**2021/04/01_壳_第4课_CrackMe.exe样本脱壳**

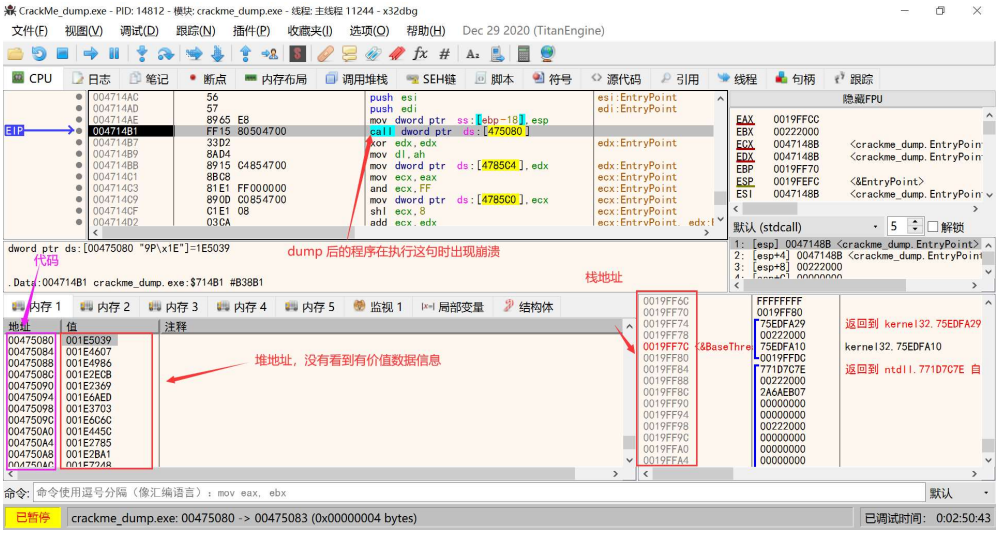
**笔记本：** 壳
**创建时间：** 2021/4/1 星期四 10:25
**作者：** ileemi

---


- 分析步骤
- 壳的对抗

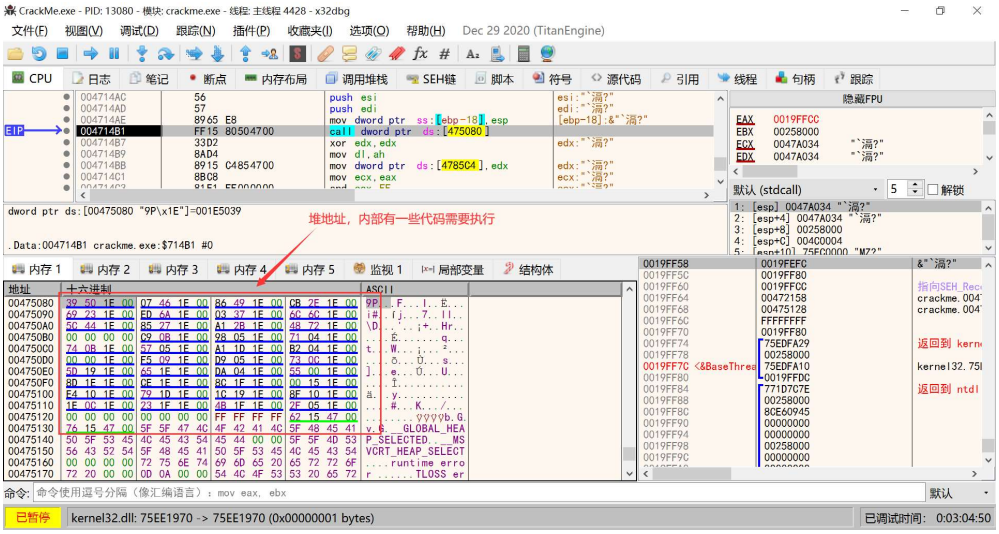

# 分析步骤

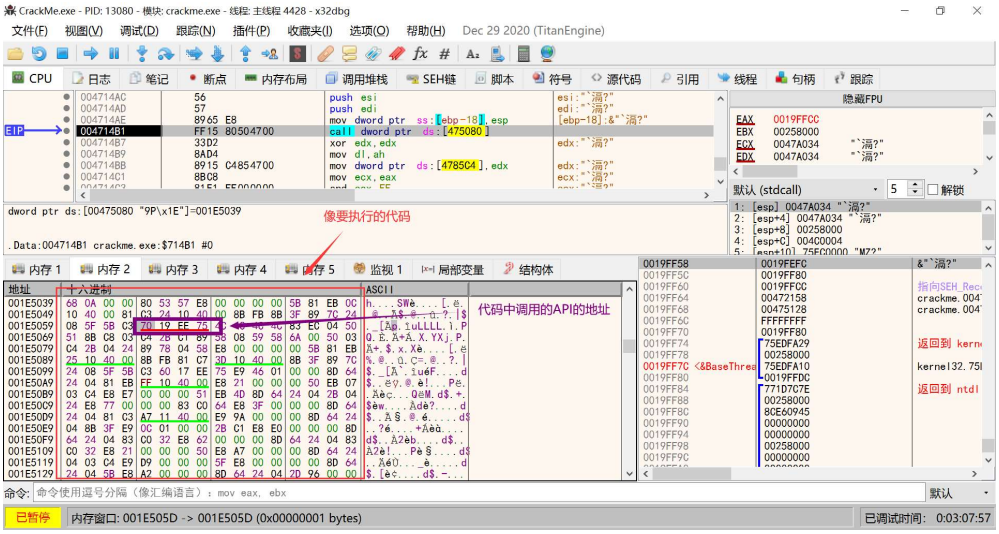
1. 定位OEP:
   0047148B | 55 | push ebp | -- OEP
   找到OEP在OEP处设一个硬件执行断点
2. 之后就可以对当前程序进行内存dump，之后，检查dump后的文件是否可以正常运行。不能正常运行在观察原程序的IAT表等。
3. 测试dump后的程序不能正常运行，使用调试器进行调试，定位崩溃的位置：
   004714B1 | FF15 80504700 | call dword ptr ds:[475080] -- 该指令是一个调用API的指令模式，475080 -- 正常情况下应该是一个导出函数的地址。

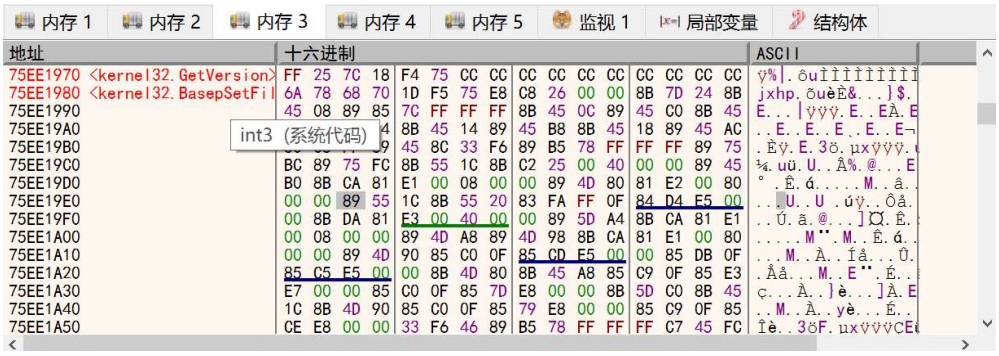



回到原程序中分析出现崩溃问题的位置上继续分析。

4. 分析源程序的 "475080" 地址：



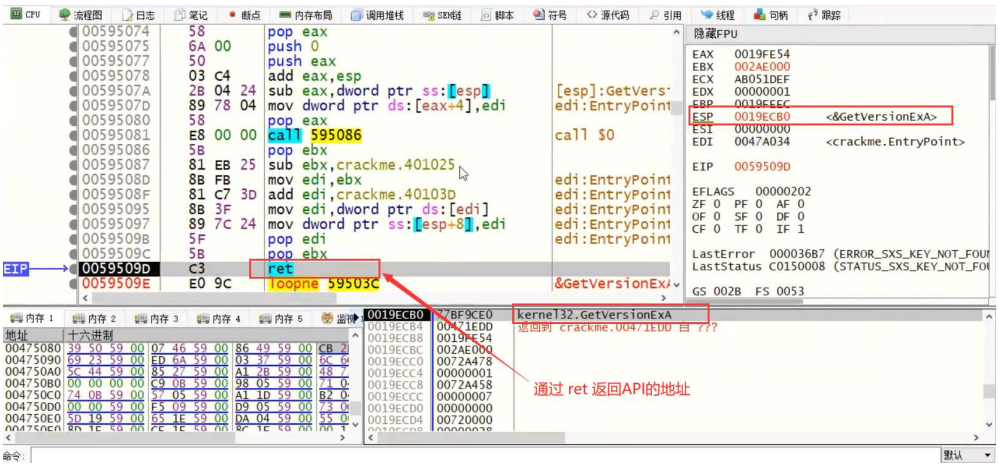



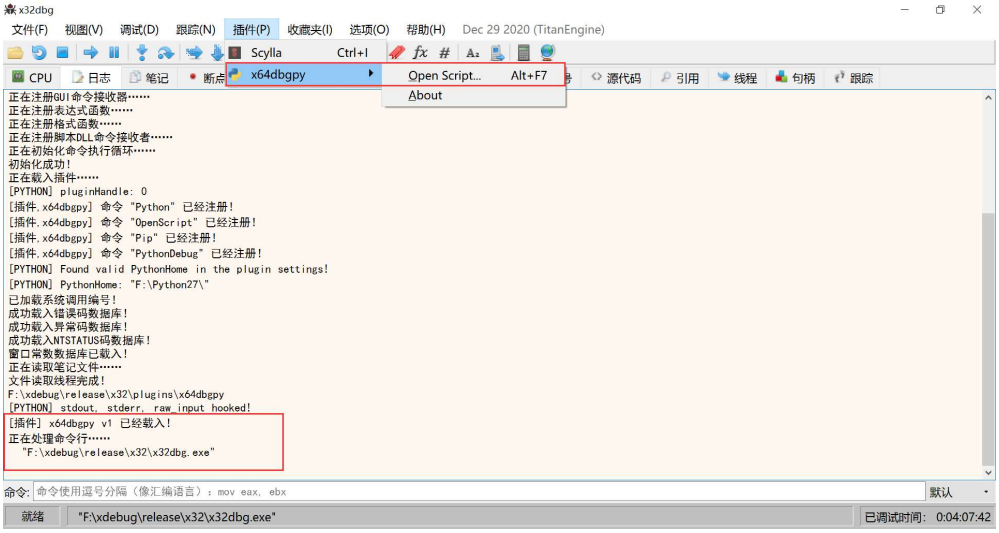

经过上面的分析，可以看出该程序的IAT表已经被加密或者混淆了。

5. 对访问导出函数加密处理的地方进行细致的分析，加密处理的代码在最后也会一定跳转到真正的导出函数的地址上，需要多分析几处，找到其实现的规律，最后针对其规律找到破解办法。
6. 调试分析时对于不确定功能的函数调用，尽量使用 "F7" 跟进到函数内部分析。经过分析可以看出混淆代码中会获取导出函数的地址，导出函数的地址放到栈

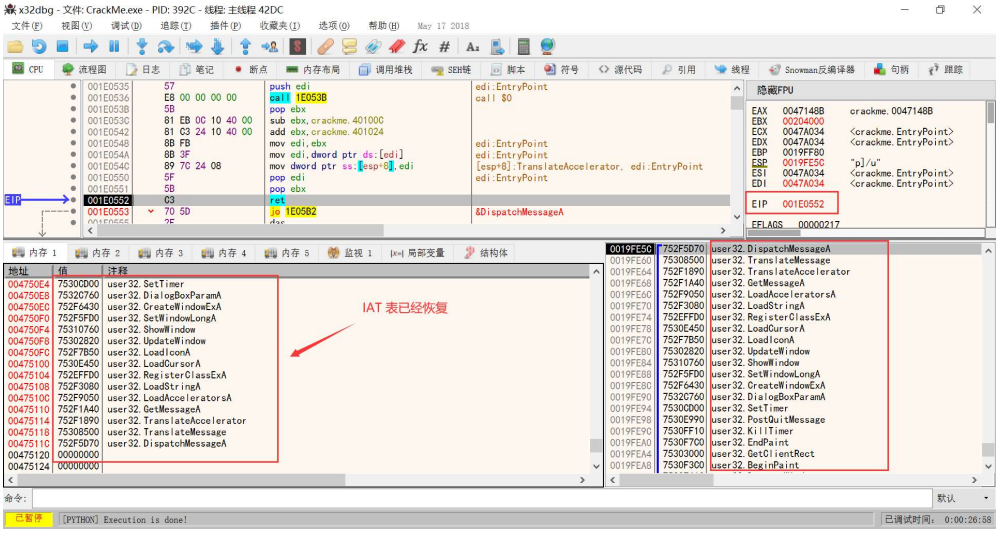
顶，再通过 "ret" 进行返回，如下图所示：



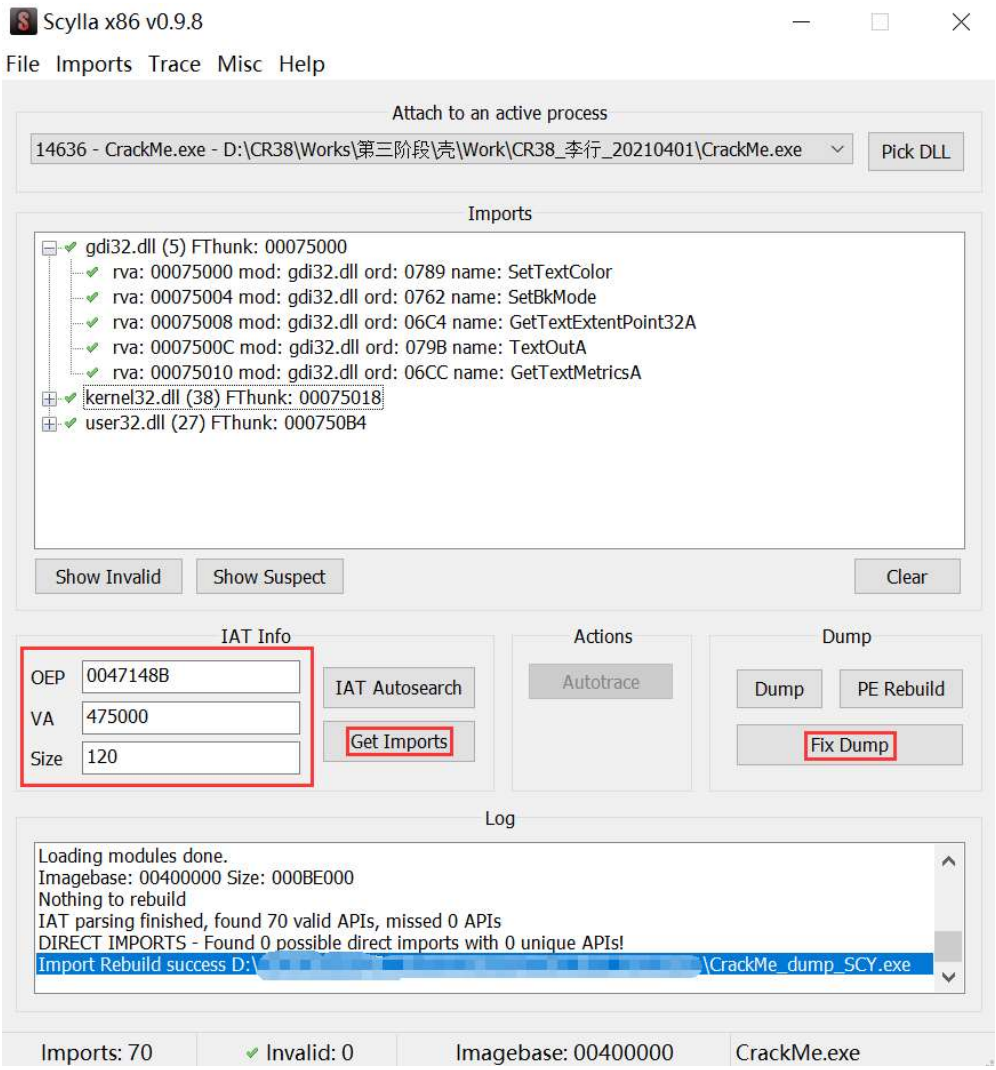

7. 修正IAT表的方法，遍历IAT表，从中取出每一个堆地址，将取出的堆地址设置为新的EIP，并执行其代码到 ret。执行到 ret 指令时就可以从栈顶获取导出函数的地址，然年将其填回IAT表中到对应的位置上。之后按照同样的方法继续遍历下一个，最后就可以还原出原IAT表。
8. 上述还原IAT表的操作手工做起来比较繁琐，可以编写一个脚本程序，让调试器去自动完成。x64dbg需要安装 "x64dbgpy" 插件，同时需要安装python。




9. 在 x64dbg 中运行编写的恢复IAT表的脚本，等待IAT表的恢复，如下图所示：




10. 使用 x64dbg 自带的 "Scylla" 插件修复之前dump的可执行文件（其IAT表未修复），如下图所示：




> x64dbgpy内部的接口实现在 "x64dbgpy\x32\plugins\x64dbgpy\x64dbgpy\pluginsdk_scriptapi" 目录下。

脚本示例：

```
# -*- coding: UTF-8 -*-
from x64dbgpy.pluginsdk._scriptapi import *

# 程序在OEP 0x0047148B 地址上设置硬件断点
SetHardwareBreakpoint(0x0047148B)
Run() # F9
Wait() # 等待硬件执行断点命中

# 从IAT表中取出每一项，设置为新的EIP，然后单步执行到ret
# 接着从栈顶取出真正的导出函数的地址，并写入到IAT表中对应的下标上

# 设置IAT表的首地址以及结束地址
dwIATBase = 0x00475000
dwIATEnd = 0x00475120

# 循环获取IAT表
```

```
dwIATItem = dwIATBase
while dwIATItem < dwIATEnd:
  # 取出IAT表项，设置为新的EIP
  dwNewEIP = ReadDword(dwIATItem)
  # 跳过IAT表项地址为0的情况
  if dwNewEIP == 0:
    dwIATItem += 4
    continue
  SetEIP(dwNewEIP)

  # 单步执行到ret
  while True:
    StepIn()
    # 判断是否单步到ret
    byteCode = ReadByte(GetEIP())
    if byteCode == 0xC3:
      break

  # 从栈顶取出导出函数对应的地址
  dwReadAPIAddr = ReadDword(GetESP())
  # 将获取到的导出函数对应的地址写回到IAT表对应的项上
  WriteDword(dwIATItem, dwReadAPIAddr)

  # 处理IAT表的下一项
  dwIATItem += 4
```

# 壳的对抗

1. 反调试
2. 混淆
    - 代码膨胀
    - 流程混淆
    - IAT
3. 虚拟机