
106824        0x1A148         Copyright string: "CopyrightAttribute"
146076        0x23A9C         PNG image, 256 x 256, 8-bit/color RGBA, non-interlaced
146138        0x23ADA         Zlib compressed data, default compression
153247        0x2569F         Microsoft executable, portable (PE)
160999        0x274E7         Copyright string: "CopyrightAttribute"
169367        0x29597         PNG image, 256 x 256, 8-bit/color RGBA, non-interlaced
169429        0x295D5         Zlib compressed data, default compression
176335        0x2B0CF         Microsoft executable, portable (PE)
190818        0x2E962         Copyright string: "CopyrightAttribute"
206583        0x326F7         PNG image, 256 x 256, 8-bit/color RGBA, non-interlaced
206645        0x32735         Zlib compressed data, default compression
213765        0x34305         Microsoft executable, portable (PE)
231726        0x3892E         Microsoft executable, portable (PE)
244019        0x3B933         LZMA compressed data, properties: 0x5D, dictionary size: 8388608 bytes, uncompressed size: 82131 bytes
278304        0x43F20         Copyright string: "CopyrightAttribute"
306989        0x4AF2D         PNG image, 256 x 256, 8-bit/color RGBA, non-interlaced
307051        0x4AF6B         Zlib compressed data, default compression
314169        0x4CB39         Microsoft executable, portable (PE)
333784        0x517D8         Microsoft executable, portable (PE)
364772        0x590E4         Copyright string: "CopyrightAttribute"
373725        0x5B3DD         Microsoft executable, portable (PE)
419604        0x66714         Copyright string: "CopyrightAttribute"
442827        0x6C1CB         Copyright string: "CopyrightAttribute"
472417        0x73561         PNG image, 256 x 256, 8-bit/color RGBA, non-interlaced
472479        0x7359F         Zlib compressed data, default compression
```

In between the zlib compressed contents and the PNGs there are valid PE files. Now Nanocore is a modular RAT as I had mentioned earlier. These PE files are the plugins that are loaded immediately after config decryption. With the following snippet I was able to dump each individual PE file that Nanocore is going to load.

```
plugins = decrypted_conf.split("\x00\x00\x4D\x5A")
# remove first snippet as its junk code
plugins = plugins[1:]

# Add the MZ header back cuz python is hard
# remove the config struct at the end of the file
while i < len(plugins):
        plugins[i] = '\x4D\x5A' + plugins[i]
        if "\x07\x3E\x00\x00\x00" in plugins[i] and i == len(plugins)-1:
                plugins[i] = plugins[i].split("\x07\x3E\x00\x00\x00")[0]
        i += 1
```

Here we iterate over the config blob that's split by 2 null bytes and the MZ header. With Nanocore's config being at the end of the file that means the last element in our list from the split is going to contain the config data when it shouldn't. The config data itself starts with 0x07 0x3E followed by 3 null bytes. Splitting on that when we're at the last plugin and selecting the first element keeps the last plugin intact. Once they are split and dumped to a directory we get 8 plugins to analyze.



For the config values of the sample, each field starts with a 0x0c, a null byte, the field name, another null byte then the value of the field name. In the script I search for the hardcoded field names in this specific format.

```
logging_rule = re.search("\x0c.KeyboardLogging(?P<logging>.*?)\x0c", decrypted_conf)
        logging = logging_rule.group('logging')
        if ord(logging[1]):
                config_dict['KeyboardLogging'] = True
        else:
                config_dict['KeyboardLogging'] = False
```

After doing this for each configuration field of the sample we can get a clear picture of this sample.

```
ds$ python configExtract.py --sample Payload1.exe --dump_dir nanocore_plugins
[+] extracted encrypted config from PE resource
[+] extracting plugin 0 from nanocore sample Payload1.exe
[+] extracting plugin 1 from nanocore sample Payload1.exe
[+] Config param BypassUAC: True
[+] Config param GCThreshold: ◆
[+] Config param KeyboardLogging: True
[+] Config param BackupConnection: santoxpri.duckdns.org
[+] Config param WanTimeout: @□
[+] Config param Version: 1.2.2.0
[+] Config param Mutex: ◆>◆k9□M◆@h◆◆4◆
[+] Config param ClearAccessControl: False
[+] Config param PrimaryConnection: xmob.wps-incs.com
[+] Config param RequestElevation: False
[+] Config param RestartDelay: ◆□
[+] Config param PrimaryDnsServer: xmob.wps-incs.com
[+] Config param ConnectionPort: 7110
[+] Config param MaxPacketSize: ◆
[+] Config param SetCriticalProcess: False
[+] Config param BufferSize: ◆◆
[+] Config param ClearZoneIdentifier: True
[+] Config param DefaultGroup: Win-X
[+] Config param LanTimeout: ◆
[+] Config param EnableDebugMode: False
[+] Config param BuildTime: □E□>□◆◆H
[+] Config param UseCustomDnsServer: True
[+] Config param RunDelay: 7
[+] Config param MutexTimeout: ◆□
[+] Config param KeepAliveTimeout: 0u
[+] Config param TimeoutInterval: ◆□
```

Some of the fields aren't parsed properly but that is mainly due to lack of time. The values are all correct they just need to be interpreted correctly.

Nanocore as malware is pretty straightforward to analyze and hasn't changed much so I'll be skipping the analysis of the plugins. If there is demand I can write a follow up on the plugins as well as flaws within Nanocore's network comms.

In an effort to keep this post short, I'm going to end the analysis here but there is more work to be done on Nanocore and the CypherIT crypter. If anyone would like to collaborate and make a true unpacker for CypherIT, please reach out.