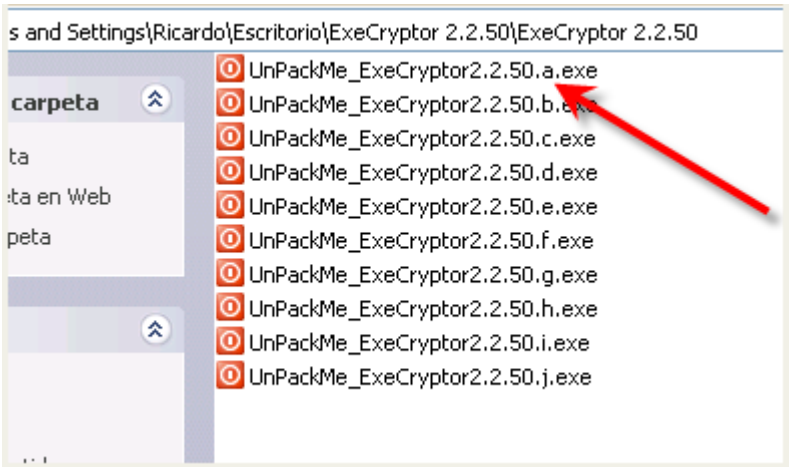
第五十四章-EXECryptor v2.2.50.a 脱壳-Part1

当我们遇到一款之前没有分析过的壳的时候,不要盲目的下手,我们最好先在网上搜一下该壳相关的 UnPackMe。如果有相关的 UnPackMe 的话,我们可以将 UnPackMe 与目标程序对照着来分析。如果实在找不到相关的 UnPackMe 的话,我们可以去下载一个该壳的加壳器,然后我们找一个比较简单的小程序(PS:或者自己编写一个小程序也可以,有源代码最好不过了)来作为加壳的对象。接着我们使用该壳的加壳器对我们的小程序进行加壳,我们逐一选择不同的加密强度,从最低保护强度到最高保护强度。接下来逐一分析该壳不同的加密选项都有什么不同,等我们把该壳的各个保护手段都研究透彻了以后,再来分析我们最初的目标程序,就会容易很多。就算以后该壳发布了新的版本,我们有了对其之前版本的深入理解,再来分析其最新版本应该也不会太困难,因为通常来说,新版本的改动不会很大。

我已经为大家准备好了 EXECryptror 的一系列 UnPackMe,虽然不是 EXECryptror 最新的版本,但是对于研究 EXECryptror 的保护机制已经足够了。(PS:EXECryptror 在当年来说算的上一款猛壳)

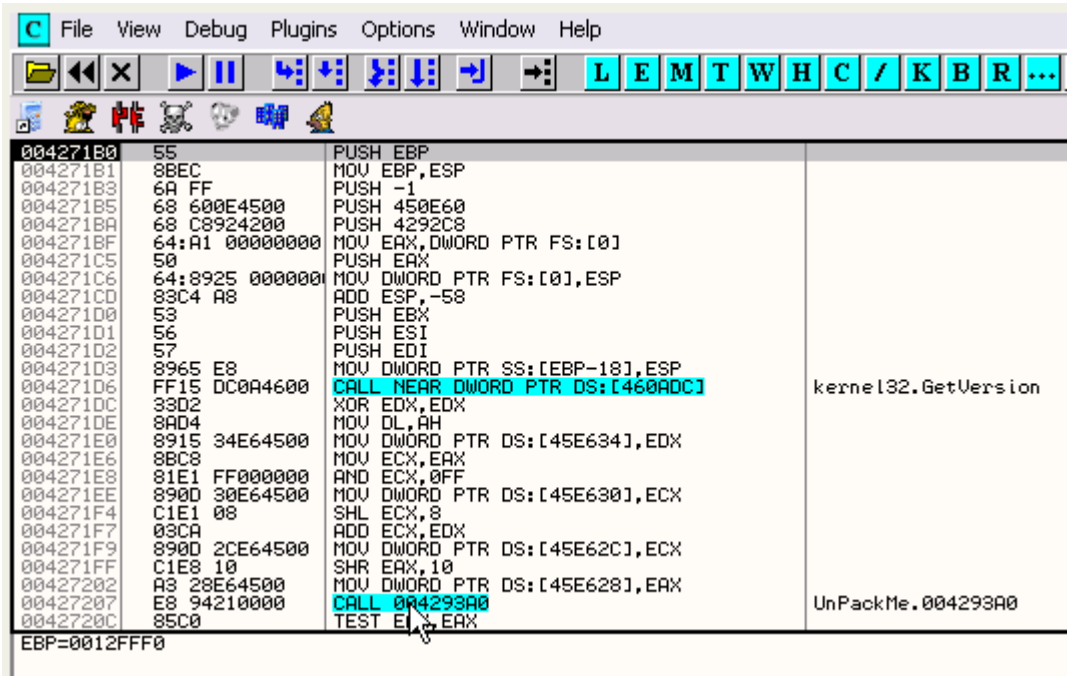
这里我们可以看到不同等级的 UnPackMe,难度逐一递增。



本章我们的目标程序是上图中的这个 UnPackMe_ExeCryptor2.2.50.a.exe。我们直接运行该程序,在弹出的对话框中可以看到保护措施。

我们可以看到这个等级的加密强度几乎为零,只是简单的压缩代码/数据/资源,跟 UPX 壳的做法很比较相似。这里如果我们将 UPX 的 UnPackMe 与该程序对照着来分析的话,就可以很容易的得知其 OEP 以及 IAT 的起始地址,大小。

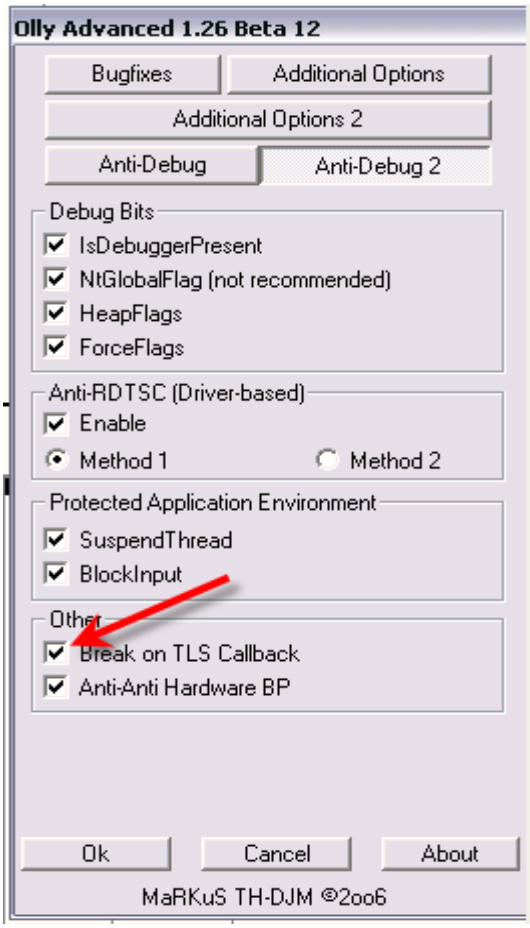
我们用 OD 加载 UPX 的 UnPackMe,断在了入口点处。



这个入口我们应该很熟悉了吧,可以说几乎本系列教程的每一个章节都可以看到。由于该 UnPackMe_ExeCryptor2.2.50.a.exe 与 UnPackMe_UPX1.91.a.exe 的原程序都是一样的,所以加壳以后,它们的 OEP,以及 IAT 也应该是一样的,它们调用的第一个 API 函数都是 GetVersion。我们可以得到这些基本的信息。

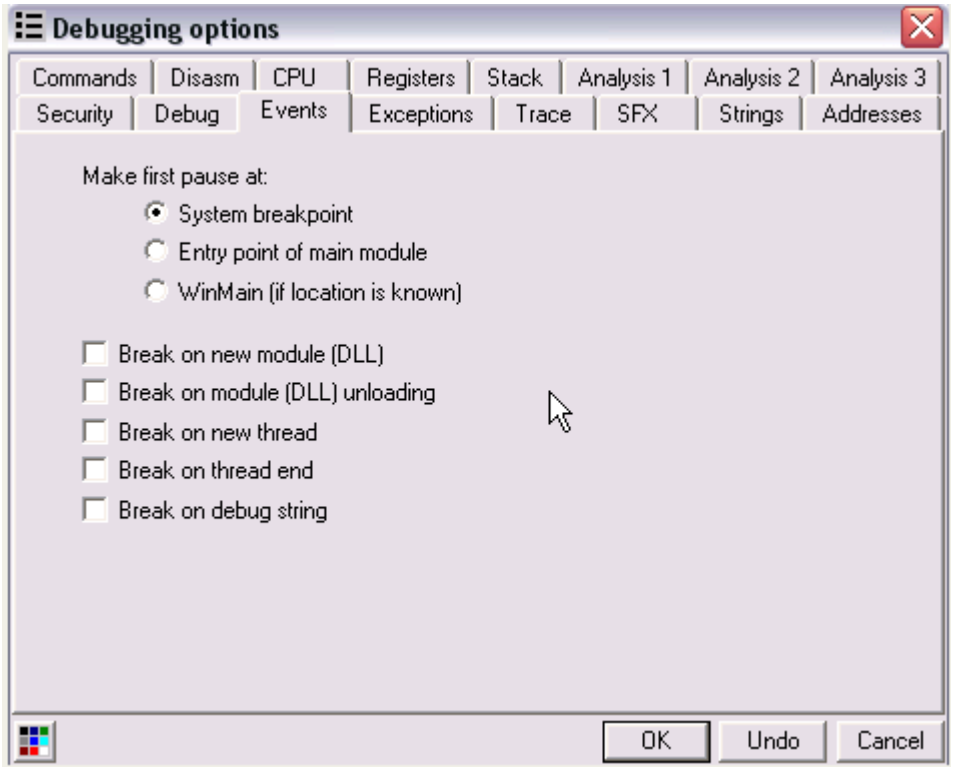
好,下面我们来分析 UnPackMe_ExeCryptor2.2.50.a.exe。

首先配置好 OllyAdvanced 这个反反调试插件。



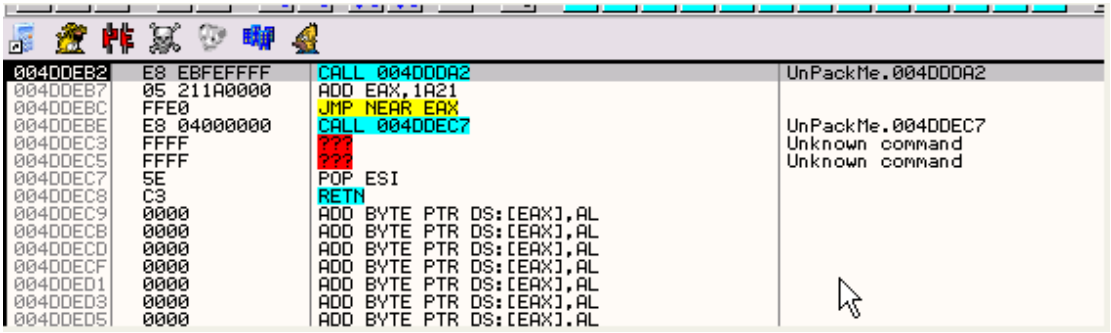
这里我用 Patched 4 这款 OD(我一般都是用这个版本),OllyAdvanced 这款插件里面有很多选项可供我们选择,其他的选项勾不勾选无所谓,但是 Break on TLS Callback 这个选项这里我们一定要记得勾选。

如果大家用 OD 加载 UnPackMe_ExeCryptor2.2.50.a.exe 的话,会发现还没有达到入口点程序就退出了。这是因为 ExeCryptor 利用了 TLS CALLBACK 这一特性在入口点之前执行代码-检测是否正在被调试,如果是,则退出进程。TLS CALLBACK 可以在入口点之前执行代码这个特性最初是由一个病毒作者发现的,后来被 ExeCryptor 的作者利用来进行反调试。

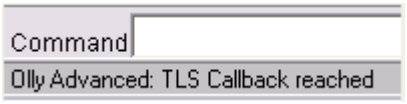


这里我们将首次中断的地方切换为 System breakpoint 处。

我们一运行起来就会断在 TLS CALLBACK 处。

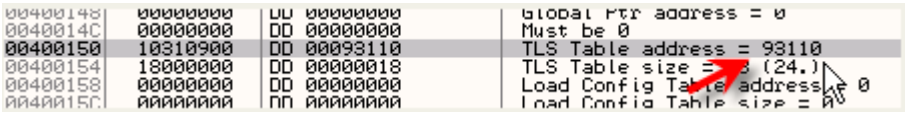


我们可以看到 OD 状态栏上的提示。

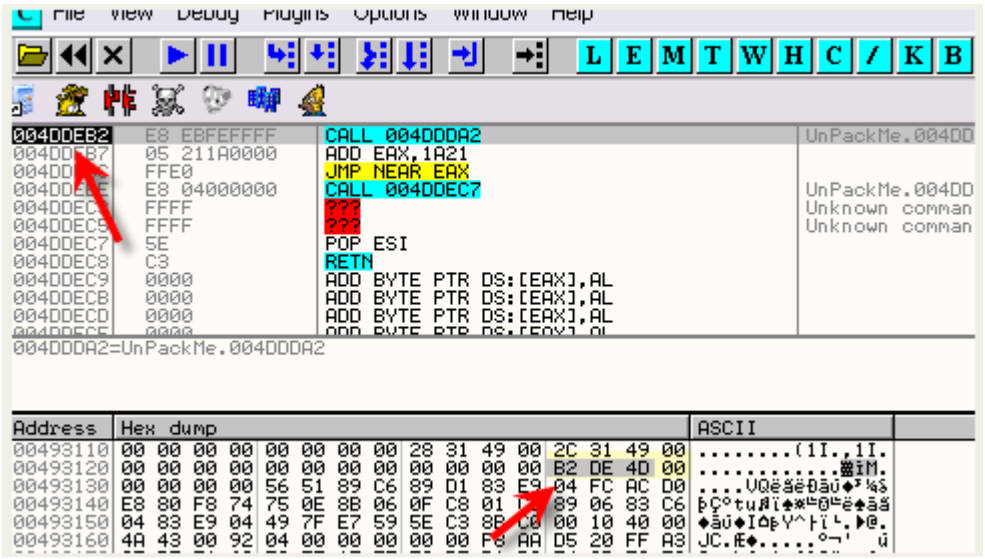


这是 OllyAdvanced 这个插件帮我们定位到的 TLS CALLBACK 回调函数的入口地址。下面我们来看看如何手工定位 TLS CALLBACK 回调函数的入口地址。

我们通过在数据窗口中按 CTRL+G 输入 400000 定位到 PE 头,然后单击鼠标右键选择-Special-PE header 将数据窗口的显示模式切换为 PE 解析模式,往下拉。

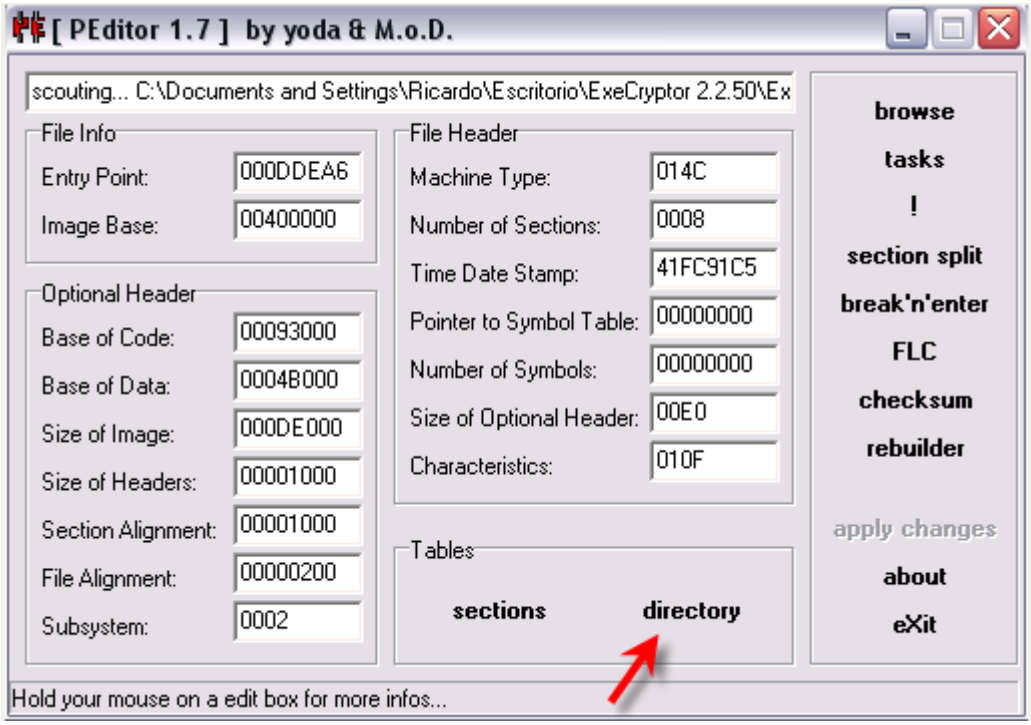


这里我们可以看到 TLS Table address 为 93110(RVA),加上映像基址 400000 就得到了 493110,即 TLS TABLE 的起始地址。我们在数据窗口中定位到该地址,我们往下面看就可以找到 TLS 回调函数的入口地址。

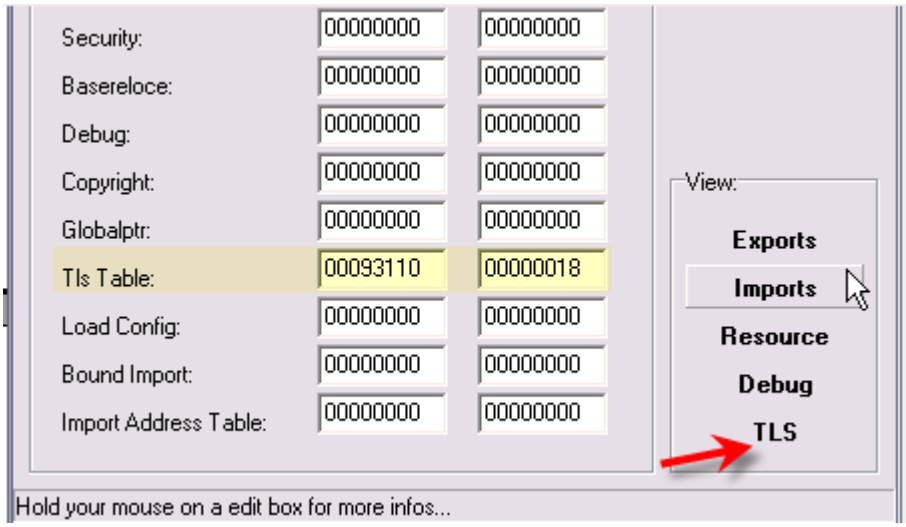


这里我们就定位到了 ExeCryptor 在到达入口点之前要执行代码的起始地址了,使用 OllyAdvanced 插件的话,它可以帮助我们直接定位到这个地址。

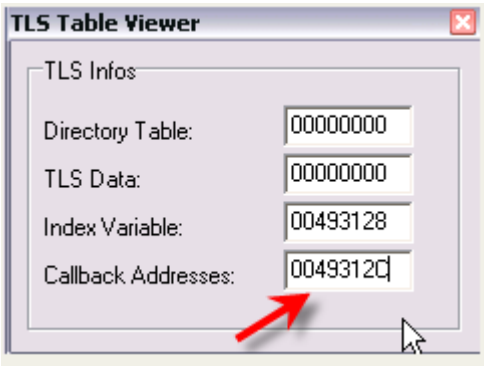
接下来我们来看看如何使用 PE 文件编辑器来定位 TLS 回调函数的入口地址。



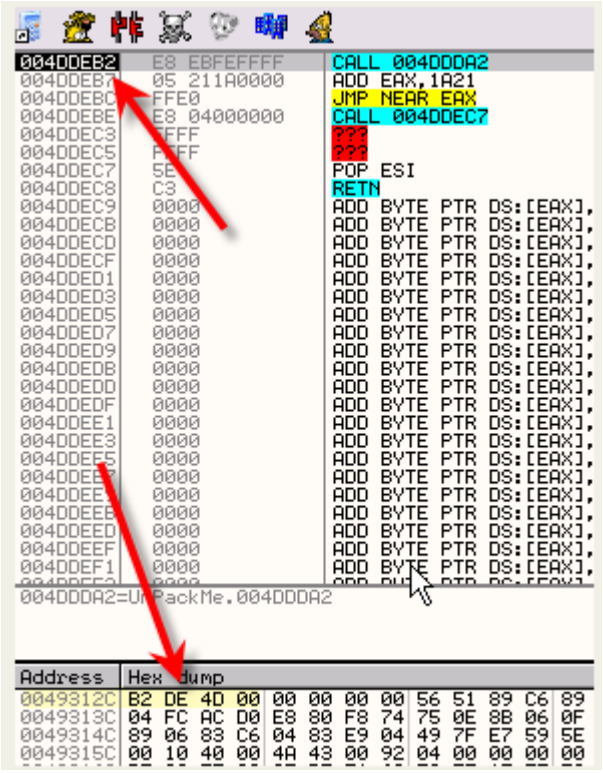
这里我们单击 directory 按钮查看数据目录。



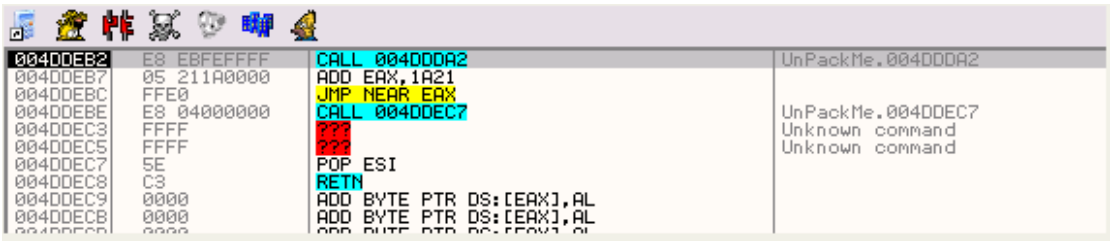
我们可以看到 TLS Table 起始地址的 RVA 为 93110,大小为 18。这右边还有个 TLS 按钮,单击该按钮我们就可以精确的查看 TLS 的回调函数入口地址的指针等信息。



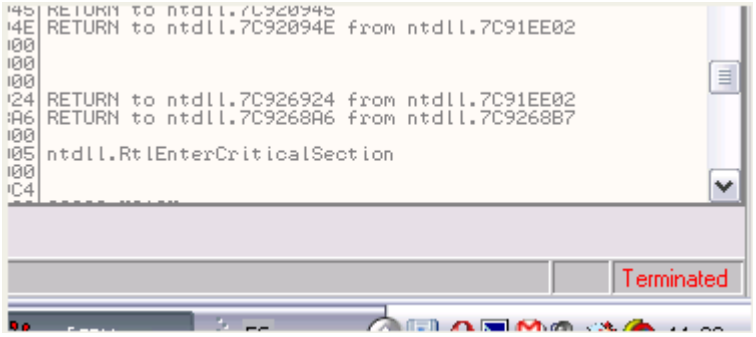
我们可以看到 TLS 回调函数的入口地址存放在 49312C 中,我们在数据窗口中定位到该地址。



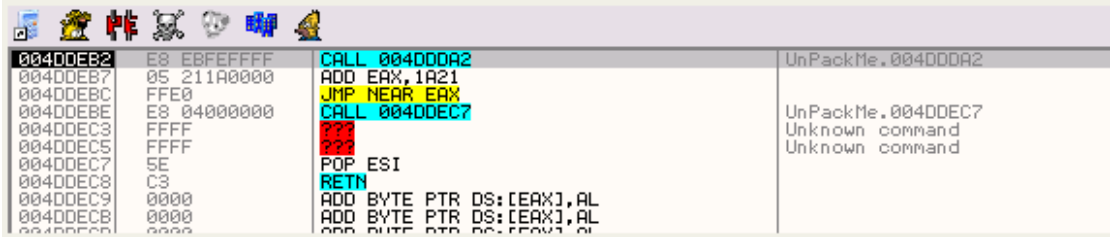
我们可以看到的确是 TLS 回调函数的入口地址,嘿嘿。好了,现在我们就知道如何用 OD,PE 文件编辑器以及 OllyAdvanced 插件来定位 TLS 回调函数的入口地址了。使用 OllyAdvanced 插件的话,我们直接就可以断在 TLS 回调函数的入口地址处,如果是手工的话,我们首先要断在系统断点处,然后在 TLS 回调函数的入口处设置一个断点,然后运行起来,就可以断在 TLS 回调函数的入口处了。



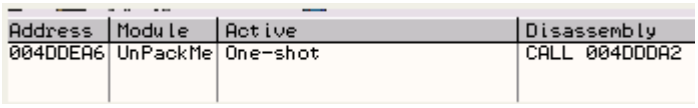
我们运行起来。



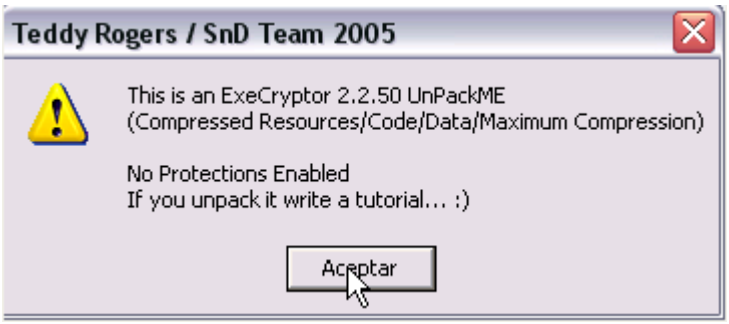
这里我们可以看到壳已经检测到自己正在被调试,退出了进程。OllyAdvanced 插件里面的选项我们都勾选上了还被检测到,那我们用 AntiDetectOlly 这个工具 Patch 一下 OD 试试看,会发现还是会被检测到。好,那我们再次回到 TLS 回调函数入口,看看为什么会被检测到。



这里我们打开断点窗口查看一下,尽管我们之前并没有设置断点,我们可以看到这里有一个一次性断点。



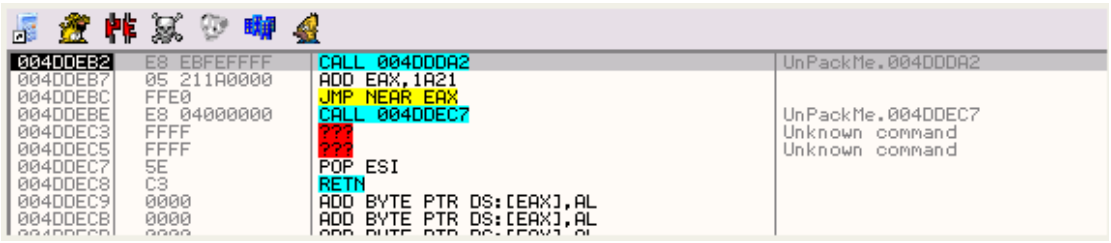
而此时壳在 TLS 回调函数中的检测代码还没有执行,当检测代码执行的时候,就会发现内存中有指令被替换成 CC,也就说明正在被调试,所以这里我们删除掉这个断点,然后运行起来,看看还会不会被检测到。



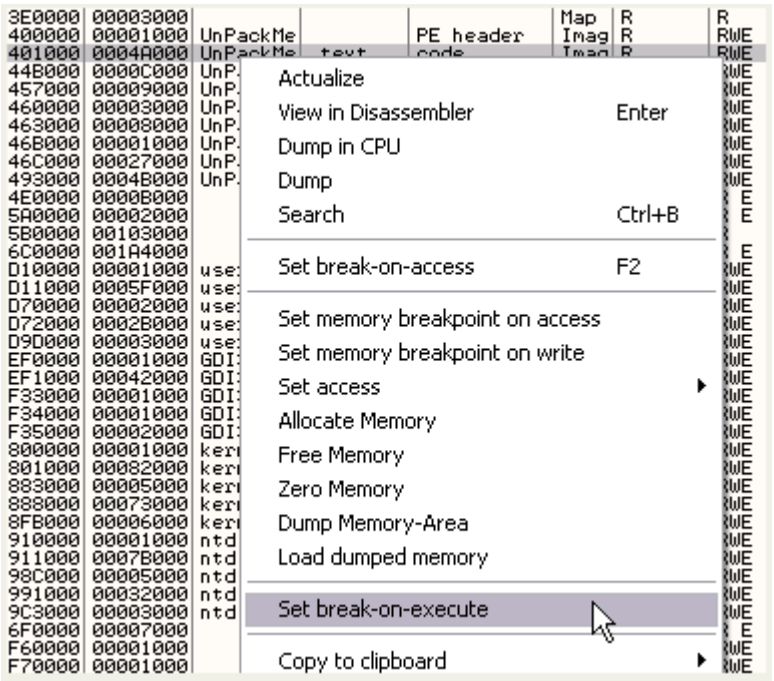
这里我们可以看到程序正常运行起来了,并没有退出。也就是说的确是这个断点被壳检测到了,才导致退出的。我们直接手动将该断点删除即可。如果遇到有的情况,删除了这个断点,还是退出的话,那么就是说出除了 OllyAdvanced 插件里面的反反调试选项以

外,我们还要添加其他的反反调试选项。

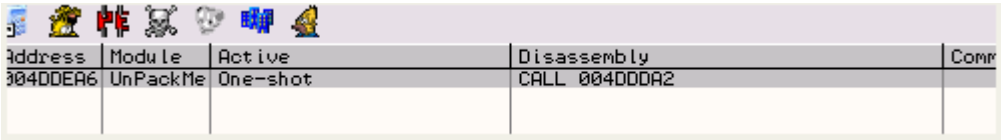
好了,现在问题我们已经解决了,我们重启 OD。



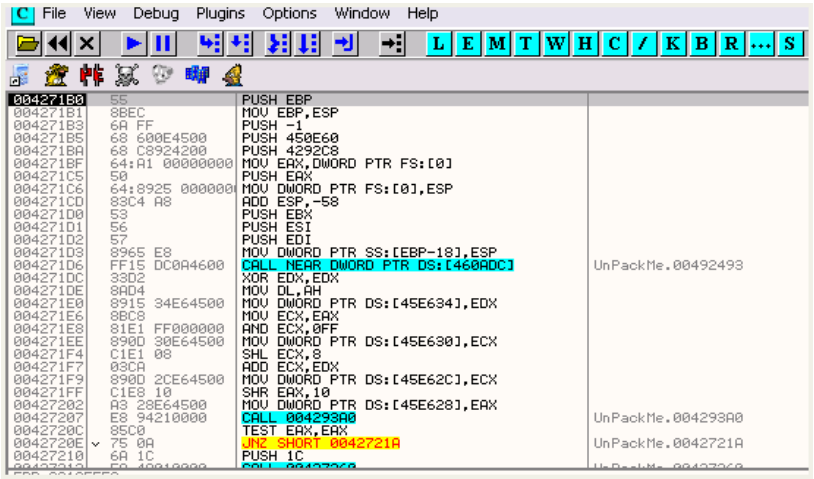
现在我们打开区段列表窗口,假设 OEP 位于代码段的话,那么我们选中起始地址为 401000 的区段(代码段),单击鼠标右键选择 Set break-on-execute。



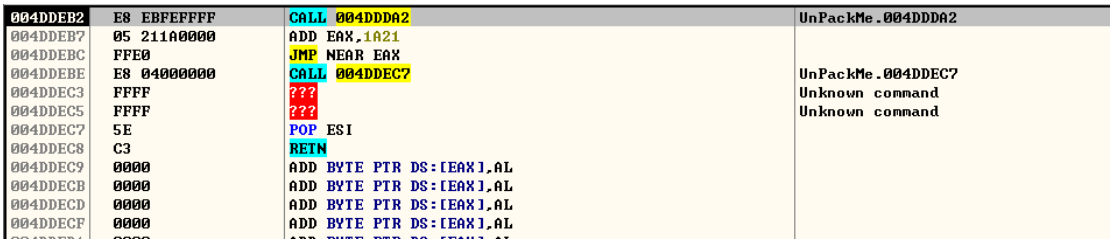
因为我们刚刚重启了 OD,所以我们还要再次删除断点列表窗口中的断点。



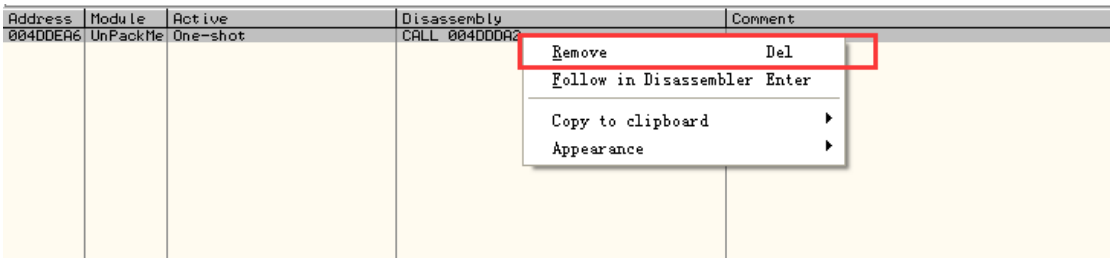
运行起来。



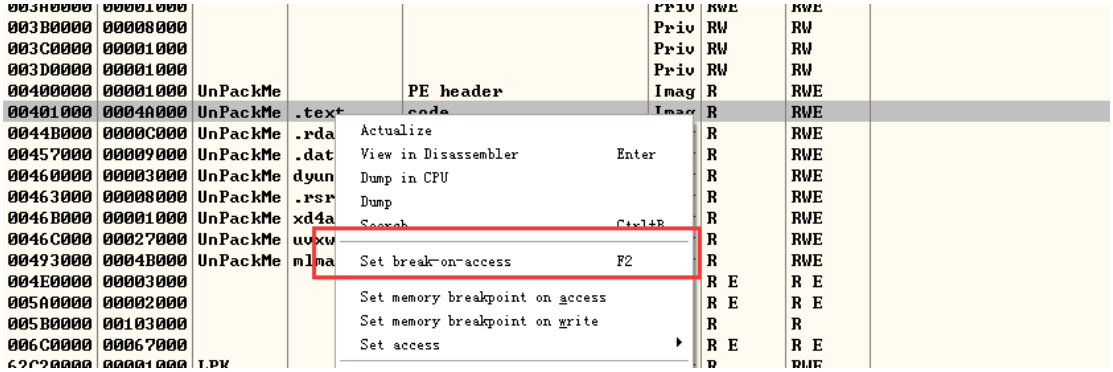
(PS:这里利用 OllyBone 这个插件的 Set break-on-execute 选项,我依然是怎么断也断不下来,无语球了,哈哈哈。等明年有时间我自己写个 break-on-execute 的插件吧!这里的话我就给大家介绍一下我断 OEP 的方法吧,嘿嘿。首先最后一次异常法大家就不要想了,因为压根就没有异常,哈哈。我呢,是用 OD 自带的 Set break-on-access 这个选项来定位 OEP 的,由于是访问断点,所以读取,写入,执行的时候都会断下来,所以肯定是没有 OllyBone 的 break-on-execute 快的,但是 OllyBone 不好用,我也没有办法。我们重启 OD,断在了 TLS CALLBACK 回调函数的入口处。



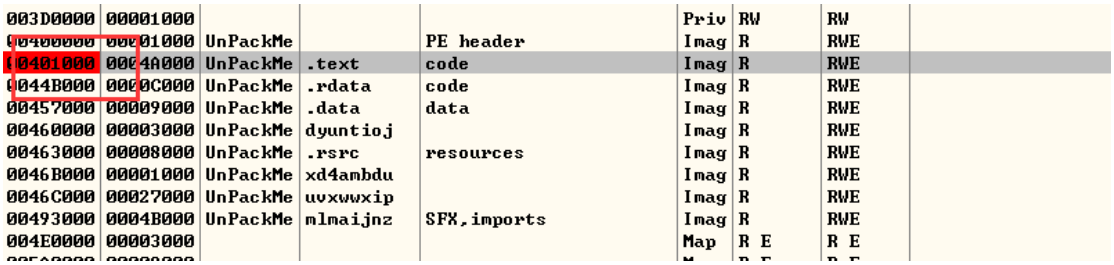
老规矩,删除掉断点列表窗口中的一次性断点。



接着给代码段设置 break-on-access 断点。



这里的大家要记住,break-on-access 是一次性断点,断下来了就没了,下次要用的话,还要设置一次。



这里我们就设置完毕 break-on-access 断点了,我们可以看到该区段的起始地址被标注为红色了。

我们运行起来。

```
004DB303   AA              STOS BYTE PTR ES:[EDI]
004DB304   EB E9           JMP SHORT 004DB2EF                  UnPackMe.004DB2EF
004DB306   E8 FC000000     CALL 004DB407                       UnPackMe.004DB407
004DB30B   0F82 97000000   JB 004DB3A8                         UnPackMe.004DB3A8
004DB311   E8 F1000000     CALL 004DB407                       UnPackMe.004DB407
004DB316   73 5B           JNB SHORT 004DB373                  UnPackMe.004DB373
004DB318   B9 04000000     MOV ECX,4
004DB31D   E8 FD000000     CALL 004DB41F                       UnPackMe.004DB41F
004DB322   48              DEC EAX
004DB323   74 DE           JE SHORT 004DB303                   UnPackMe.004DB303
004DB325   0F89 C7000000   JNS 004DB3F2                        UnPackMe.004DB3F2
004DB32B   E8 D7000000     CALL 004DB407                       UnPackMe.004DB407
004DB330   73 1B           JNB SHORT 004DB34D                  UnPackMe.004DB34D
```

断在了这里,从 OD 状态栏中的提示信息,我们可以知道这是由于写入导致的中断。

```
00457100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
00457110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
00457120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
Command :
Break-on-access when writing to [00401000]
```

我们要的是执行导致的中断,而不是读取或者写入导致的中断。

这里大家不要盲目的再次设置 break-on-access 断点,然后按 F9 键直接运行起来。

如果基础好的童鞋的话,一眼就可以看出这里是一个循环,大家看出来没有?我的天!有童鞋说木有看出来!

我们用鼠标选中接下来的 4DB304 这个地址处的跳转指令。

```
004DB2EE   43              INC EBX
004DB2EF   31C0            XOR EAX,EAX
004DB2F1   E8 11010000     CALL 004DB407
004DB2F6   73 0E           JNB SHORT 004DB306
004DB2F8   8B4D F0         MOV ECX,DWORD PTR SS:[EBP-10]
004DB2FB   E8 1F010000     CALL 004DB41F
004DB300   0245 EF         ADD AL,BYTE PTR SS:[EBP-11]
004DB303   AA              STOS BYTE PTR ES:[EDI]
004DB304   EB E9           JMP SHORT 004DB2EF
004DB306   E8 FC000000     CALL 004DB407
```

看到没,出现了向上指的红色箭头,这款代码不是循环操作是什么?嘿嘿。那么怎么样跳过循环呢?很简单,直接对下一条语句即 4DB306 处设置一个断点,然后运行起来,就可以跳过这个循环了。但是大家不要慌,将代码往下拉,看看这是不是一个嵌套循环。

```
004DB2E9   BA 00000080     MOV EDX,80000000
004DB2EE   43              INC EBX
004DB2EF   31C0            XOR EAX,EAX
004DB2F1   E8 11010000     CALL 004DB407                       UnPackMe.004DB407
004DB2F6   73 0E           JNB SHORT 004DB306                  UnPackMe.004DB306
004DB2F8   8B4D F0         MOV ECX,DWORD PTR SS:[EBP-10]
004DB2FB   E8 1F010000     CALL 004DB41F                       UnPackMe.004DB41F
004DB300   0245 EF         ADD AL,BYTE PTR SS:[EBP-11]
004DB303   AA              STOS BYTE PTR ES:[EDI]
004DB304   EB E9           JMP SHORT 004DB2EF                  UnPackMe.004DB2EF
004DB306   E8 FC000000     CALL 004DB407                       UnPackMe.004DB407
004DB30B   0F82 97000000   JB 004DB3A8                         UnPackMe.004DB3A8
004DB311   E8 F1000000     CALL 004DB407                       UnPackMe.004DB407
004DB316   73 5B           JNB SHORT 004DB373                  UnPackMe.004DB373
004DB318   B9 04000000     MOV ECX,4
004DB31D   E8 FD000000     CALL 004DB41F                       UnPackMe.004DB41F
004DB322   48              DEC EAX
004DB323   74 DE           JE SHORT 004DB303                   UnPackMe.004DB303
004DB325   0F89 C7000000   JNS 004DB3F2                        UnPackMe.004DB3F2
004DB32B   E8 D7000000     CALL 004DB407                       UnPackMe.004DB407
004DB330   73 1B           JNB SHORT 004DB34D                  UnPackMe.004DB34D
004DB332   55              PUSH EBP
004DB333   BD 00010000     MOV EBP,100
004DB338   E8 D7000000     CALL 004DB414                       UnPackMe.004DB414
004DB33D   8807            MOV BYTE PTR DS:[EDI],AL
004DB33F   47              INC EDI
004DB340   4D              DEC EBP
004DB341   75 F5           JNZ SHORT 004DB338                  UnPackMe.004DB338
004DB343   E8 BF000000     CALL 004DB407                       UnPackMe.004DB407
004DB348   72 E9           JB SHORT 004DB333                   UnPackMe.004DB333
004DB34A   5D              POP EBP
004DB34B   EB A2           JMP SHORT 004DB2EF                  UnPackMe.004DB2EF
004DB34D   B9 01000000     MOV ECX,1
004DB352   E8 C8000000     CALL 004DB41F                       UnPackMe.004DB41F
004DB357   83C0 07         ADD EAX,7
004DB35A   8945 F0         MOV DWORD PTR SS:[EBP-10],EAX
004DB35D   C645 EF 00      MOV BYTE PTR SS:[EBP-11],0
004DB361   83F8 08         CMP EAX,8
004DB364   74 89           JE SHORT 004DB2EF                   UnPackMe.004DB2EF
004DB366   E8 A9000000     CALL 004DB414                       UnPackMe.004DB414
004DB36B   8845 EF         MOV BYTE PTR SS:[EBP-11],AL
```

我们会发现 4DB364 处也是一个向上的跳转,说明这块代码是一个双层嵌套循环。

好,那么我们直接对 4DB364 这个地址的下一条语句处即 4DB336 地址处设置一个断点。

```
004DB2E9    BA 00000080         MOV EDX,80000000
004DB2EE    43                  INC EBX
004DB2EF    31C0                XOR EAX,EAX
004DB2F1    E8 11010000         CALL 004DB407                    UnPackM
004DB2F6    73 0E               JNB SHORT 004DB306               UnPackM
004DB2F8    8B4D F0             MOV ECX,DWORD PTR SS:[EBP-10]
004DB2FB    E8 1F010000         CALL 004DB41F                    UnPackM
004DB300    0245 EF             ADD AL,BYTE PTR SS:[EBP-11]
004DB303    AA                  STOS BYTE PTR ES:[EDI]
004DB304    EB E9               JMP SHORT 004DB2EF               UnPackM
004DB306    E8 FC000000         CALL 004DB407                    UnPackM
004DB30B    0F82 97000000       JB 004DB3A8                      UnPackM
004DB311    E8 F1000000         CALL 004DB407                    UnPackM
004DB316    73 5B               JNB SHORT 004DB373               UnPackM
004DB318    B9 04000000         MOV ECX,4
004DB31D    E8 FD000000         CALL 004DB41F                    UnPackM
004DB322    48                  DEC EAX
004DB323    74 DE               JE SHORT 004DB303                UnPackM
004DB325    0F89 C7000000       JNS 004DB3F2                     UnPackM
004DB32B    E8 D7000000         CALL 004DB407                    UnPackM
004DB330    73 1B               JNB SHORT 004DB34D               UnPackM
004DB332    55                  PUSH EBP
004DB333    BD 00010000         MOV EBP,100
004DB338    E8 D7000000         CALL 004DB414                    UnPackM
004DB33D    8807                MOV BYTE PTR DS:[EDI],AL
004DB33F    47                  INC EDI
004DB340    4D                  DEC EBP
004DB341    75 F5               JNZ SHORT 004DB338               UnPackM
004DB343    E8 BF000000         CALL 004DB407                    UnPackM
004DB348    72 E9               JB SHORT 004DB333                UnPackM
004DB34A    5D                  POP EBP
004DB34B    EB A2               JMP SHORT 004DB2EF               UnPackM
004DB34D    B9 01000000         MOV ECX,1
004DB352    E8 C8000000         CALL 004DB41F                    UnPackM
004DB357    83C0 07             ADD EAX,7
004DB35A    8945 F0             MOV DWORD PTR SS:[EBP-10],EAX
004DB35D    C645 EF 00          MOV BYTE PTR SS:[EBP-11],0
004DB361    83F8 08             CMP EAX,8
004DB364    74 89               JE SHORT 004DB2EF                UnPackM
004DB366    E8 A9000000         CALL 004DB414                    UnPackM
```

运行起来。

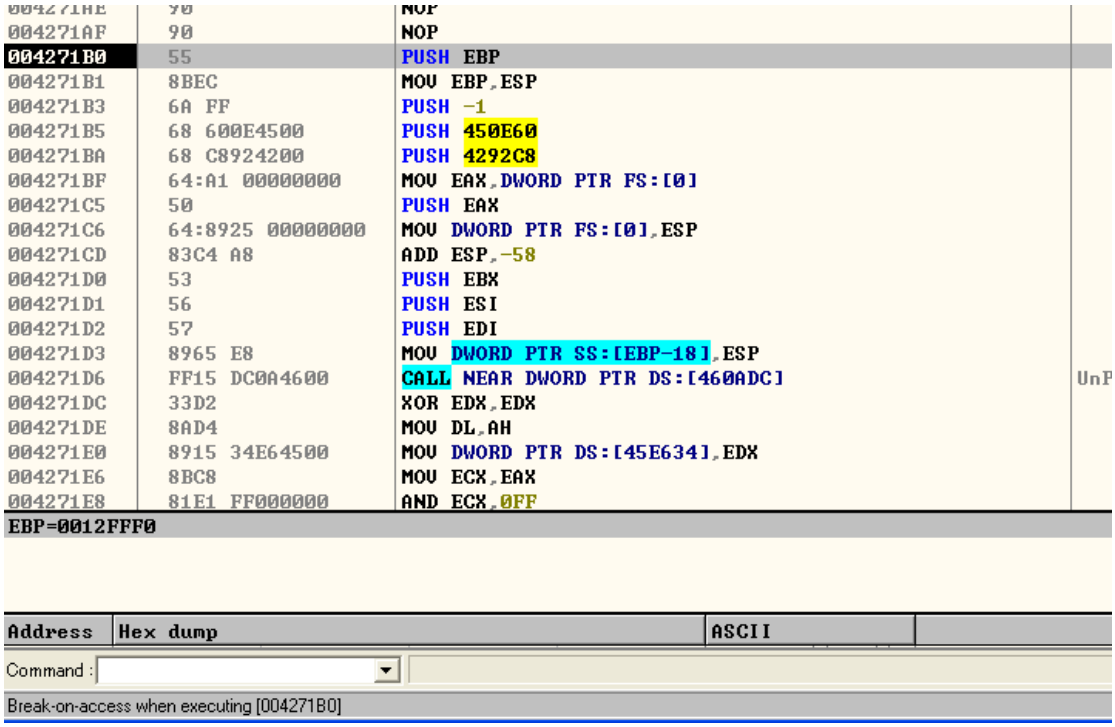
```
004DB357    83C0 07             ADD EAX,7
004DB35A    8945 F0             MOV DWORD PTR SS:[EBP-10],EAX
004DB35D    C645 EF 00          MOV BYTE PTR SS:[EBP-11],0
004DB361    83F8 08             CMP EAX,8
004DB364    74 89               JE SHORT 004DB2EF                U
004DB366    E8 A9000000         CALL 004DB414                    U
004DB36B    8845 EF             MOV BYTE PTR SS:[EBP-11],AL
004DB36E    E9 7CFFFFFF         JMP 004DB2EF                     U
004DB373    B9 07000000         MOV ECX,7
004DB378    E8 A2000000         CALL 004DB41F                    U
004DB37D    50                  PUSH EAX
```

我们可以看到断在了 4DB366 地址处,也就是我们跳过了这个双层嵌套循环。

接下来我们删除掉 4DB366 地址处的断点,然后依然是对代码段设置 break-on-access 断点。

```
003B0000  00008000                                  Priv  RW    RW
003C0000  00001000                                  Priv  RW    RW
003D0000  00001000                                  Priv  RW    RW
00400000  00001000  UnPackMe                        Imag  R     RWE
00401000  0004A000  UnPackMe  .text     code        Imag  R     RWE
0044B000  0000C000  UnPackMe  .rdata    code        Imag  R     RWE
00457000  00009000  UnPackMe  .data     data        Imag  R     RWE
00460000  00003000  UnPackMe  dyuntioj               Imag  R     RWE
00463000  00008000  UnPackMe  .rsrc     resources   Imag  R     RWE
0046B000  00001000  UnPackMe  xd4ambdu               Imag  R     RWE
0046C000  00027000  UnPackMe  uvxwwxip               Imag  R     RWE
00493000  0004B000  UnPackMe  mlmaijnz  SFX,imports  Imag  R     RWE
004E0000  00003000                                  Map   R E   R E
005A0000  00002000                                  Map   R E   R E
005B0000  00103000                                  Map   R     R
```

运行起来。



这样我们就断在了 OEP 处,注意到 OD 状态栏中的提示信息没有?该中断是由执行导致的。

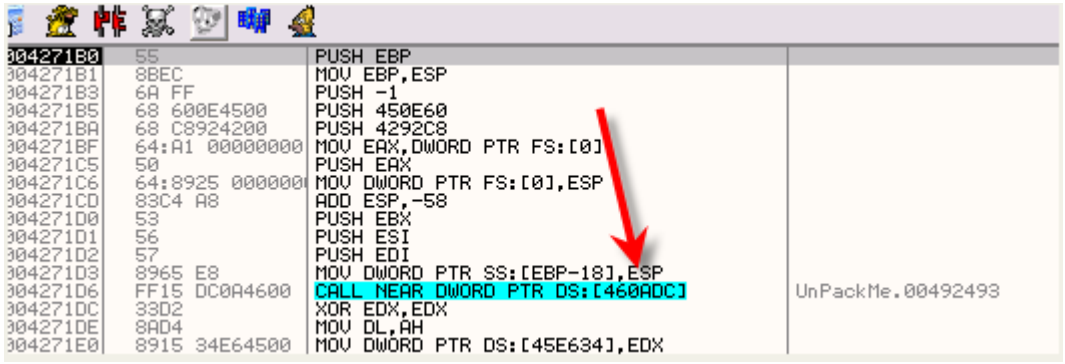
以上就是我定位 OEP 的方法。

不知道有木有童鞋的 OllyBone 插件能够断下来,如果你们中有人能断下来,请分享一下经验,谢谢,反正我是一次都没有断下来。

~~~~(>_<)~~~~

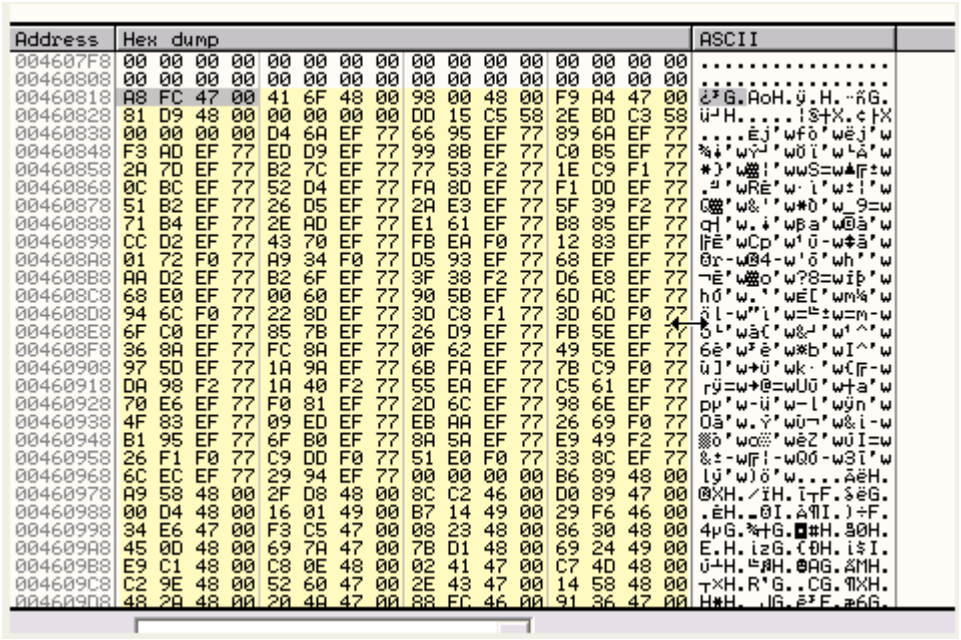
)

这里我们可以看到断在了我们熟悉的 OEP-4271B0 处了,这个等级的保护并不存在 stolen bytes,我们直接就可以定位到 OEP。

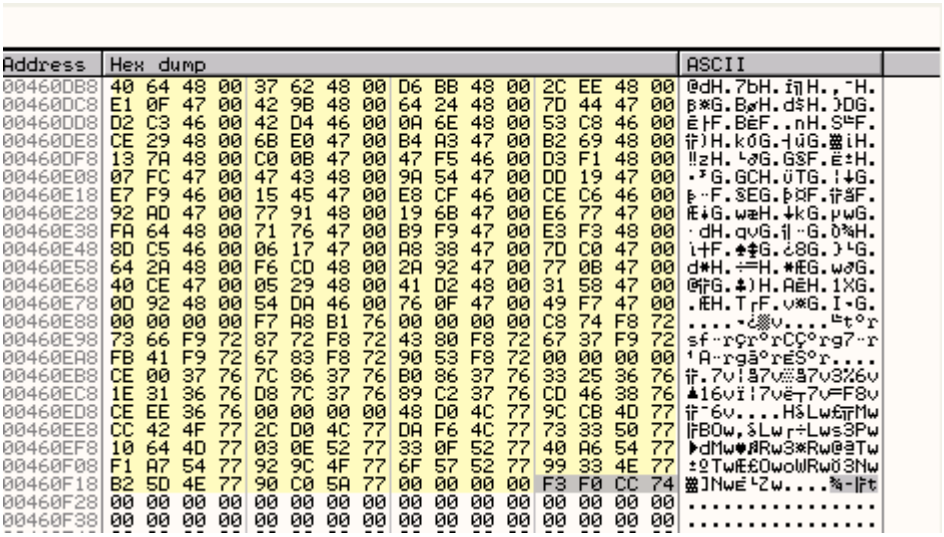


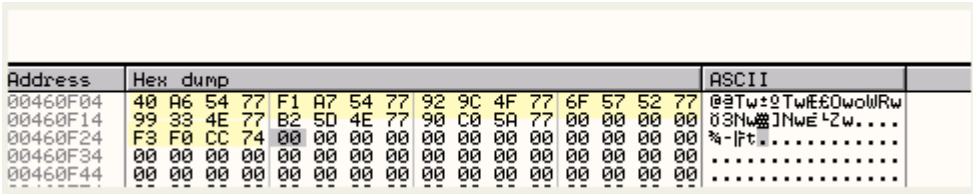
我们对比着 UPX 的 UnPackMe 来看,它到达 OEP 后以后,下面的 GetVersion 的调用处并没有被重定向,而我们这里被重定向了,下面我们来修复 IAT。

首先我们来定位 IAT 的起始地址和结束位置。
~~~~

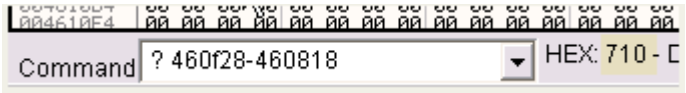


这里我们可以看到 IAT 的起始地址为 460818,跟 UPX 的 UnPackMe 一样。我们继续往下定位 IAT 的结束位置。





这里我们可以看到 IAT 的结束地址为 460F28。我们来计算一下 IAT 的长度。



计算得出 IAT 的长度为 710。

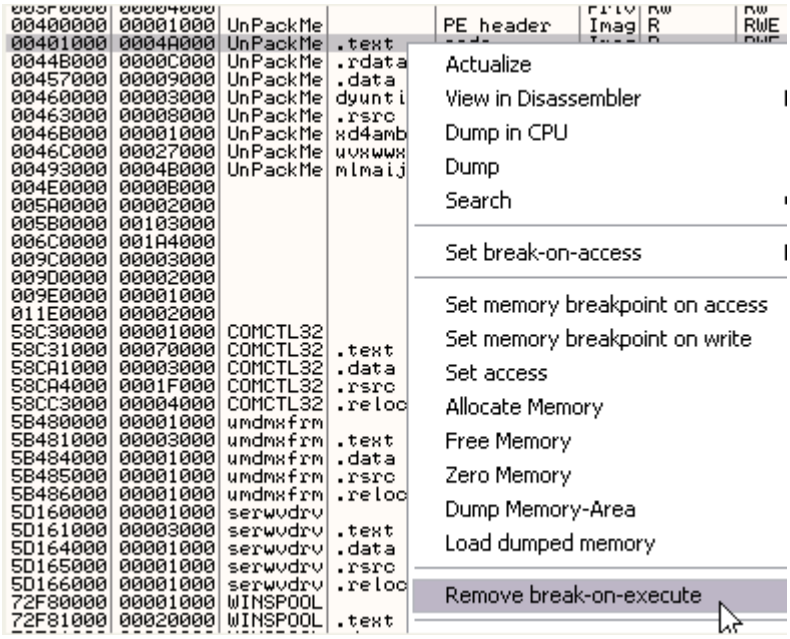
OEP(RVA):271B0

IAT 起始地址(RVA):60818

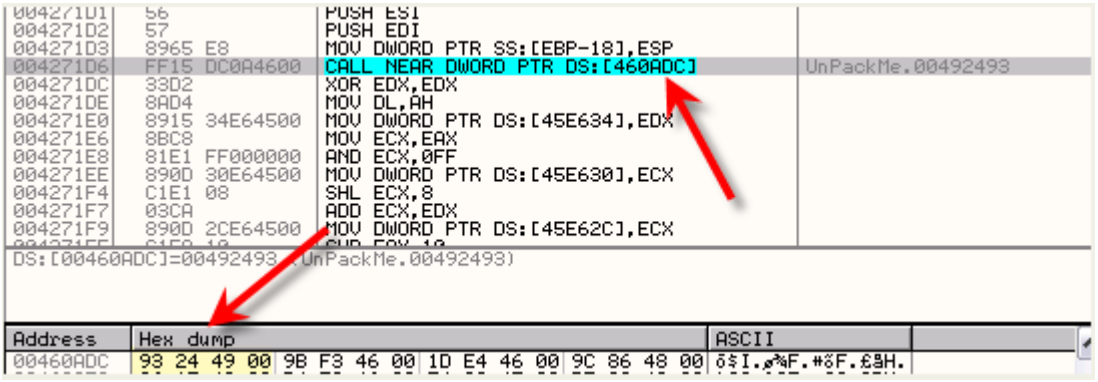
IAT 的大小:710

这里 IMP REC 重建 IAT 所需要的数据我们都有了。

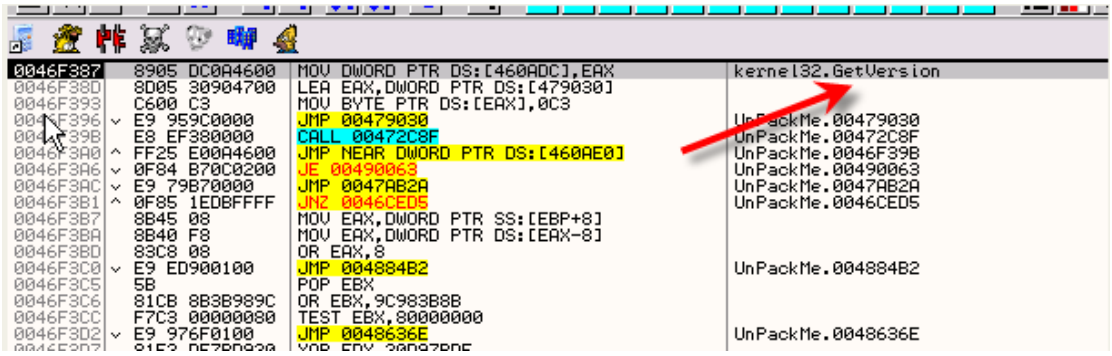
下面我们的任务就是来修复IAT,ExeCryptor并不存在我们前面章节介绍过的关键跳,它是怎么做的呢?它会在特定的时候将正确的API函数地址填充到对应的IAT项中,所以这里我们给GetVersion所在的IAT项设置内存写入断点。



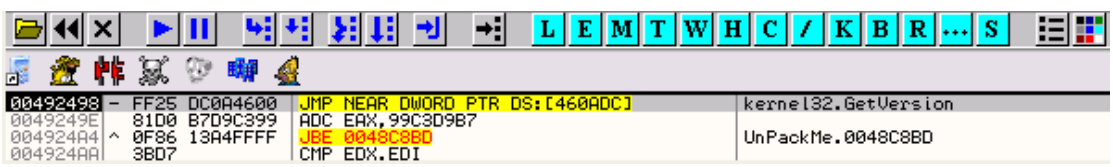
这里我们首先要选择Remove break-on-execute将break-on-execute断点删除掉,以免出错。



这里我们给将要调用的第一个API函数所在的IAT项设置了内存写入断点,运行起来,看看会发生什么。



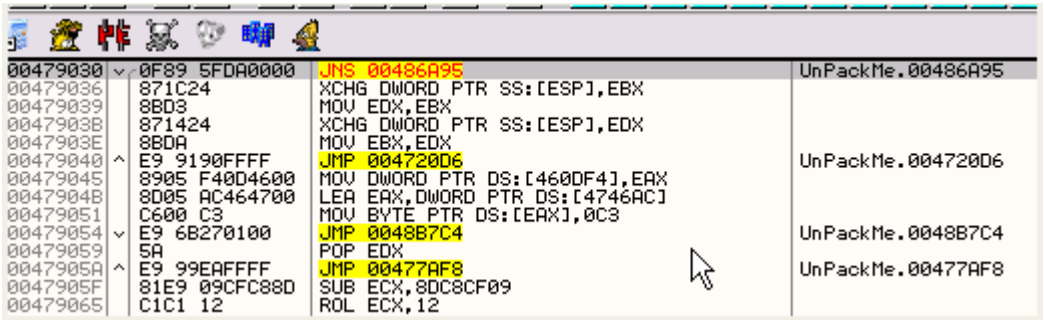
断在了这里,我们可以看到这一条指令是将正确的API函数地址保存到对应的IAT项中。



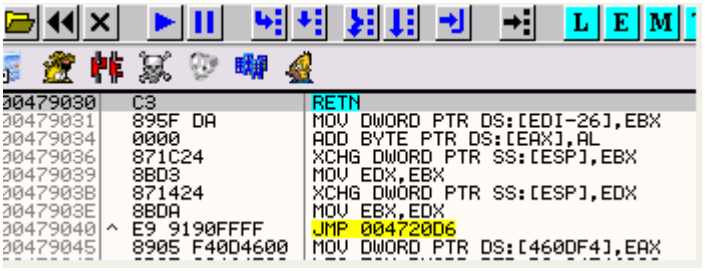
下面几行,我们会发现其会将 479030 处的代码修改为 C3,即 RET 指令,然后利用 RET 指令返回到某地址处再去调用实际的要调用的 API 函数,下面我们来详细跟踪一下这个流程。

我们可以看到在正确的 API 地址被保存到对应的 IAT 项中后,下面会向 479030 地址处写入一个 C3,我们看看 479030 地址处之前是什么内容。

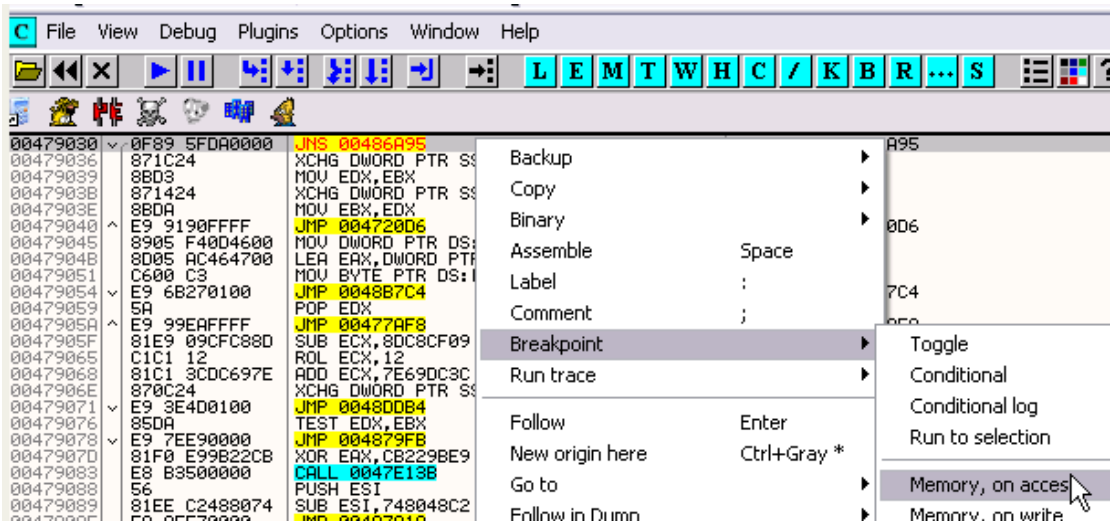
479030 之前是这样的:



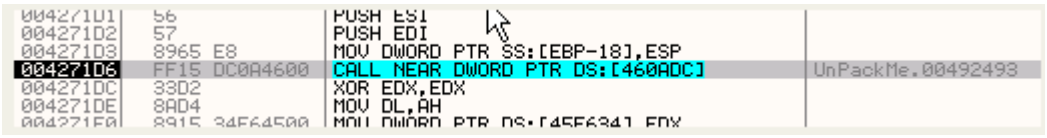
写入 C3 后变成了:



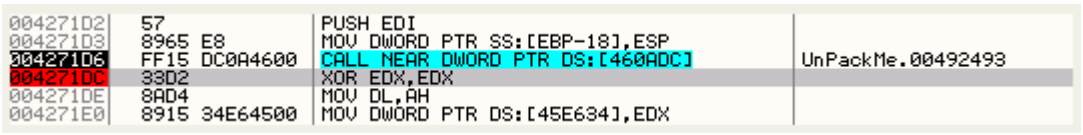
这里我们可以看到被修改为了 RET 指令,相当于该壳在修复 IAT 项以后再进行自修改(修改自身代码)。但是该壳的代码并不位于第一个区段,所以这里我们重启 OD,达到 OEP 以后对 479030 地址处的指令设置内存访问断点。



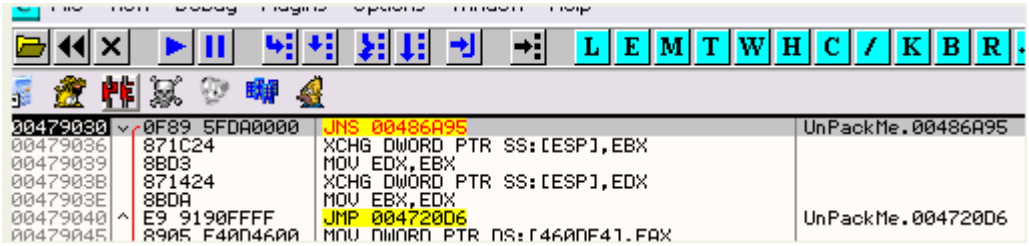
首先我们到达第一个 API 函数调用处。



下面我们给返回地址 4271DC 处设置一个断点。

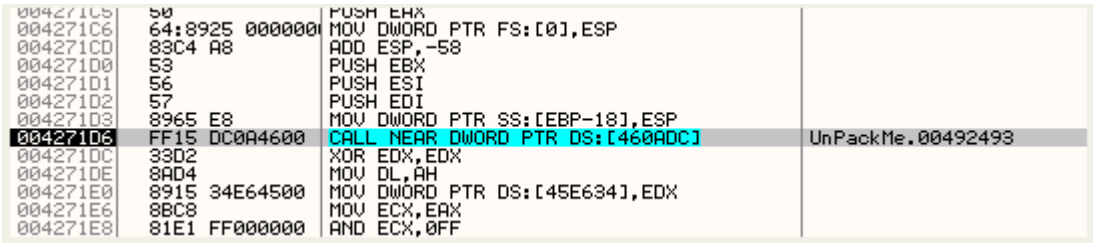


现在我们运行起来。

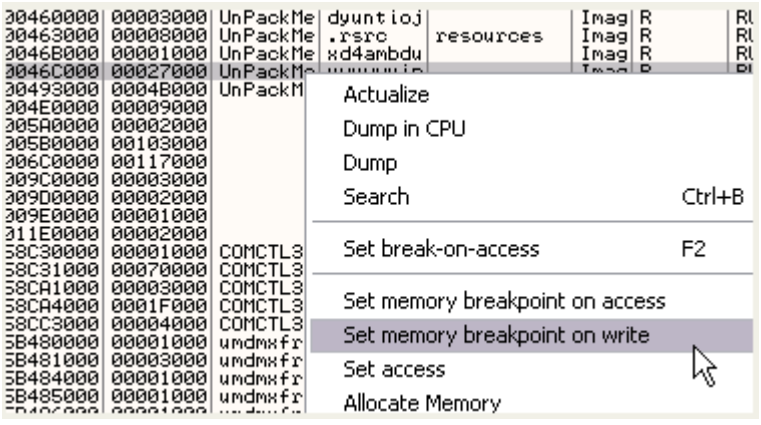


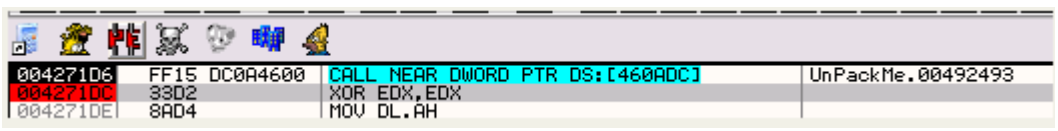
我们可以看到断在了条件跳转处,当 API 函数地址被填充到对应 IAT 项中以后,这里就会被自修改为 RET 指令,壳的自修改是我们要重点关注的,也就是说我们在到达 OEP 处以后,可以对壳修复 IAT 项代码所在的区段设置内存写入断点,当断下来时,就到了自修改的地方,

现在我们重启 OD,到达 OEP 处。

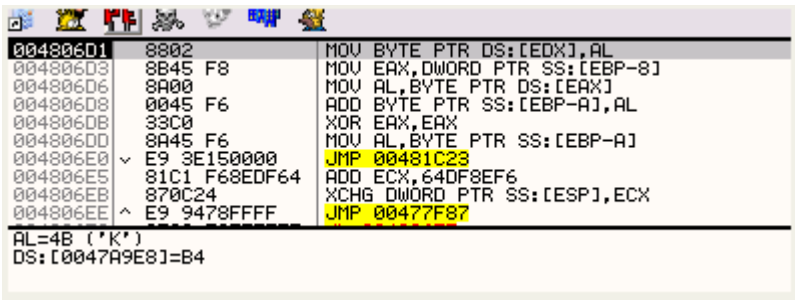


这里我们再次到达第一个 API 函数调用处,这里是 CALL 重定向后的地址 492493,当 479030 处被修改为了 RET 指令,重定向的 IAT 项已经被恢复为正常的 API 函数地址了,所以这里我们给壳所在的区段设置内存写入断点,接着运行起来,看看会发生什么。

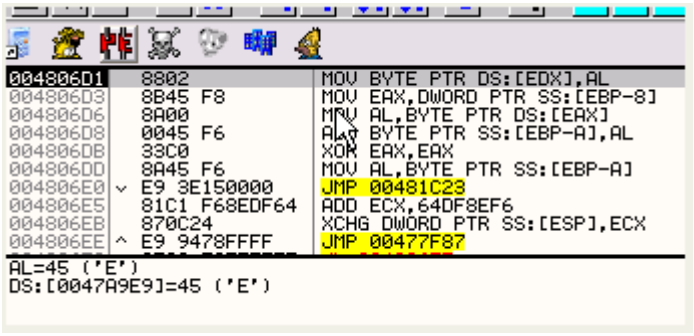


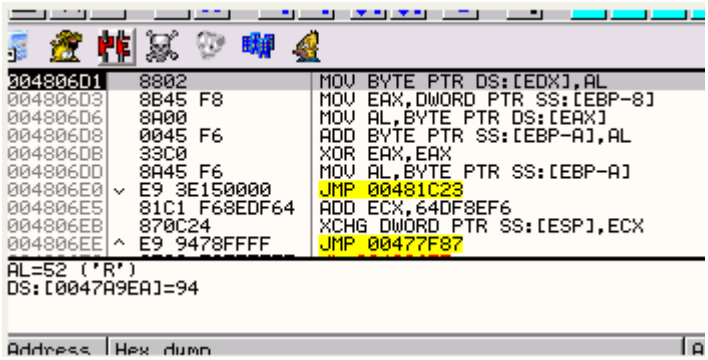


接着我们还是对返回地址 4271DC 处设置一个断点,运行起来,看看会发生什么。

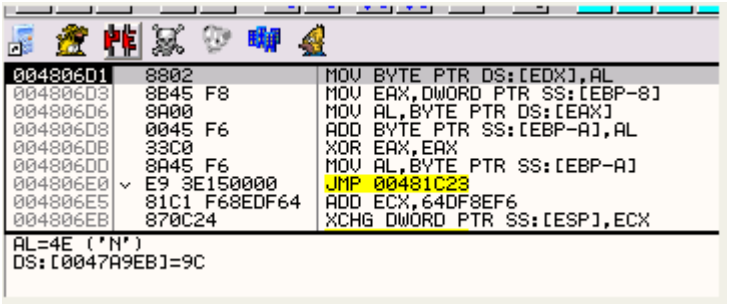


继续:

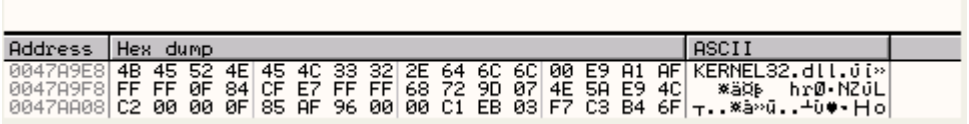




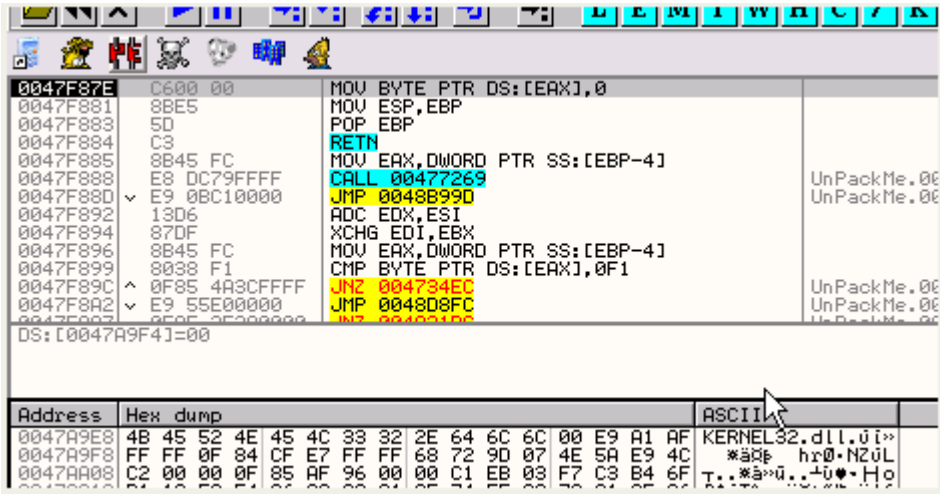
下一个:



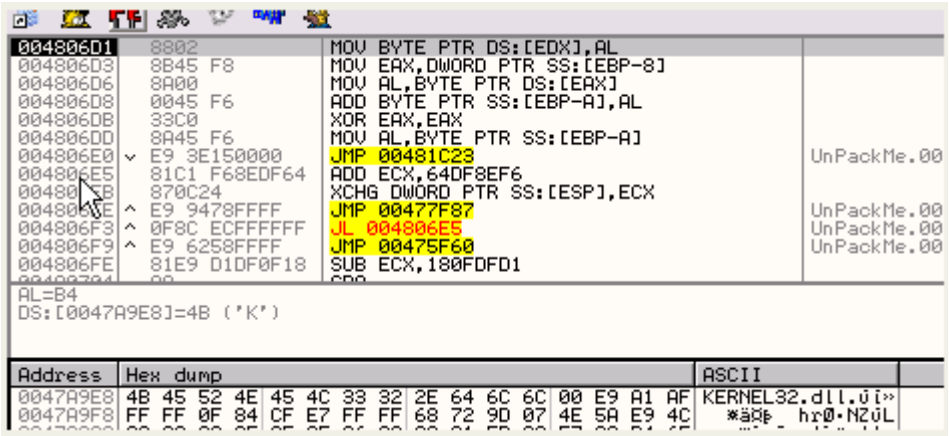
这是实际上是在填充一个字符串,我们就不一个字母一个字母的看了,我们直接在数据窗口中查看填充完毕后的整个字符串是什么。



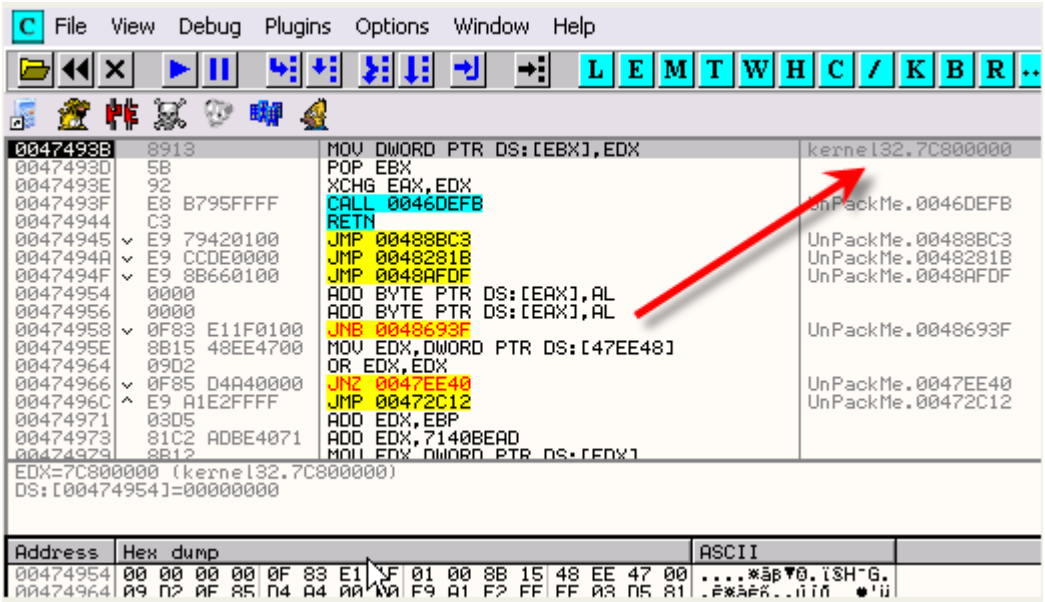
我们可以看到实际上是一个 DLL 的名称字符串,可能下面会被用来获取 API 函数的地址,我们继续。



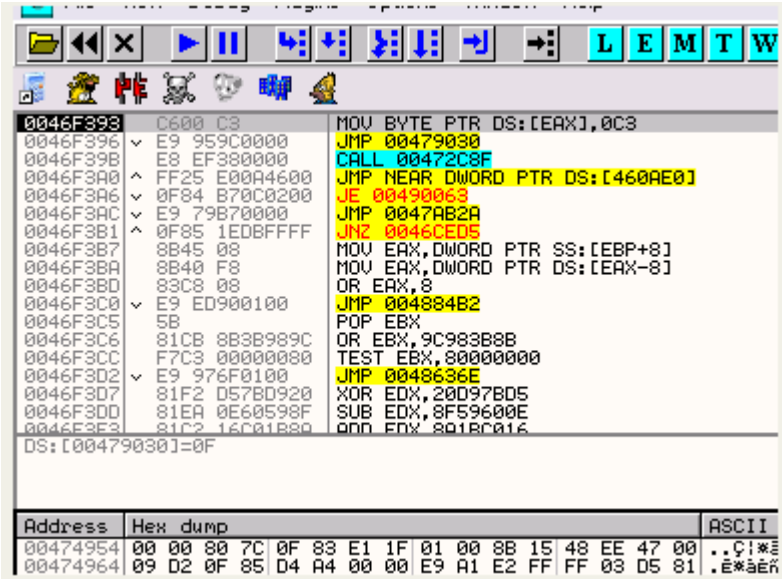
这里是添加字符串结束符'\0'。



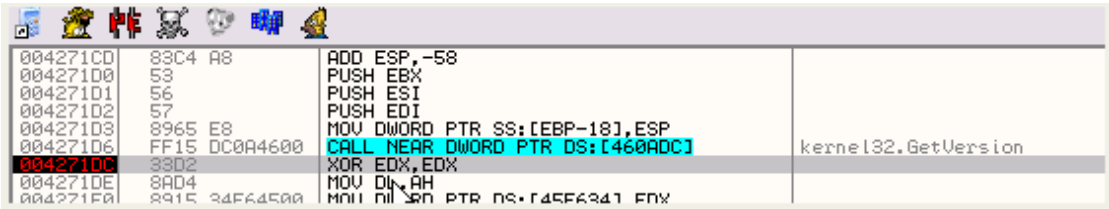
这里又是重复上面的步骤,但是这次是填充字符'K'的小端存储方式,4B 反过来就是 B4,这么做的目的可能是为了隐藏字符串。



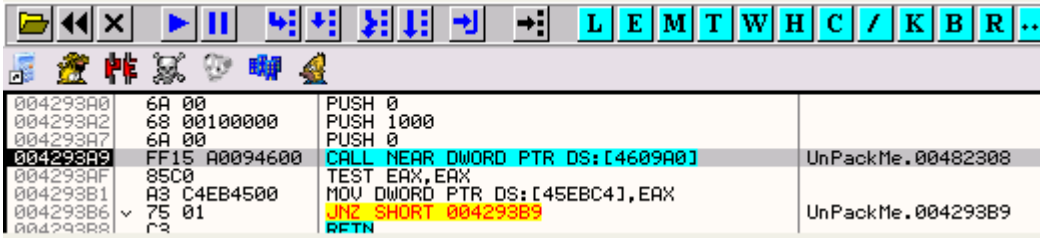
接着到了这里,我们可以看到是保存 Kernel32.dll 的基地址,我这里是 7C800000。



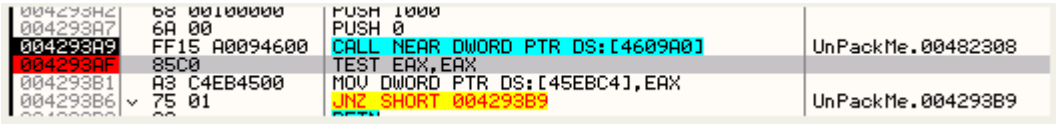
接着再次到了这里,填充 C3,此时对应 IAT 项中的值已经被修复为正常的 API 函数地址了。

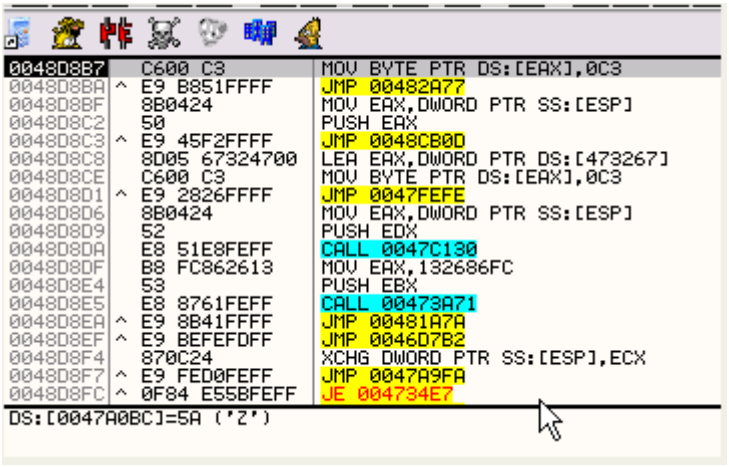


我们继续运行就断在了返回地址处,我们按 F7 单步继续往下跟踪到下一个 API 函数调用处。

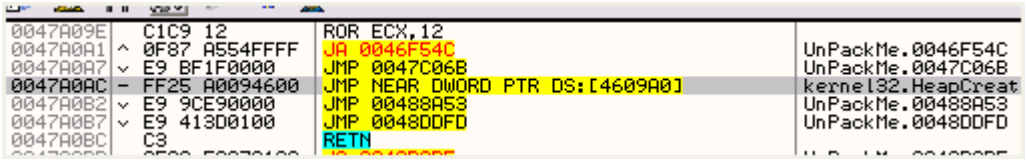


给返回地址处设置一个断点,运行起来。



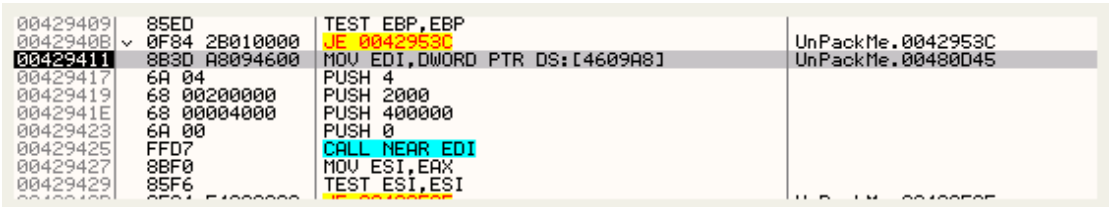


这里是将 47A0BC 处的首字节修改为 RET 指令。

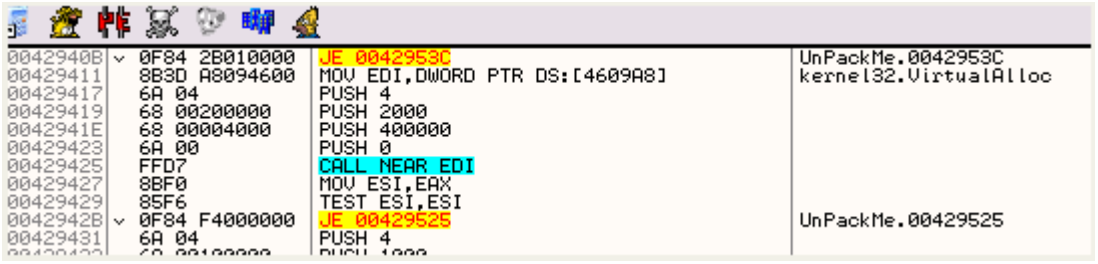


我们单步往下跟踪到 47A0BC 的 RET 指令处,可以看到对应 IAT 项中的值已经被修改为正确的 API 函数地址了,该 API 函数是 Kernel32.dll 导出的。

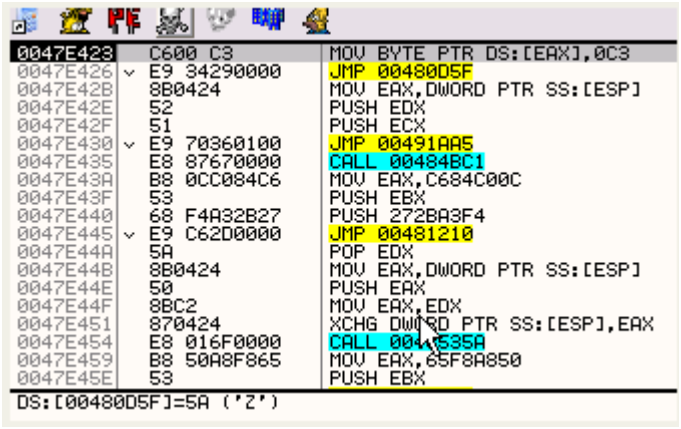
我们继续看接下来要调用的这个 API 函数。



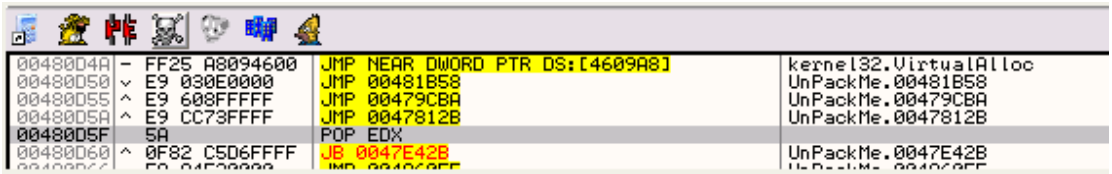
到了这里,我们对比着 UPX 的 UnPackMe 来看。



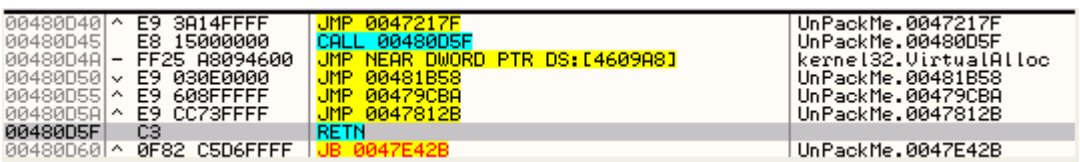
我们可以看到这里实际要调用的 API 函数是 VirtualAlloc,我们还是在返回地址处设置一个断点,运行起来。



这里又是填充另一个 RET,我们来看看修改之前的代码是什么:



被修改为 RET 之前是 POP EDX,修改为 RET 以后,就会返回到 480D4A 处去调用实际要调用的 API 函数,即 VirtualAlloc。

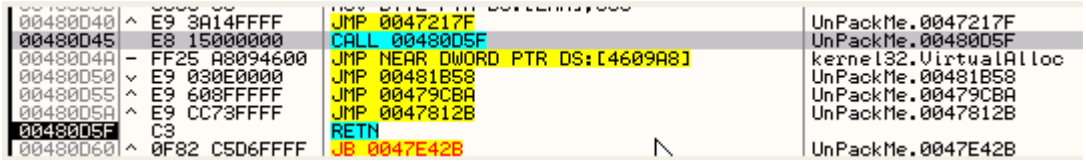


也就是说调用 API 函数的过程是,首先将对应 IAT 项中重定向的值修改为正确的 API 函数地址,然后通过自修改得到 RET 指令,

接着通过该 RET 指令返回到相应的地址处去调用实际要调用的 API 函数。

我们继续往下单步跟踪来验证一下:

可以看到这里 480D5F 处的 POP EDX 被修改为了 RET,此时我们可以注意到 4609A8 这个 IAT 项中的值已经被修复为正常的 API 函数地址了,接着执行 RET 指令,就会返回到 480D45 处,调用实际要调用的 API 函数。



好了,下面我来给大家总结一下整个过程:首先获取相应模块的基地址(可能是调用的 LoadLibraryA),然后是获取对应的 API 函数地址(可能是调用 GetProcAddress),接着将 IAT 项中重定向的值修改为正确的 API 函数地址,然后修改自身区段的代码来达到调用实际要调用 API 函数的目的。

好了,本章就到这里,下一章我们来尝试编写脚本修改 ExeCryptor 的 IAT。