

## 03.customm\_.exe 壳：

### 一、特征：

1. 由于 CustomM 壳是定制的，传统的壳检测工具（如 PEiD、ProtectionID）可能无法准确识别壳类型。
2. 特征不明确：特征码不如常见壳（如 UPX、ASPack 等）明显，通常依赖动态分析、调试跟踪来了解其保护机制。

### 二、工具：x32dbg(scylla 插件)+IAT 表恢复脚本

### 三、总结：

#### 1、如何检查程序没有壳？

PEID 等 PE 文件结构检测工具看壳类型、入口点、IAT、节表数量、字符串等。

#### 2、工具检测未发现 IAT 表，那程序是如何正常调用 API 呢？

此文中待脱壳程序使用工具检测未发现 IAT 表具有加壳表现，通过 X32dbg 调试器动态分析。发现 TEB 结构的访问，继续监控分析：API 获取流程是通过 TEB->PEB->Ldr 获取模块和 API 地址。

#### 3、壳功能？

通过 TEB 查找 API 地址，加密并存到另外内存中，使用时动态解密函数地址。修复 IAT 办法：赶在壳加密 API 地址前就把地址保存下来，更新到 IAT 表中。

#### 4、为什么 x32dbg 中脚本不能使用 JNE 反而要使用 jne 指令，也不能用 JMP 反而使用 jmp？

官方帮助文档的语法就是 jne 小写。注：【x64dbg.chm 是官方帮助文档，下载的安装包里面自带了此文档。CHM Editor 工具可以用来汉化编辑帮助文档】

#### 5、dwEAX 变量的值在哪里看？地址在哪？IAT 表地址在哪里？如何转储文件？

X32dbg 命令行输入：msg {dwEAX}可显示变量的值。

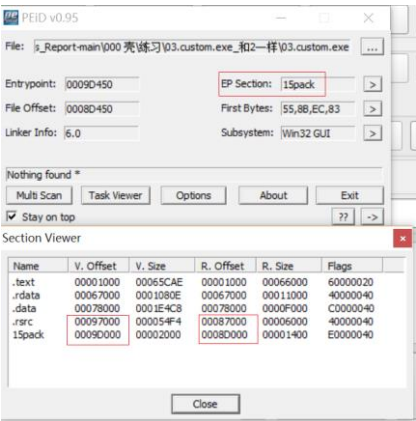
IAT 表地址看程序动态分配的地址，此程序中加密的 API 地址存放位置在 EDI 寄存器处。

转储文件：找到正确 OEP，能正常识别 IAT 表(找到加密前的 API 地址并更新存到 EDI 中，使得 Scylla 能正常识别 IAT 表，修复转储即可)。

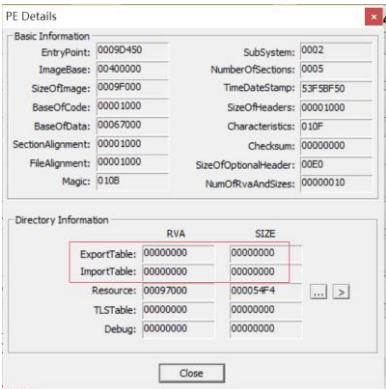
四、脱壳示例：

1、查看待脱壳文件信息：EP 有异常（不是通常的指向.text 节）、没有 IAT 表。

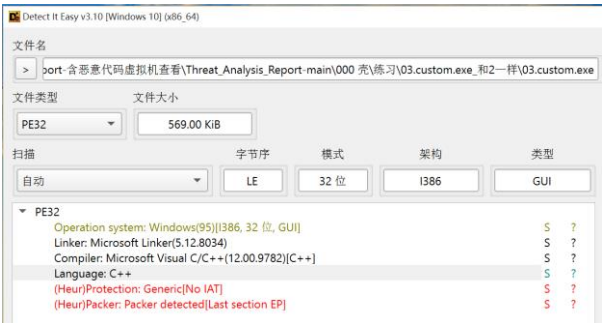
EP 指向 15pack 节：



没有导入导出表：那程序执行时是如何调用 API 的？

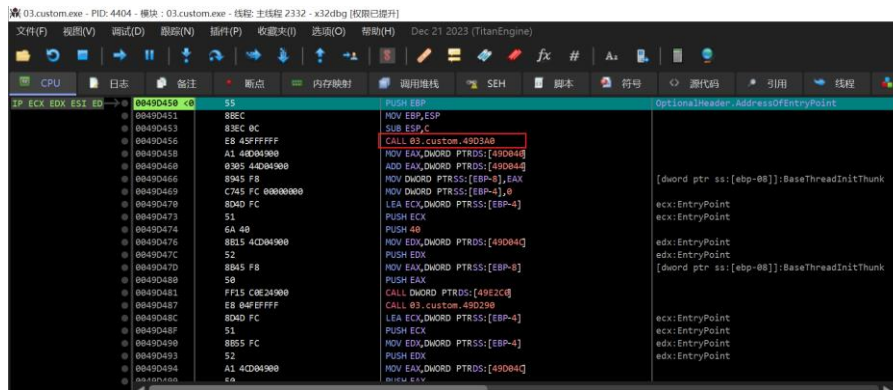


其他工具检测：

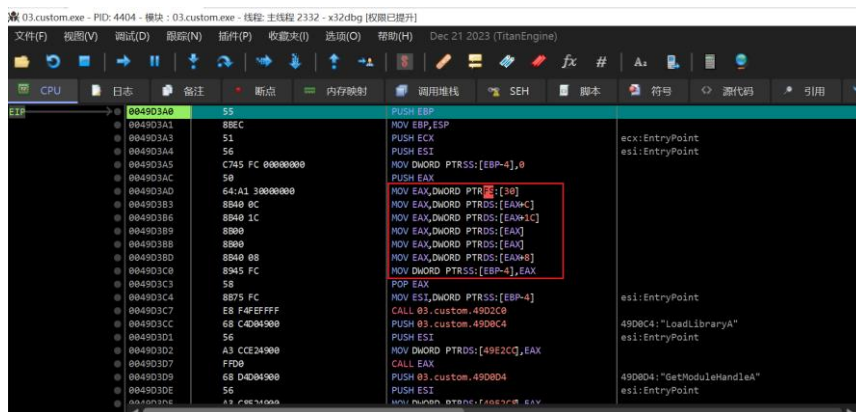


2、x32dbg 调试：

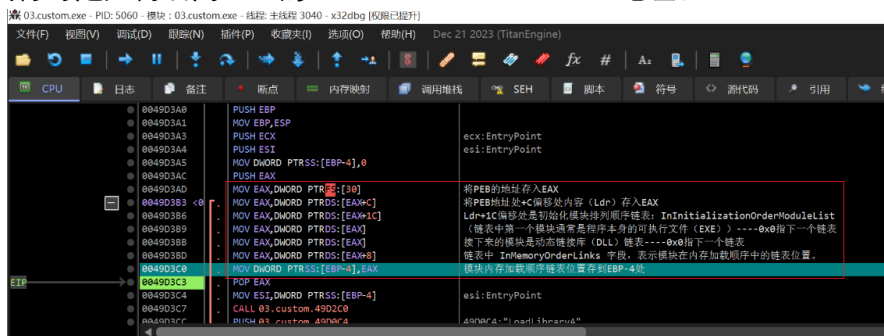
运行程序跳到 0x49D450 地址并不像 OEP:



单步运行进 call 0x49D3A0:



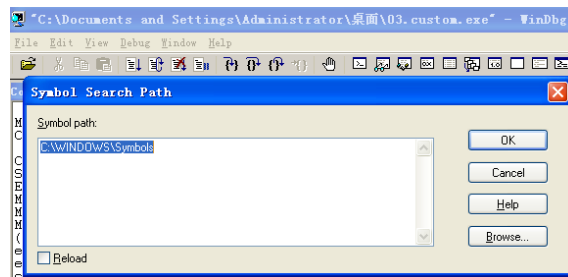
那程序执行时是如何调用 API 的？TEB->PEB->Ldr->API 地址。



下面详细解释如何用 WinDbg 查看相关 TEB 和 PEB 的数据结构!!!

Windbgxp 中查看 fs:[30]处数据结构:

添加符号文件:



查看 TEB 地址: 0x7ffdf000, 即 FS 寄存器中的值为 7ffdf000

```
"C:\Documents and Settings\Administrator\桌面\03.custom.exe" - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Command
Microsoft (R) Windows Debugger Version 6.12.0002.633 X86
Copyright (c) Microsoft Corporation. All rights reserved.
CommandLine: "C:\Documents and Settings\Administrator\桌面\03.custom.exe"
Symbol search path is: C:\WINDOWS\Symbols
Executable search path is:
ModLoad: 00400000 00400000 image00400000
ModLoad: 7c900000 7c9b3000 ntdll.dll
ModLoad: 7c800000 7c91e000 C:\WINDOWS\system32\kernel32.dll
(10c7c): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=77fd7000 ecx=00000001 edx=00000002 esi=00241f48 edi=00241eb4
eip=7c92120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c92120e cc          int     3
(0:000) > !teb
TEB at 77fd0000
ExceptionList: 0012fd0c
StackBase: 00130000
StackLimit: 0012e000
SubSystemTib: 00000000
FiberData: 00001e00
ArbitraryUserPointer: 00000000
Self: 77fd0000
EnvironmentPointer: 00000000
ClientId: 0000010c . 0000007c
RpcHandle: 00000000
Tls Storage: 00000000
PEB Address: 77fd7000
LastErrorValue: 0
LastStatusValue: 0
Count Owned Locks: 0
HardErrorMode: 0
```

TEB内存地址

输入 dt \_TEB 可以查看 TEB 的具体结构，可以看到 0x30 处指向了 PEB 结构

```
"C:\Documents and Settings\Administrator\桌面\03.custom.exe" - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Command
0:000> dt _TEB
ntdll!_TEB
+0x000 NtTib : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId : CLIENT_ID
+0x028 ActiveRpcHandle : Ptr32 Void
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
+0x034 LastErrorValue : Uint4B
+0x038 CountOfOwnedCriticalSections : Uint4B
+0x03c CsrClientThread : Ptr32 Void
+0x040 Win32ThreadInfo : Ptr32 Void
+0x044 User32Reserved : [26] Uint4B
+0x04c UserReserved : [5] Uint4B
+0x050 WOW32Reserved : Ptr32 Void
+0x054 CurrentLocale : Uint4B
+0x058 FpSoftwareStatusRegister : Uint4B
+0x05c SystemReserved1 : [54] Ptr32 Void
+0x064 ExceptionCode : Int4B
+0x068 ActivationContextStack : _ACTIVATION_CONTEXT_STACK
+0x06c SpareBytes1 : [24] UChar
+0x070 GdiTebBatch : _GDI_TEB_BATCH
+0x074 RealClientId : _CLIENT_ID
```

PEB

可以继续查看 PEB 的结构，能够看到偏移 c 处为 Ldr(当前进程加载模块的相关信息)，该成员为 \_PEB\_LDR\_DATA 结构体：

```
"C:\Documents and Settings\Administrator\桌面\03.custom.exe" - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Command
0:000> dt _PEB
ntdll!_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 SpareBool : UChar
+0x004 Mutant : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData : Ptr32 Void
+0x018 ProcessHeap : Ptr32 Void
+0x01c FastPebLock : Ptr32 _RTL_CRITICAL_SECTION
+0x020 FastPebLockRoutine : Ptr32 Void
+0x024 FastPebUnlockRoutine : Ptr32 Void
+0x028 EnvironmentUpdateCount : Uint4B
+0x02c KernelCallbackTable : Ptr32 Void
+0x030 SystemReserved : [1] Uint4B
+0x034 AtlThunkSListPtr32 : Uint4B
+0x038 FreeList : Ptr32 _PEB_FREE_BLOCK
+0x03c TlsExpansionCounter : Uint4B
```

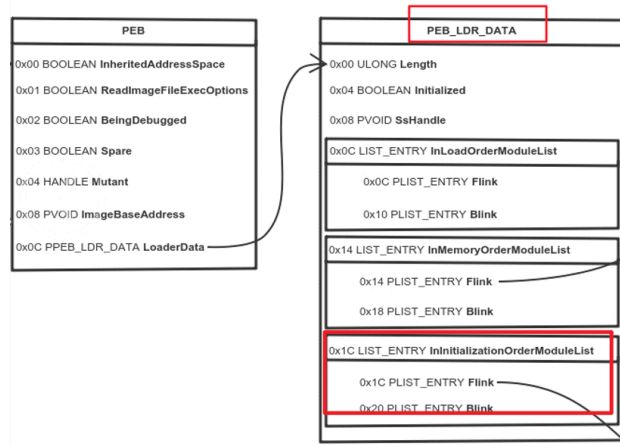
那么用 dt \_PEB\_LDR\_DATA 命令查看 \_PEB\_LDR\_DATA 结构体的结构，偏移 1c 指向哪里呢？可以看到指向了 InInitializationOrderModuleList (// 按初始化顺序排列的模块链表)，这个成员是个 LIST\_ENTRY 结构体：

```
0:000> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
+0x000 Length : Uint4B
+0x004 Initialized : UChar
+0x008 SsHandle : Ptr32 Void
+0x00c InLoadOrderModuleList : _LIST_ENTRY
+0x014 InMemoryOrderModuleList : _LIST_ENTRY
+0x01c InInitializationOrderModuleList : _LIST_ENTRY
+0x024 EntryInProgress : Ptr32 Void
```

Flink 指前一个链表地址，Blink 指下一个链表地址：

```
0:000> dt _LIST_ENTRY
ntdll!_LIST_ENTRY
+0x000 Flink      : Ptr32 _LIST_ENTRY
+0x004 Blink      : Ptr32 _LIST_ENTRY
```

通过图看下 PEB\_LDR\_DATA 的成员，下图中的 0x1c 处的 InInitializationOrderModuleList



每个链表（InInitializationOrderModuleList、InMemoryOrderModuleList、InLoadOrderModuleList）的数据结构：

dt \_LDR\_DATA\_TABLE\_ENTRY

```
0:000> dt _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x010 InInitializationOrderLinks : _LIST_ENTRY
+0x018 DllBase : Ptr32 Void
+0x01c EntryPoint : Ptr32 Void
+0x020 SizeOfImage : UInt4B
+0x024 FullDllName : _UNICODE_STRING
+0x02c BaseDllName : _UNICODE_STRING
+0x034 Flags : UInt4B
+0x038 LoadCount : UInt2B
+0x03a TlsIndex : UInt2B
+0x03c HashLinks : _LIST_ENTRY
+0x03c SectionPointer : Ptr32 Void
+0x040 CheckSum : UInt4B
+0x044 TimeDateStamp : UInt4B
+0x044 LoadedImports : Ptr32 Void
+0x048 EntryPointActivationContext : Ptr32 Void
+0x04c PatchInformation : Ptr32 Void
```

\_LDR\_DATA\_TABLE\_ENTRY 结构中：

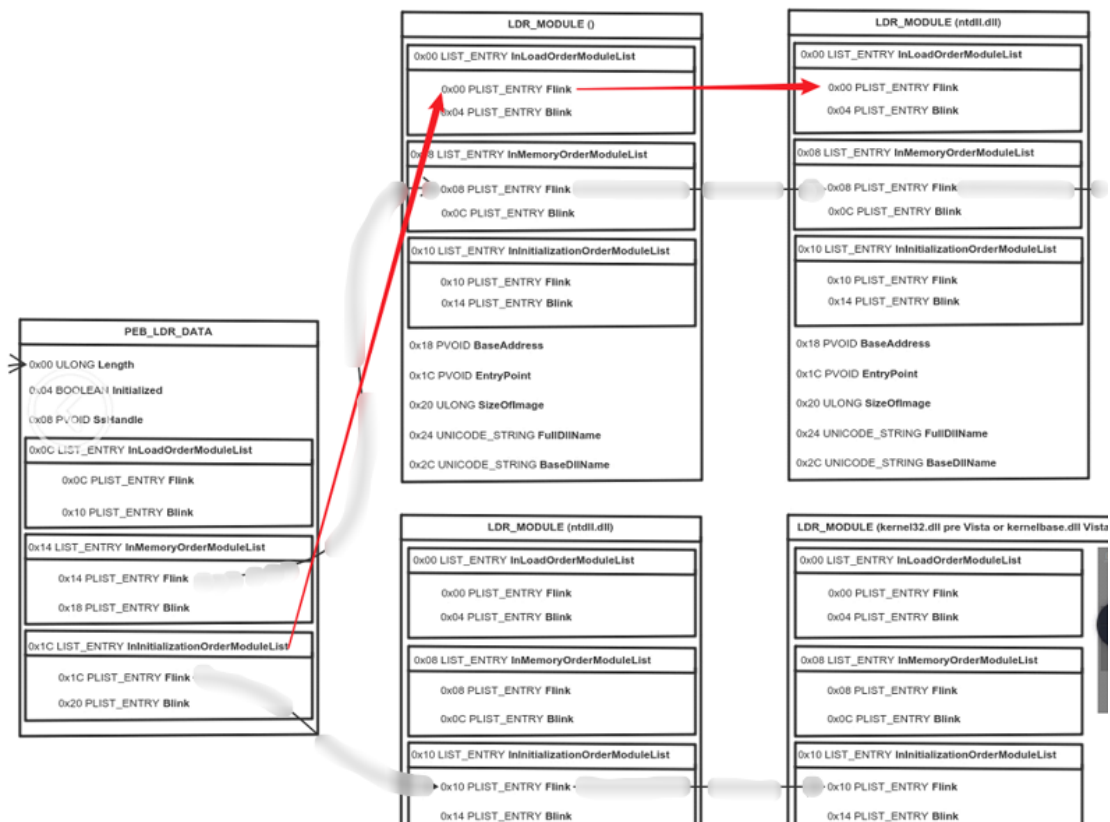
**DllBase**：模块（DLL 或 EXE）的基地址。通常，所有函数和数据的地址都是相对于这个基地址的偏移。

**EntryPoint**：模块的入口点地址，也就是当模块被加载到进程中时，操作系统首先调用的函数地址。在 EXE 文件中，这通常是程序的 main 或 WinMain 函数的地址；对于 DLL 文件，则是 DllMain 函数的地址。

下面是一个实例：按初始化顺序排列的模块链表中第一个链表情况。

[illegible]

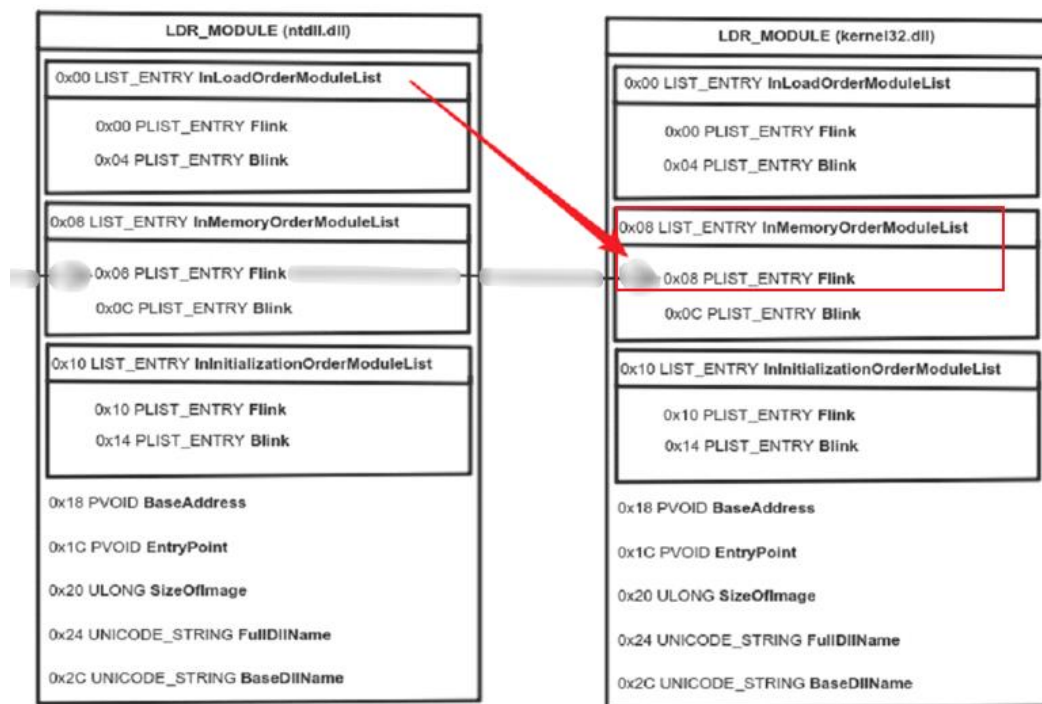
两条指令：MOV EAX,DWORD PTR DS:[EAX], 将 EAX 的当前值（链表头）指向的第一个节点加载到 EAX 中，即链表中的第一个模块（实际就是获取下一条链表的地址存入 EAX 中）。



MOV EAX, DWORD PTR DS:[EAX+08]:

EAX+08 偏移量对应的就是 InMemoryOrderLinks 字段。InMemoryOrderLinks 是一个链表节点，表示该模块在内存中的加载顺序。

如果你遍历这个链表，就可以按照内存中实际加载模块的顺序来获取每一个模块的 LDR DATA TABLE ENTRY。



03.custom.exe - PID: 5060 - 模块: 03.custom.exe - 线程: 主线程 3040 - x32dbg [权限已提升]

文件(F) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选项(O) 帮助(H) Dec 21 2023 (TitanEngine)

CPU 日志 备注 断点 内存映射 调用堆栈 SEH 脚本 符号 源代码 引用 线程

0049D3A0	PUSH EBP	
0049D3A1	MOV EBP, ESP	
0049D3A3	PUSH ECX	
0049D3A4	PUSH ESI	
0049D3A5	MOV DWORD PTR SS:[EBP-4], 0	
0049D3AC	PUSH EAX	
0049D3AD	MOV EAX, DWORD PTR DS:[30]	ecx:EntryPoint
0049D3B3	MOV EAX, DWORD PTR DS:[EAX*4C]	esi:EntryPoint
0049D3B6	MOV EAX, DWORD PTR DS:[EAX*1C]	
0049D3B9	MOV EAX, DWORD PTR DS:[EAX]	
0049D3BB	MOV EAX, DWORD PTR DS:[EAX]	
0049D3BD	MOV EAX, DWORD PTR DS:[EAX*8]	
0049D3C0	MOV DWORD PTR SS:[EBP-4], EAX	
0049D3C3	POP EAX	
0049D3C4	MOV ESI, DWORD PTR SS:[EBP-4]	
0049D3C7	CALL 03.custom.49D2C8	
0049D3CC	PUSH 03.custom.49D4C4	49D4C4: "LoadLibraryA"

将PEB的地址存入EAX  
将PEB地址处+4C偏移处内容(Ldr)存入EAX  
Ldr+1C偏移处是初始化模块排列顺序链表: InInitializationOrderModuleList  
(链表中第一个模块通常是程序本身的可执行文件(EXE)) --- 0x0指下一个链表  
接下来的模块是动态链接库(DLL)链表 --- 0x0指下一个链表  
链表中 InMemoryOrderLinks 字段, 表示模块在内存加载顺序中的链表位置。  
模块内存加载顺序链表位置存入到EBP-4处

那么这样也就知道了这个壳确实是通过查找 TEB 结构来查找所有的导入函数的。

“那么接下来壳要做的就是加密找到的 IAT 中的 API (逐个加密 API 的地址即可), 然后将加密的地址放到壳新申请的内存空间, 而原来 IAT 中的地址就被填充为指向新申请的内存空间的地址, 新的空间中除了加密的 API 地址, 还有解密代码。这样当程序运行到原来 IAT 中的地址时, 就会自动指向壳申请的空间, 通过解密代码把加密的 API 地址还原, 进行调用。”

频繁调用 LoadLibrary 导入 dll 和 GetProcAddress 获取函数模块地址:



03.custom.exe - PID: 376 - 模块: 03.custom.exe - 线程: 主线程 3312 - x32dbg [权限已提升]

文件(F) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选项(O) 帮助(H) Dec 21 2023 (TitanEngine)

CPU 日志 备注 断点 内存映射 调用堆栈 SEH 脚本 符号 源代码 引用

```
0049D3CC . PUSH 03.custom.49D0C4
0049D3D1 . PUSH ESI
0049D3D2 . MOV DWORD PTR DS:[<&GetProcAddress],EAX
0049D3D7 . CALL EAX
0049D3D9 . PUSH 03.custom.49D0D4
0049D3DE . PUSH ESI
0049D3DF . MOV DWORD PTR DS:[<&LoadLibraryA],EAX
0049D3E4 . CALL DWORD PTR DS:[<&GetProcAddress]
0049D3EA . PUSH 03.custom.49D0E8
0049D3EF . PUSH ESI
0049D3F0 . MOV DWORD PTR DS:[<&GetModuleHandleA],EAX
0049D3F5 . CALL DWORD PTR DS:[<&GetProcAddress]
0049D3FB . PUSH 03.custom.49D0F8
0049D400 . MOV DWORD PTR DS:[<&VirtualProtect],EAX
0049D405 . CALL DWORD PTR DS:[<&LoadLibraryA]
0049D40B . PUSH 03.custom.49D104
0049D410 . PUSH EAX
0049D411 . CALL DWORD PTR DS:[<&GetProcAddress]
0049D417 . PUSH 03.custom.49D110
0049D41C . PUSH ESI
0049D41D . MOV DWORD PTR DS:[<&MessageBoxA],EAX
0049D422 . CALL DWORD PTR DS:[<&GetProcAddress]
0049D428 . PUSH 03.custom.49D11C
```

49D0C4: "LoadLibraryA"  
eax: MessageBoxA  
eax: MessageBoxA  
49D0D4: "GetModuleHandleA"  
eax: MessageBoxA  
49D0E8: "VirtualProtect"  
eax: MessageBoxA  
49D0F8: "user32.dll"  
eax: MessageBoxA  
49D104: "MessageBoxA"  
eax: MessageBoxA  
49D110: "ExitProcess"  
eax: MessageBoxA  
49D11C: "VirtualAlloc"

当加壳程序框被用户确认后，才会执行申请空间和加密的逻辑，下面的 jmp 执行了一次大跳转，跳转到真正的 OEP，因此 jmp 前的 call 有加密 API 地址的最大嫌疑。

03.custom.exe - PID: 376 - 模块: 03.custom.exe - 线程: 主线程 3312 - x32dbg [权限已提升]

文件(F) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选项(O) 帮助(H) Dec 21 2023 (TitanEngine)

CPU 日志 备注 断点 内存映射 调用堆栈 SEH 脚本 符号 源代码 引用

```
0049D493 . PUSH EDI
0049D494 . MOV EAX,DWORD PTR DS:[49D04C]
0049D499 . PUSH EAX
0049D49A . MOV ECX,DWORD PTR SS:[EBP-8]
0049D49D . PUSH ECX
0049D49E . CALL DWORD PTR DS:[<&VirtualProtect]
0049D4A4 . PUSH 4
0049D4A6 . PUSH 03.custom.49D12C
0049D4AB . PUSH 03.custom.49D138
0049D4B0 . PUSH 0
0049D4B2 . CALL DWORD PTR DS:[<&MessageBoxA]
0049D4B8 . MOV DWORD PTR SS:[EBP-C],EAX
0049D4BB . CMP DWORD PTR SS:[EBP-C],6
0049D4BF . JNE 03.custom.49D4CC
0049D4C1 . CALL 03.custom.49D4E0
0049D4C6 . JMP DWORD PTR DS:[49D093]
0049D4CC . PUSH 0
0049D4CE . CALL DWORD PTR DS:[<&ExitProcess]
0049D4D4 . MOV ESP,EBP
0049D4D6 . POP EBP
0049D4D7 . RET
0049D4D8 . INT3
0049D4D9 . INT3
```

49D12C: "Hello "  
49D138: "欢迎使用免费加壳程序，是否运行主程序?"

03.custom.exe - PID: 1156 - 模块: 03.custom.exe - 线程: 主线程 5848 - x32dbg [权限已提升]

文件(F) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选项(O) 帮助(H) Dec 21 2023 (TitanEngine)

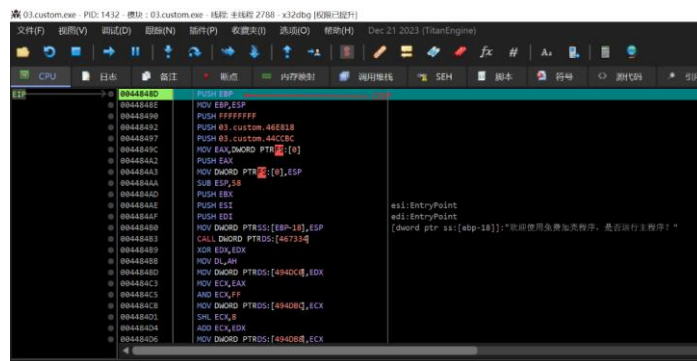
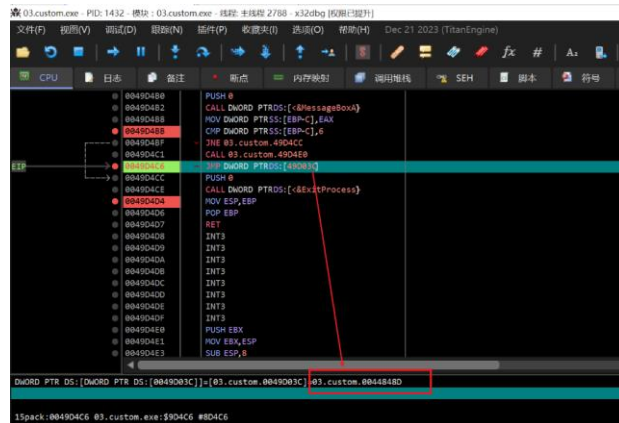
CPU 日志 备注 断点 内存映射 调用堆栈 SEH 脚本 符号 源代码 引用

```
0049D4B0 . PUSH 0
0049D4B2 . CALL DWORD PTR DS:[<&MessageBoxA]
0049D4B8 . MOV DWORD PTR SS:[EBP-C],EAX
0049D4BB . CMP DWORD PTR SS:[EBP-C],6
0049D4BF . JNE 03.custom.49D4CC
0049D4C1 . CALL 03.custom.49D4E0
0049D4C6 . JMP DWORD PTR DS:[49D093]
0049D4CC . PUSH 0
0049D4CE . CALL DWORD PTR DS:[<&ExitProcess]
0049D4D4 . MOV ESP,EBP
0049D4D6 . POP EBP
0049D4D7 . RET
0049D4D8 . INT3
0049D4D9 . INT3
```

大跳

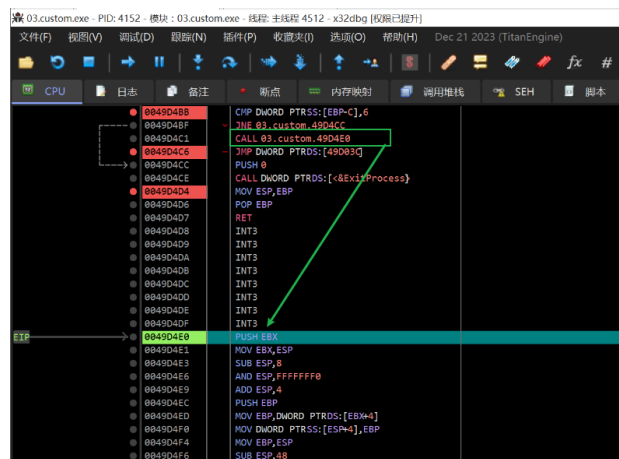
跳到 OEP:



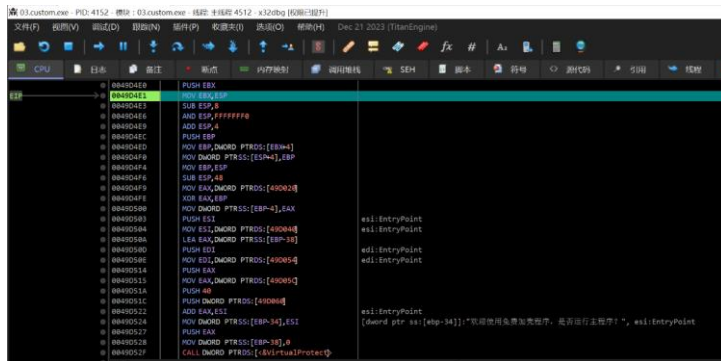


下面详细分析壳找 API 和加密 IAT 和存放加密后 IAT 的地方：分析  
Call

进 call: **0049D4C1** **CALL 03.custom.49D4E0**

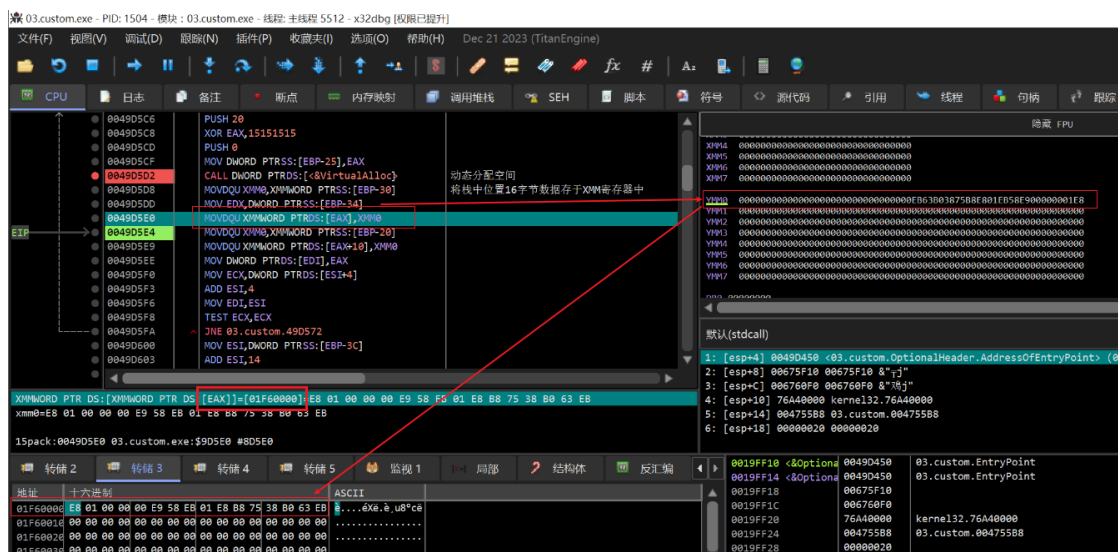
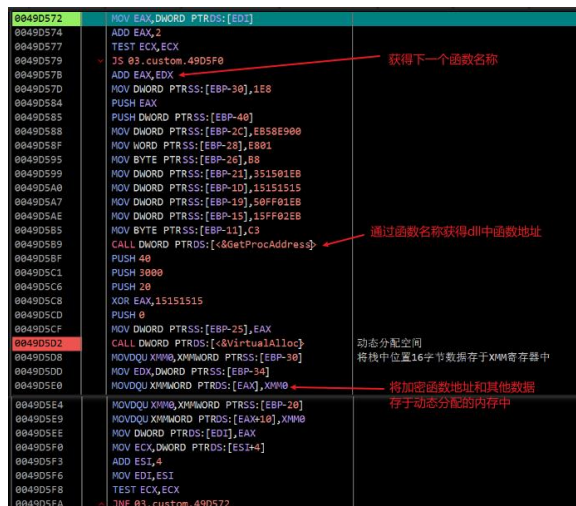


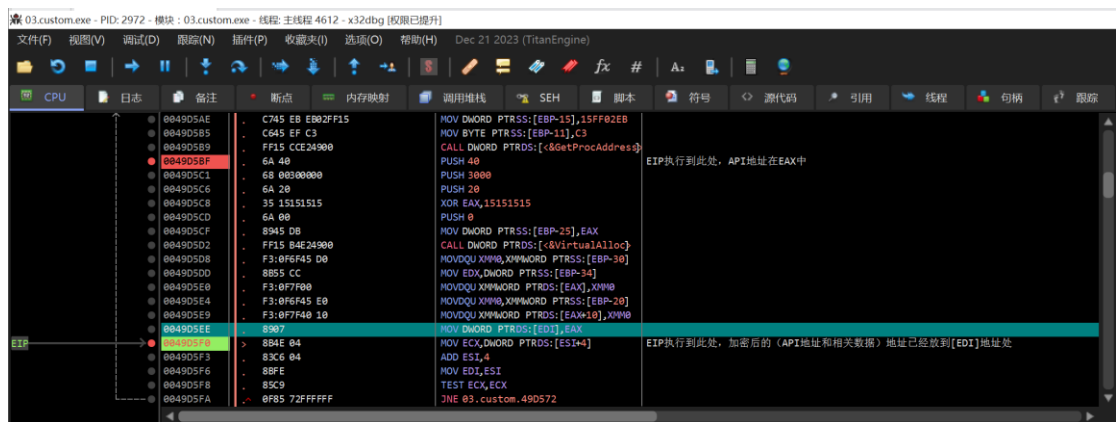
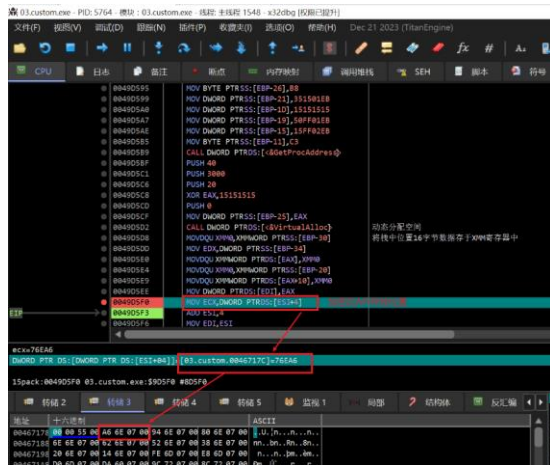
仔细跟踪指令：



## 找 API、加密 API、存到另外内存

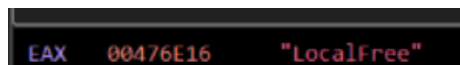
加密数据将 API 地址加密后存到另外的地方。XMM 寄存器中间过渡存储加密函数和一些其他数据。





下面是 API 加密详细跟进过程:跟进的是 Kernel32.dll 中的 LocalFree 函数。

ADD EAX,EDX----获取下一个函数名称



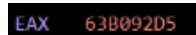
地址	十六进制	ASCII
00476E16	4C 6F 63 61 6C 46 72 65 65 00 8A 00 46 69 6C 65	LocalFree..File
00476E26	54 69 6D 65 54 6F 53 79 73 74 65 6D 54 69 6D 65	TimeToSystemTime
00476E36	00 00 89 00 46 69 6C 65 54 69 6D 65 54 6F 4C 6F	....FileTimeToLo
00476E46	63 61 6C 46 69 6C 65 54 69 6D 65 00 71 02 53 65	calFileTime.q.Se
00476E56	74 4C 61 72 74 45 72 73 65 73 00 00 05 03 6C 73	blatFileTime..le

CALL DWORD PTR DS:[<&GetProcAddress>]----根据函数名称获取函数在 dll 中的地址



76A587C0	8B55FF8B	.yU.
76A587C4	25FF5DEC	ijy%
76A587C8	76AC0548	H.-v
76A587CC	CCCCCCCC	iiii
76A587D0	CCCCCCCC	iiii

XOR EAX,15151515----如: 异或加密 kernel32.dll 中函数地址数据。

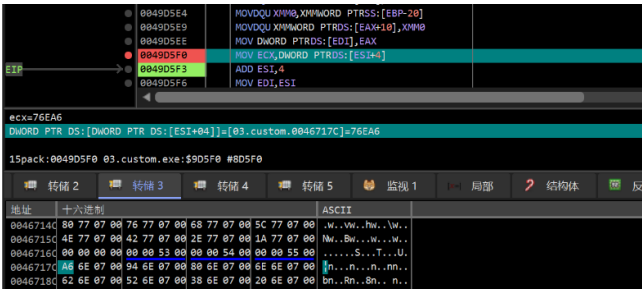
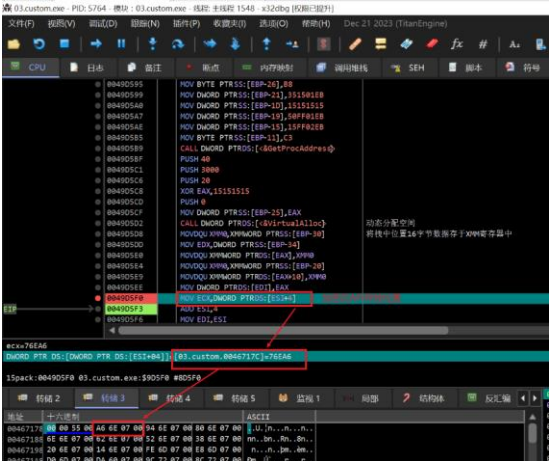
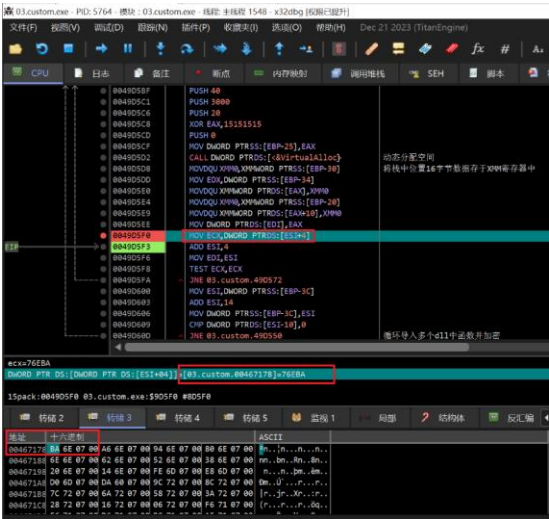


CALL DWORD PTR DS:[<&VirtualAlloc>]----动态分配一个内存地址, 用于存放加密后的

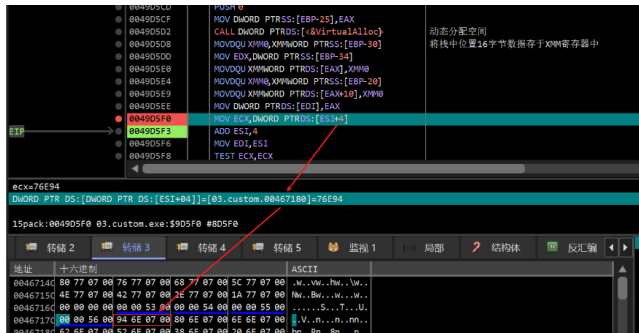
内存地址和一些其他数据。

EAX 01F50000	
地址	十六进制
01F50000	E8 01 00 00 00 E9 58 EB 01 E8 B8 D5 92 B0 63 EB
01F50001	01 15 35 15 15 15 15 EB 01 FF 50 EB 02 FF 15 C3
01F50002	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
	ASCII
	.....è.ë.ö.°cè
	..S....ÿPä.ÿ.Ä
	.....

加密数据存放位置：

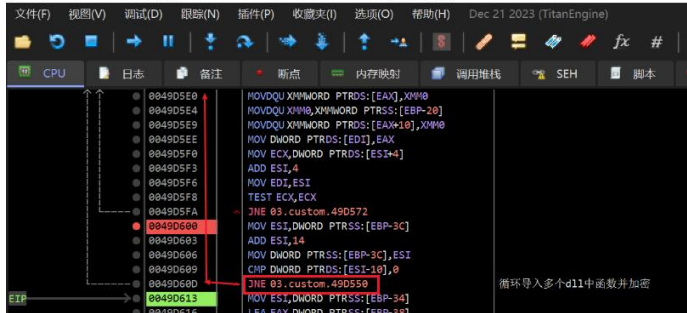


之前加密的都变成序号了：



上面是单个函数加密存储的步骤，下面是循环导入多个 dll 中函数并加密的结尾部分：

03.custom.exe - PID: 1504 - 模块: 03.custom.exe - 线程: 主线程 5512 - x32dbg (权限已提升)



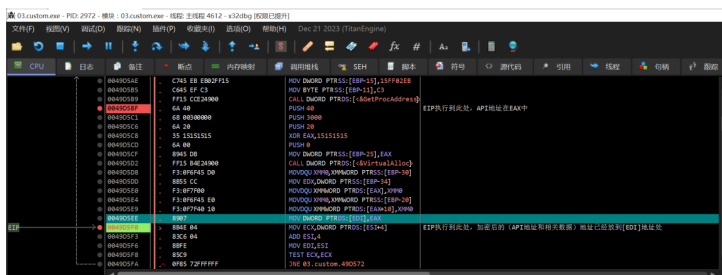
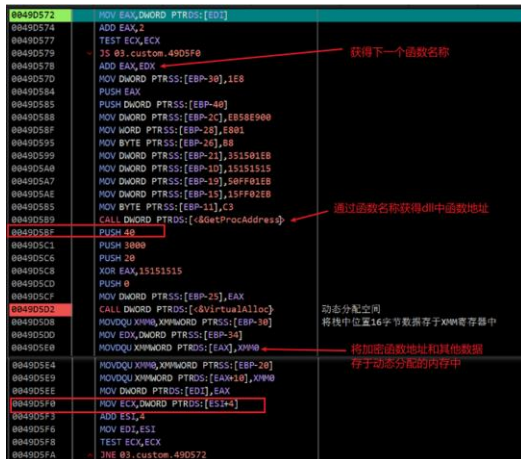
怎样解密壳加密后的 API 地址？那要怎样修复 IAT 表呢？ ---- 直接利用脚本将加密前的 API 地址存到加密后要存放的内存位置。

到现在一共三个地址：

OEP: 0044848D

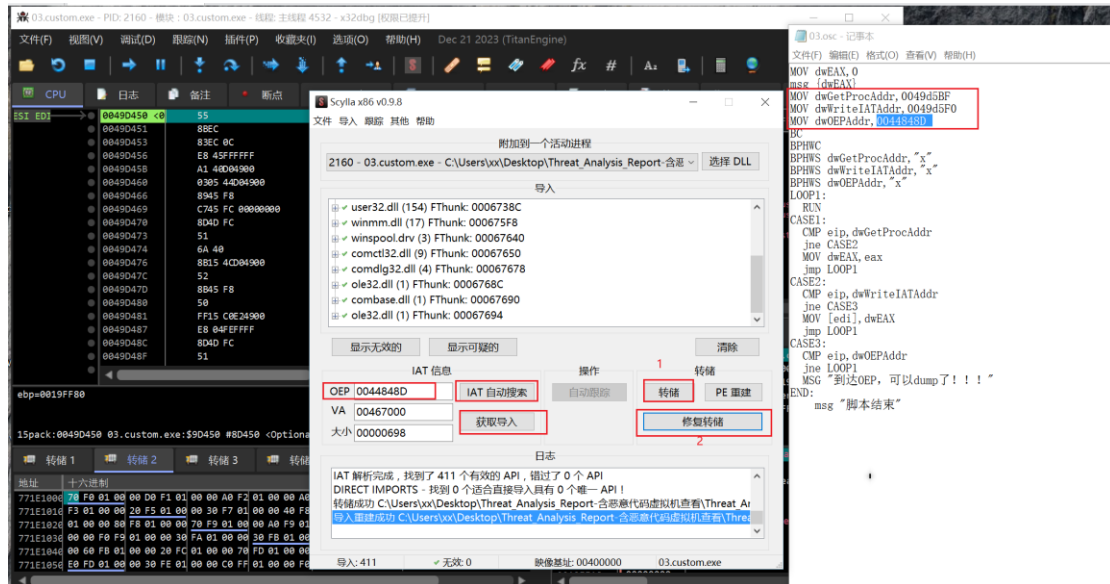
0049D5BF: EIP 执行到这里时，获取的 API 地址在 EAX 中

0049D5F0: 当 EIP 执行到这个地址的时候，[EDI]该地址中保存的是加密后的 API 的地址



## 利用脚本脱壳：(脚本修复 IAT 表)+Scylla 插件 dump 内存

需要点击确认：(脚本中 3 个地址千万不要写错了!!!!)、OEP 填对 (自动识别 IAT)、转储文件、再修复转储的文件。





```
MOV dwEAX,0
msg {dwEAX}

MOV dwGetProcAddr,0049d5BF    //EIP 执行到此处时， 获取的未加密的 API 地址在 EAX 中
MOV dwWriteIATAddr,0049d5F0    //加密后 API 地址已存放到动态分配地址
MOV dwOEPAAddr,0044848D        //OEP 地址

BC

BPHWC

BPHWS dwGetProcAddr,"x"
BPHWS dwWriteIATAddr,"x"
BPHWS dwOEPAAddr,"x"

LOOP1:
    RUN
CASE1:
    CMP eip,dwGetProcAddr
    jne CASE2
    MOV dwEAX,eax
    jmp LOOP1
CASE2:
    CMP eip,dwWriteIATAddr
    jne CASE3
    MOV [edi],dwEAX            //更新 IAT 地址表
    jmp LOOP1
CASE3:
    CMP eip,dwOEPAAddr
    jne LOOP1
    MSG "到达 OEP， 可以 dump 了!!! "
END:
    msg "脚本结束"
```