

**恶意代码分析与防治课程实验报告**

**实验12**

****

专 业 信息安全

学 号 2113662

姓 名 张丛

班 级 信安一班

1. **实验目的**

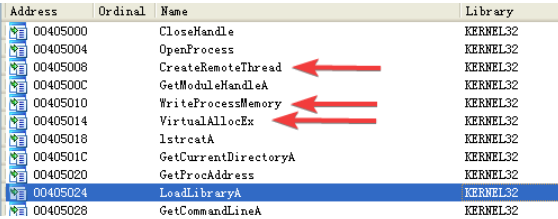
完成课本Lab 12的实验内容，编写Yara规则，并尝试IDA Python的自动化分析。

1. **实验过程**

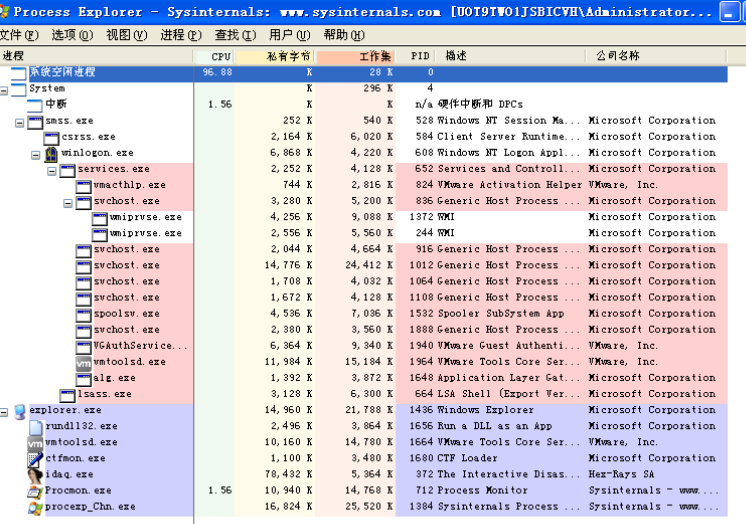
## Lab 12-1

**分析：**

导入函数：



Proexproer检查：



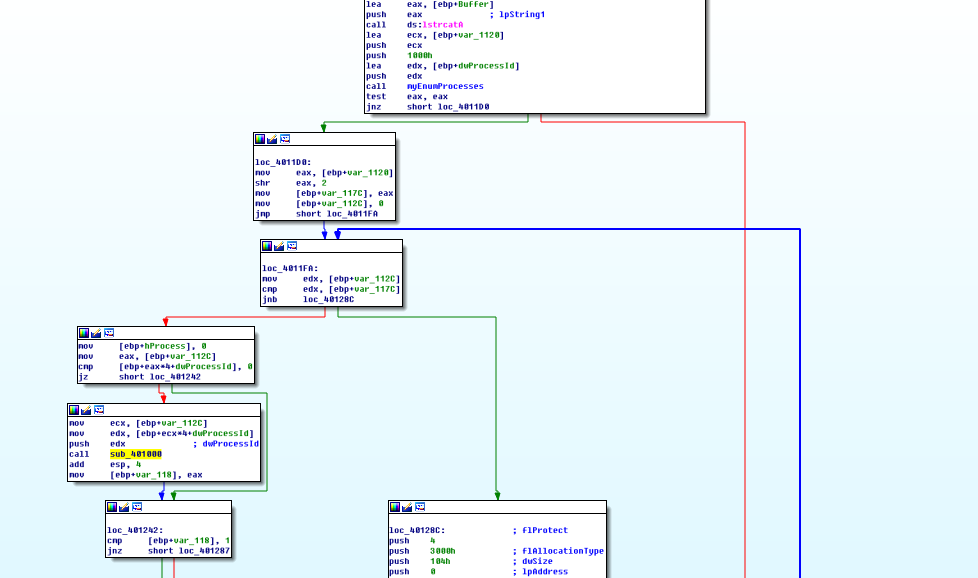
没有什么有用的信息。

IDA中：

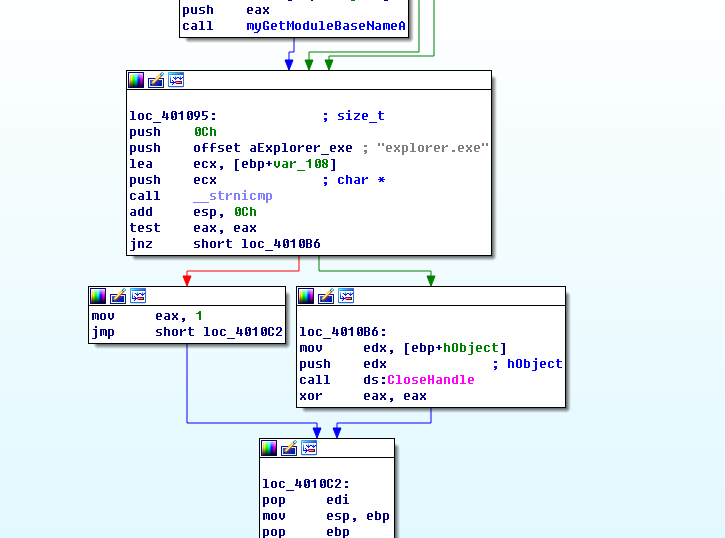


将这几个字符串重命名，便于识别。

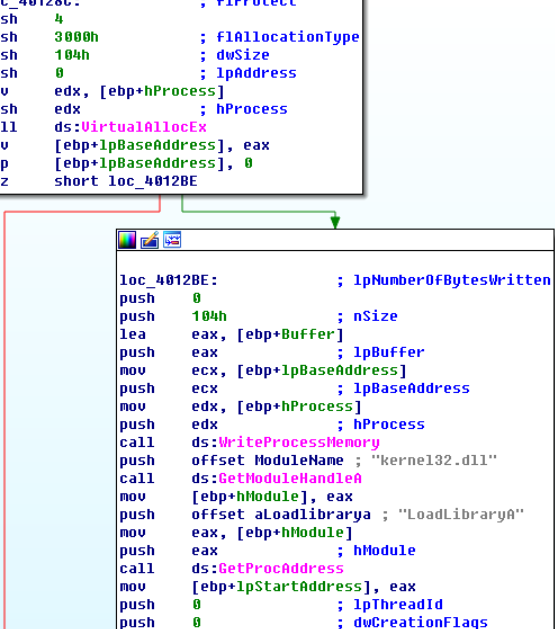
该程序是一个对于PID的遍历。



这个函数的作用是检查是否存在explorer.exe这个程序。

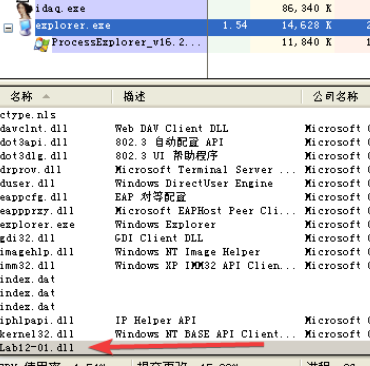


Buffer表示的字符串是Lab12-01.dll.



这段代码所表示的就是对explorer.exe程序进行注入。

根据已知分析结果，我们取proexp中查找被注入的程序：



### 在你运行恶意代码可执行文件时，会发生什么？

运行这个恶意代码之后，每分钟在屏幕上显示一次弹出消息，会显示已经弹出的次数。



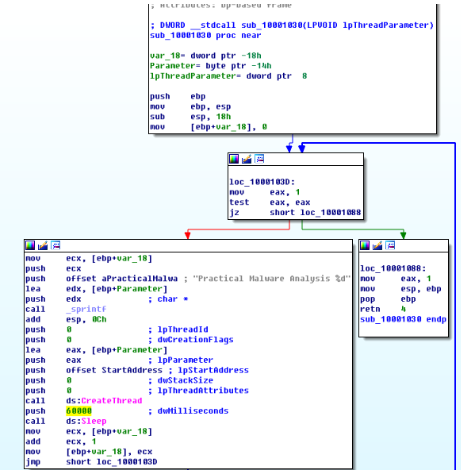
### 哪个进程会被注入？

被注入的程序是explorer.exe。

### 你如何能够让恶意代码停止弹出窗口？

重启explorer.exe进程。

### 这个恶意代码样本是如何工作的？



这个恶意代码执行DLL注入，在explorer.exe中启动Lab12-01.dll，一旦Lab12-01.dll被注入，它在屏幕上每分钟显示一个消息框，并通过一个计数器，来显示已经弹出了多少次。

## Lab 12-2

**分析：**

导入函数：

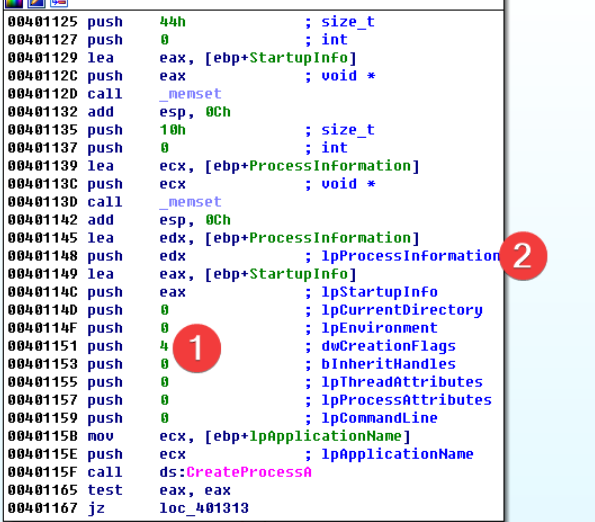


CreateProcessA、GetThreadContext以及SetThreadContext暗示着这个程序创建新的进程，并修改进程中线程的上下文。

导入函数ReadProcessMemory和WriteProcessMemory告诉我们这个程序对进程内存空间进行了直接的读写。

导入函数LockResource和SizeOfResource告诉我们这个进程比较重要的数据可能保存在哪里。

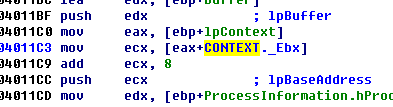
分析CreateProcessA函数的目的：



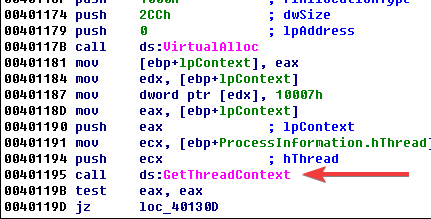
在代码①处，我们看到一个push 4指令。而它被IDA Pro标记为参数dwCreationFlags。这个进程将不会被执行，除非等到这个主进程调用API ResumeThread函数时，它才会被启动。

在如下所示的位置，我们看到这个程序正在访问一个线程上下文。GetThreadContex的hThread参数与②处传递给CreateProcessA的参数处于同一个缓冲区，他告诉我们在这个程序正在访问挂起进程的上下文。获取进程句柄非常重要，因为程序将使用这个进程句柄与怪气进程进行交互。

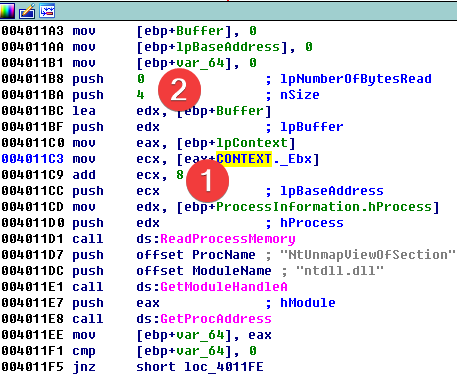
调用GetThreadContext以后，我们看见这个进程奖杯用于ReadProcessMemory的调用。为了更好地判断这个程序用进程上下文做了什么，我们需要在IDA Pro中添加CONTEXT结构体。要添加这个标准结构题，需要单击Structures标签页，按INS键。接着单击ADD Standard Structure按钮，并放置名为CONTEXT的结构体。



0x4A偏移实际上调用了这个进程的EBX寄存器。

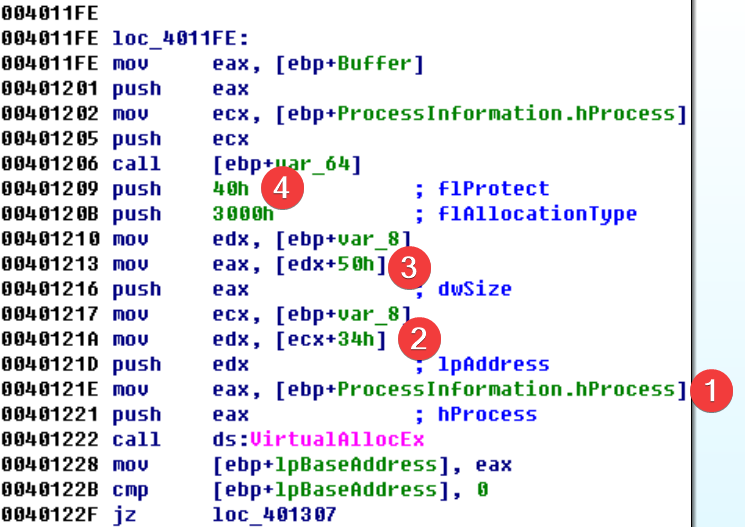


这个新创建就被挂起的进程EBX寄存器总是包含一个指向进程环境块（PEB）的数据结构。如下所示，在①处，程序以8字节递增结构体，并将这个值压到栈上，作为要读取内存的起始地址。

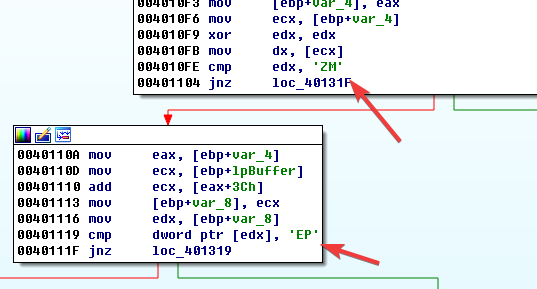


因为PEB数据结构并不是标准IDA Pro数据结构中而定一部分，我们使用其他的工具来判断PEB数据结构的8字节偏移处是什么：一个指向ImageBaseAddress（被加载的可执行文件起始部分）的指针。 将这个地址作为读取位置，并在②处读取4个字节，我们看到IDA Pro已标记为Buffer的变量将包含被挂起进程的ImageBase。

这个程序使用在0x004011E8处的GetProcessA，手动解析导入函数UnMapViewOfSection，并且在0x004011FE处，ImageBaseAddress作为UnMapViewOfSection的一个参数传入。UnMapViewOfSection的调用从内存中溢出这个被挂起的进程，此时程序不再被执行。

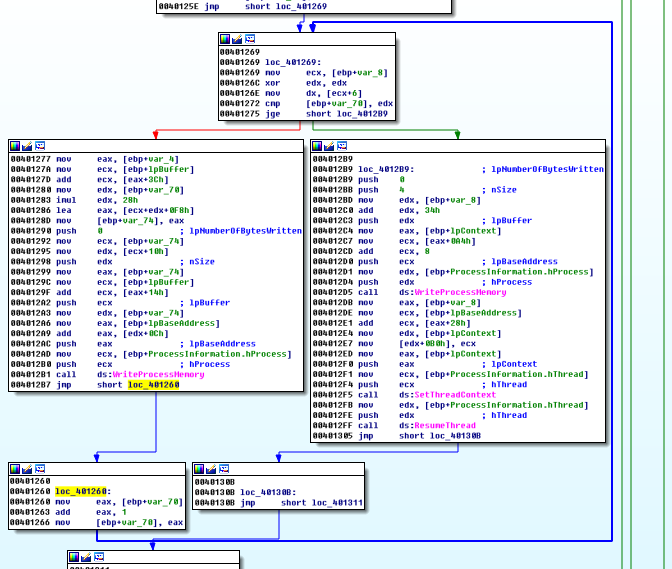


在①处，显示了这个程序在被挂起进程的地址空间分配内存。在函数的开头，程序检查在0x004010FE处的魔术值MZ和0x00401119处的魔术值PE。

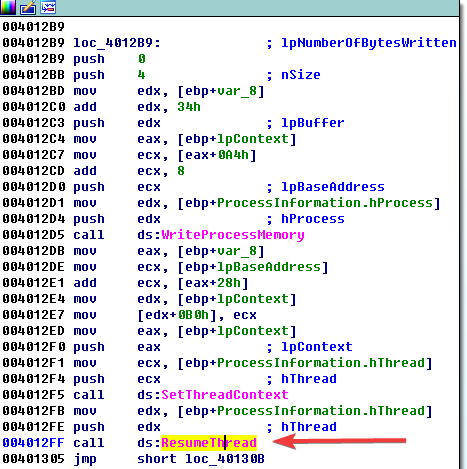


在②处，这个程序要求内存被分配在PE文件ImageBase的地址，它告诉Windows加载器这个可执行文件更倾向于被加载到哪个位置。 在③处，这个程序请求由PE头ImageSize属性指定内存的大小。 最终，在④处，我们查询MSDN文档判定这部分内存是以PAGE\_EXECUTE\_READWRITE权限分配的。

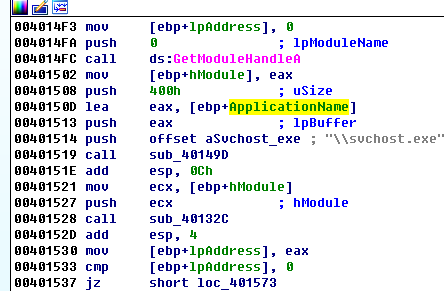
一旦这部分内存被分配后，在0x00401251处指令会调用writeProcessMemory函数。从这个PE文件的开头写入数据到挂起进程时分配的内存中。要写入的字节数是从PE文件头的偏移0x54处取得的，也就是SizeOfHeaders。第一次WriteProcessMemory调用将PE文件头复制到被挂起的进程中，这表明了这个程序正移动一个PE文件到另一个进程的地址空间。



整体有一个循环函数。这个循环函数的作用试讲PE字节复制到内存中。

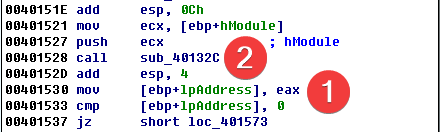


恢复了被挂起的进程。

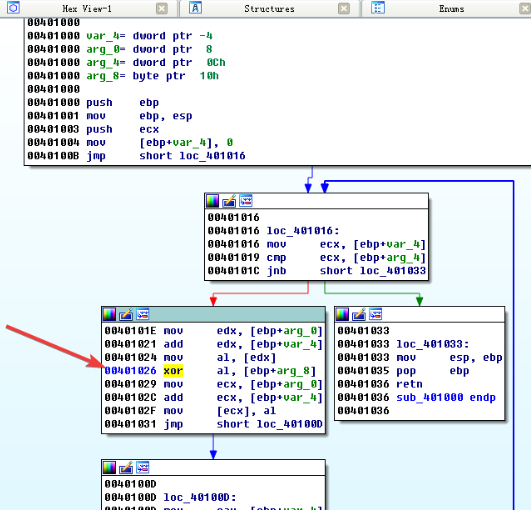


被替换的进程是%SystemRoot%\System32\svchost.exe。

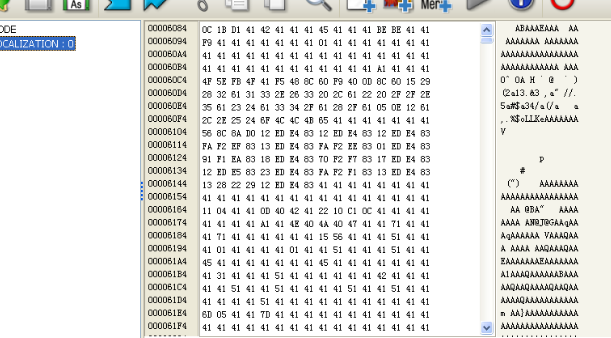
现在我们这个这个程序启动的是svchost.exe，但我们还需要判断要替换svchost.exe的进程。我们通过跟随在0x00401539处的变量lpBuffer，来跟踪传递给sub\_4010EA的PE缓冲区，就像原路返回到Dst变量一样。



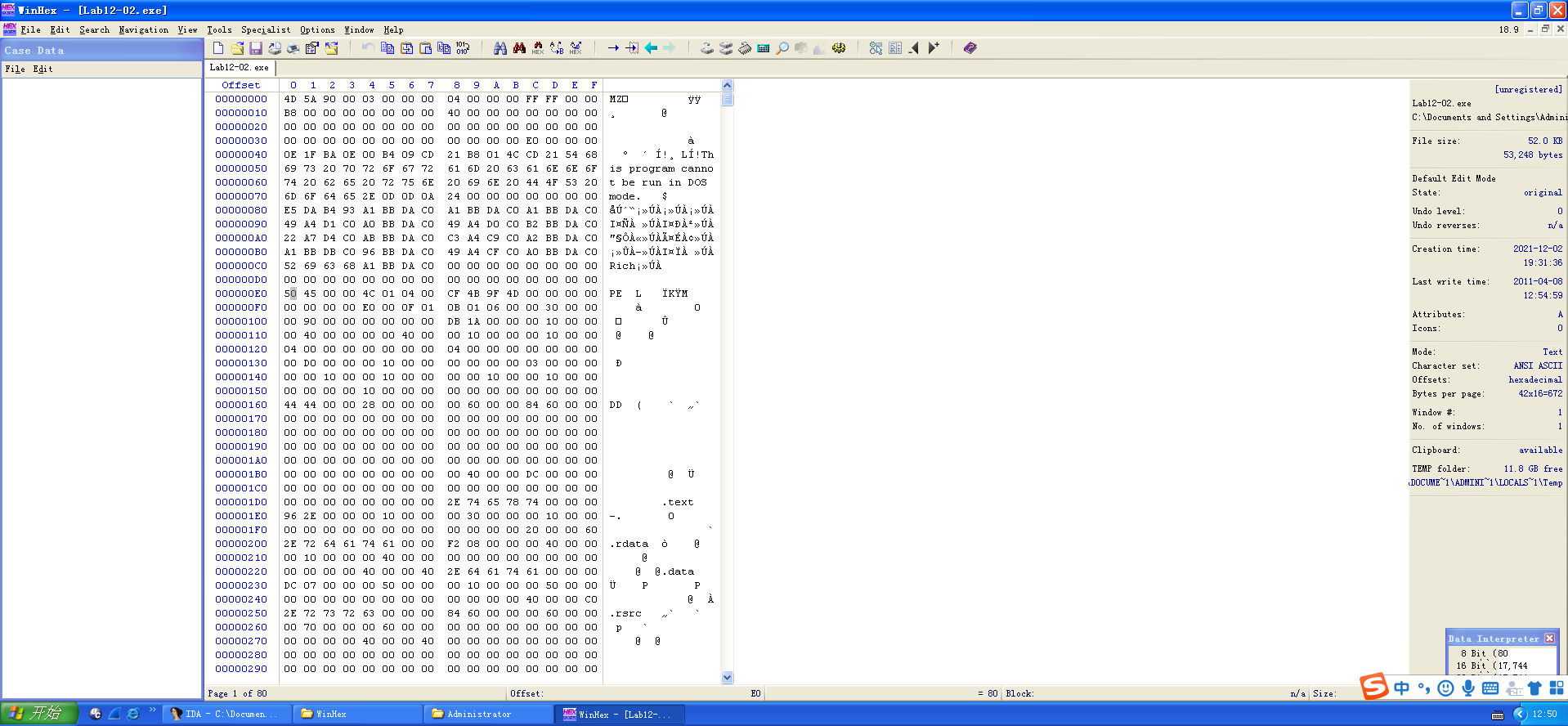
我们定位到lpBuffer，它在①处接受EAX，通过检查之前的指令，可以发现在②处的一个函数调用。记住EAX用作一个函数的返回值，我们知道这个缓冲区来自函数sub\_40132C,很明显，它使用变量hModule，即一个指向程序本身——Lab12-02.exe———内存指针。



使用Resourcehack查看：



使用winhex解密：



### 这个程序的目的是什么？

这个程序的目的是秘密地启动另一个程序。

### 启动器恶意代码是如何隐蔽执行的？

这个程序使用进程替换来秘密执行。

### 恶意代码的负载存储在哪里？

这个恶意的有效载荷（payload）被保存在这个程序的资源节中。这个资源节的类型是UNICODE，且名字是LOCALIZATION。

### 恶意负载是如何被保护的？

保存在这个程序资源节中的恶意有效载荷是经过XOR编码过的。这个解码例程可以在sub\_40132C处找到，而XOR字节在0x0040141B处可以找到。

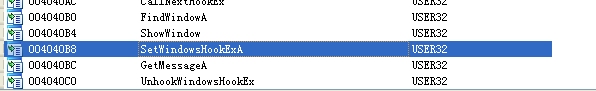
### 字符串列表是如何被保护的？

这些字符串是使用sub\_401000处的函数，进行XOR编码的。

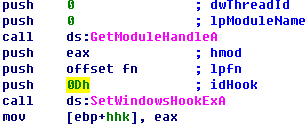
## Lab 12-3

**分析：**

导入函数：



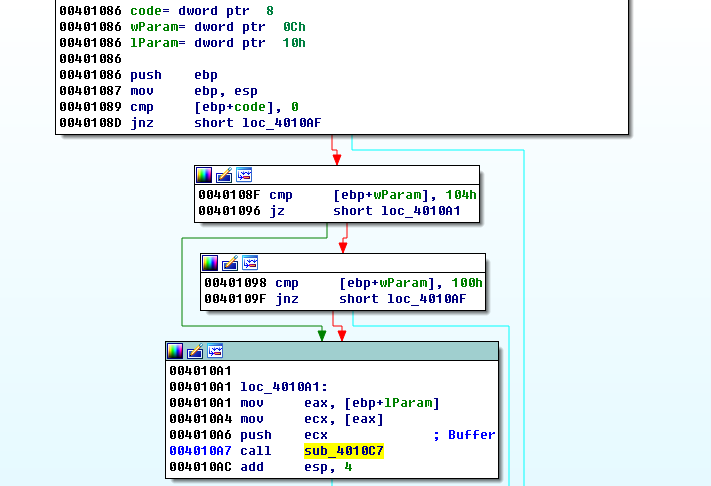
SetWindowsHookExA，这是一个允许应用程序挂钩或监控微软Windows内部事件的API。



我们看到SetWindowsHookExA在main中被调用，第一个参数是）0Dh，对应WH\_KEYBOARD\_LL，它使用IDA Pro标记为fn的挂钩函数启用键盘事件监控。这个函数可能对键击消息做些手脚，而这个fn函数正在接受这击键记录。

注册接受键盘事件的钩子之后，程序在0x00401076处的循环中调用了GetMessageA。本例中，程序一定要调用GetMessageA，因为Windows不会讲消息发送到程序进程的钩子函数中，知道循环产生错误，它才会停止。

移动fn函数，我们开始查看程序对截获的击键事件做了什么处理。fn是一个通用函数，它带有三个参数，它有一个定义为HOOKPROC的原型。查询MSDN文档，我们确定WH\_KEYVOARD\_LL回调函数，实际上是LowLevelKeyboardProc回调函数。

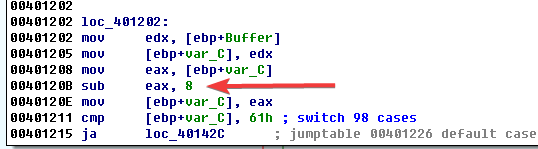


程序利用cmp指令检查按键的类型，将虚拟按键码传递到随后的函数sub\_4010C7函数中。

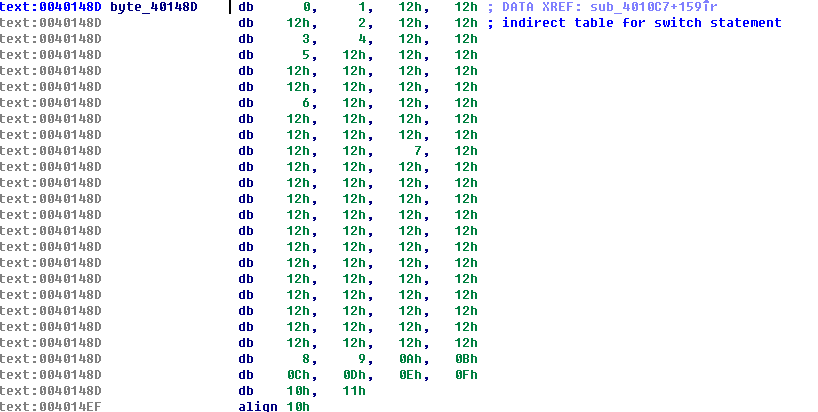
检查sub\_4010C7函数，我们看到程序首先打开一个文件practicalmalwareanalysis.log。打开文件之后，恶意代码调用GetForegroundWindows和GetWindoTextA，如下所示。GetForegroundWindows选择按键按下时的活动窗口，然后使用GetWindoTextA获得窗口的标题。通过这些操作，帮助程序提供按键来源的上下文。



一旦程序将窗口标题写入到日志文件，它就进入下一个跳转表。确定变量var\_c包含传入函数sub\_4010C7的虚拟按键码后，我们看到虚拟按键码作为一个查询表的索引。查询表得到的值作为跳转表off\_401441的一个索引。



在sub eax,8处，剩下的值为0x8。在byte\_40148D查找偏移量0x8。得到值3,3被存入到ecx中。然后，ecx乘以4，得到0xC，将它作为off\_401441的一个偏移量，然后返回到位置loc\_401249，这里我们发现字符串[SHIFT]被写入到日志文件中。



### 这个恶意负载的目的是什么？

这个程序是一个击键记录器。

### 恶意负载是如何注入自身的？

这个程序使用挂钩注入，来偷取击键记录。

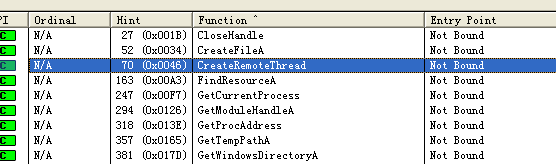
### 这个程序还注入了哪些文件？

这个程序创建文件practicalmalwareanalysis.log，来保存击键记录。

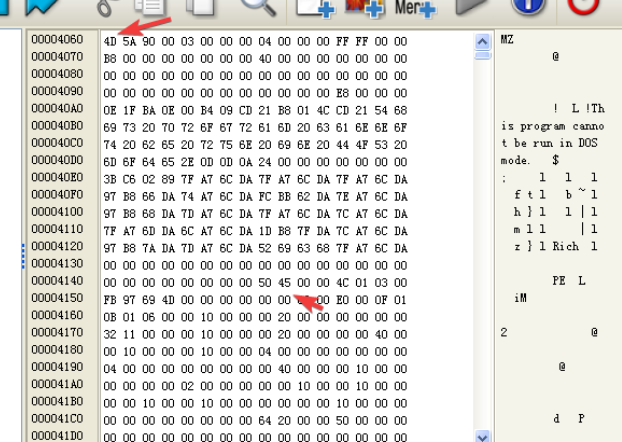
## Lab 12-4

**分析：**

查看导入表：

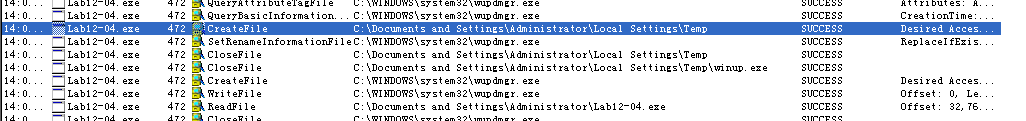


使用resourcehack查看：

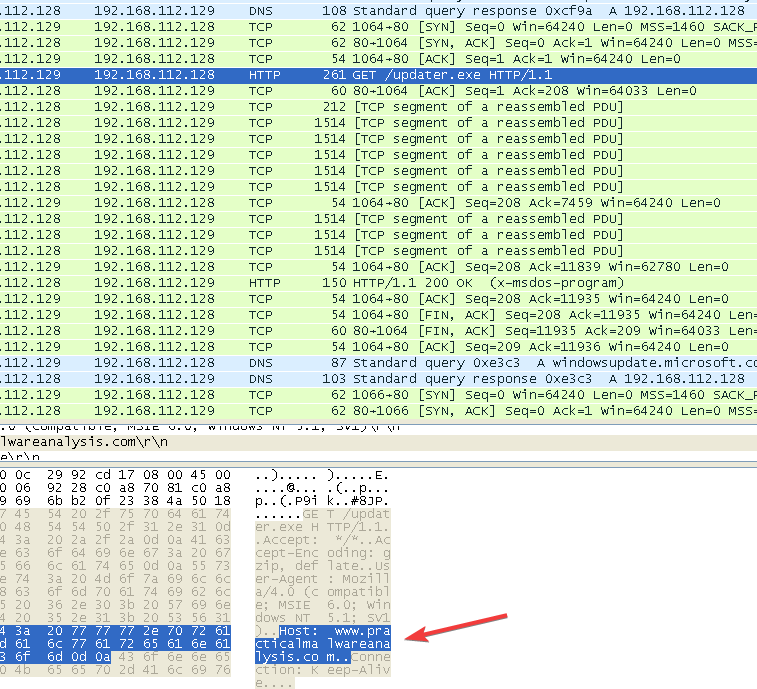


确实为一个PE文件，我们将它提取出来。

启动wireshark和promon，进行动态分析。

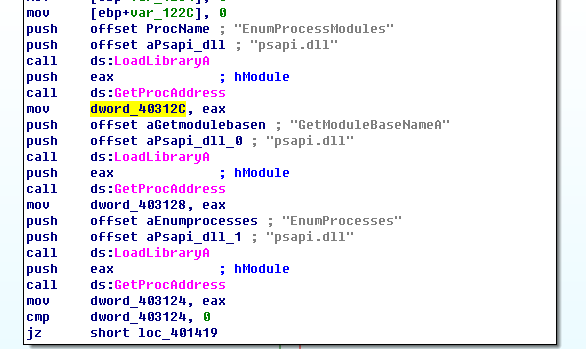


程序添加了一个文件，使用winmd5可知这个文件和我们在上从程序中提取出来的文件是同一个文件。

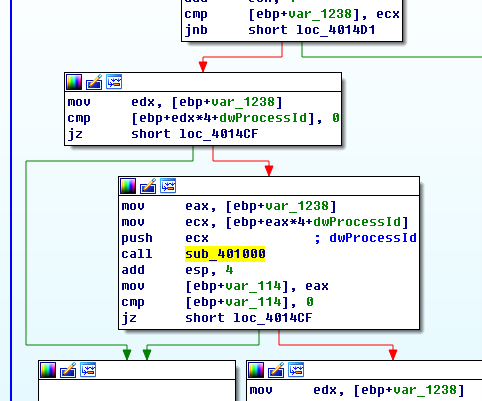


试图从一个网址下载程序。

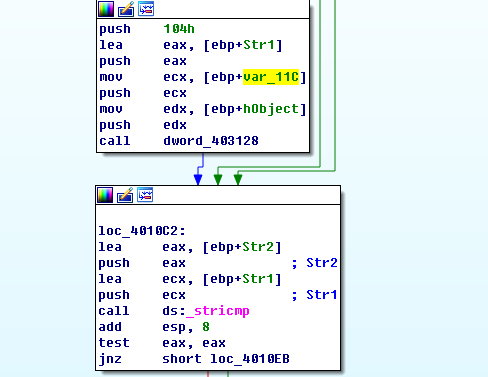
### 位置 0x401000 的代码完成了什么功能？



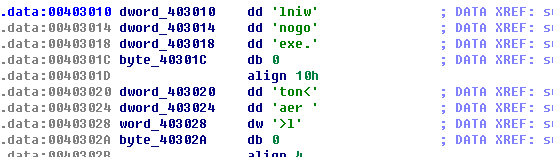
首先是调用函数，重命名。  
 然后是遍历PID。



进入函数查看。  
 查看str1和str2中保存了什么字符串。

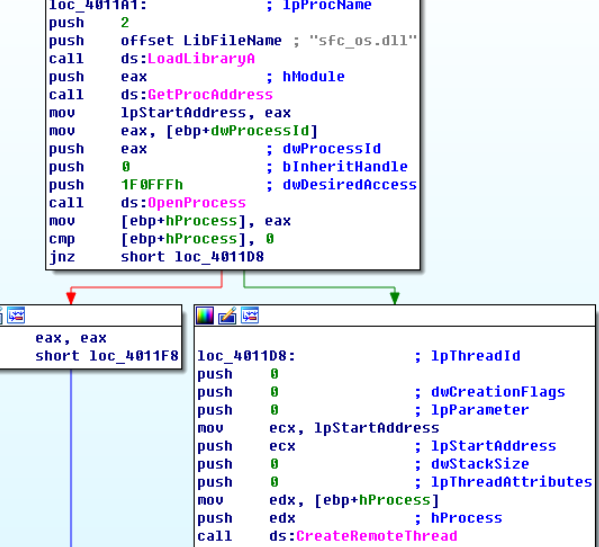


比对str1和str2。



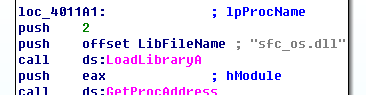
这个函数的作用是对比当前进程是否是winlogon.exe。

### 代码注入了哪个进程？



被远程注入的程序就是winlogon.exe。

### 使用LoadLibraryA装载了哪个DLL程序？



装载的DLL程序是sfc\_os.dll，调用的函数可以禁用文件保护机制。

### 传递给CreateRemoteThread调用的第4个参数是什么？

指向的是一个函数而定指针，就是sfc\_os.dll中序号为2的函数的指针。

### 二进制主程序释放出了哪个恶意代码？

从网页中下载文件，重命名到目录：\system32\wupdmgrd.exe。

恶意文件wuodmgrd.exe比正常wuodmgr.exe多了一个字母d。

恶意代码从资源段中释放一个二进制文件，并且将这个二进制文件覆盖旧的Windows更新程序( wupdmgr.exe)。覆盖真实的wupdmgr.exe之前，恶意代码将它复制到%TEMP%目录，供以后使用。

### 释放出恶意代码的目的是什么？

恶意代码向winlogon.exe注入一个远程线程，并且调用sfc\_ os.dll 的一个导出函数(序号为2的SfcTerminateWatcherThread)，在下次启动之前禁用Windows的文件保护机制。

因为这个函数一定要运行在进程winlogon.exe 中，所以CreateRemoteThread调用十分必要。恶意代码通过用这个二进制文件来更新自己的恶意代码。

## yara

rule Lab12

{

meta:

description = "rules for Lab12"

date = "2021/12/11"

strings:

$a0 = "LOCALIZATION"

$a1 = "paticalmalwareanalysis.log，"

$a2 = "wupdmgrd.exe"

$a3 = "http://www.practicalmalwareanalysis.com/updater.exe"

condition:

any of them

}

1. **实验结论及心得体会**

复习到课上的知识点，隐蔽的恶意代码启动，包括直接注入、挂钩等等。

熟练使用恶意代码分析工具。