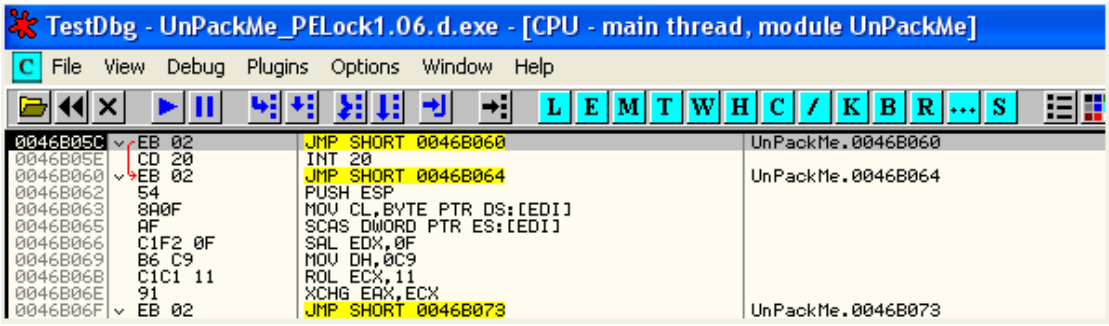
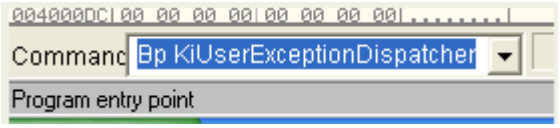
第三十九章-神马是 stolen bytes

本章与下一章节将介绍 stolen bytes(PS:壳偷代码)以及 OD 脚本编写方面的内容。我们拿 UnPackMe_PELock1.06 来讲解。这款壳拿来介绍 stolen bytes 正合适,很多经典的教程都拿它作为范例来讲解。

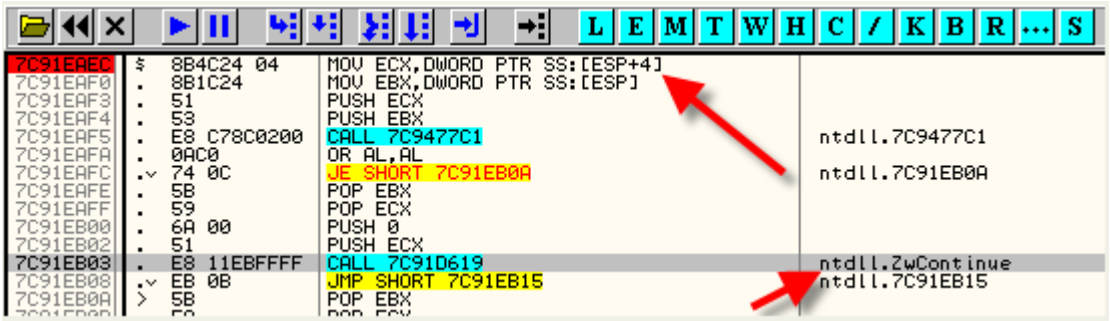
用 OD 加载它,停在了入口点处。



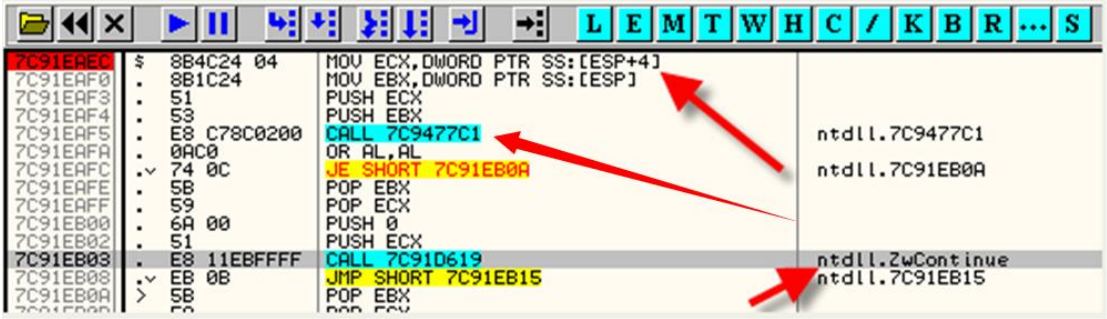
直接按 F9 键运行起来,会发现该程序会异常报错弹个错误框出来,直接 shift + F9 忽略异常继续运行,弹出主程序对话框,这里可以通过日志窗口中最后异常发生处来定位 OEP,大家可以自行尝试,我就不再赘述了,下面我们换种方法来定位最后一次异常发生处。



这里我们给 Ring3 异常分发函数 KiUserExceptionDispatcher 设置一个断点,所有的异常都会经过这一个点,我们在 OD 中定位到这个函数。



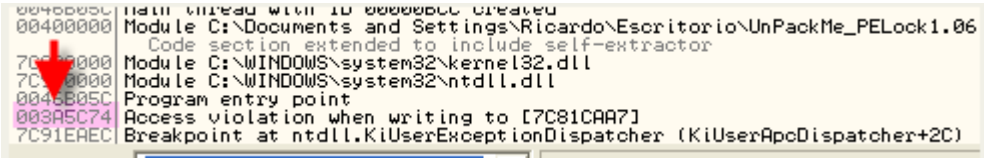
我们来分析一下这个函数。



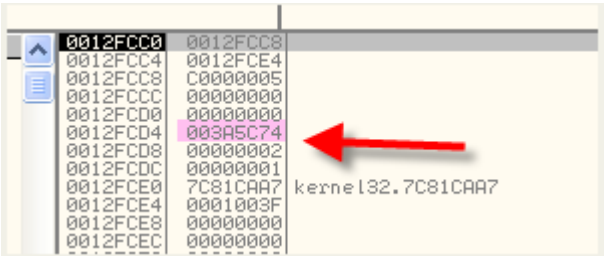
最上面的红色箭头标注的是 NTDLL.DLL 中的 KiUserExceptionDispatcher 的起始地址,紧接着下面有一个 CALL,这里是调用 SEH 链中的异常处理函数,执行完毕以后返回,根据返回的结果来决定是否继续执行程序。

这里我们就没法继续往里面跟了,因为 OD 不能跟进 RING0,这里我们只能对异常处理函数下断点或者对触发异常的区段设置内存访问断点。这样 RING0 的部分执行完了以后,OD 就断在异常产生处所在区段。

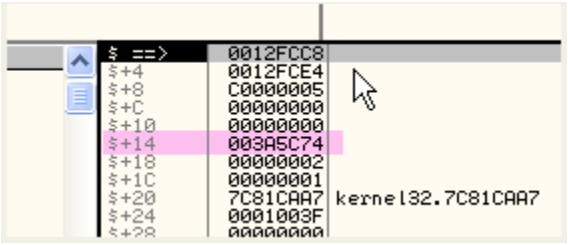
好,现在我们运行起来,第一次异常断了下来。



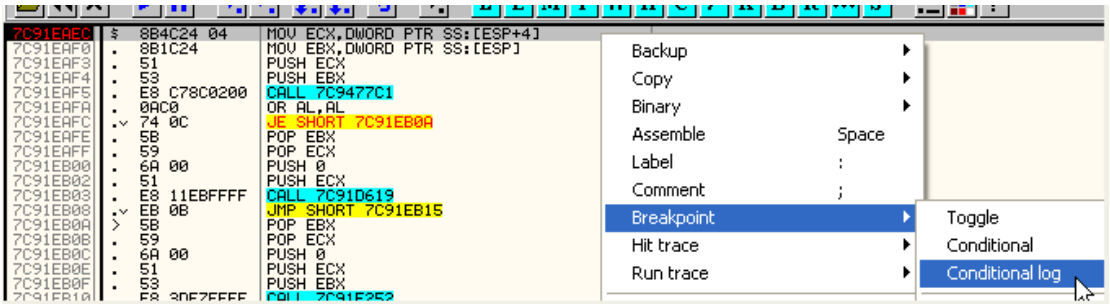
我们在日志窗口中可以看到(在我的机器上)异常发生在 3A5C74 处,好,接下来把忽略异常的调试选项都勾选上,我们给该异常发生处设置一个断点,接着我们看看堆栈窗口。



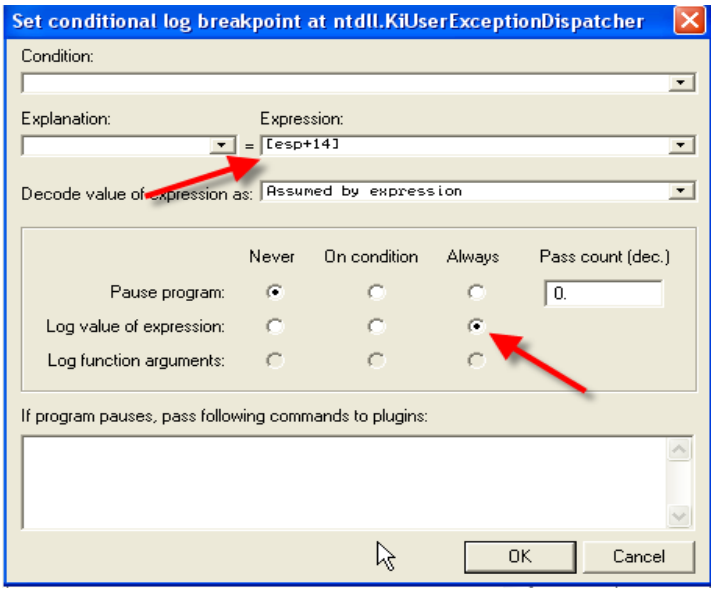
我们可以看到红色箭头标注处,也存放了异常发生的地址,确切的说是[ESP + 14]。



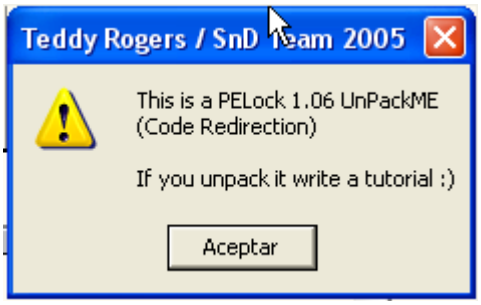
因此,这里我们用条件记录断点代替 INT 3 断点来记录[ESP + 14]的值,即记录异常触发的地址,嘿嘿。



这里我们在 KiUserExceptionDispatcher 的起始地址处单击鼠标右键选择-Breakpoint Conditional log。



这里我们将 Expression 设置为[ESP + 14](异常发生的地址),Pause program 设置为 Never,Log value of expression 设置为 Always,运行起来。



我们可以看到程序运行起来了,奇怪,这个壳居然没有检测条件断点,也没有对 KiUserExceptionDispatcher 下断进行检测。

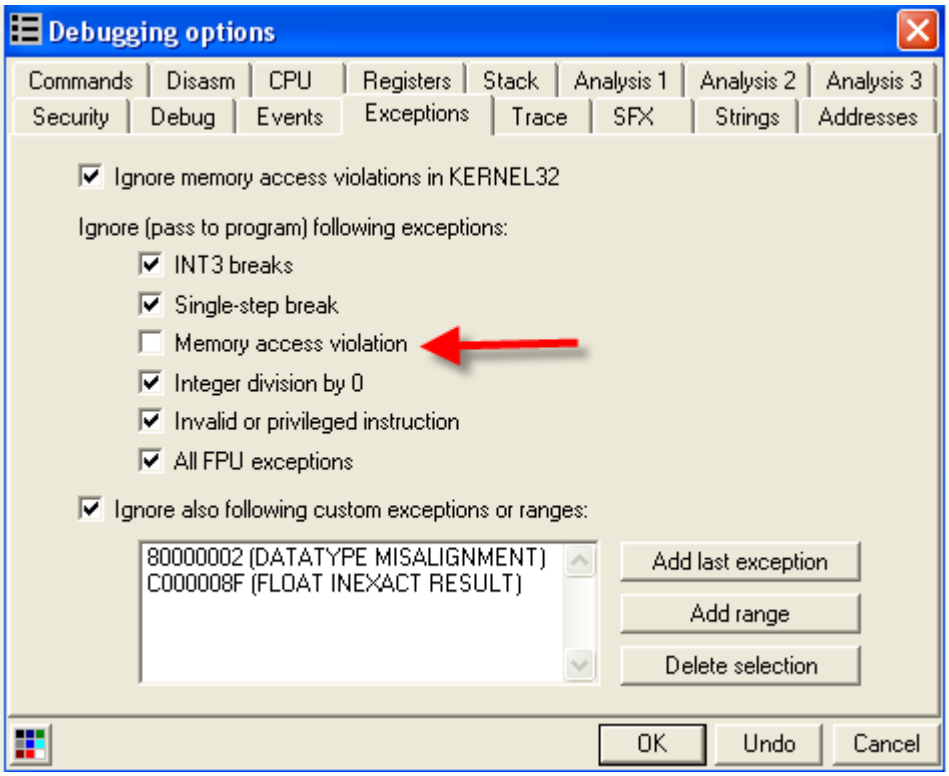
好,下面我们来看看日志窗口中记录的结果。

```
Address  | Message
0046B05C | Program entry point
003A5C74 | Access violation when writing to [7C81CAA7]
7C91EAEC | Breakpoint at ntdll.KiUserExceptionDispatcher (KiUserApcDispatcher+2C)
003A1CED | Integer division by zero
7C91EAEC | COND: 003A1CED
003A24EB | Access violation when writing to [00000000]
7C91EAEC | COND: 003A24EB
003A24EE | Integer division by zero
7C91EAEC | COND: 003A24EE
003A2698 | Access violation when writing to [00000000]
7C91EAEC | COND: 003A2698
003A269B | Integer division by zero
7C91EAEC | COND: 003A269B
003A2851 | Access violation when writing to [00000000]
7C91EAEC | COND: 003A2851
003A2854 | Integer division by zero
7C91EAEC | COND: 003A2854
003A3374 | Access violation when reading [FFFFFFFF]
7C91EAEC | COND: 003A3374
003A33CB | Illegal instruction
7C91EAEC | COND: 003A33CB
003A3374 | Access violation when reading [FFFFFFFF]
7C91EAEC | COND: 003A3374
003A33CB | Illegal instruction
7C91EAEC | COND: 003A33CB
003A3374 | Access violation when reading [FFFFFFFF]
7C91EAEC | COND: 003A3374
003A33CB | Illegal instruction
7C91EAEC | COND: 003A33CB
003A3374 | Access violation when reading [FFFFFFFF]
7C91EAEC | COND: 003A3374
003A33CB | Illegal instruction
7C91EAEC | COND: 003A33CB
003A3652 | Access violation when writing to [00000000]
7C91EAEC | COND: 003A3652
003A3655 | Integer division by zero
7C91EAEC | COND: 003A3655
003A37DE | Access violation when writing to [00000000]
7C91EAEC | COND: 003A37DE
003A37E1 | Integer division by zero
7C91EAEC | COND: 003A37E1
003A3873 | Access violation when writing to [00000000]
7C91EAEC | COND: 003A3873
003A3876 | Integer division by zero
7C91EAEC | COND: 003A3876
76B00000 | Module C:\WINDOWS\system32\WINMM.dll
77D10000 | Module C:\WINDOWS\system32\USER32.dll
```

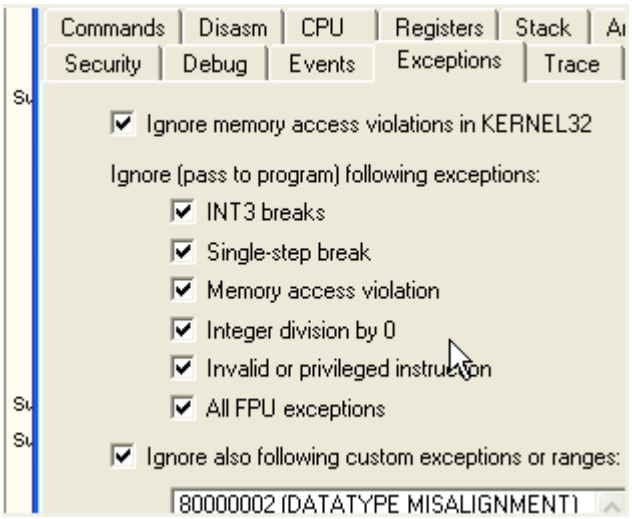
我们可以看到记录了很多异常触发的地址,我们往下看。

```
774B0000 | Module C:\WINDOWS\system32\ole32.dll
7C81084E | Breakpoint at kernel32.7C81084E
003A6744 | Access violation when writing to [00000000]
7C91EAEC | COND: 003A6744
7C810856 | New thread with ID 00000110 created
770F0000 | Module C:\WINDOWS\system32\OLEAUT32.dll
5B150000 | Module C:\WINDOWS\system32\uxtheme.dll
746B0000 | Module C:\WINDOWS\system32\MSCTF.dll
```

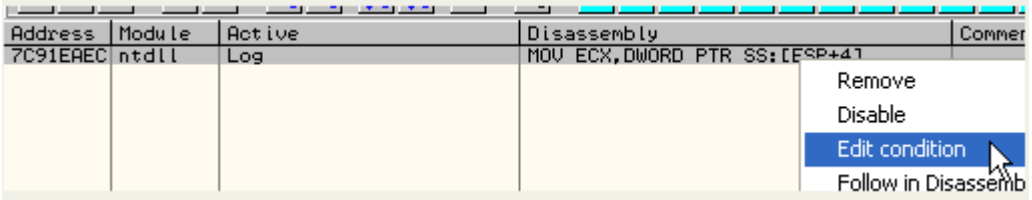
我们可以看到最后一个异常发生在 3A6744 处,是非法访问异常,这里我们不能对该处设置 INT 3 断点,因为该壳会有检测导致程序无法运行。

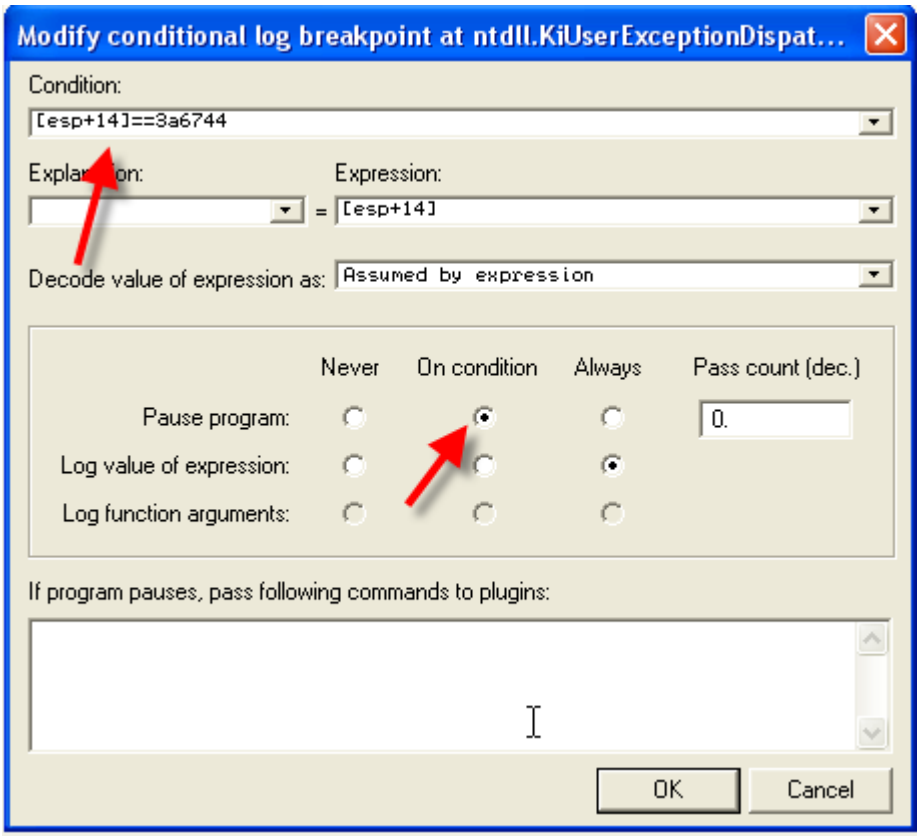


这里我们可以将 Memory access violation 这一项对勾去掉,当产生内存访问异常的时候就会断下来,虽然这种方法可以奏效,都是只要是内存访问异常就会断下来,会断很多次,其实我们还有更快速定位的方法,这里我们还是将 Memory access violation 这一项勾选上。



我们还是来设置条件断点,我们打开断点列表,在断点上单击鼠标右键选择-Edit condition。

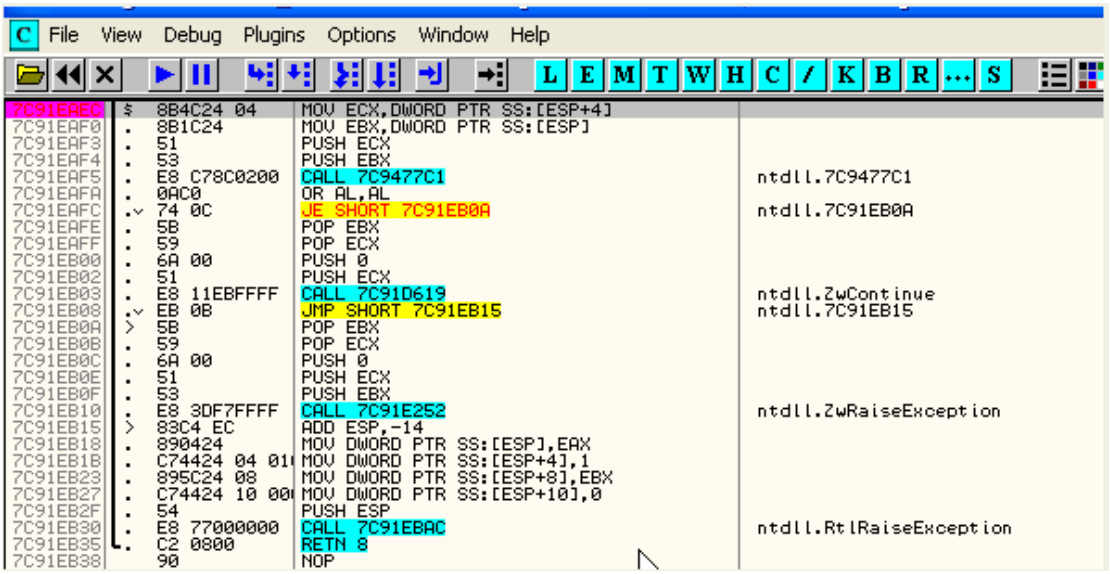




这里根据日志信息中显示的最后一次异常发生在 3A7644 地址处,我们只需添加一个条件就能让其断在最后一次异常处,我们设置条件为[ESP + 14] == 3A7644,这样当最后异常触发时就会断下来。

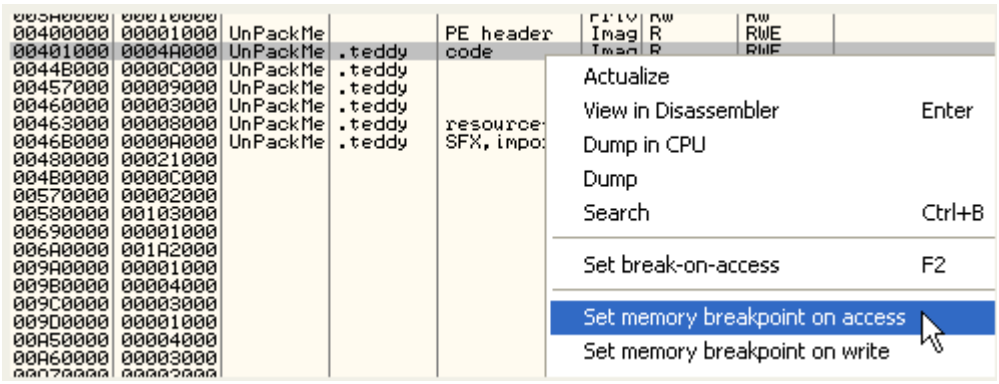
```
774B0000 Module C:\WINDOWS\system32\ole32.dll
7C81084E Breakpoint at kernel32.7C81084E
003A6744 Access violation when writing to [00000000]
7C91EAEC COND: 003A6744
7C810856 New thread with ID 00000110 created
770F0000 Module C:\WINDOWS\system32\OLEAUT32.dll
5B150000 Module C:\WINDOWS\system32\uxtheme.dll
746B0000 Module C:\WINDOWS\system32\MSCTF.dll
```

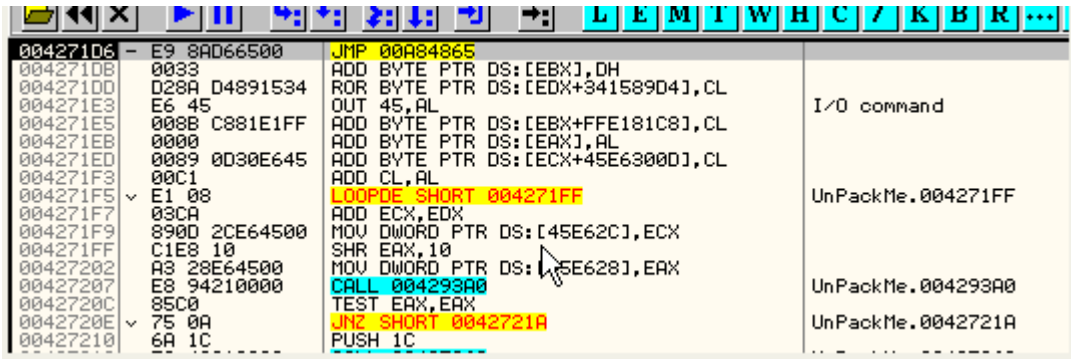
接着将 Pause program 这一项设置为 On condition,这样当条件满足时就会断下来,我们运行起来。



这里,就断到了最后一次异常处,我们没有必要一个异常一个异常的去定位。

现在如果我们对第一个区段设置内存访问断点就能定位到 OEP(真的是这样吗?嘿嘿,我们一起来看看)





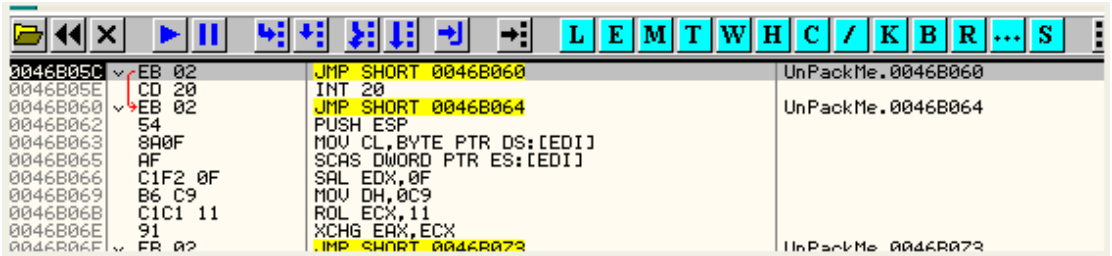
这里我们可以看到断在了第一个区段中,正常来说,这里应该就是 OEP 了呀,但是看这些代码怎么不像 OEP 呀,这是怎么回事呢?这里就要给大家介绍 stolen bytes 了。stolen bytes:即某些壳在处理 OEP 代码的时候,会把 OEP 处固定的代码 NOP 掉,然后把这些代码放到壳代码的空间中去(而且常伴随着花指令)!使原程序的起始代码从壳空间开始执行,然后再 JMP 会原程序空间。如果我们脱掉壳,这一部分代码就会遗失,也就达到了反脱壳的目的,这就是 stolen code(或者 stolen bytes)技术,或者更确切的说是 stolen OEP code 技术。

stolen bytes 了以后会怎么样呢?很简单,如果我们 dump 以后,修复 IAT 的时候,这里 OEP 就会被指定为 4271D6,但是这样的话程序就无法运行了,因为前面几行代码还没有执行,这几行代码在壳空间中,所以是不会被转储的,因此也得不到执行。

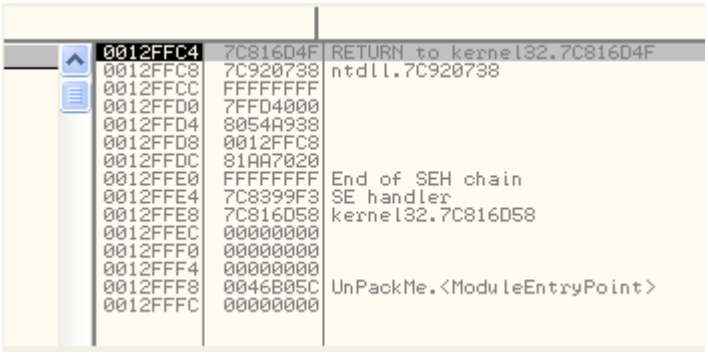
那我们该怎么办呢?

一种方式就是尝试跟踪壳的代码,当所有异常都触发以后我们就会到达错误 OEP 处,我们将跟踪过程中执行过的代码都保存到一个 txt 中,便于我们的分析,这里我们在 JMP 到错误的 OEP 4271D6 之前都执行些什么呢?

下面我们重启程序,看看哪些可能是 stolen bytes。

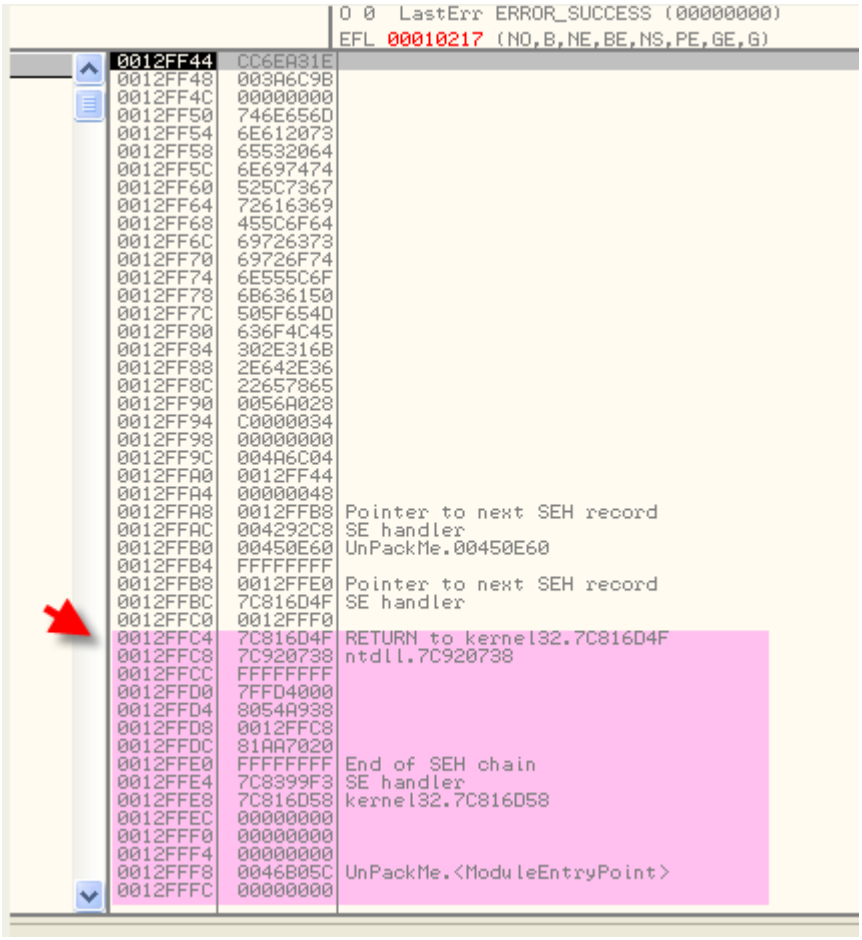


我们看下栈顶:

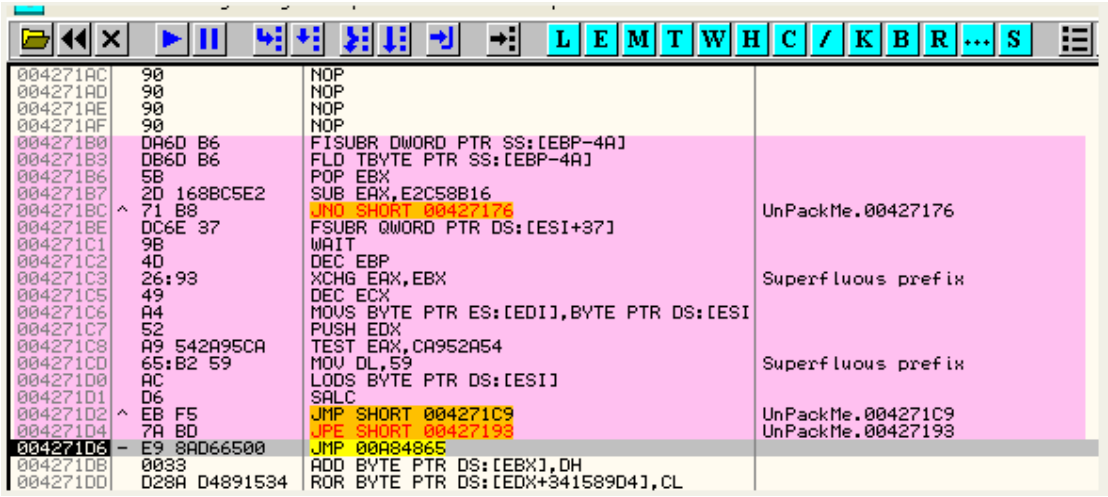


我的机器,当前指针为 12FFC4,大家有可能是其他的值,当到达真正的 OEP 处时,一般来说,栈顶指针应该也会指向 12FFC4 或者附近的地址,但是如果是假的 OEP 呢?

我们再次定位到假的 OEP 处。

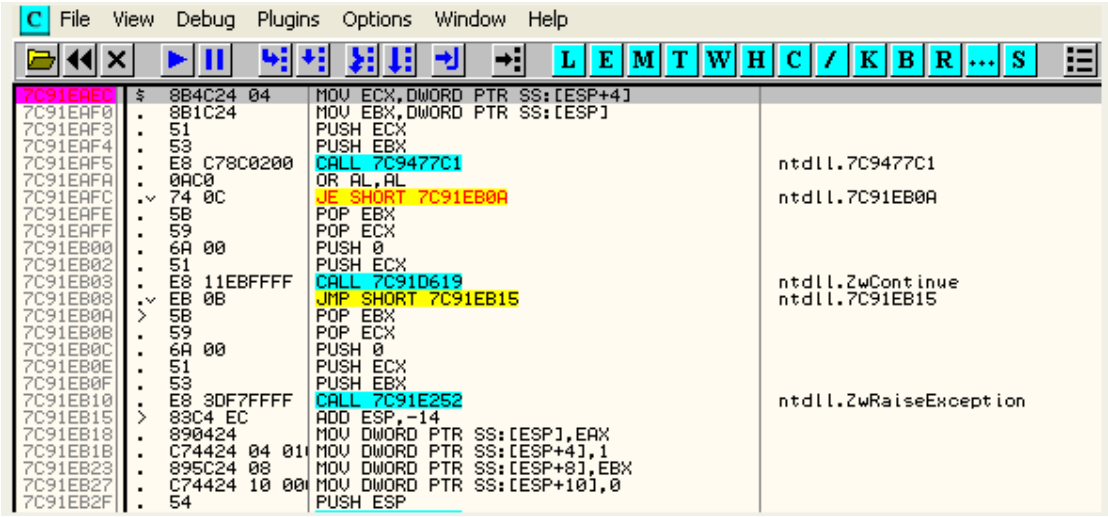


如果是到达真正的 OEP 处时,栈顶应该是 12FFC4 或者附近的地方,但是该壳把 OEP 处前几行的代码清空掉了,并填入了垃圾数据,并且把这前几行代码放到壳代码的空间中去!接着在壳空间执行原程序的前几条指令,然后再 JMP 到原程序的假 OEP 处。大家应该还记得该 crackme 的 OEP 是 4271B0,我们来看看该地址处是什么。

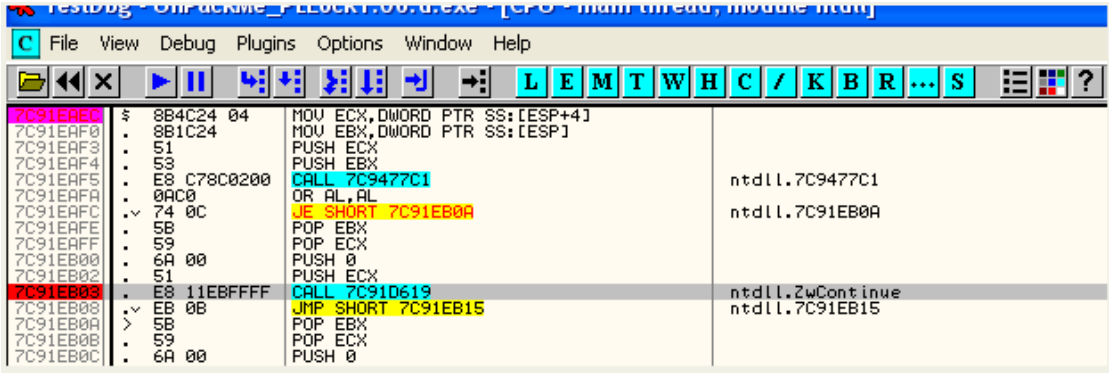


这里我们可以看到,壳将 OEP 处的原始字节都填充为了垃圾指令,我们对该区段设置内存访问断点会断在 4271D6 处,不会断在前面。

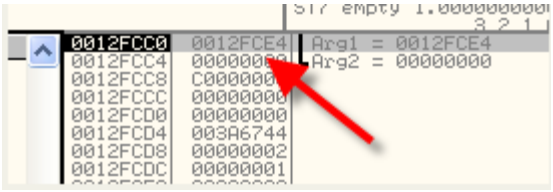
有很多方法可能定位 stolen bytes,但最为经典,最为常用的方法还是在最后一次异常产生后单步跟踪,我们现在还是跟到最后一次异常处。



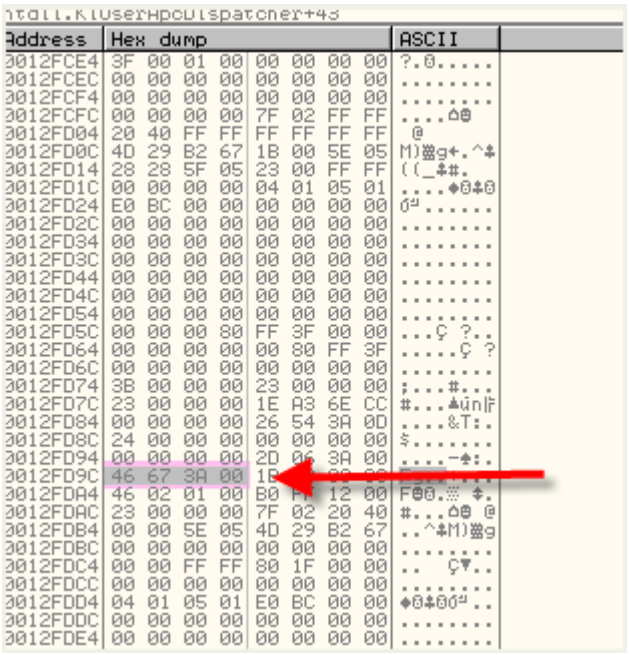
我们知道最后一次异常是在 3A6744 地址处产生的,这里我们对异常处理程序并不感兴趣,我们直接在其下面 7C91EB03 地址处下断。



运行起来,接下来程序将返回到哪里呢?我们要对哪里下断?一起来看看堆栈。



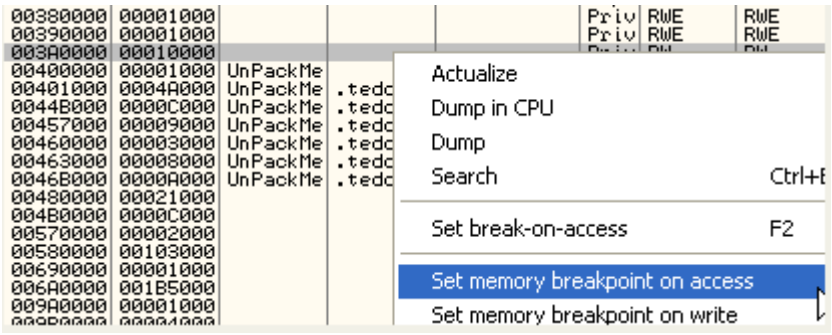
这里第一个参数是 CONTEXT 结构体的指针,我们在数据窗口中定位到该地址。



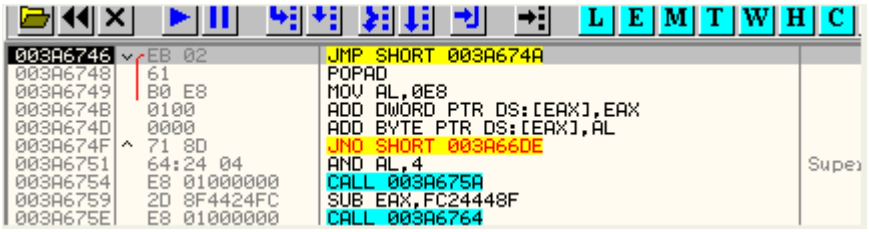
这里如果大家熟悉 CONTEXT 结构的话,就会知道其中有个字段标识的是 EIP 寄存器,其他一些字段标识的是其他一些寄存器,当异常处理完毕以后会根据 CONTEXT 结构中的 EIP 值来决定返回到哪里,后面章节我们会详细这个 CONTEXT 结构,现在我们只需要知道程序返回到哪里,EIP 字段位于偏移 B8 处(PS:可以参考 VC 中 CONTEXT 结构体的定位)。

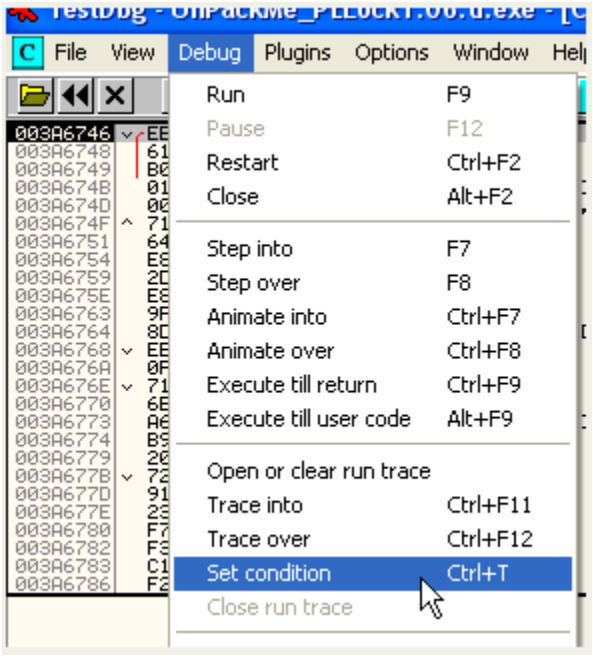
```
typedef struct _CONTEXT {
    DWORD ContextFlags;
    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD   SegGs;
    DWORD   SegFs;
    DWORD   SegEs;
    DWORD   SegDs;
    DWORD   Edi;
    DWORD   Esi;
    DWORD   Ebx;
    DWORD   Edx;
    DWORD   Ecx;
    DWORD   Eax;
    DWORD   Ebp;
    DWORD   Eip;                    // +0xB8
    DWORD   SegCs;
    DWORD   EFlags;
    DWORD   Esp;
    DWORD   SegSs;
    BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];

} CONTEXT;
```

因此,程序将返回到 3A6746 地址处。我们给该地址所在区段设置一个内存访问断点。

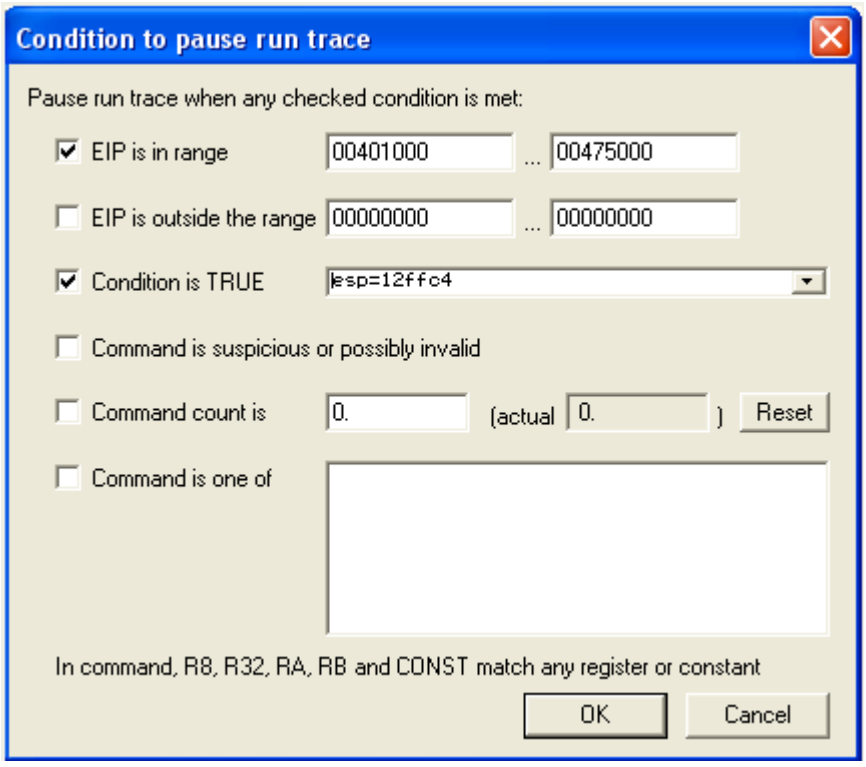


运行起来,我们可以看到断在了这里。

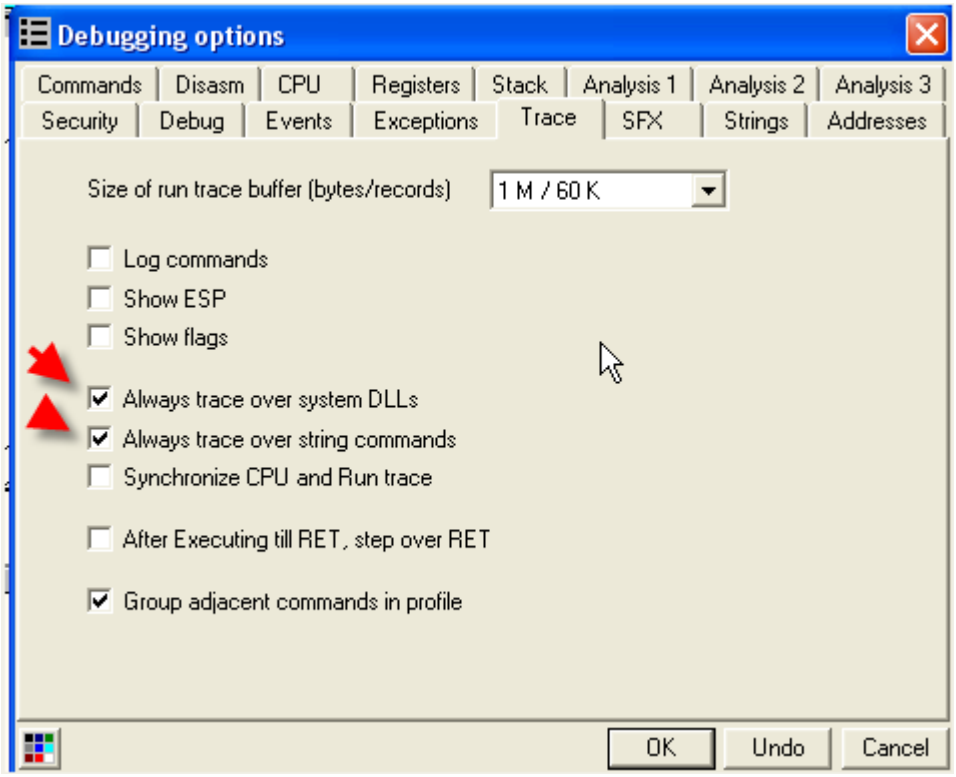




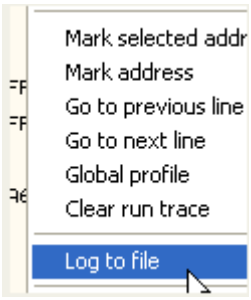
接下来我们来设置自动跟踪的条件,有好几个选项可供设置,比如说我们可以定位恢复寄存器环境指令,当恢复寄存器环境后,接下来就要执行 OEP 处的代码了,当前还可以设置其他条件,比如说,我们设置如下两项。



EIP is Range 这一项我们设置为 401000~475000,即原程序的所有区段,这样我们就能在 Run Trace 窗口中看到跟踪过程中执行了哪些指令,下面 Condition is TRUE 这一项我们设置 ESP = 12FFC4,即跟踪到栈顶指针跟假的 OEP 一致时,停止跟踪。

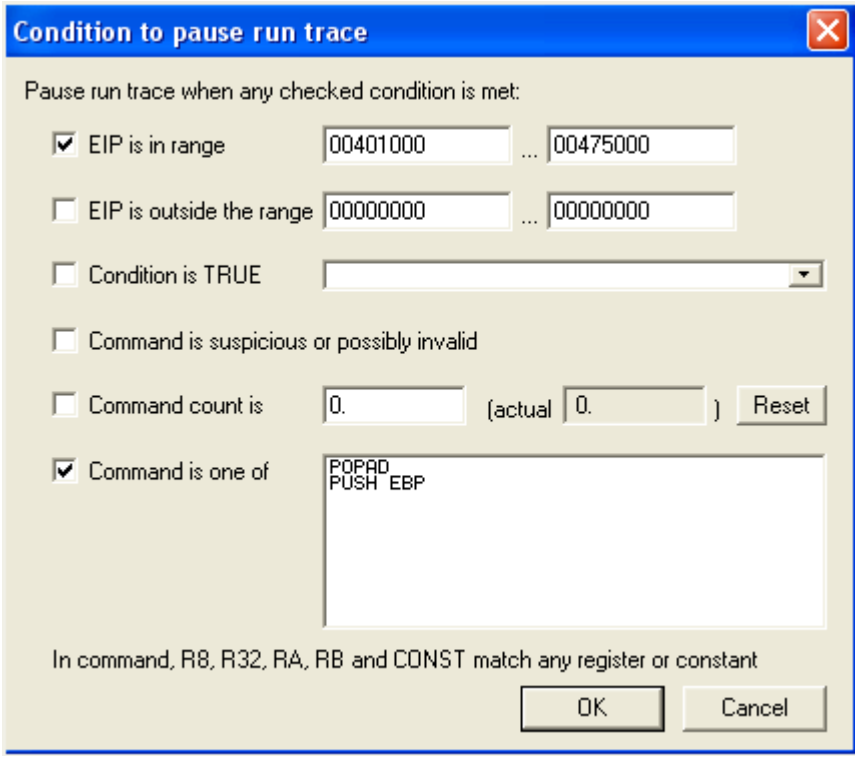


注意,这里调试选项中的 Trace 选项卡中的这两项我们也可以尝试勾选上,看看自动跟踪的效果如何,如果效果不好,我们再来去掉这两个对勾。

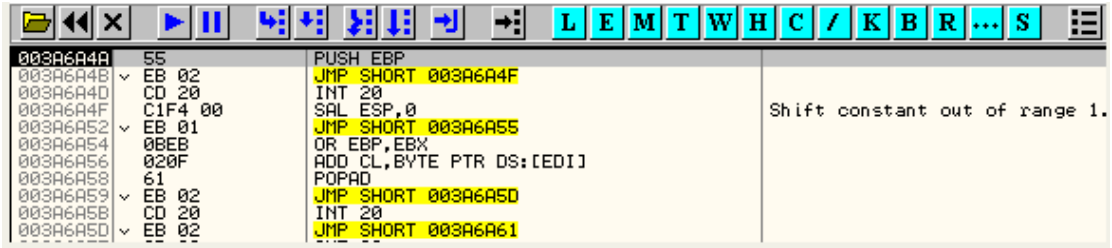


跟踪完毕以后,我们可以单击鼠标右键选择-Log to file(将日志信息保存到文件),再来进一步分析。

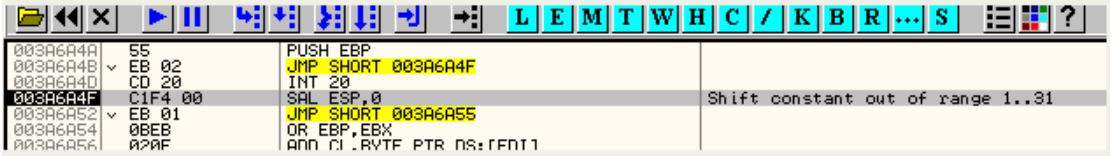
我们选择主菜单中的 Debug-Trace into,断了下来,但是并没有到达假的 OEP 处,我们继续 Trace into 多次以后才会到达假 OEP 处,Run Trace 窗口中记录了自动跟踪所执行的指令,我们依然看出来哪里是真正的 OEP 所在。



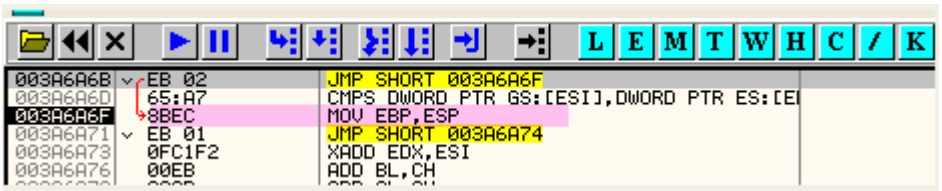
既然不奏效,那我们来换个条件试试,我们设置 Command is one of 这一项,当自动跟踪过程中遇到设置的命令序列中的任意一个就会停止跟踪,这里,我们填入两条命令,POPAD 和 PUSH EBP,即当遇到恢复寄存器环境或者 OEP 处常见的第一条指令时停止自动跟踪,我们来看看能不能奏效,单击主菜单中的 Debug-Trace into。



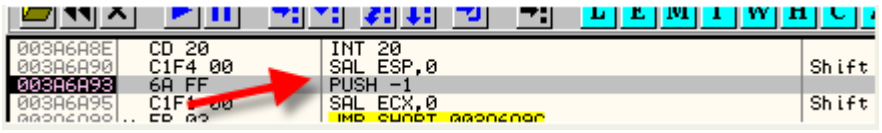
这里有可能是真正的 OEP,我们往下跟。



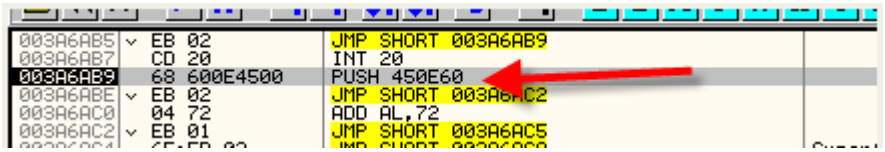
这里很明显是垃圾指令,紧接着下面是一个无条件跳转,没有实际作用,相当于花指令。继续往下跟。

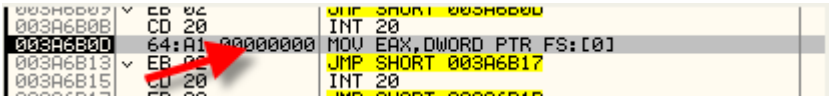


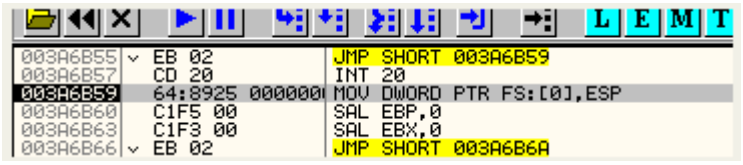
这里应该是缺失的第二条指令。

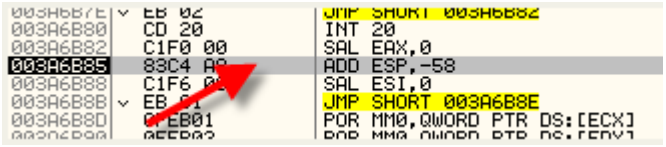


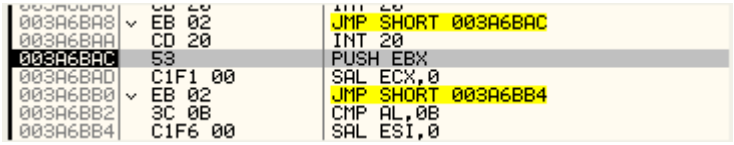
我们继续往下跟,这里应该是一条正常的指令。

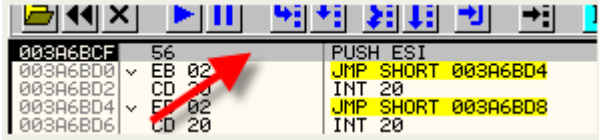


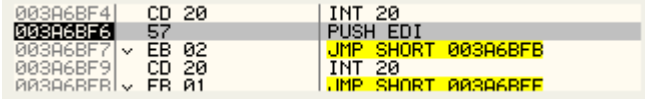


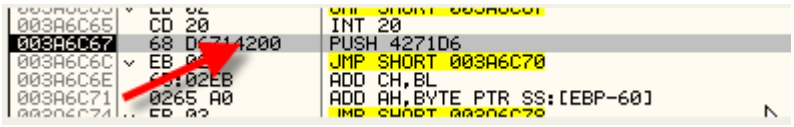




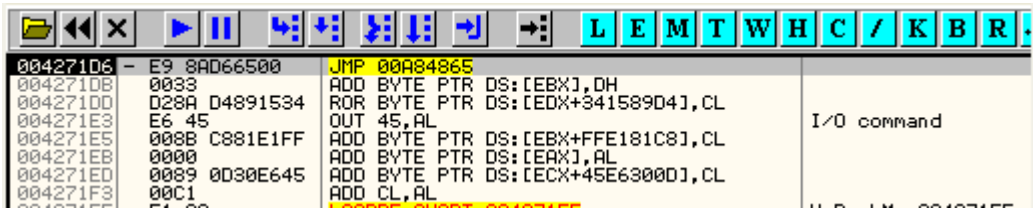




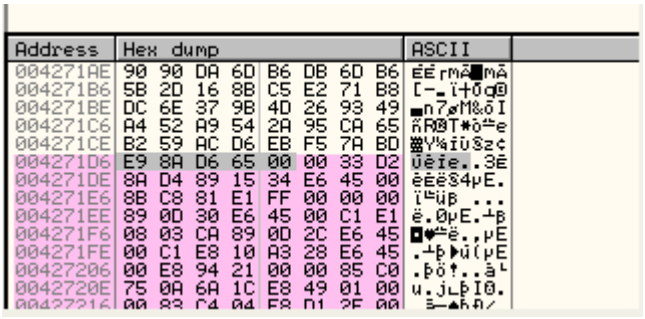




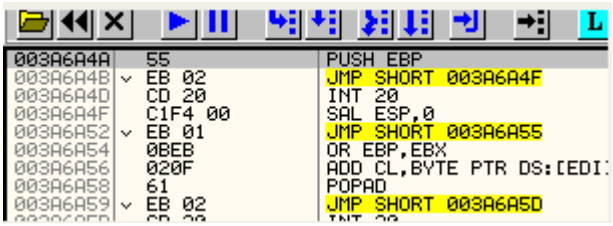
接下来我们就会到达假的 OEP 4271D6 处,因为这里是 PUSH 4271D6,接着通过 RET 就可以返回到假的 OEP 处。



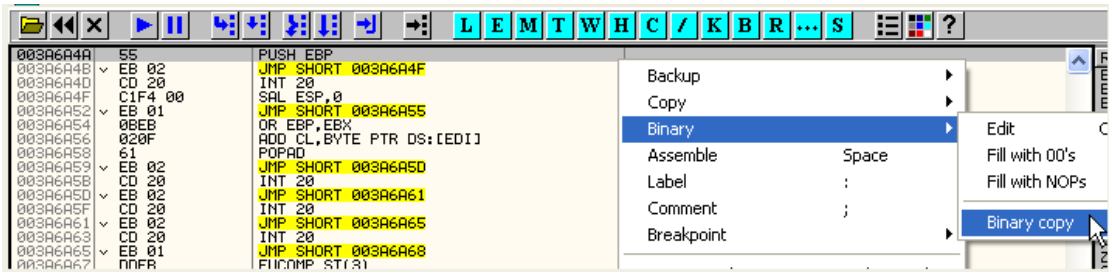
接下来我们将可以将 stolen bytes 拷贝出来,我们首先在数据窗口中看看假 OEP 之前的内存单元的情况。

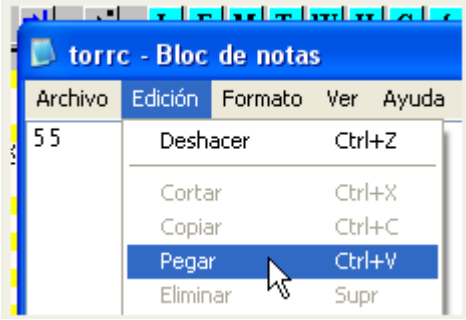


我们需要将 stolen bytes 填充到假 OEP 前面的内存单元中,好,现在我们按减号键返回到刚刚真正 OEP 的第一条指令处。



我们依次将缺失的指令所对应的字节码拷贝到记事本中,最后一个 PUSH 指令和 RET 指令不需要拷贝,这两条指令时用来跳转到假 OEP 处的,并不是 stolen bytes。



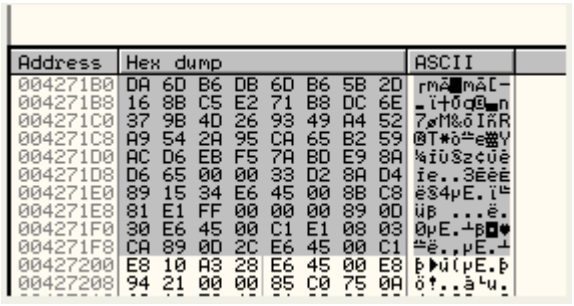


55 8B EC 6A FF 68 60 0E 45 00 68 C8 92 42 00 64 A1 00 00 00 00 50 64 89 25 00 00 00 00 83 C4 A8 53 56 57 89 65 E8

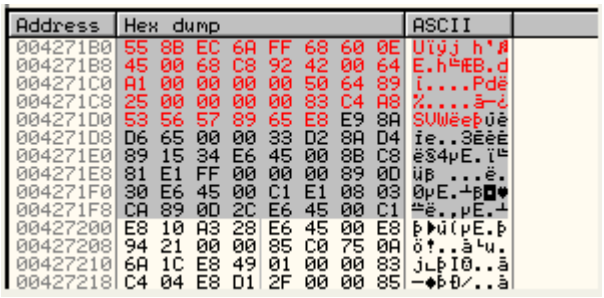
以上是 38 个 stolen bytes,也即是 16 进制的 26。

所以应该把假 OEP 往下抬高 26(十六进制)个字节。

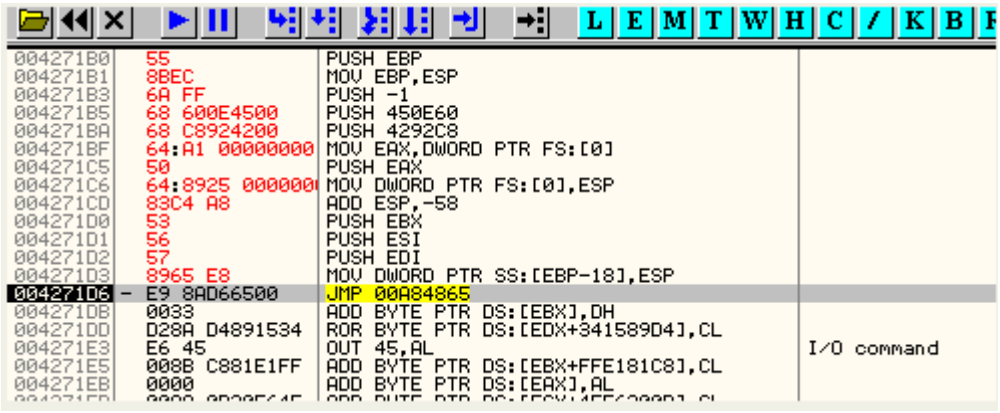
即如果壳没有抽取 OEP 处的代码的话,真正的 OEP 应该是 4271B0,我们需要将被抽取的代码填充到 4271B0 处。



全选记事本中的字节拷贝出来,并在 4271B0 为起始地址,长度为 26(16 进制)的区域上面单击鼠标右键选择 Binary paste(二进制粘贴)。



我们来看看拷贝的指令。



好了,stolen bytes 被找回来了,下面我们进行 dump 的时候需要将 OEP 修改为 4271B0,这样我们就解决了 stolen bytes 的问题。

下一章节我们来讨论如何编写脚本修改该程序的 IAT。