实验：二进制程序补丁实验——修改软件时间限制

1. 实验概述

在“ESP定律脱壳实验”中，脱壳后的DiskProbe_dump_SCY.exe程序就可以进入下一步的分析工作，为了模拟恶意代码分析训练，本实验要求使用动态分析的方法，分析已经脱壳后的DiskProbe_dump_SCY.exe，找到其时间限制的代码，并进行代码修补，最终得到一个破解时间限制的DiskProbe软件。

1.1 实验软件

- x32dbg 【http://172.17.200.225/gyf/Malware/x96dbg.zip】
- Ida Pro （软件在教学电脑上已经安装）【http://172.17.200.225/gyf/Malware/IDAPro.zip】
- DiskProbe.exe 【http://172.17.200.225/gyf/Malware/DiskProbe.exe】

1.2 实验目标

- 掌握动态分析工具x32dbg的使用
- 掌握静态分析工具ida pro的使用
- 了解Windows应用程序文件逆向操作流程

2. 实验原理及方法

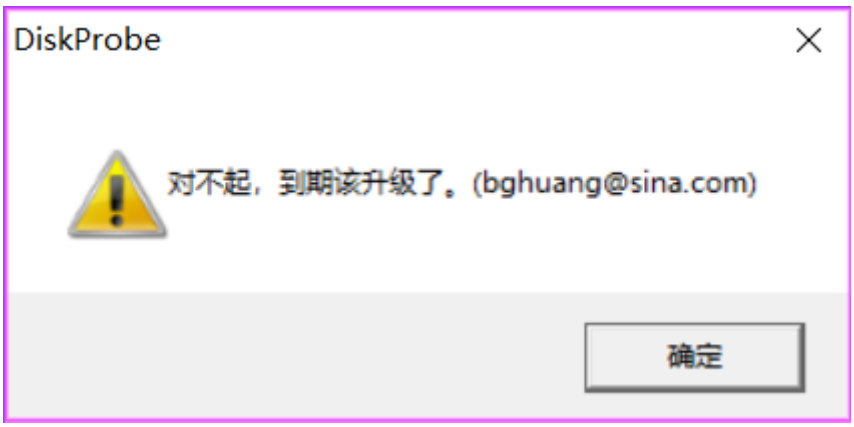

图1 DiskProbe提示框

实验思路：

DiskPro软件在启动后会显示更新提示对话框，如图1，表明程序在启动后会自动版本检测。如果能够在代码中定位检测代码的位置，就可以实现对这个时间限制的修改。

版本的检测有多种方式，比较简单的方式是检测发布时间和用户当前电脑时间的差值，当时间差超过特定值就认为版本过期。这是早期的一些软件常用的检测方法，因为其实现简单，现在还有很多软件采用这种方式。这里首先假设是采用的这种时间检测法，然后展开对应的分析和破解流程。（实际分析后发现diskprobe确实使用了简单的时间检测）。

因为时间检测法必须要获得当前系统的时间，所以必定会调用读取当前系统时间的API函数，通过查阅Window API，常见的获取系统时间的API是GetSystemTime函数。以下是函数原型。

```
void GetSystemTime(
  [out] LPSYSTEMTIME lpSystemTime
);
```

按照这个假设，使用动态调试的方式，在调试器添加API函数断点，然后启动调试，调试代码在访问的API函数的地址时，断点会被触发，此时就能找到函数被调用的代码位置。

注意：

实验软件提供的diskpro.exe是未脱壳版本软件，请按照"使用ESP定律脱壳实验"中的方法首先将软件脱壳，得到脱壳后的软件，DiskProbe_dump_SCY.exe，再进行以下实验。

以下分别使用x32dbg和Ida Pro两款软件进行实验。

2.1 使用x32dbg动态分析实验

2.1.1 添加断点

- 使用x32dbg打开脱壳后的软件 DiskProbe_dump_SCY.exe

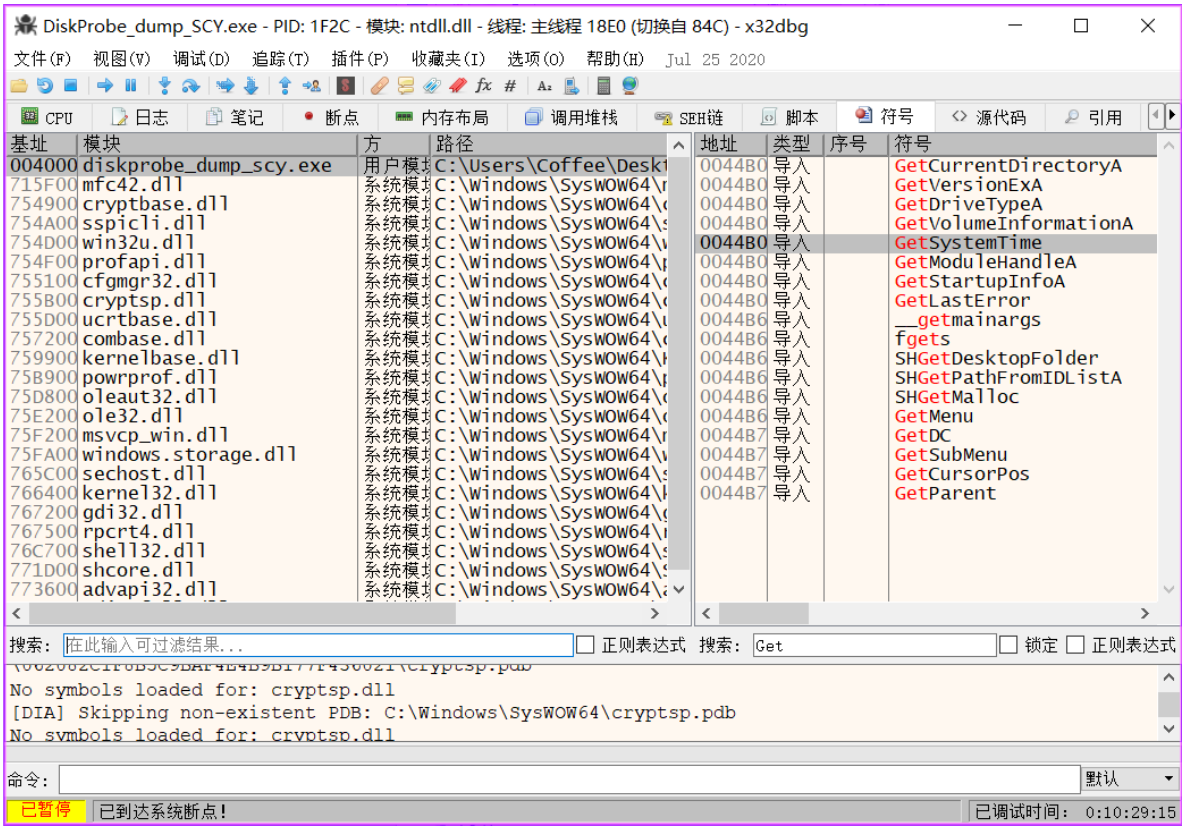

图2 查找GetSystemTime API

- 在x32dbg中切换到"符号"标签页面，并选择DiskProbe_dump_SCY.exe模块，然后在右侧搜索中输入`GetSystemTime`。如图2中，输入"Get"字符后就已经筛出所需的API函数了，不一定需要输入全部的函数名称。
- 单击右键GetSystemTime，然后单击"切换断点"菜单项添加断点（选择GetSystemTime项后，使用`F2`快捷键也可以添加断点）。添加断点后，可以点击"断点"标签页查看是否添加成功。

> 以上查找GetSystemTime的过程可以直接使用命令而不需要进行前面两部的操作。对于熟练使用了x32dbg，推荐使用命令添加断点。
>
> ```
> # 在调试器底端的命令输入框输入命令添加访问GetSystemTime函数的断点
> bp GetSystemTime
> ```

- 断点添加完毕后，可以打开“断点”子页面查看是否正确添加断点，另外开启调试器后会自动添加一些断点，可以把这些断点删除。

2.1.2 分析断点代码

- 在上一小节中正确添加断点后，可以使用 `F9` 快捷键启动调试。

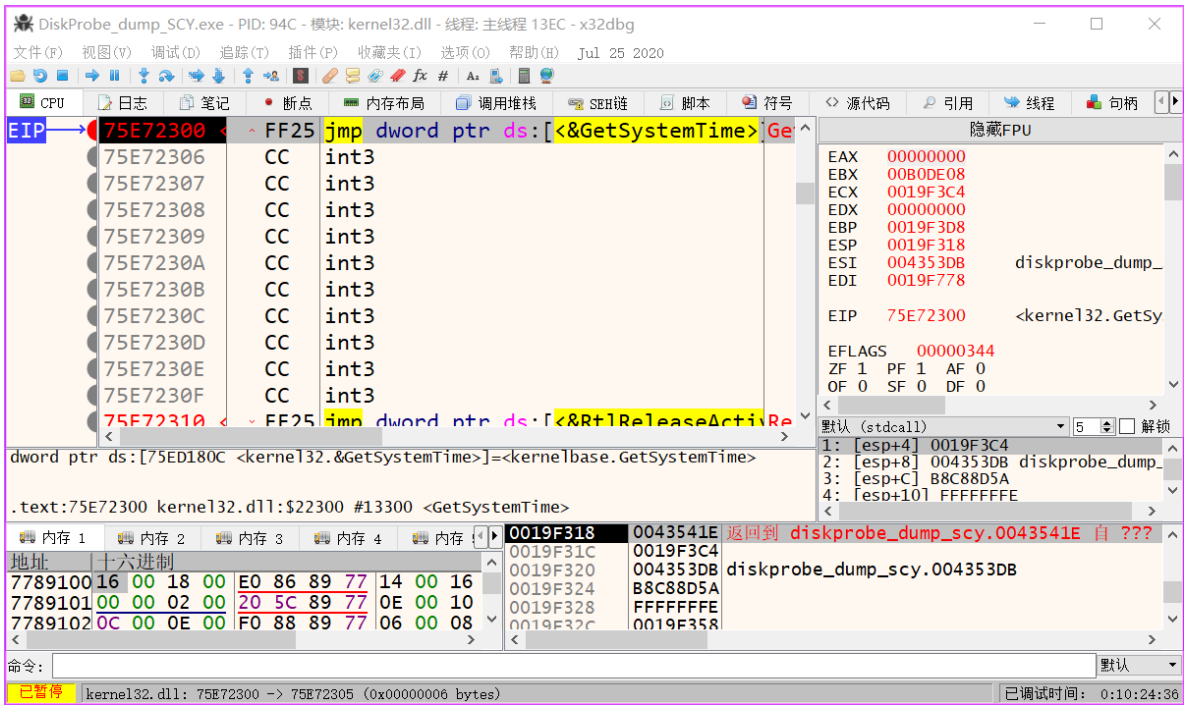

图3 GetSystemTime断点

在图3中EIP指向地址为75E72300，为系统动态链接库的载入地址，此时调试器中断在GetSystemTime函数内部，但实验需要分析得到的是调用GetSystemTime函数的**用户代码**的位置，也就是调用GetSystemTime这个函数的代码段。找到位置可以有以下两种方法，同学们可以都尝试一下。

（1）利用函数调用栈帧的内存结构布局，可以查看当前栈顶数据，这个数据地址就是调用GetSystemTime函数后面下一条指令地址，就是实验所需要定位的用户代码段。通过**右键单击栈顶地址**，在弹出的菜单中选择“在反汇编中转到指定DWORD”，如图4，就能进入GetSystemTime函数被调用后下一条指令的地址。

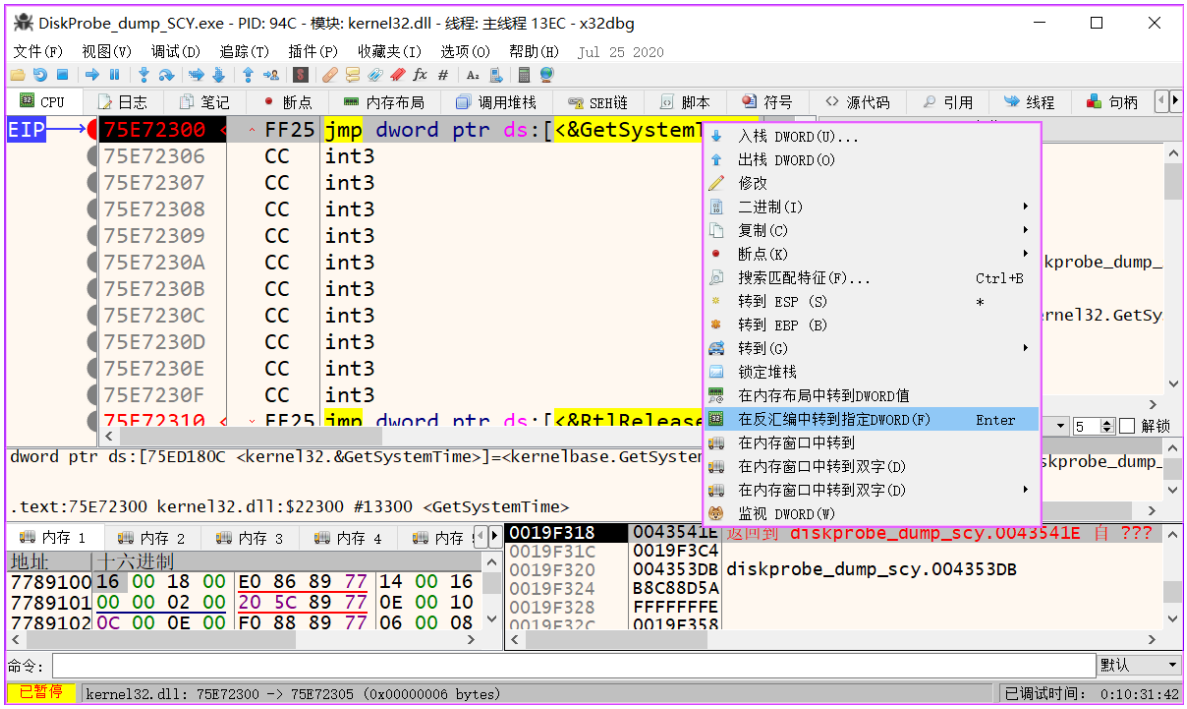

图4 跳转到函数调用位置

（2）利用调试器的*运行到用户代码*功能，执行代码到GetSystemTime函数后的指令位置。事实上调试器中断的位置是GetSystemTime函数开头代码，但GetSystemTime是系统函数代码，属于系统地址空间，这时使用快捷键`Alt+F9`（*运行到用户代码*功能）就能快速执行GetSystemTime函数代码，暂停在用户代码位置，即GetSystemTime调用后的位置。

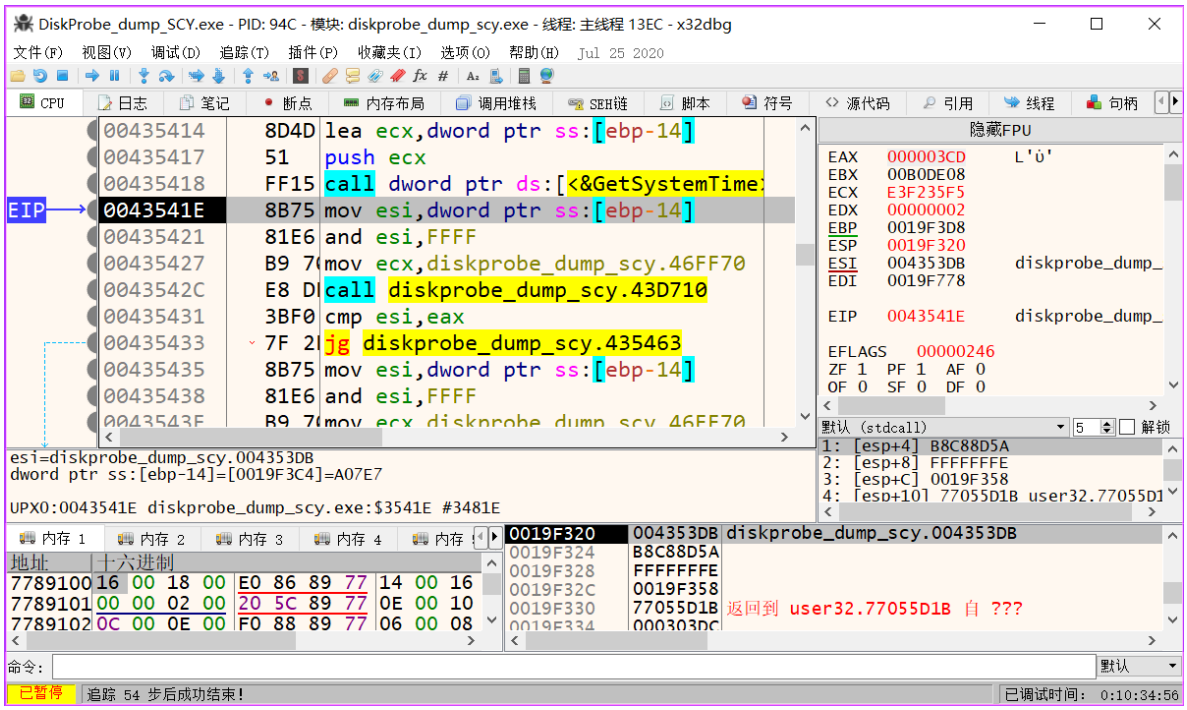

图5 GetSystemTime调用代码位置

使用以上两种方法都能正确的导航到GetSystemTime被调用的位置下一条指令，滚动上翻几行指令，就能看到如图5所示的完整调用代码。

- 代码功能分析

找到调用函数位置后，需要阅读调用后的相关代码，按照推测思路，查找对时间比较的代码。

在汇编指令代码中，常见的比较都会使用`cmp`指令进行比较操作，在比较操作后，会使用各类**条件跳转指令**进行分支操作，常见的**条件跳转指令**有`jg,jle,jb,je,jz,.....`

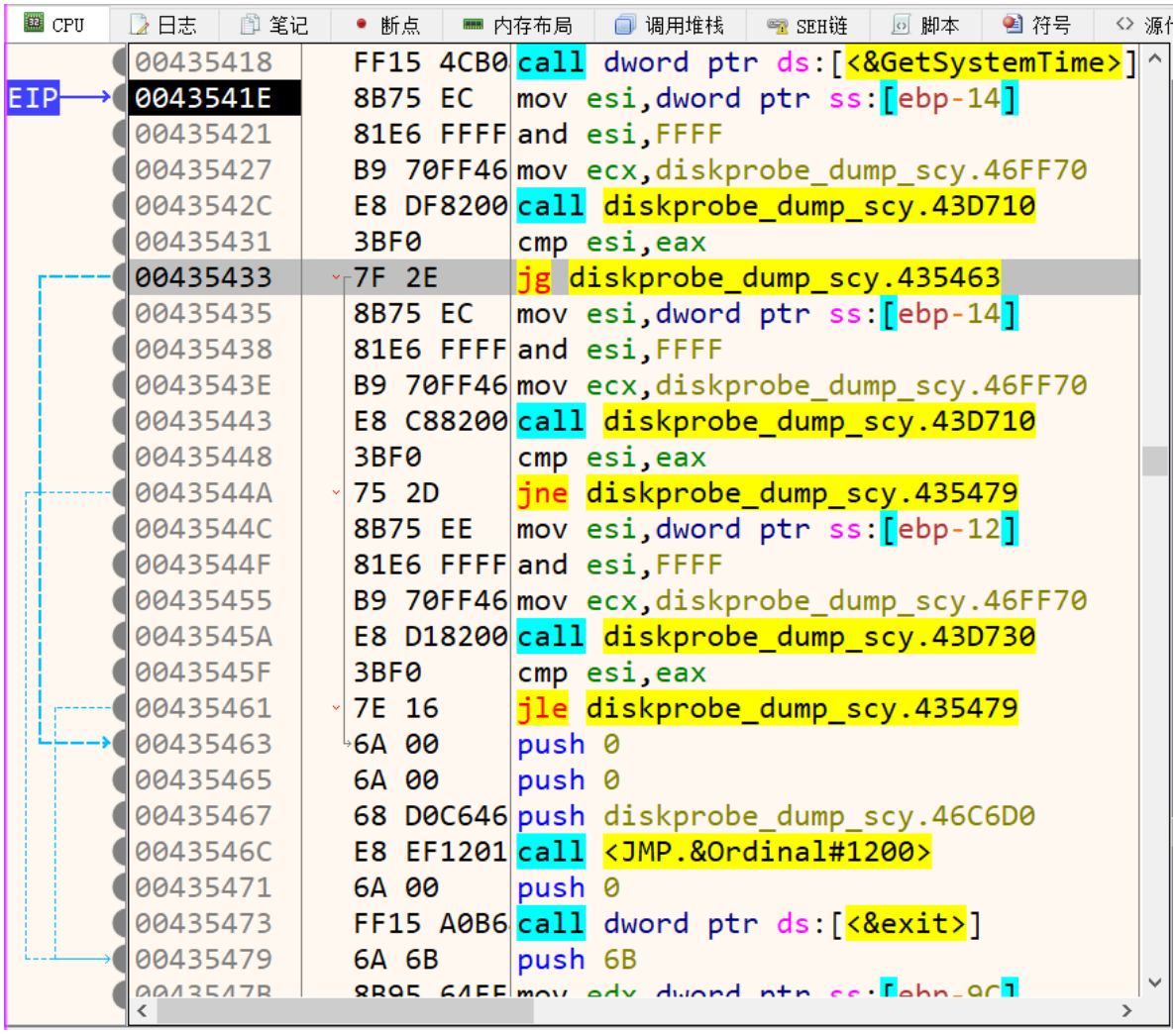

图6 时间比较指令

查看在GetSystemTime函数调用后代码段，如图6所示。其中有一条指令：

```
00435471  | 6A 00       | push 0                          |
00435473  | FF15 A0B644 | call dword ptr ds:[<&exit>]     |  <= 退出程序
```

这与软件在运行后，弹出升级对话框后，点击确定就退出程序的行为一致。所以主要分析GetSystemTime和这条指令之间的代码。

地址0x00435431是在获取到当前系统时间后的一次比较，然后下一条指令按照比较结果，可以跳转到地址0x00435463，而这个地址向下执行就到达了退出程序的代码段，因此这里可以将0x00435433地址处的比较跳转进行逻辑上的相反，那么程序就不会退出了。按照这个思路可以做如下操作：

具体步骤：

（1）单击图6中地址为 00435433指令，使其被选中状态。

（2）单击空格键，在汇编窗口中，将原有的`jg`指令替换成`jle`，其他不变。（这里也可以用NOP填充代码进行修改）

（3）单击快捷键`F9`,继续调试运行程序代码。

完成以上3步操作后，发现diskprobe程序主界面已经正常显示，且没有了原来的时间对话框，修改代码有效。

> 以上指令修改中，`jg`和`jle`分别是"大于跳转"和"小于等于跳转"两种相反逻辑的跳转指令，因此当修改指令后，程序不再弹出时间对话框，软件正常运行全部功能。

> 以上的步骤是经过整理优化的分析过程，实际的分析过程中需要多次的尝试，除了阅读代码，还需要经历大量的失败才能找到最终的结果，所以实际工作不会如实验这样顺利完成。

2.1.3 补丁文件

当以上动态调试成功后，可以使用x32dbg的补丁功能，将内存中已经修改好的程序保存在磁盘上。

图7 补丁修改文件

- 单击工具栏上补丁按钮或使用快捷键`Ctrl+P`,打开图7对话框。
- 单击“修补文件”完成补丁修改。
- 测试：打开修补后的可执行程序文件。

2.2 使用Ida pro静态分析实验

Ida Pro是典型的恶意代码静态分析工具，可以将各类二进制执行文件逆向出汇编指令，甚至是类C语言的伪代码，极大的提高了逆向分析效率，其中自动对函数的识别，标识变量的自动标注非常的方便。本节实验利用其静态分析功能，分析关键代码并使用补丁功能最终完成对目标软件的功能破解。

2.2.1 查找GetSystemTime函数调用代码

- 查找GetSystemTime函数调用内存地址

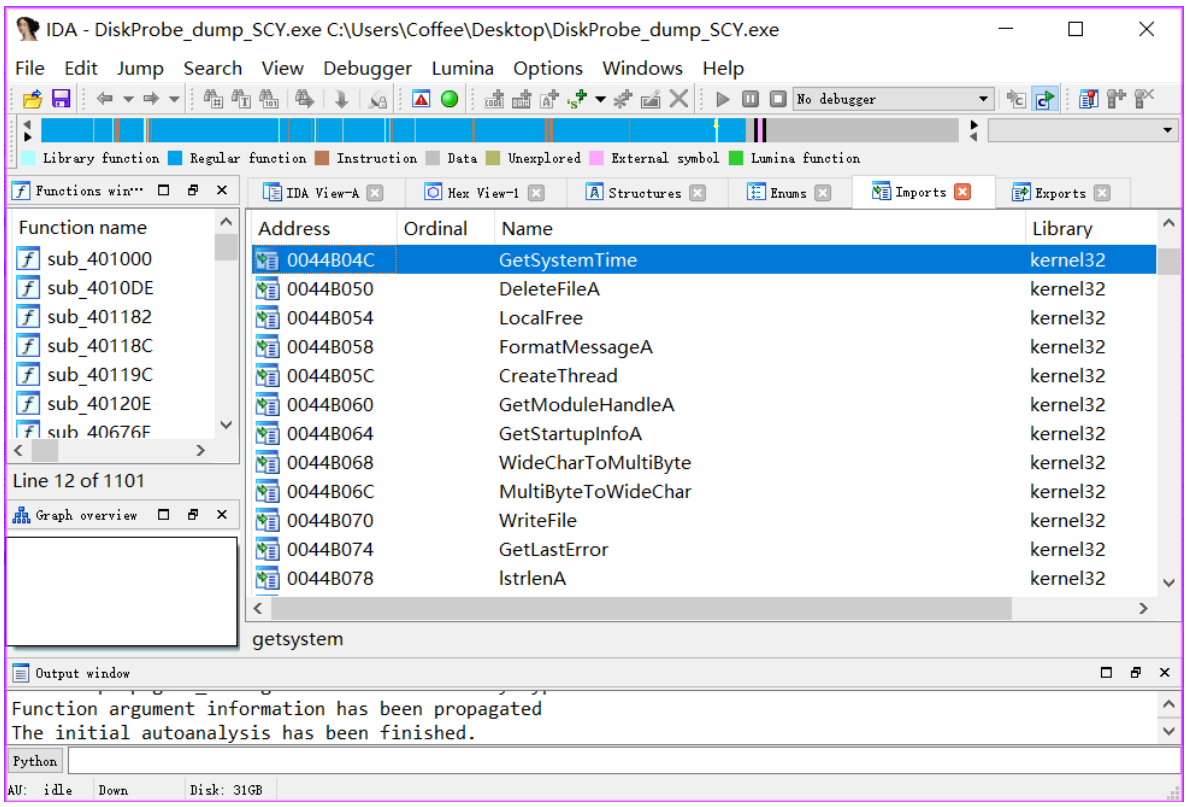

图8 查找导入函数

IDA Pro的Import子视图能够显示程序导入的外部函数，GetSystemTime是系统kernel32.dll中的函数，因此可以从这里查看到导入的内存地址。默认启动IDA Pro就会自动在主界面中显示Imports页面，如果没有显示，可以在菜单中打开，菜单路径为：`View->Open subviews->imports`。

在Imports视图中会显示大量的导入函数，为了方便查找，可以输入查找函数的前几个符号筛选。查找到GetSystemTime函数的内存地址为0044B04C，记录这个地址。

- 定位GetSystemTime函数内存

  在图8界面中切换显示反汇编视图，即IDA-View-A页面，然后单击快捷键`g`,然后输入记录的GetSystemTime内存地址，就能在在反汇编窗口显示GetSystemTime函数，如图9。

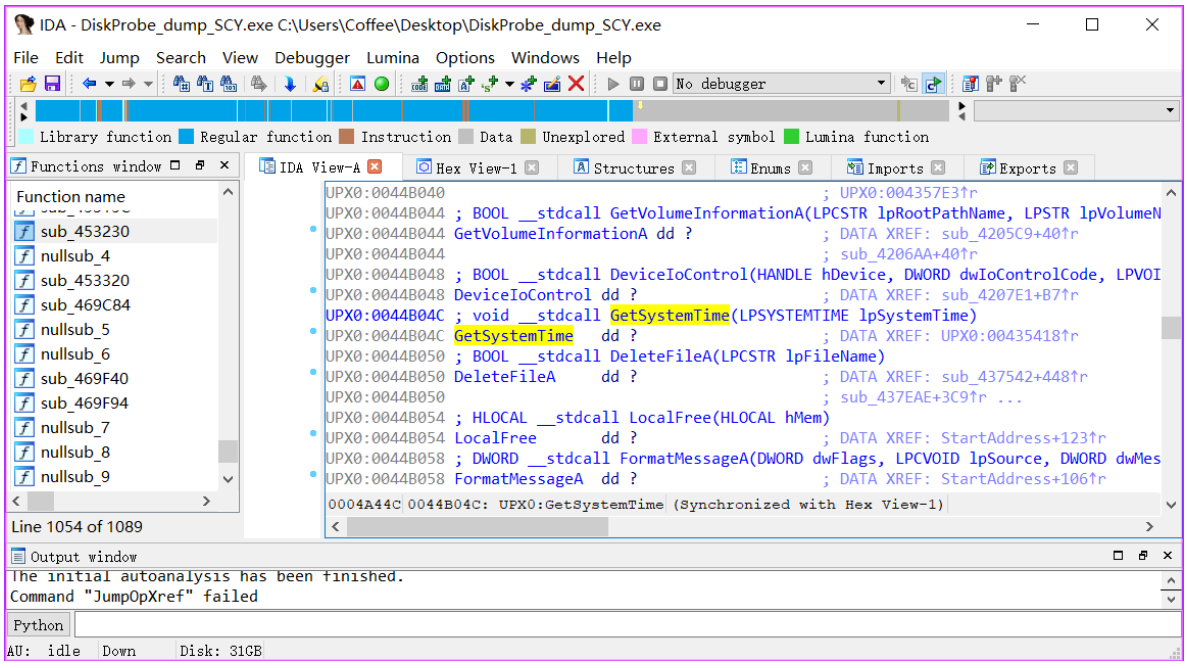

图9 GetSystemTime函数

- 定位GetSystemTime函数调用位置

在图9界面中，在单击GetSystemTime，然后使用快捷键 x ,打开交叉引用列表，如图10，在这个列表中，列出了GetSystemTime被调用的位置，选择其中项目单击OK就能定位到对应的函数调用的位置。

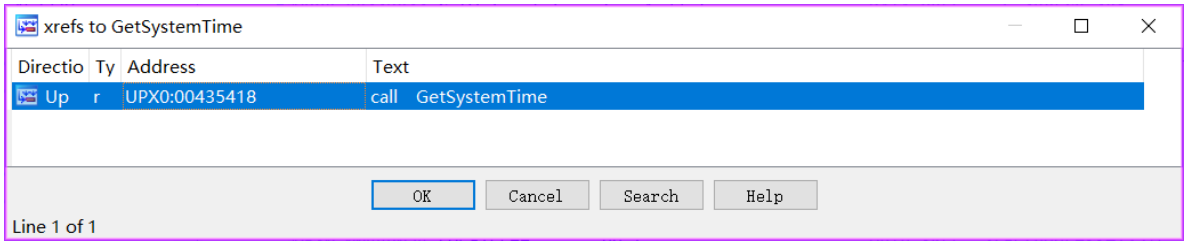

图10 查找引用GetSystemTime的引用

这里只有一处引用了GetSystemTime代码，打开后就能显示函数调用处的反汇编，如图11。

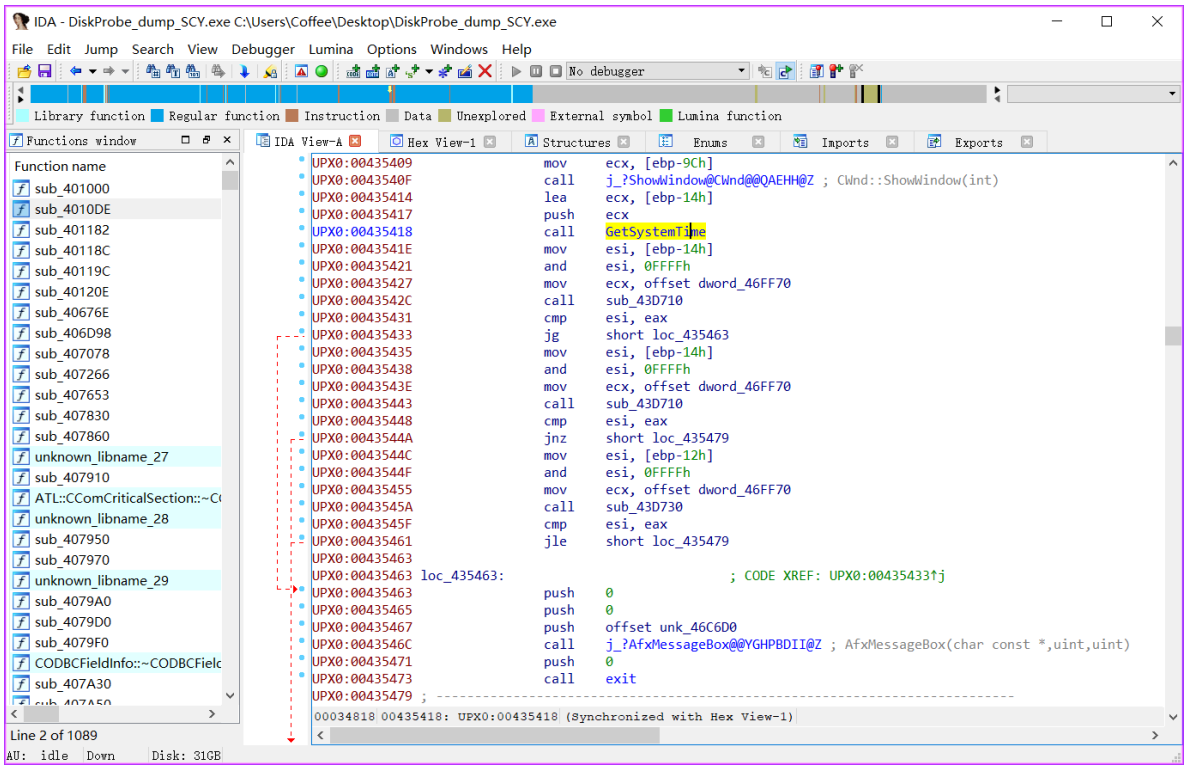

图11 GetSystemTime调用的用户代码段

2.2.2 分析代码并制作补丁文件

- 代码分析

在图11中，能直观的看到GetSystemTime调用后有两条红色的虚线箭头，这表示有两处进行了条件指令跳转，特别是顺着第一个跳转，可以看到是调用AfxMessageBox函数。

> AfxMessageBox是使用Visual C++的MFC开发框架中的个显示对话框的函数，由于这是一个C++函数，所以在编译后函数名会有些变化，看上去会增加一些杂乱的符号。IDA Pro软件识别了这个函数，并且在右侧使用灰色文字注释了这个函数的调用原型。

通过查看AfxMessageBox函数的调用原型，可以发现其调的第一个参数是char const*，也就是输入的是字符串，联系前面的代码，可以确定是call指令上一条的push代码中引用的参数，offset unk_46C6D0就是这个参数字符串。IDA Pro中，如果确定所数据是字符串数据，可以使用快捷键 a 对二进制的数据进行转换。

（1）双击offset offset unk_46C6D0后，在窗口打开其对应的数据。

（2）在显示的窗口中，单击 a 键，此时会自动对数据进行字串解析，转换后显示如下。

```
:0046C6CF                 align 10h
UPX0:0046C6D0 aBghuangSinaCom db '对不起，到期该升级了。(bghuang@sina.com)',0
UPX0:0046C6D0                                         ; DATA XREF:
UPX0:00435467↑o
```

（3）单击键盘左上角`ESC`键，退出，此时发现此时反汇编代码更新了标识和备注，如下：

```
UPX0:00435463       push    0
UPX0:00435465       push    0
UPX0:00435467       push    offset aBghuangSinaCom ; "对不起，到期该升级了。
(bghuang@sina.com)"
UPX0:0043546C       call    j_?AfxMessageBox@@YGHPBDII@Z ; AfxMessageBox(char
const *,uint,uint)
UPX0:00435471       push    0
UPX0:00435473       call    exit
```

到此已经能确认这里代码就是程序中弹出对话框的代码位置，图1显示的对话框，就是以上代码的执行效果。再参考图11中的代码，可以确认代码中第一个条件跳转指令是去除对话框的关键。

- 补丁制作

（1）单击选择地址00435433，使用菜单`Edit -> Patch Program -> Assemble...`打开反汇编窗口，将原来jg指令修改为jle指令。

（2）使用菜单命令`Edit -> Patch Program -> Apply patches to input file...`，打开补丁文件对话框。单击确定，就能将原来的文件进行补丁。如果需要备份补丁前文件，可以勾选backup选项。

2.2.3 测试

直接打开补丁后的文件，发现已经去除了对话框，说明修改有效。

3. 实验结论

- 使用动态分析软件的行文相对比较简单，因此可以通过对比参考软件运行效果，查看运行数据，以及使用断点来提高对关键代码的定位。
- 使用静态分析需要更高的技术知识储备，需要非常熟悉分析的代码相关知识，例如对Windows MFC开发程序，就需要了解框架中相关的关键类以及函数，这里通过AfxMessageBox函数快速确定了弹出的对话框，辅助对代码功能的分析。
- 无论动态分析工具还是静态分析工具，都提供了自动化较强的“汇编”和“补丁”工具，这样可以方便对原程序的修改。

4. 习题

1. 对于本实验内容，除了修改为jle指令，还可以怎么修改？
2. 在使用ida pro静态分析一段数据时，如果怀疑是一段字符串，那么如何使其能正确显示其内容，而不是默认显示16进制编码？
3. 请查阅资料，说明本文提到的各类条件跳转指令的含义以及区别。
4. 每当进入一个函数，开辟新的栈帧，此时栈顶数据是什么？