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。

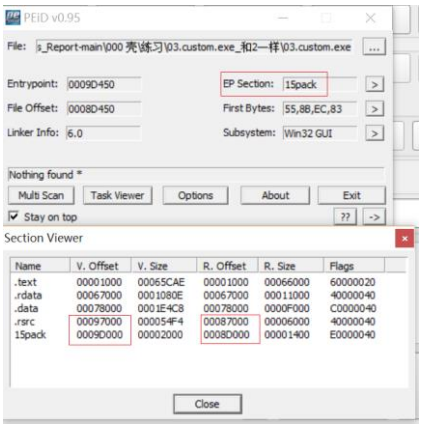
IAT 表地址看程序动态分配的地址, 此程序中加密的 API 地址存放位置在 EDI 寄存器处。

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)。

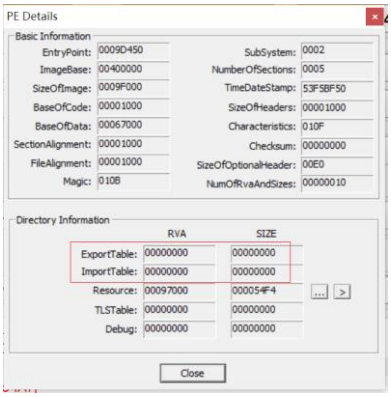
四、脱壳示例：

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：



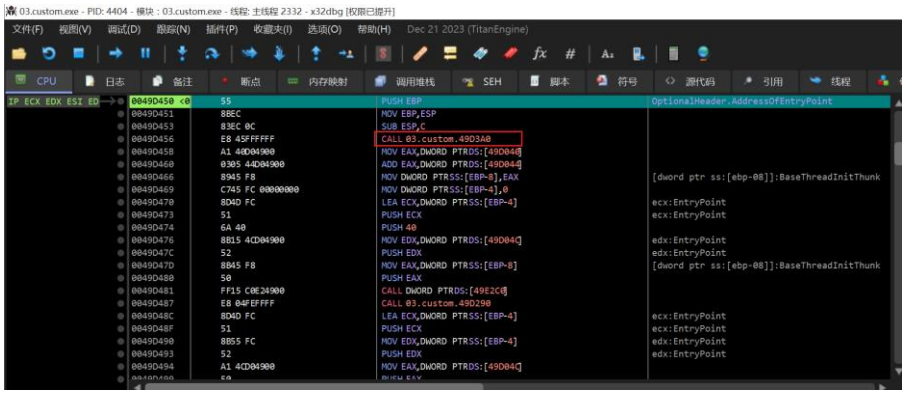
没有导入导出表：那程序执行时是如何调用 API 的？



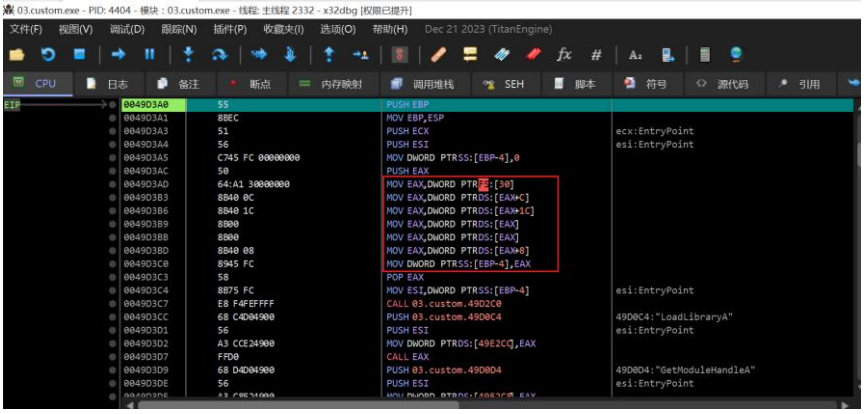
其他工具检测：

2、x32dbg 调试：

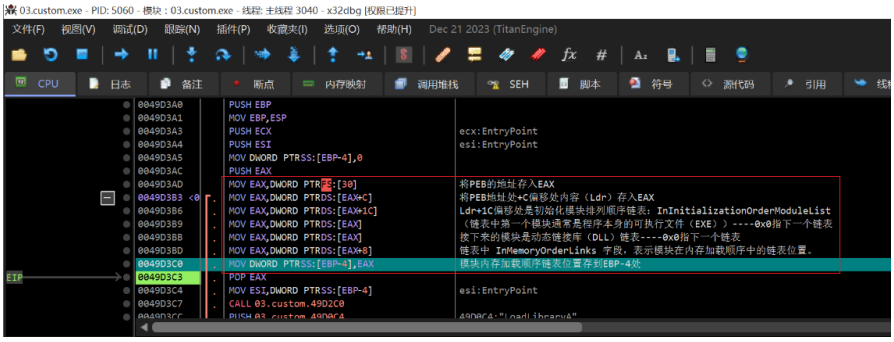
运行程序跳到 0x49D450 地址并不像 OEP：



单步运行进 call 0x49D3A0:



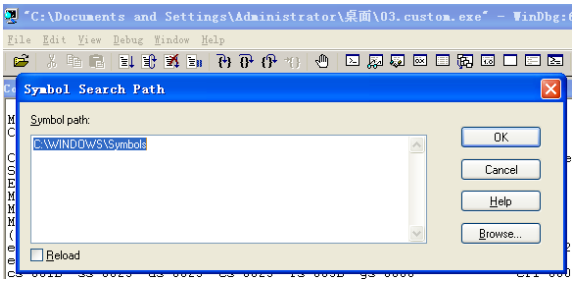
那程序执行时是如何调用 API 的？TEB->PEB->Ldr→API 地址。



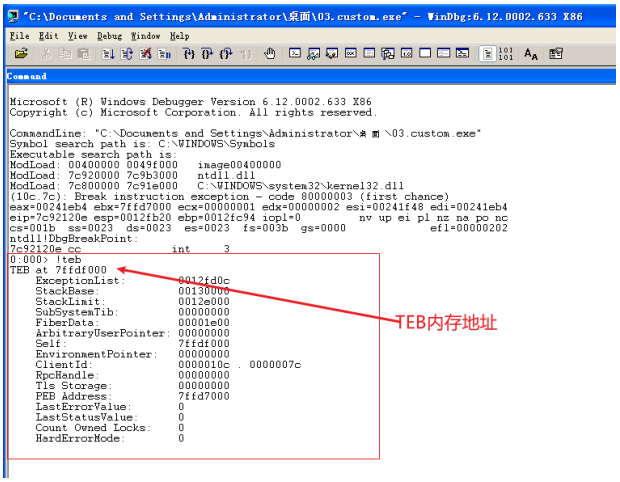
下面详细解释如何用 WinDbg 查看相关 TEB 和 PEB 的数据结构!!!!

Windbgxp 中查看 fs:[30]处数据结构:

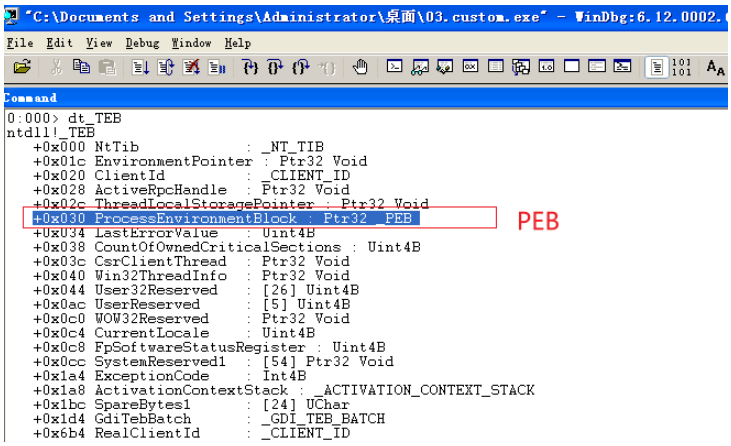
添加符号文件：



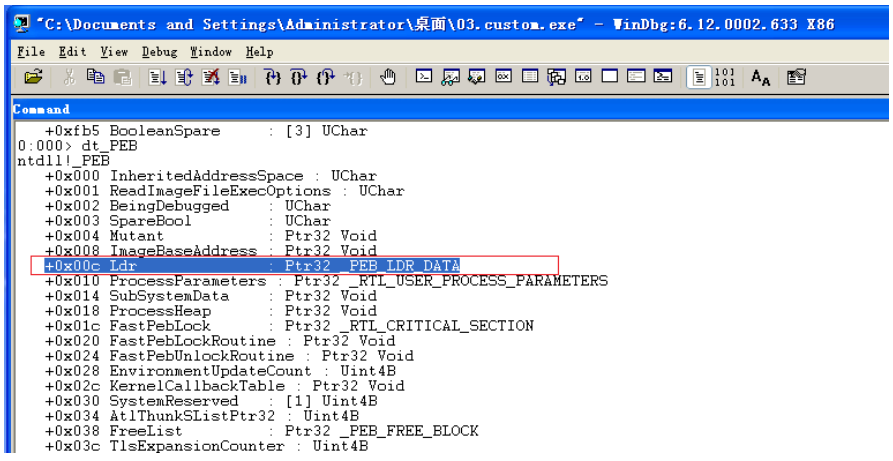
查看 TEB 地址：0x7ffdf000，即 FS 寄存器中的值为 7ffdf000



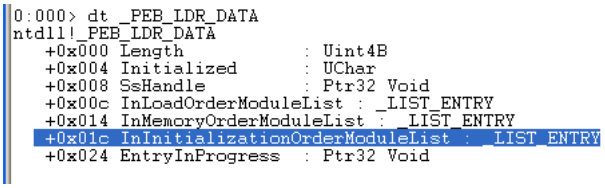
输入 dt _TEB 可以查看 TEB 的具体结构，可以看到 0x30 处指向了 PEB 结构



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：



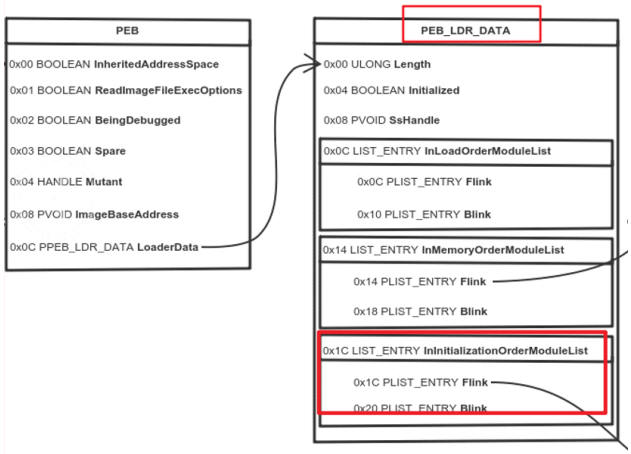
那么用 dt _PEB_LDR_DATA 命令查看_PEB_LDR_DATA 结构体的结构，偏移 1C 指向哪里呢？可以看到指向了 InInitializationOrderModuleList (// 按初始化顺序排列的模块链表)，这个成员是个_LIST_ENTRY 结构体：



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。

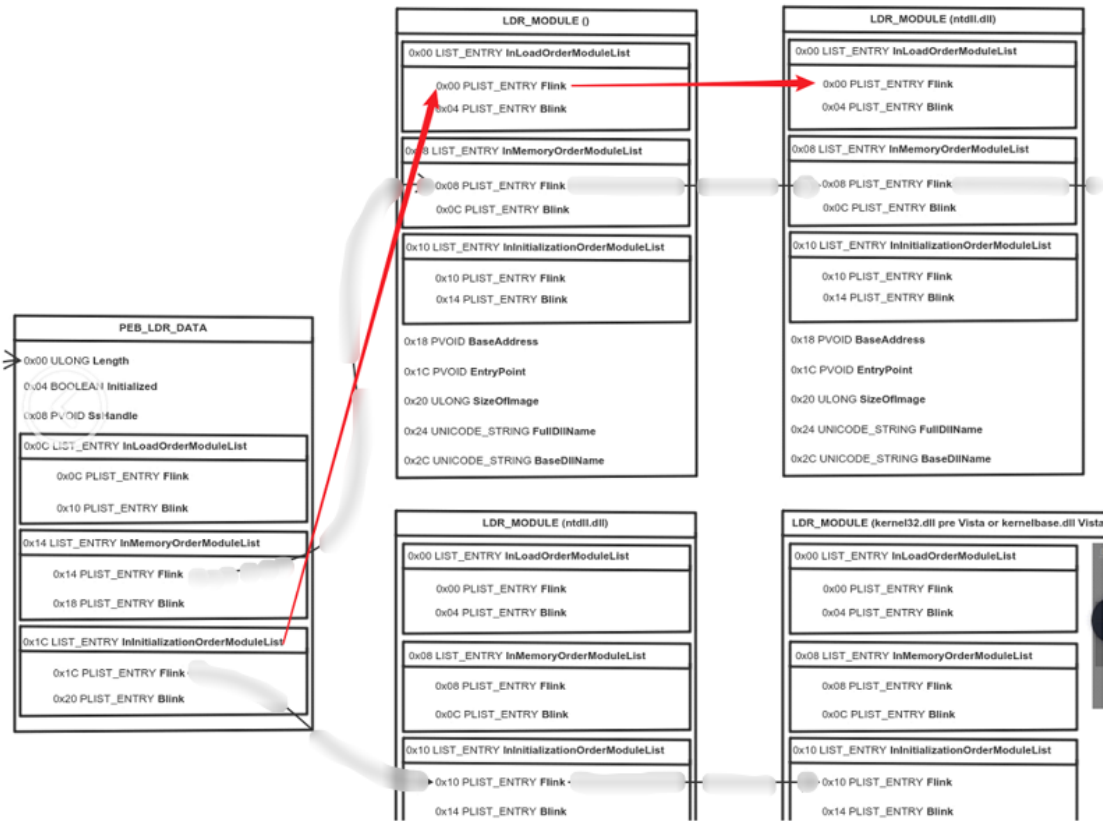
```
0:000> !peb
PEB at 7ffde000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
    Ldr                       00241ea0
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00241f58 . 00242020
    Ldr.InLoadOrderModuleList:           00241ee0 . 00242010
    Ldr.InMemoryOrderModuleList:         00241ee8 . 00242018
```

PEB中第一个初始化链表地址

```
0:000>  dt _LDR_DATA_TABLE_ENTRY 00241f58
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x242020 - 0x241ebc ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x7c920000 - 0x7c932c28 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x93000 - 0x208003a ]
   +0x018 DllBase          : 0x7c99d028 Void
   +0x01c EntryPoint       : 0x00140012 Void
   +0x020 SizeOfImage      : 0x7c942158
   +0x024 FullDllName      : _UNICODE_STRING "--- memory read error at address 0x0000ffff ---"
   +0x02c BaseDllName      : _UNICODE_STRING "[illegible]
   +0x034 Flags            : 0x4802bdc5
   +0x038 LoadCount        : 0
   +0x03a TlsIndex         : 0
   +0x03c HashLinks        : _LIST_ENTRY [ 0x0 - 0xabababab ]
   +0x03c SectionPointer   : (null)
   +0x040 CheckSum         : 0xabababab
   +0x044 TimeDateStamp    : 0xabababab
   +0x044 LoadedImports    : 0xabababab Void
   +0x048 EntryPointActivationContext : (null)
   +0x04c PatchInformation : (null)
```

第一个初始化顺序的链表

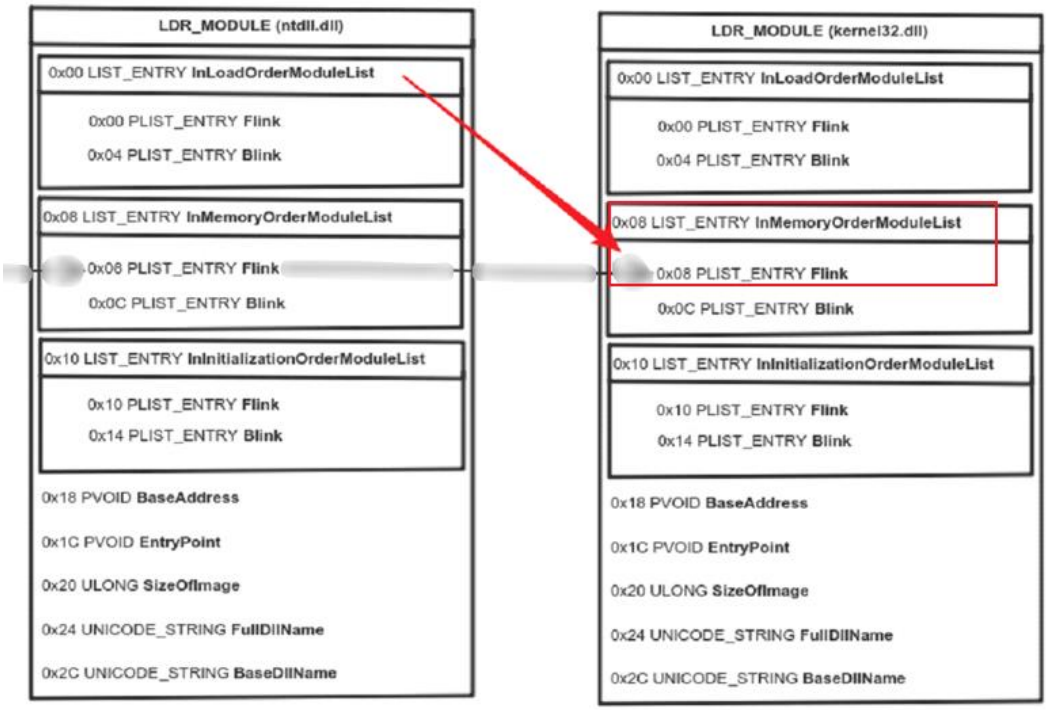
两条指令：MOV EAX,DWORD PTR DS:[EAX]，将 EAX 的当前值（链表头）指向的第一个节点加载到 EAX 中，即链表中的第一个模块（实际就是获取下一条链表的地址存入EAX 中）。

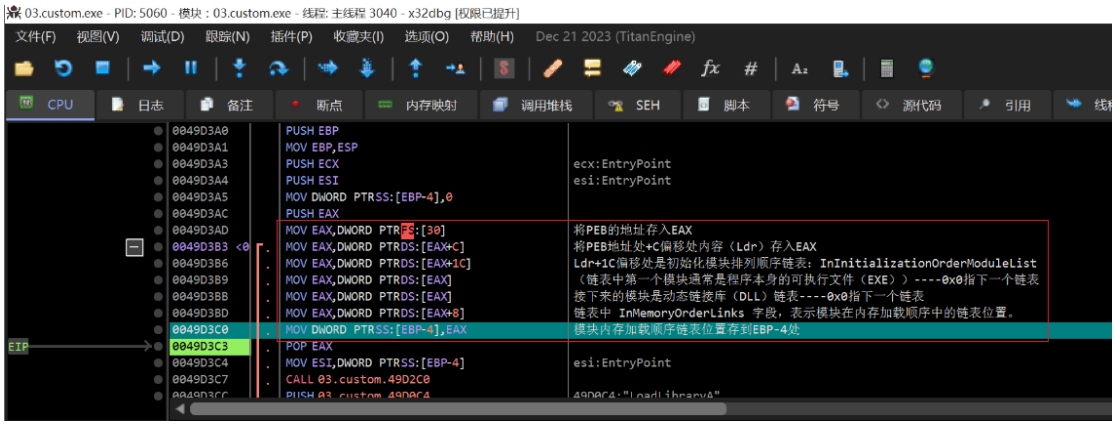


MOV EAX, DWORD PTR DS:[EAX+08]：

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。

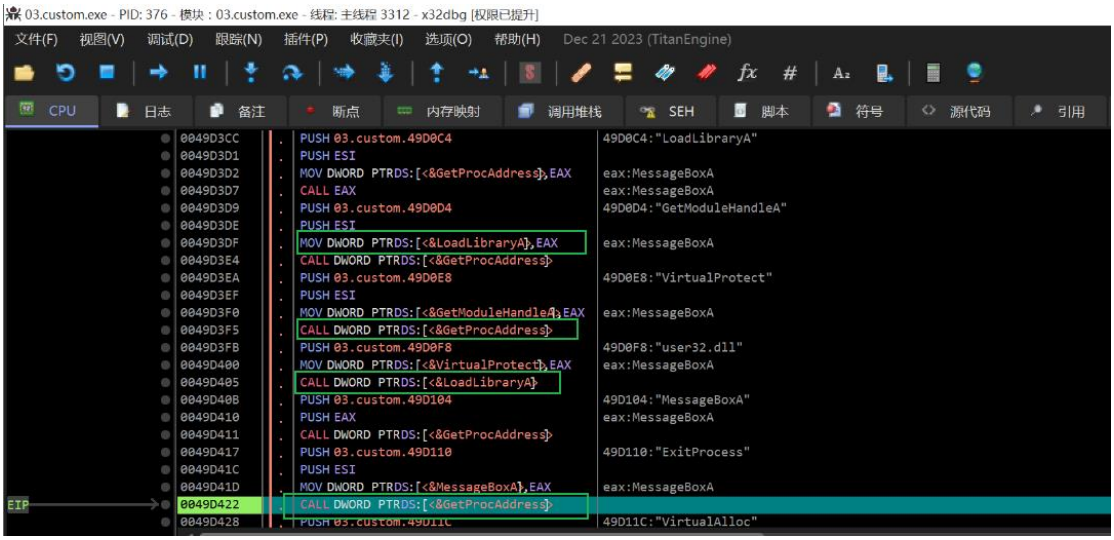




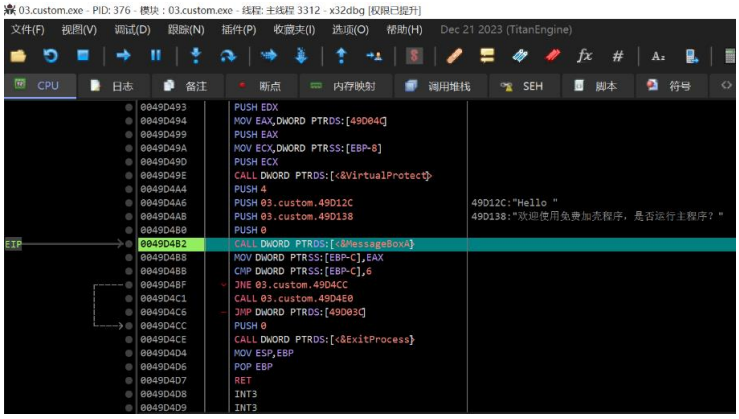
那么这样也就知道了这个壳确实是通过查找 TEB 结构来查找所有的导入函数的。

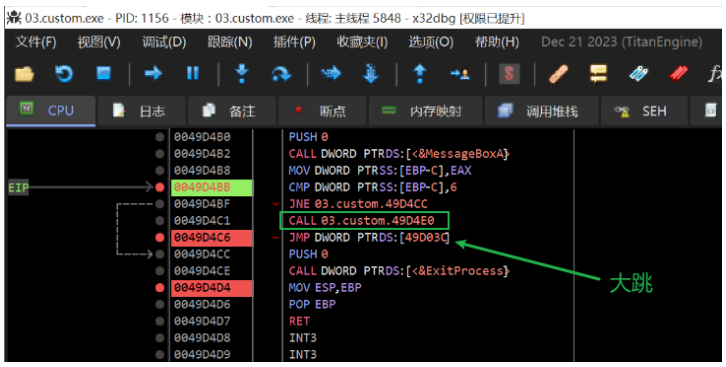
“那么接下来壳要做的就是加密找到的 IAT 中的 API（逐个加密 API 的地址即可），然后将加密的地址放到壳新申请的内存空间，而原来 IAT 中的地址就被填充为指向新申请的内存空间的地址，新的空间中除了加密的 API 地址，还有解密代码。这样当程序运行到原来 IAT 中的地址时，就会自动指向壳申请的空间，通过解密代码把加密的 API 地址还原，进行调用。”

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

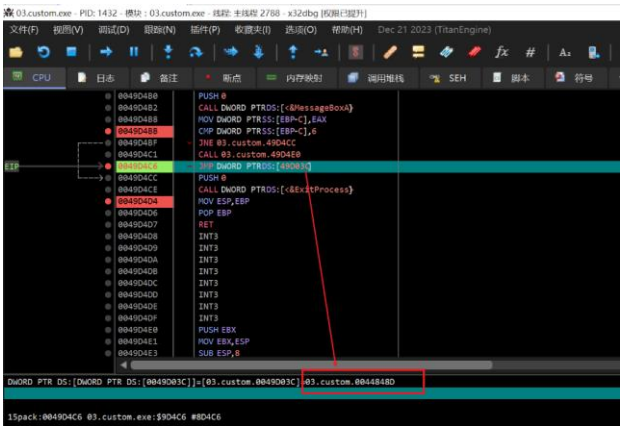


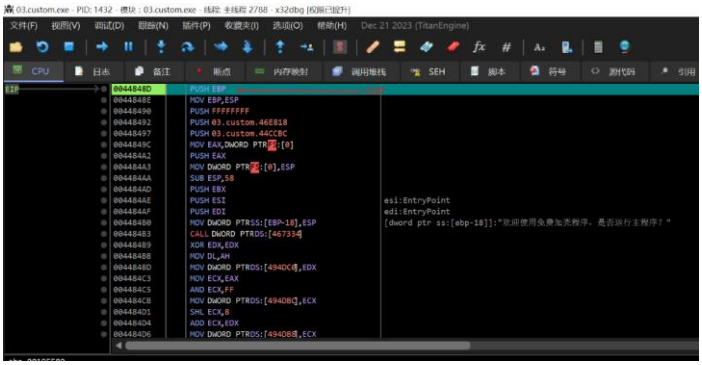
当加壳程序框被用户确认后，才会执行申请空间和加密的逻辑，下面的 jmp 执行了一次大跳转，跳转到真正的 OEP，因此 jmp 前的 call 有加密 API 地址的最大嫌疑。





跳到 OEP：

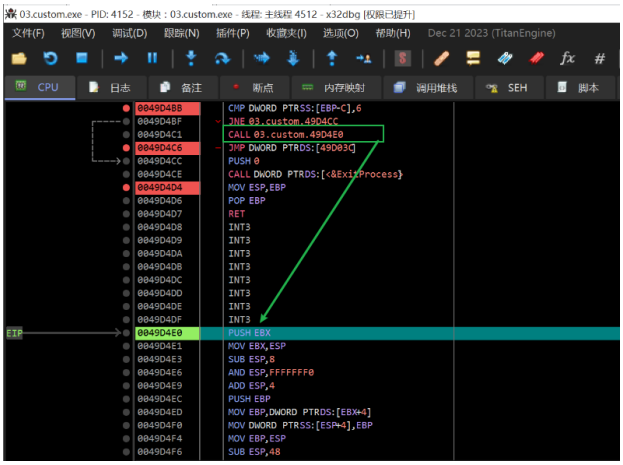




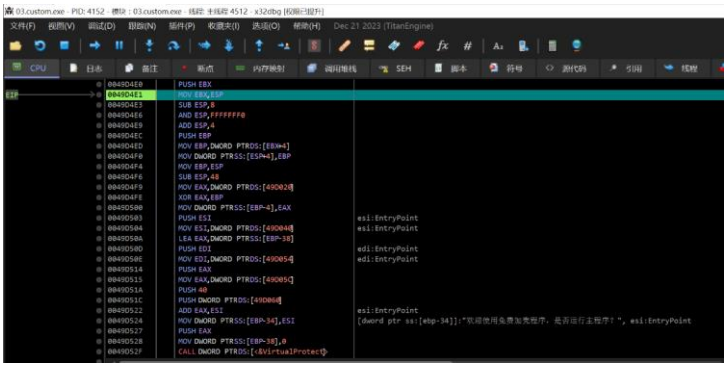
下面详细分析壳找 API 和加密 IAT 和存放加密后 IAT 的地方：分析 Call

进 call:

```
0049D4C1    CALL 03.custom.49D4E0
```

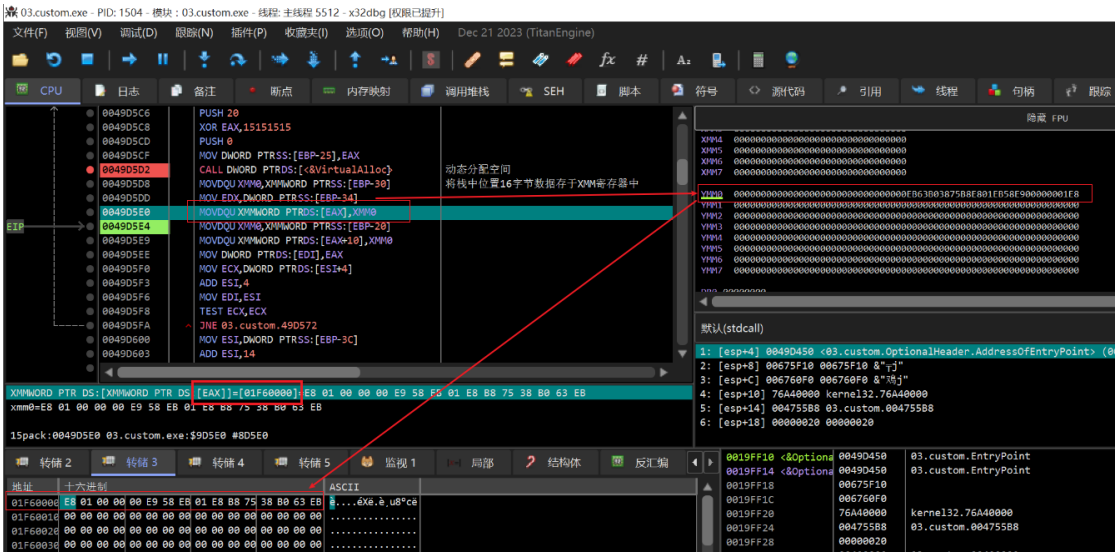


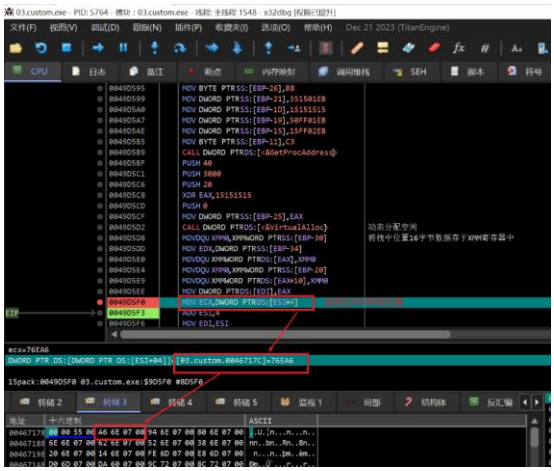
仔细跟踪指令:

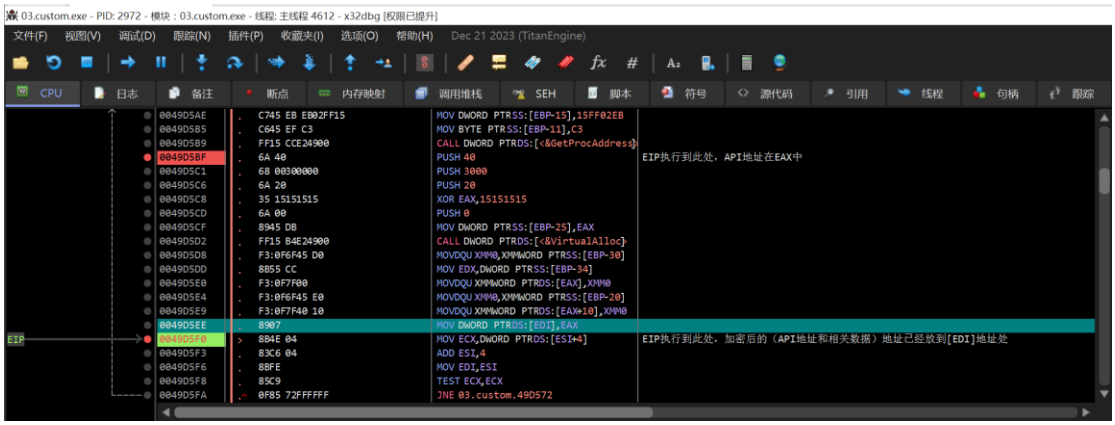


找 API、加密 API、存到另外内存

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。

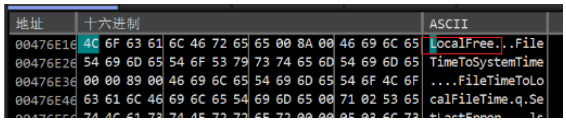




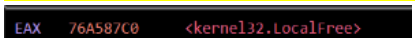


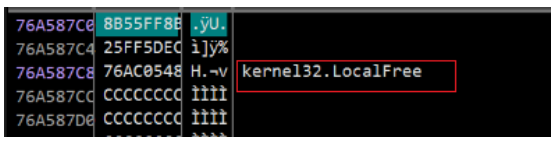
下面是 API 加密详细跟进过程：跟进的是 Kernel32.dll 中的 LocalFree 函数。

ADD EAX,EDX----获取下一个函数名称

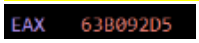


CALL DWORD PTR DS:[<&GetProcAddress>]----根据函数名称获取函数在 dll 中的地址



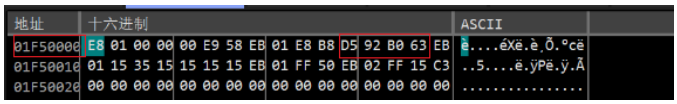


XOR EAX,15151515----如：异或加密 kernel32.dll 中函数地址数据。

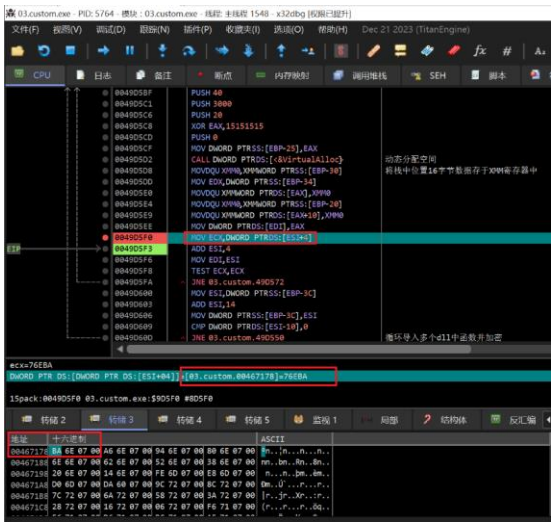


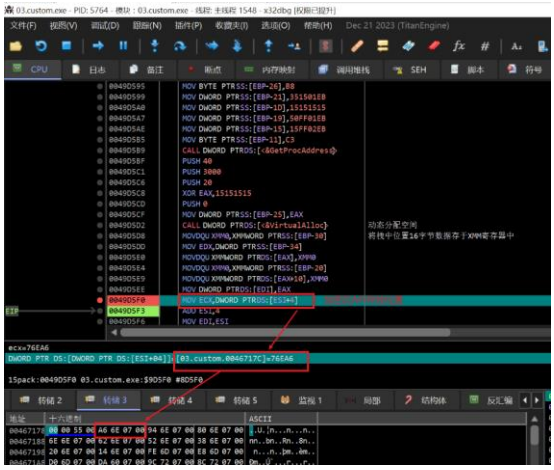
CALL DWORD PTR DS:[<&VirtualAlloc>]----动态分配一个内存地址，用于存放加密后的

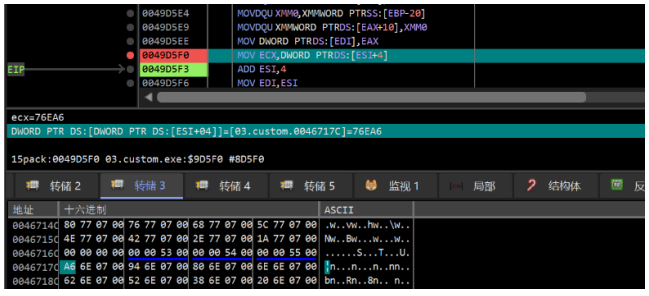
内存地址和一些其他数据。



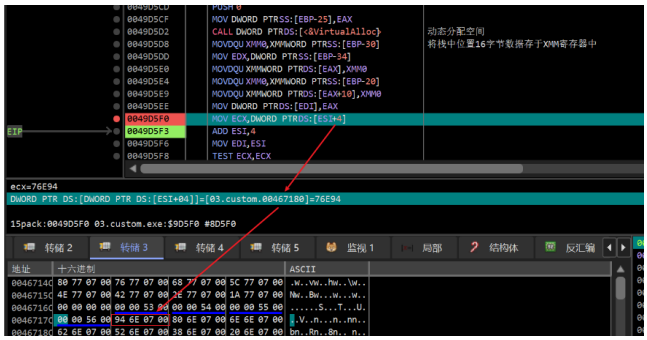
加密数据存放位置：



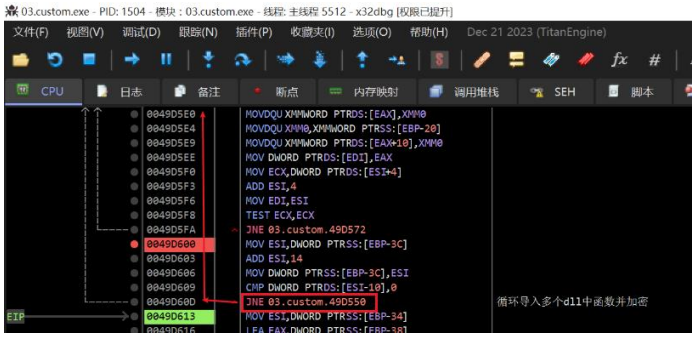




之前加密的都变成序号了：



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：



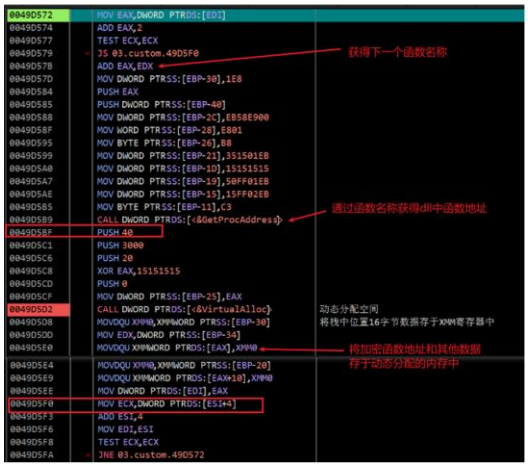
怎样解密壳加密后的 API 地址？那要怎样修复 IAT 表呢？ ----直接利用脚本将加密前的 API 地址存到加密后要存放的内存位置。

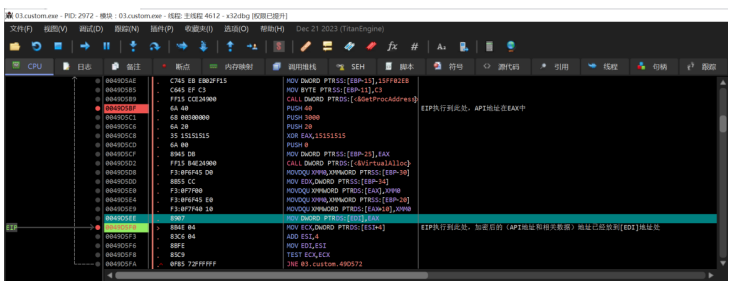
到现在一共三个地址：

OEP：0044848D

0049D5BF：EIP 执行到这里时，获取的 API 地址在 EAX 中

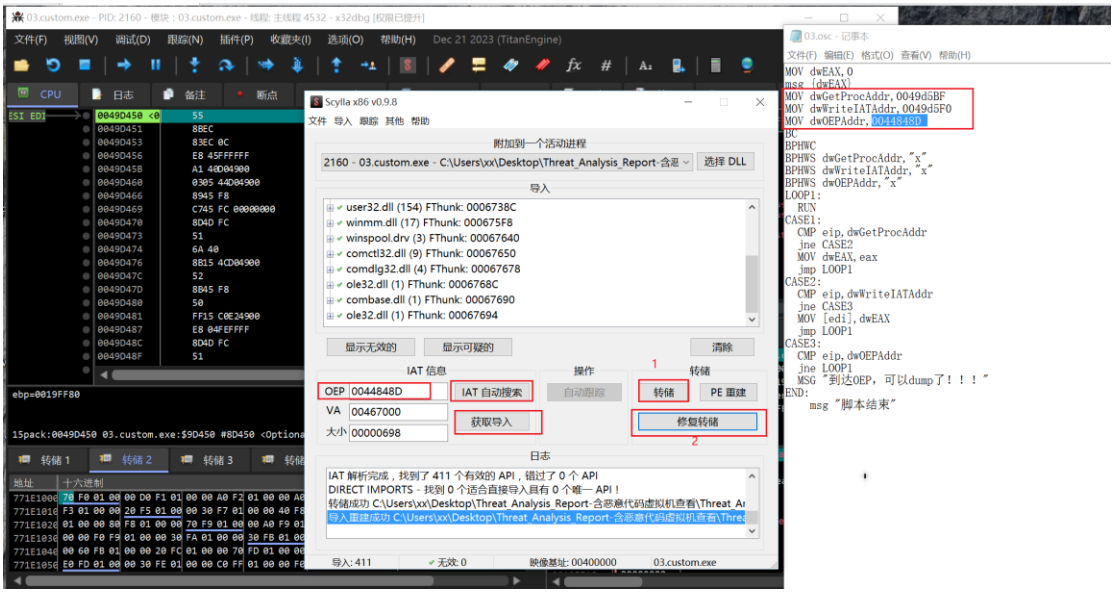
0049D5F0：当 EIP 执行到这个地址的时候，[EDI]该地址中保存的是加密后的 API 的地址





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 dwOEPAddr,0044848D       //OEP 地址
BC
BPHWC
BPHWS dwGetProcAddr,"x"
BPHWS dwWriteIATAddr,"x"
BPHWS dwOEPAddr,"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,dwOEPAddr
   jne LOOP1
   MSG "到达 OEP，可以 dump 了!!!  "
END:
      msg "脚本结束"
```