## 课本实验12实验报告

姓名：魏伯繁 学号：2011395 专业：信息安全

### **一、实验工具：**

IDAPro7.0 IDAPro7.7 win10虚拟机 winxp虚拟机、Wireshark、Ollydbg、strings、ProcessMonitor、Process Explorer、RegShot等

### 二、实验环境：

静态分析主要在win10虚拟机完成。

内存4GB

主频2.9GHZ

### 三、实验内容

第一部分：Lab12-01

1、在你运行这个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，并且还有计数的次数的显示并且无法被正常关闭**

2、哪个进程会被注入

**被注入的进程是explorer.exe。**

3、你如何能够让恶意代码停止弹出窗口？

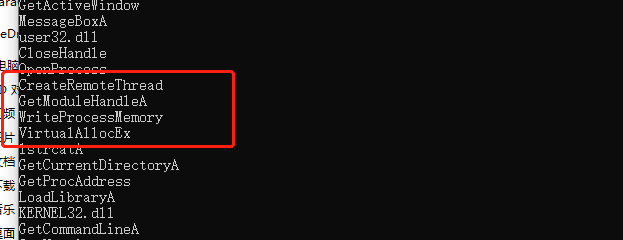
**使用Process Explorer强行终止进程**

4、这个恶意代码样本是如何工作的？

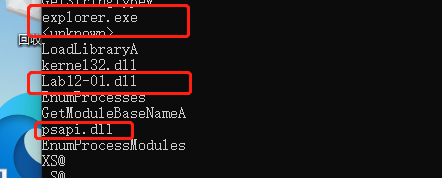
**这个恶意代码执行DLL注入，来在explorer.exe中启动Lab12-01dll。它在屏幕上每分钟显示一个消息框，并通过一个计数器，来显示已经过去了多少分钟。**

具体分析过程：

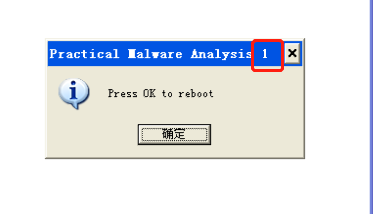
第一步：基本静态分析，在基本静态分析中，我们可以看到基于Lab12-01.exe的导入函数，我们看见了：CreateRemoteThread、WriteProcessMemory以及VirtualAllocEx，我们可以判定这可能是基于某种形式的进程注入恶意代码

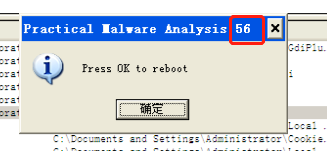


当然，我们还可以看到一些其他的有意思的字符串，比如：explorer.exe、Lab12-01.dll以及psapi.dll

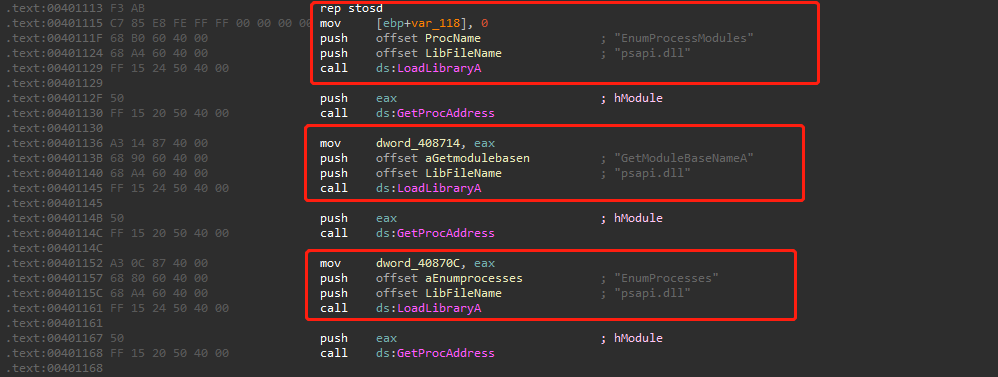


接下来，我们使用动态分析技术，当我们运行这个恶意代码时他每分钟都会弹出一个消息框，并且还会对弹出的次数进行计次，但是Procmon以及Process Explorer都没有什么明显的恶意进程以及恶意行为可以被观察到

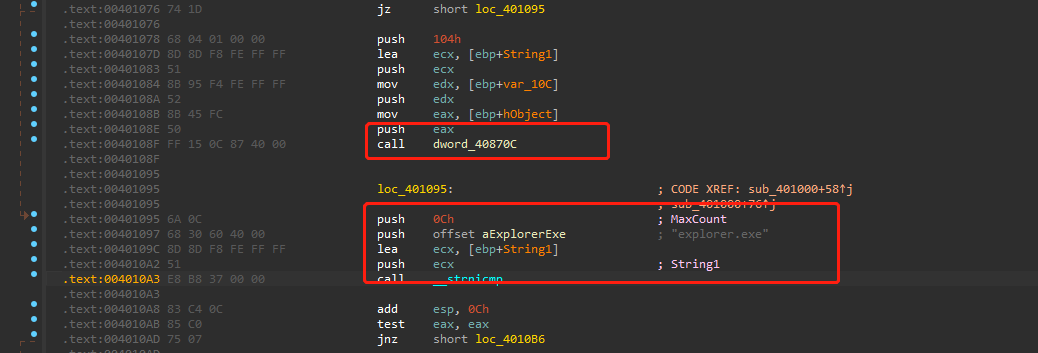




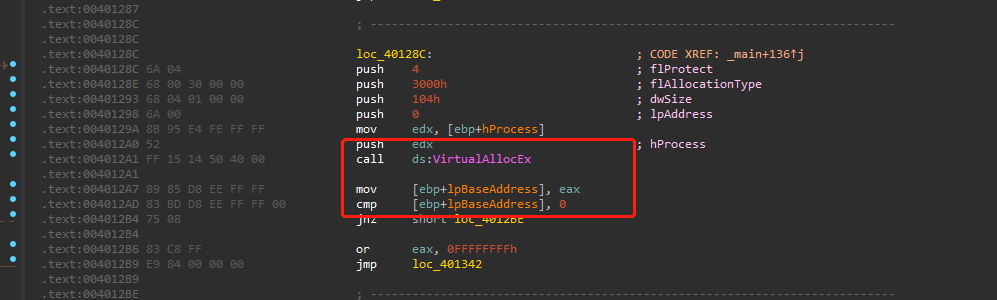
然后，我们使用IDAPRO对恶意代码进行分析，在main函数刚开始的几行里可以看到这个代码正在解析psapi.dll中Windows枚举进程的函数，恶意代码使用了LoadLibrary和GetProcAddress来手动的解析3个函数：恶意代码保存指向408714/40870C以及408710的函数指针，他们分别代表了EnumProcessModules、GetModuleBaseNameA以及EnumProcesses，在动态解析结束后这段代码调用EnumProcess来获取在系统中每一个进程对象的PID。EnumProcesses返回一个由局部变量dwProcessId引用的PID数组，该引用将一个列表中的进程列表调用sub\_401000



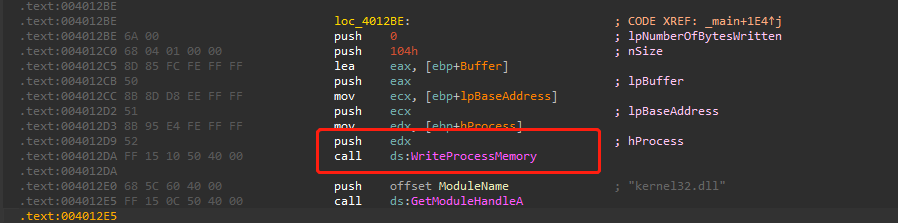
然后，我们来查看401000的代码，当动态解析的导入函数EnumProcessMoudles在PID被传给OpenProcess函数后被调用，然后我们就调用GetModuleBaseNameA，其中dword\_40870C代表着GetModuleBaseNameA函数将PID翻译为进程名，然后我们将得到的字符串和explorer.exe进行比较赖在内存空间中找到explorer.exe



如果成功找到了该进程那么则返回1，main函数会调用OpenProcess来打开一个指向他的句柄，如果这个恶意代码成功改的返回了一个这个进程的句柄那么就执行下面的代码：调用一个VirtualAllocEx动态的explorer.exe中分配内存，上面提到的0x104字节通过压栈的方式当做参数传递，如果内存分配成功，一个指向被分配空间的lpBaseAddress会被移动到该句柄中。



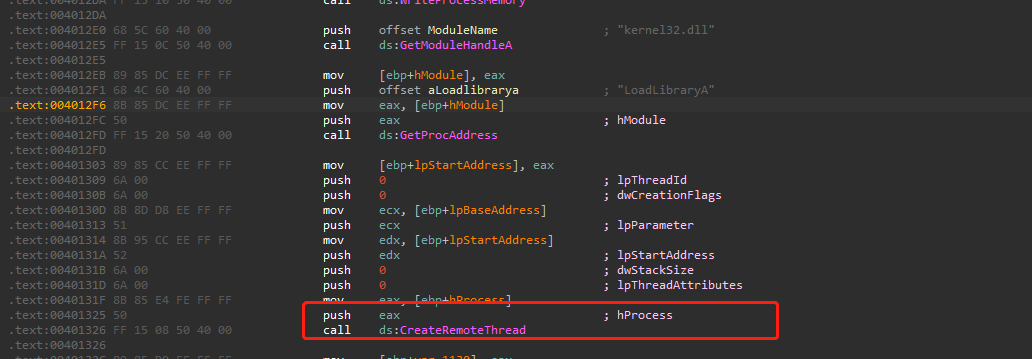
该句柄和进程句柄hprocess会被当做参数一起传递给WriteProcessMemory以便向explorer.exe中传递数据，写入这个进程的数据存储在了lpBuffer中



如果上面的内容成功写入了则会进入下面代码的执行首先调用GetModuleHandleA以及GetProcAddress来获取LoadLibraryA的地址。在本恶意样本中LoadLibrary和explorer.exe的地址相同，于是使用指令

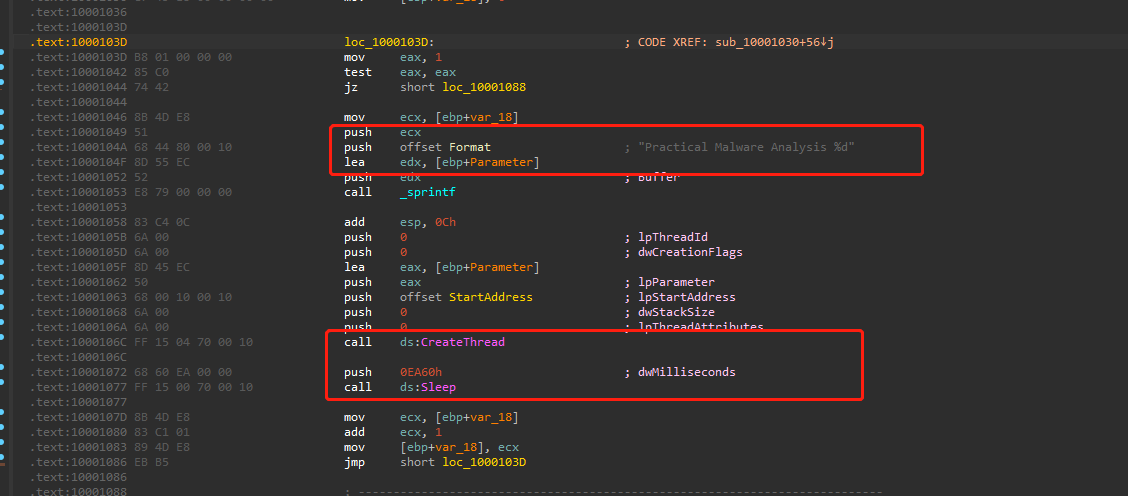
mov [ebp+lpStartAddress]， eax

来讲LoadLibraryA的地址写入lpStarAddress中，CreateRemoteThread的参数lpStarAddress参数强制explorer.exe调用LoadLibraryA。他的参数通过CreateRemoteThread中的lpParameter传递，也即是包含Lab12-01的字符串，于是，在远程进程中启动一个线程，这个线程以参数Lab12-01.dll来调用LoadLibraryA。我们现在可以断定这个恶意代码进行了DLL注入，将Lab12-01.dll注入到explorer.exe中



当我们知道什么东西被注入了时就可以想办法去停止他，使用Process Explorer来进行停止恶意代码的运行，我们可以看到在explorer.exe中载入了Lab12-01.dll，在确定我们的分析无误后，我们可以使用Process Explorer来停止该进程的执行，然后再通过在Run中输入explorer重启进程

于是我们可以明确的是，恶意代码的功能都是由dll文件提供的，于是我们分析dll文件：我们可以看到其开启了一个新的线程，线程调用的函数的句柄为lpStartAddress，他用来创建消息框并显示Press OK to reboot，他用一个sprintf拼接的参数作为这个消息框的标题，这个参数用Format字符串来显示“Practical Malware Analysis”+其对话框弹出的次数，每次执行都都会睡眠60s然后再次执行



第二部分：Lab12-02

1、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**

2、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，也就是先将一个正常的程序以挂起启动，然后替换他的每一个节**

3、恶意代码的负载存储在哪里？

**这个恶意的有效载荷被保存在这个程序的资源节中。可以通过ResourceHacker进行提取**

4、恶意负载是被如何保护的？

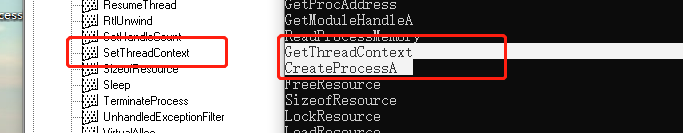
**保存在这个程序资源节中的恶意有效载荷是经过XOR编码过的。这个解码例程可以在sub 40132C4处找到，而XOR字节在0x0040141B处可以找到。如果想要获得解密后的程序可以通过WINHEX进行解密**

5、字符串列表是被如何保护的？

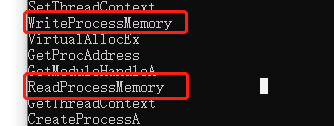
**字符串是使用在sub40100处的函数，来进行XOR编码的**

具体分析过程：

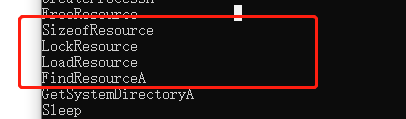
通过静态分析，除了导入表的函数名外似乎找不到更多的有用的信息，于是我们针对导入表的信息进行分析，可以看到CreateProcessA、GetThreadContext以及SetThreadContext暗示这个程序创建新的进程并修改进程中的线程的上下文



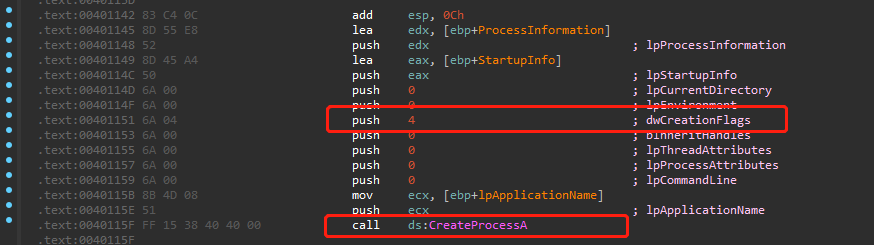
接下来导入函数ReadProcessMemory和WriteProcessMemory告诉我们这个程序对进程内存空间进行了直接读写



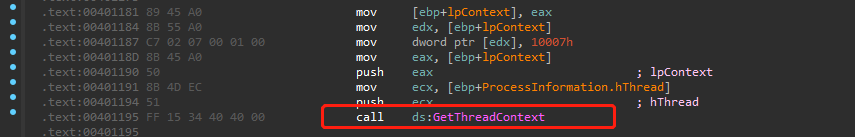
导入函数LockResource和SzieOfResource告诉我们这个进程比较重要的数据可能存在资源节中



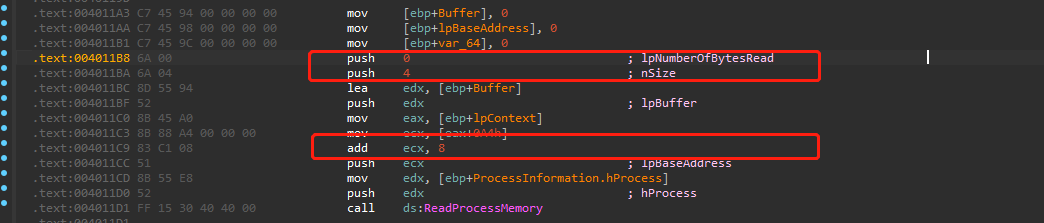
随后我们启用高级静态分析来准确识别其实现的具体功能形式：首先我们将注意力集中在CreateProcessA--创造进程的函数调用上，首先，他压入了一个参数为4，其对应的参数名为dwCreationFlags。微软官方对该函数的描述表明该位置是描述CREATE\_SUSPENDED标志，他允许这个进程被创建但是不被启动。这个进程不会被执行，除非等待这个主进程的API ResumeThread函数时才会被启动



而在后面它又调用了访问线程上下文的函数，而作为函数参数传递的hThread在前面一张图中的lpProcessInformation处于同一缓冲区，它告诉我们这个程序正在访问挂起的进程的上下文。或其与进程的句柄非常重要，因为程序将使用进程句柄与挂起进程进行交互



调用GetThreadContext之后，这个进程被用于ReadProcessMemory的调用。代码如下图所示：这个新创建就被挂起EBX寄存器总是指向一个指向进程环境块PEB的数据结构，通过在004011c9处递增8字节递增结构体并将这个值压入到栈上，作为要读取内存的起始地址。因为PEB不是IDAPRO数据结构中的一部分，但我们可以在网上查找到关于PEB的相关资料，在偏移为8的位置为ImageBase

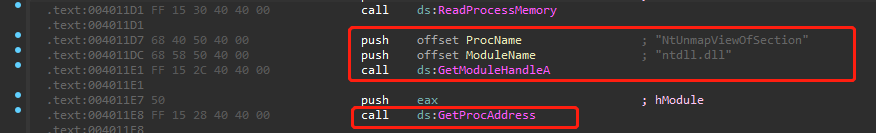


其中，进程环境块的定义如下：参考网址如下：

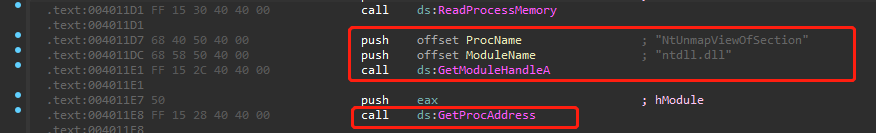
[(117条消息) Process Environment Block（PEB）\_寻梦&之璐的博客-CSDN博客](https://blog.csdn.net/CSNN2019/article/details/113113347)

typedef struct \_PEB { // Size: 0x1D8  
 000h UCHAR InheritedAddressSpace;  
 001h UCHAR ReadImageFileExecOptions;  
 002h UCHAR BeingDebugged; //Debug运行标志  
 003h UCHAR SpareBool;  
 004h HANDLE Mutant;  
 008h HINSTANCE ImageBaseAddress; //程序加载的基地址  
 00Ch struct \_PEB\_LDR\_DATA \*Ldr //Ptr32 \_PEB\_LDR\_DATA  
 010h struct \_RTL\_USER\_PROCESS\_PARAMETERS \*ProcessParameters;  
 014h ULONG SubSystemData;  
 018h HANDLE DefaultHeap;  
 01Ch KSPIN\_LOCK FastPebLock;  
 020h ULONG FastPebLockRoutine;  
 024h ULONG FastPebUnlockRoutine;  
 028h ULONG EnvironmentUpdateCount;  
 02Ch ULONG KernelCallbackTable;  
 030h LARGE\_INTEGER SystemReserved;  
 038h struct \_PEB\_FREE\_BLOCK \*FreeList  
 03Ch ULONG TlsExpansionCounter;  
 040h ULONG TlsBitmap;  
 044h LARGE\_INTEGER TlsBitmapBits;  
 04Ch ULONG ReadOnlySharedMemoryBase;  
 050h ULONG ReadOnlySharedMemoryHeap;  
 054h ULONG ReadOnlyStaticServerData;  
 058h ULONG AnsiCodePageData;  
 05Ch ULONG OemCodePageData;  
 060h ULONG UnicodeCaseTableData;  
 064h ULONG NumberOfProcessors;  
 068h LARGE\_INTEGER NtGlobalFlag; // Address of a local copy  
 070h LARGE\_INTEGER CriticalSectionTimeout;  
 078h ULONG HeapSegmentReserve;  
 07Ch ULONG HeapSegmentCommit;  
 080h ULONG HeapDeCommitTotalFreeThreshold;  
 084h ULONG HeapDeCommitFreeBlockThreshold;  
 088h ULONG NumberOfHeaps;  
 08Ch ULONG MaximumNumberOfHeaps;  
 090h ULONG ProcessHeaps;  
 094h ULONG GdiSharedHandleTable;  
 098h ULONG ProcessStarterHelper;  
 09Ch ULONG GdiDCAttributeList;  
 0A0h KSPIN\_LOCK LoaderLock;  
 0A4h ULONG OSMajorVersion;  
 0A8h ULONG OSMinorVersion;  
 0ACh USHORT OSBuildNumber;  
 0AEh USHORT OSCSDVersion;  
 0B0h ULONG OSPlatformId;  
 0B4h ULONG ImageSubsystem;  
 0B8h ULONG ImageSubsystemMajorVersion;  
 0BCh ULONG ImageSubsystemMinorVersion;  
 0C0h ULONG ImageProcessAffinityMask;  
 0C4h ULONG GdiHandleBuffer[0x22];  
 14Ch ULONG PostProcessInitRoutine;  
 150h ULONG TlsExpansionBitmap;  
 154h UCHAR TlsExpansionBitmapBits[0x80];  
 1D4h ULONG SessionId;  
 1d8h AppCompatFlags : \_ULARGE\_INTEGER  
 1e0h AppCompatFlagsUser : \_ULARGE\_INTEGER  
 1e8h pShimData : Ptr32 Void  
 1ech AppCompatInfo : Ptr32 Void  
 1f0h CSDVersion : \_UNICODE\_STRING  
 1f8h ActivationContextData : Ptr32 Void  
 1fch ProcessAssemblyStorageMap :Ptr32 Void  
 200h SystemDefaultActivationContextData : Ptr32 Void  
 204h SystemAssemblyStorageMap : Ptr32 Void  
 208h MinimumStackCommit : Uint4B  
} PEB, \*PPEB;

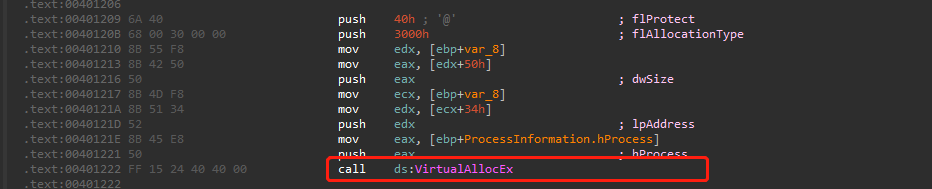
这个程序在0x004011E8的GetProcessAddress手动解析导入函数NtUnmapViewOfSection



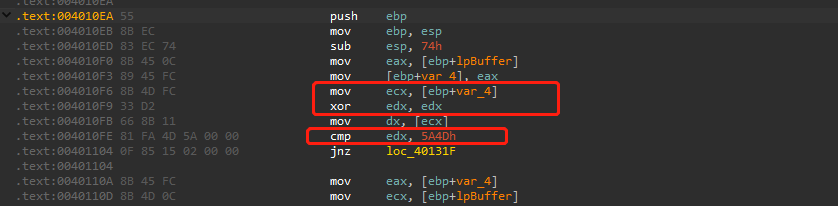
并且在004011FE处，ImageBaseAddress作为UnmapViewOfSection的一个参数传入。UnmapViewOfSection的调用从内存中移除这个被挂起的进程，此时这个车程序将不再执行



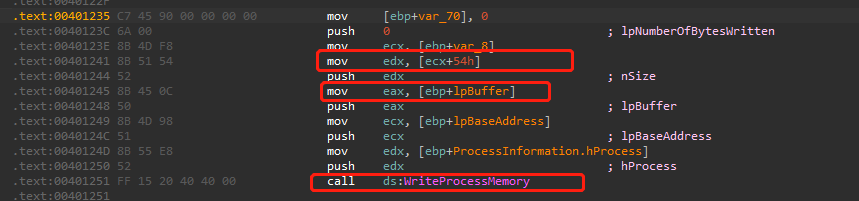
随后，我们可以看到参数被压倒栈上，然后调用VirtualAllocEx，其中作为参数被压入的40h显示了这个程序再被挂起进程的地址空间中分配内存，然后咋ecx+34得位置，这个程序要求内存被分配在PE文件的ImageBase的地址，它告诉Windows加载器这个可执行文件更偏向于加载到哪个位置，随后在eax，[edx+50h]这个程序由PE头imageBase属性指定内存的大小，最终在分配的权限包括可执行以及可写



早在这个程序开始前，程序检查0x004010FE处的magic value值MZ也就是4D5A同理检查0x00401119处的MZ，如果这个检查成功则var\_8指向内存的PE头



一旦这部分内存被分配后，在0x00401251的指令会调用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，要写入的字节数从PE头的0x54处获得，也即是SizeOfHeaders。第一次WriteProcessMemory调用将PE文件头复制到被挂起的进程中，这个行为表明了程序正在移动一个PE文件到另一个进程空间中：



接下来，我们可以看到一个循环，这个循环计数器var\_70在0x00401257处被初始化为0，这个循环计数器用来与00401272位置的第六子街偏移处的值作比较，也就是NumberOfSection。因为这个可执行代码包含一个可执行文件所必须的数据--如代码。数据。重定位等等，所以我们知道这个循环正在赋值PE可执行段到这个挂起线程

具体的进行分析：var\_4变量包含了一个指向内存中PE文件的指针，也就是lpBuffer，他在0x004010F3的位置被初始化，在接下来的0x0040127D的位置，我们看到程序在将MZ头缓冲区的偏移量加上0x3C，也就是到PE头的变异量，这使得ECX现在指向了PE头的位置，于是在后面我们可以看到程序获取了一个指针，在edx这个循环中第一次迭代时的值为0，所以我们可以在指针计算中将EDX移除，这样就只剩ECX和0xF8了

PE文件头的结构中0XF8是Image\_Header\_Section的起始位置，一个乘法将其结构所占字节算出可以告诉我们他占用了40字节，这与16进制值28h符合



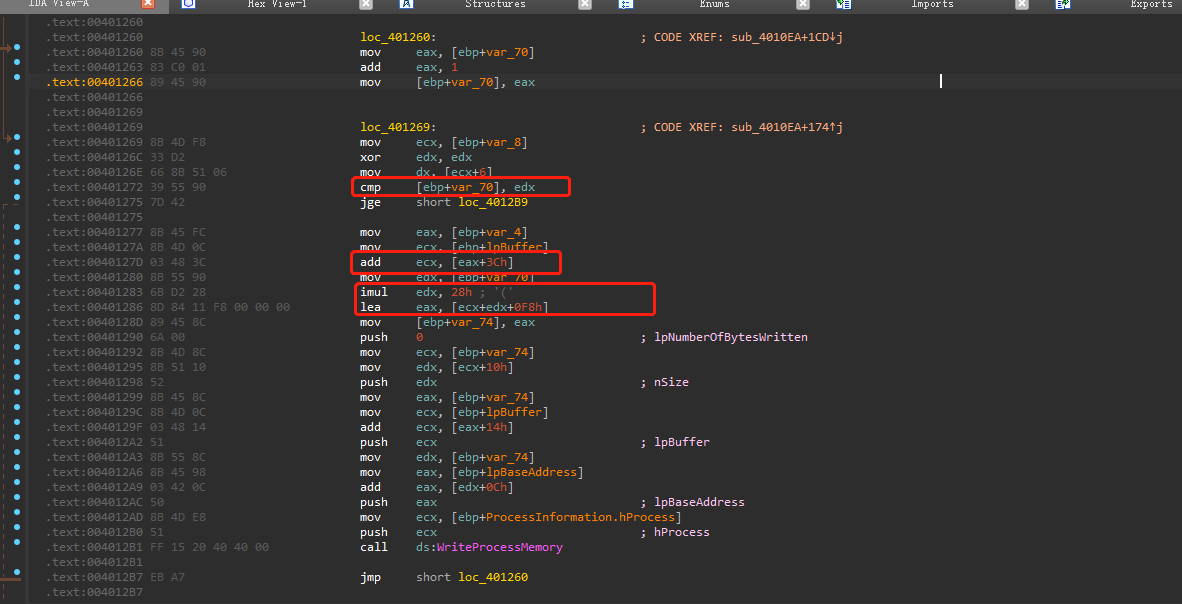
那么，根据前面的分析，我们就可以知道后面的代码进行的工作，起始就是将PE文件对应的每一个段都复制到被挂起的进程空间中，至此，这个程序就完成了加载一个可执行文件到另一个进程的地址空间的工作。在下面的代码中，可以在0x00401272处看到一个循环，这个循环在后面的图片中展示了，他被初始化为0

上面所说的循环计数器将用来与PE头部的第六个字节的偏移作比较，也就是NumberOfSection，因为可执行代码包含一个可执行文件所必须的数据，比如代码、数据、重定位信息等等，所以我们可以知道这个循环可能正在复制PE可执行段到这个挂起进程

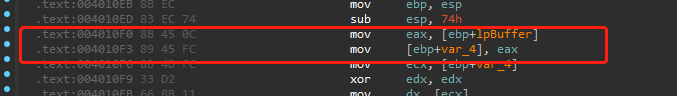
其中var\_4变量包含一个指向内存中PE文件的指针，也就是lpBuffer，这在下面的图片中也展示了其初始化的过程，在0x0040127D的位置这个程序将MZ头的缓冲区偏移量加上0X3C，也就是到PE头的偏移，这让ecx指向了PE头的位置，随后马上一个指针被获取。EDX在这个循环第一次迭代时的值为0，所以现在指针就只剩ECX以及0XF8

0XF8在PE文件头中时是IMAGE\_HEADER\_SECTION数组的位置，这个结构的大小也可以很简单的被计算出为40字节

下面代码的意义就变得清晰了起来，他复制了每一个段，也就是程序赋值每一个段到挂起的进程空间中，于是通过下面的代码就可以完成加载一个可执行文件到另一个进程的地址空间的必要步骤



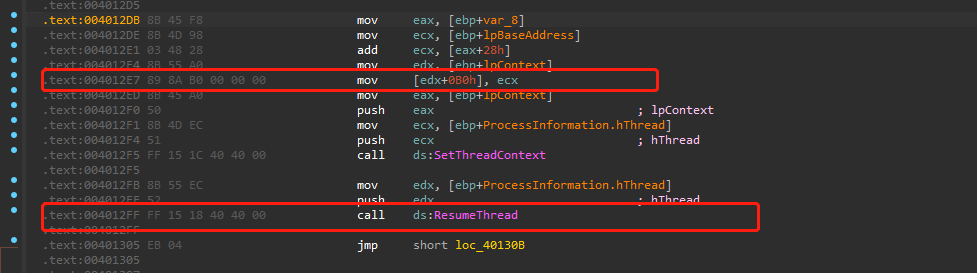
lpBuffer以及var\_4的初始化过程：



对于循环开始次数的初始化

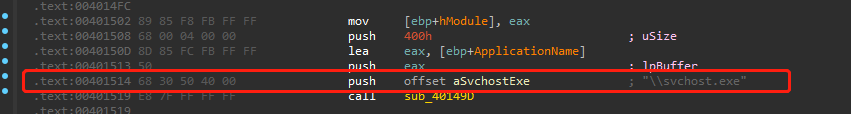


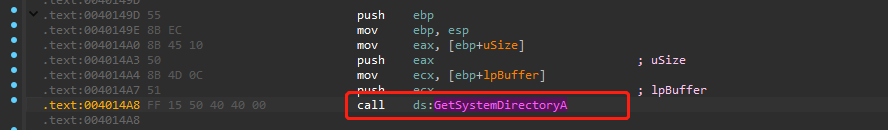
在下面的函数中，我们看到这个程序调用了SetThreadContext，这个函数在0x004012E7处将ecx寄存器加载为被挂起进程内存空间中可执行文件的入口点，如果这个程序在004012FF处执行了ResumeThread函数，他就会成功到达之前CreateProcessA创建的进程替换为另一个进程的目的



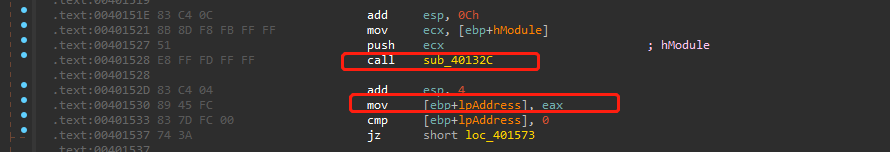
我们经过前面的分析已经明白了这个恶意代码的目的就是完成对挂起进程的替换，那么我们现在就需要知道其到底对哪一个进程完成了进程替换。于是代码中使用的lpApplicationName的来源便成为了重中之重，使用交叉引用可以在00401508的位置找到对该变量的初始化

我们可以发现，在00401514中出现了svchost.exe的字符串，并且在这之后马上将其作为参数压入栈中并调用另外一个函数，起作用就是调用函数获取系统目录，然后将二者拼接，那么我们不难推断出这个恶意代码完成了对svchost.exe的替换



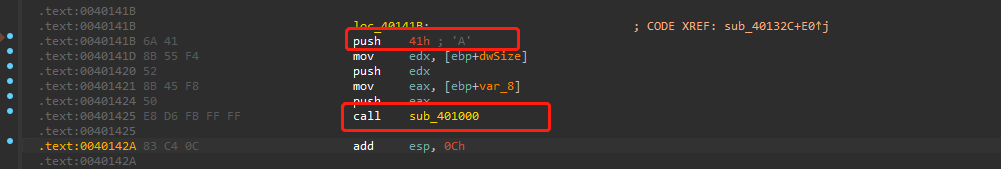


在svchost程序启动后还需要判断要替换的svchost进程，我们通过对0x00401539的变量lpBuffer的跟踪发现可以定位到0x00401521位置，在第一个call位置更新了EAX的值，很明显，他马上使用变量hmoudle即一个指向程序本身的内存指针：

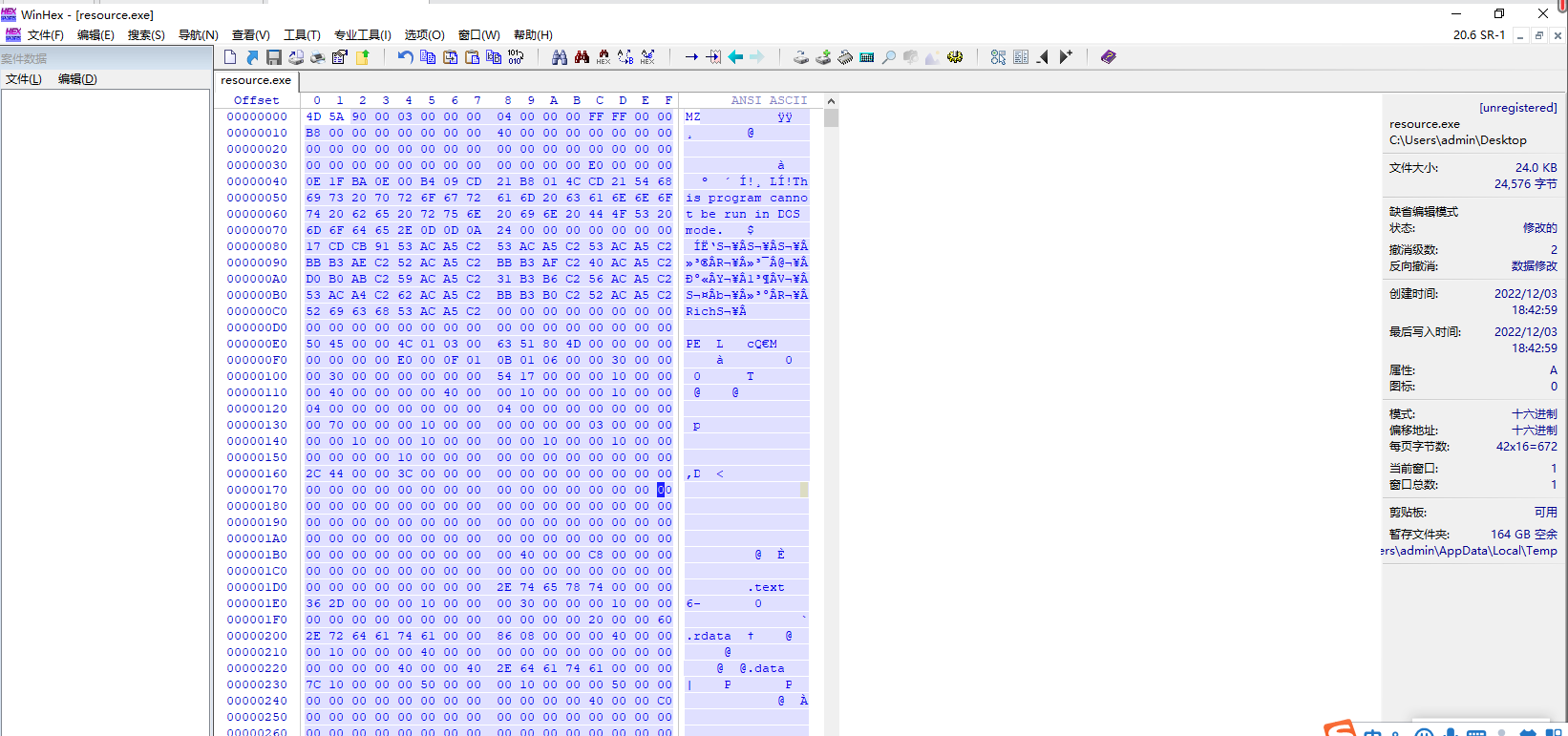


函数40132C调用函数FindResource、LoadResource、LockResource、SizeOfResource、virtualAlloc以及memcpy，这个程序从可执行文件的资源端复制数据到内存中，我们将使用ResourceHacker来查看资源端中的向并将它导出到单独文件中

我们继续查看反汇编代码，在0x00401425的位置，我们看到这个缓冲区被传递给参数sub\_401000，这个函数看起来向一个XOR程序，回顾传递给0x0040141B的第三个参数0x41可以发现异或的密钥就是0x41



那么，我们可以利用工具WINHEX来进行异或解密，发现解密结果是一个exe文件，也就是替换svchost的可执行文件



第三部分：Lab12-03

1、这个恶意负载的目的是什么

**这个程序是一个击键记录器**

2、恶意负载是如何注入自身的

**这个程序使用hook挂钩，窃取击键记录**

3、这个程序还创建了哪些其他文件

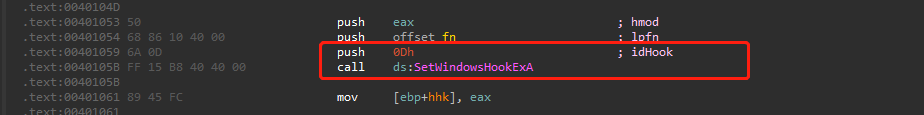
**这个程序创建文件praticalmalwareanalysislog，来保存击键记录**

具体分析过程：

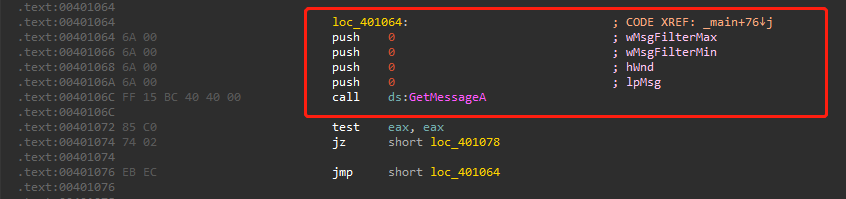
我们首先还是使用基本静态分析来对该恶意代码进行分析，使用strings工具得到的答案不难发现他很有可能是一个击键记录器，大量的特殊键的名字暴露了他的功能



接下来，我们使用IDAPro来对其进行分析：不难观察到，一个挂钩函数在main函数中被调用，其中第一个参数0Dh对应WH\_KEYBOARD\_LL，其使用IDAPro标记为fn的挂钩函数启用键盘事件监控，这个程序很有可能对击键消息进行了记录

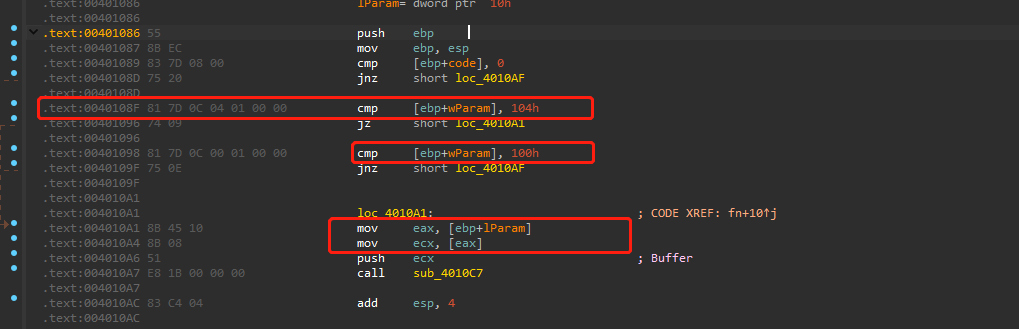


注册该钩子之后，程序在0x00401076处的循环中调用了GetMessageA，在本例中程序一定要调用GetMessageA，因为Windows不会将消息发送到程序进程的钩子函数中，所以一直到函数循环结束才会终止。

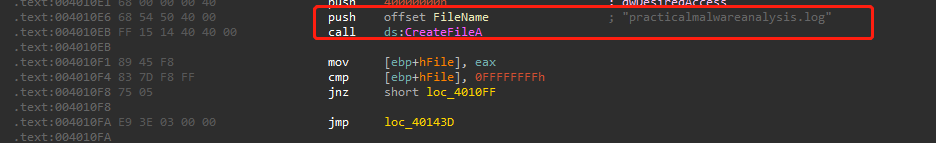


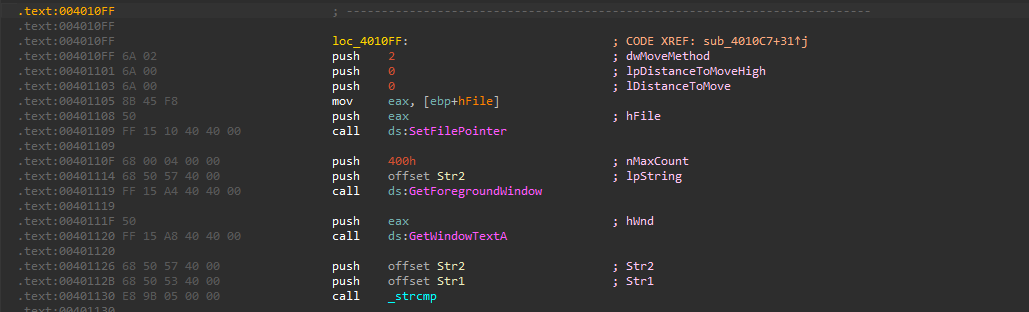
下面我们负fn函数进行分析，fn是一个通用函数，它一共需要3个参数，其中存在一个定义为HOOKPROC的原型。我们可以通过查阅资料确定WH KEYBOARD LL回调函数实际上是LowLevelKeyboardProc回调函数。

为了处理每次按键，程序用cmp指令在O和e处检查按键的类型。程序将虚拟按键码传递到随后以粗体显示的sub 401C7函数中。



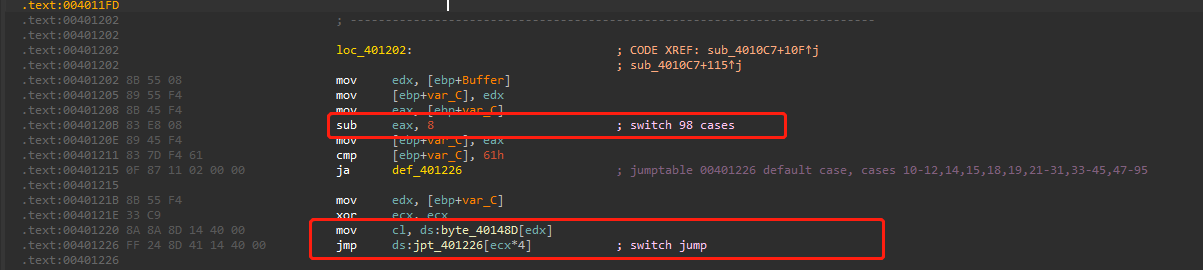
下面我们查看sub 4010C7函数，可以看到程序首先打开文件practicalmalwareanalysis.log，然后恶意代码调用GetForegroundwindow和GetWindowTextA，首先，GetForegroundWindow选择按键按下时的活动窗口，然后使用GetwindowTextA获得窗口的标题。通过这些操作，帮助程序提供按键来源的上下文。



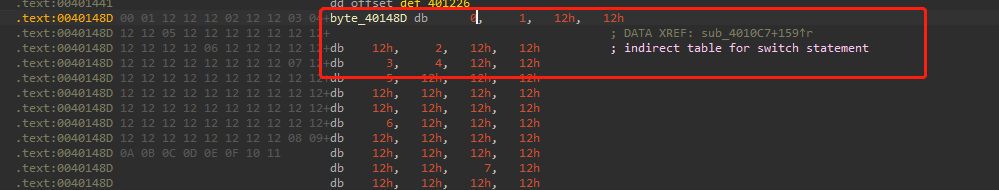


一旦程序将窗口标题写入到日志文件，它就进入一个跳转表，确定变量var C包含传入函数sub 401C7的虚拟按键码后，可以看到虚拟按键码作为一个查询表的索引2。查询表得到的值作为跳转表off401441的一个索引

选择和键盘 SHIFT对应的值，可以在程序中跟随查询过程。从这个值中减去8，剩下的值为0x8(0x10-0x08)。在byte 40148D查找偏移量0x8。可以得到结果为值3，3被存入到ECX中。然后,ECX乘以40,得到0xC,将它作为off 41441的一个偏移量,然后返回到位置10c 401249,这里我们发现字符串[SHIFT1被写入到日志文件中。



于是，我们可以判断他是一个击键记录器，他讲击键记录到文件practicalmalwareanalysisi.log中这个击键记录器使用挂钩来实现相应功能



Lab12-04

1、位置0x401000的代码完成了什么功能？

**恶意代码判断进程是否为winlogon.exe进程**

2、代码注入了哪个进程？

**winlogon.exe**

3、使用LoadLibraryA装载了哪个DLL程序？

**sfc\_os.dll被装载，他是用来禁用Windows的文件保护机制**

4、传递给CreateRemoteThread调用的第四个参数是什么

**函数指针，指向sfcosdll中一个未命名的序号为2的函数**

5、二进制主程序释放出了哪个恶意代码

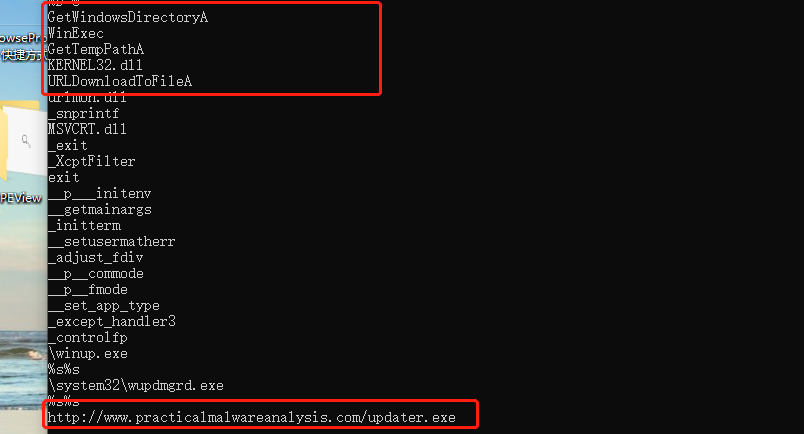
**恶意代码从资源段中释放一个二进制文件，并且将这个二进制文件覆盖旧的 Windows可执行文件wupdmgr.exe。覆盖真实的wupdmgr.exe之前，恶意代码将它复制到一个中间目录，供以后使用。**

6、释放出的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.dl 的一个序号为2的导出函数，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，并且调用原始的二进制文件来特洛伊木马化wupdmgr.exe文件。**

具体分析过程：

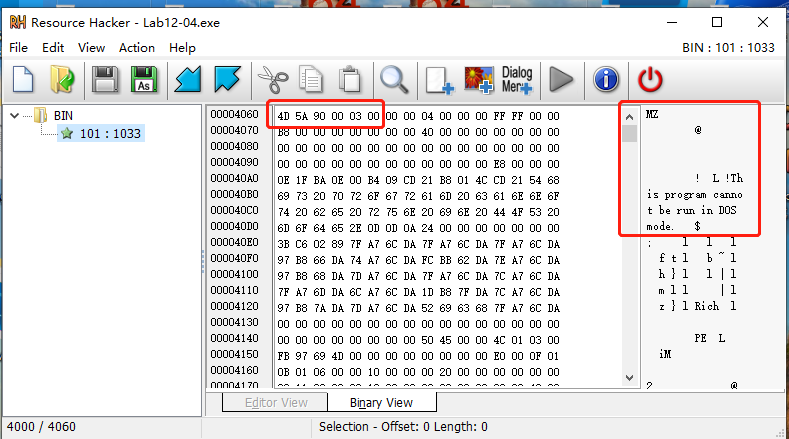
首先，进行基本静态分析和基本动态分析，检查恶意代码的导入函数表，我们看到CreateRemoteThread，但是没有WriteProcessMemory或VirtualAllocEx这些更有意思的函数，这就比较奇怪了，那么他是怎么完成替换的过程就需要我们去重点关注，同时，我们也看到有资源操作函数的导入，例如LoadResource和FindResourceA等。用ResourceHacker检查恶意代码，我们注意到一个名为BIN的程序存储在资源段中。



从上图的URLDownLoad函数以及网址我们就不难得出恶意代码会从该网址完成恶意文件或代码的下载工作

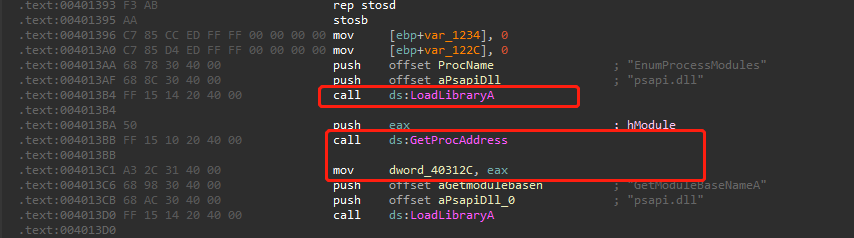


并且很显然，资源节中的内容是一个可执行文件或者dll文件，因为他很明显具有PE文件结构：

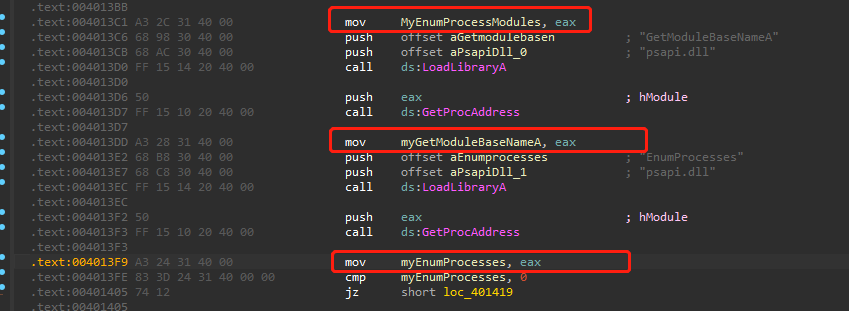


如果使用基本动态分析，我们会发现他实际上会对一个网页进行不停的访问，并且根据procmon，恶意代码创建一个文件，并且覆盖了位于系统目录下的wupdmgr.exe的Windows更新二进制文件。比较恶意代码释放的wupdmgrexe和恶意代码资源节中BIN文件，我们看到它们是相同的。

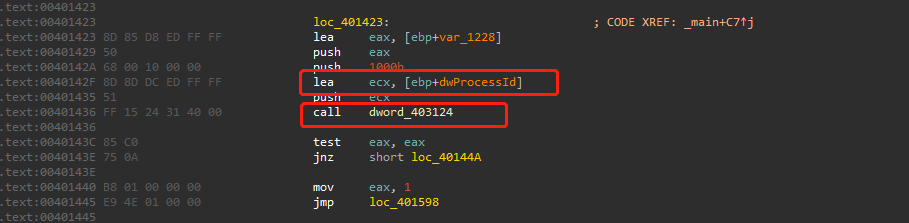
于是，接下来我们完成一些高级分析，我们将恶意代码载入IDAPro中，在main函数开始的几行中，我们看到恶意代码从psapi.dl1中解析Windows进程枚举的函数，同样我们也可以很轻松的观察到恶意代码使用LoadLibraryA和GetProcAddress手工解析的列举出的函数



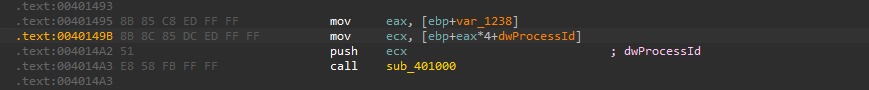
我们可以手动的将几个地址Rename为其函数名，这样以后我们可以很方便的识别他们：



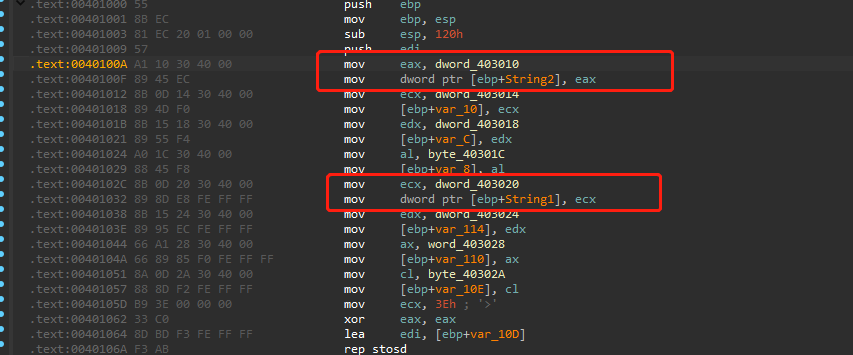
之后恶意代码检查这些函数指针的值，当达到0x00401423和myEnumProcesses调用时，所列代码的目的是返回系统进程PID的数组。数组被局部变量dwProcessId所引用



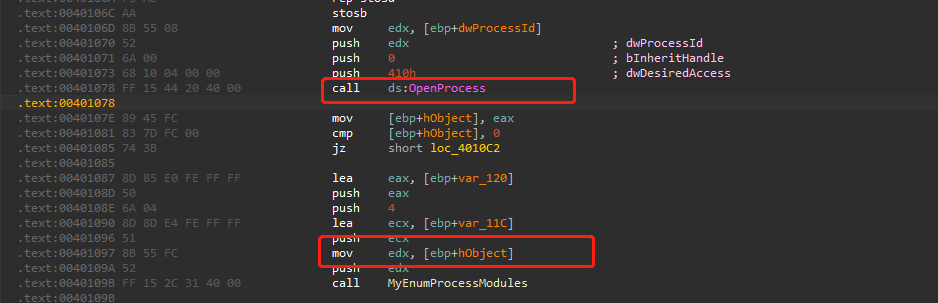
然后，恶意代码开始循环遍历PID，并且将每个进程的PID传递给0x00401000的子例程，可以看到一个到dwProcessId引用数组的索引，在调用sub 40100之前计算它。



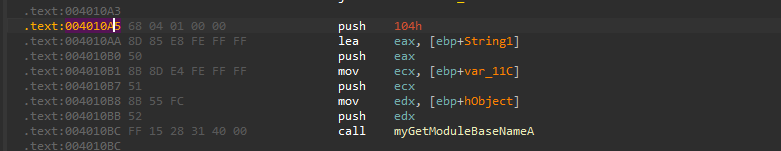
我们检查sub4010函数时可以看到两个局部变量集合分别是string1和string2，通过对16进制字符串的查表转换，我们不难得出，其二者分别对应字符串“winlogon.exe”和""



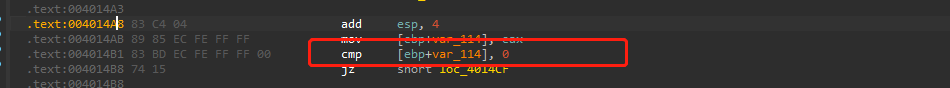
接下来，恶意代码将循环中的参数(dwProcessId)传递到OpenProcess，来获得进程的句柄。OpenProcess返回的句柄j将被存入到EAX，并且传递给00401097处的myEnumProcessModules函数，这个函数将返回加载到这个进程所有模块的句柄数组

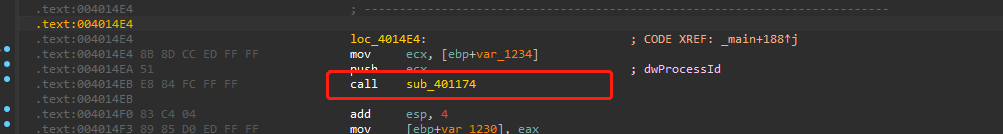


恶意代码在后面还试图使用GetModuleBaseNameA来获得模块PID的基本名称。如果成功，str1将会包含传递给子例程的模块PID的基本名称的字符串，如果失败，它就会保持初始值""。原有的初始化字符串""应该含有GetModuleBaseNameA返回的基本模块的名字。将这个字符串与字符串"winlogon.exe"比较，如果相同，EAX将会等于0，函数返回时EAX将会等于1。如果不相同，则函数返回值EAX等于0现在我们可以有把握地说，sub41000函数正在确定与winlogonexe关联的PID。

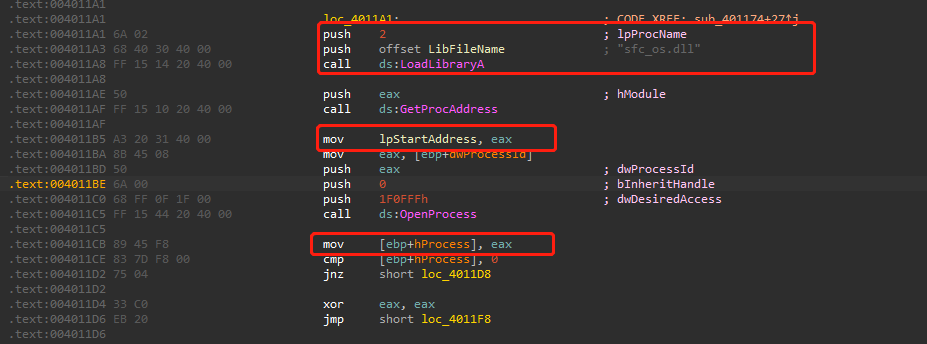


现在，我们已经知道了sub401000函数所做的操作，而在恶意代码中，他还需要测试EAX中的返回值，来查看它是否等于1，如果等于1，代码跳转到loc\_4014CF，递增循环计数器并且用新的PID运行PIDLookup。否则，如果PID与winlogonexe相匹配，则PID将被传递给sub481174

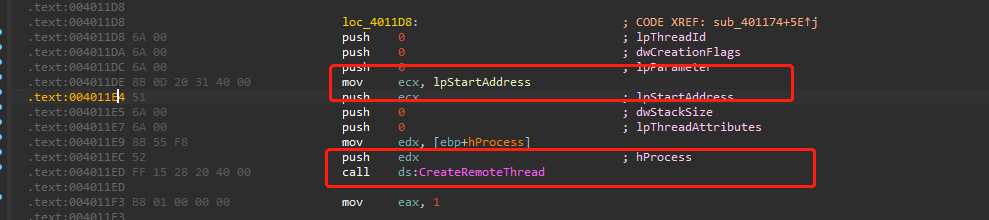




我们马上对sub401174函数进行分析，我们看到它用参数SeDebugPrivilege立即调用另外一个例程。跟着SeDebugPrivilege提取函数，我们看到字符串sfcosd11传递给LoadLbraryA，接下来，用sfcosdll的句柄和序号2调用GetProcAddress。序号2被压入到堆栈。序号2的函数指针被存入到lpStartAddress。然后，恶意代码用winlogonexe的PID和值为0xIFOFFF的dwDesiredAccess，调用OpenProcess函数。winlogonexe的句柄被保存到hProcess中

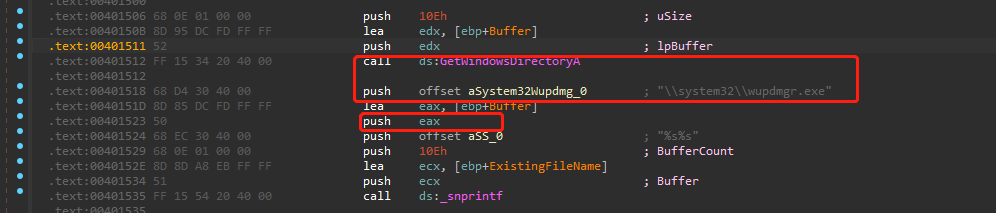


随后，函数调用CreateRemoteThread，检查CreateRemoteThread的参数，在0处我们看到hProcess参数是EDX，即winlogonexe句柄。在004011DE位置处传入的IpStartAddress是sfcosdll中序号2函数的指针，它负责向winlogonexe注入一个线程，但因为sfc\_os.dll已经载入到winlogonexe中，在新创建的线程中不需要加载这个DLL，所以没有WriteProcessMemory调用。该线程是sfcosdll中序号为2的函数。

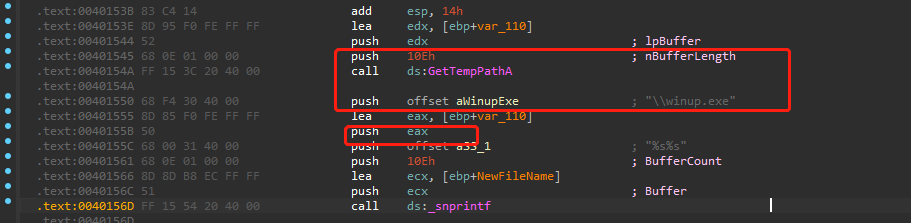


sfcosdl部分负责Windows文件保护机制，它有一系列运行在winlogonexe中的线程。sfc osdl的序号2函数是一个未公开的导出函数，它被称为SfcTerminatewatcherThread。如果星耀要成功地运行sfcTerminateWatcherThread函数，则必须将它运行在winlogonexe中。强制运行sfcTerminatewatcherThread函数，恶意代码可以在下次系统重启之前，禁用Windows文件保护机制。如果正确地注入线程，则代码代码会运行并且创建一个字符串。运行代码清单中的代码，恶意代码中的GetWindowsDirectoryA会返回当前Windows目录，

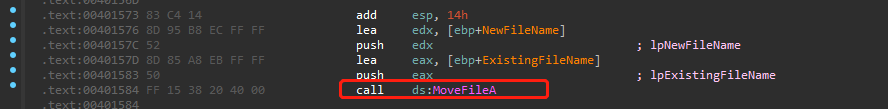
同时，恶意代码也会将这个字符串和\system32\wupdmgrexe传递给 snprintf调用，如00401518和00401523所示这段代码很显然是构建字符串"C:\windows\system32\wupdmgr.exe”，构建的字符串被存储在ExistingFileName中在WindowsXP系统下，wupdmgrexe被用作Windows系线的更新。



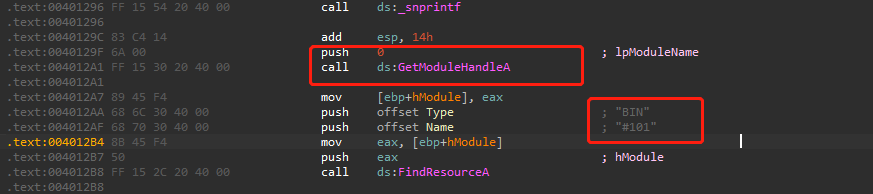
在分析后面的代码过程中我们看到另一个正在构建的字符串。0040154A处的GetTempPathA调用为我们提供了当前用户临时目录的指针。随后临时目录的路径和参数\lwinup.exe一起传递给另外一个 snprintf调用，如0040155B和00401550所示，这个\_snprintf调用创建了字符"C:\Documents and Settings\username\Local\Temp\winup.exe"并且存储在NewFileName中。



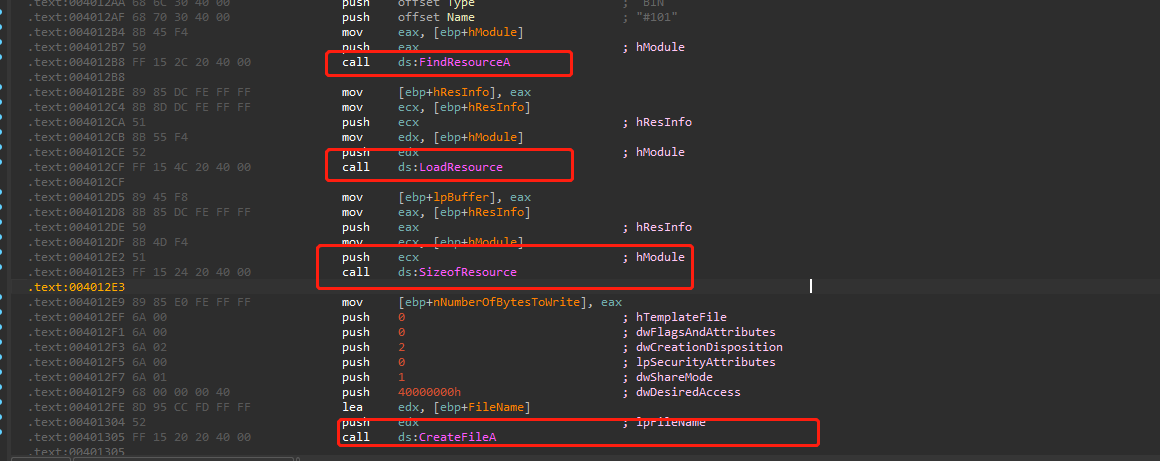
MoveFileA调用中使用了两个局部变量，MoveFileA将把Windows更新二进制文件移到用户的临时目录中。



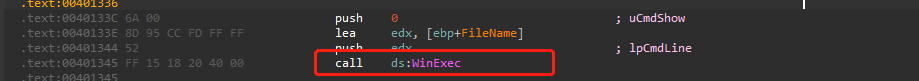
我们接下来继续按照顺序分析恶意代码，我们看到恶意代码在004012A1的位置调用了GetModuleHandleA，GetModuleHandlel返回当前进程的模块柄。随后，我们看到一系列的资源段操作API，特别是带有参数#101和BIN的FindResourceA调用。这与我们前面基础分析过程所猜想的结果一样，恶意代码正在提取它的资源节到硬盘上。



在这个函数中，FindResourceA调用之后便调用LoadResource、SizeofResourceCreateFileA以及writeFile，这些函数调用的组合，从资源段BIN中提取文件，并且写入到文件C:WindowslSwsem32\wupdmgrexe中在本实验中，恶意代码是创建一个新的Window更新处理程序。在正常情况下，因为Windows文件保护机制会探测到文件的改变以及用一-个新创建文件罚盖，所以恶意代码尝试创建一个新的更新程序通常会失败。但是，因为恶意代码禁用了Windows文件保护机制的功能，所以它可以覆盖通常受保护的Windows二进制文件



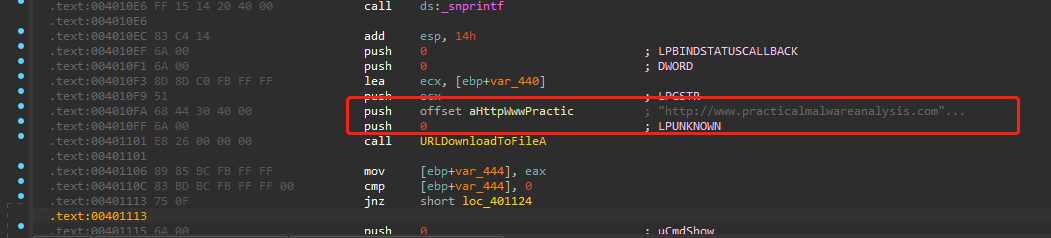
这个函数做的最后一件事情，就是使用winExec启动新的wupdmgrexe。为了隐藏这个程序的窗口，用0或者SW\_HIDE作为uCmdShow参数的值来启动



接下来，我们可以使用ResourceHacker将资源节的二进制文件提取出来进行单独分析，恶意代码创建了一个临时目录中的字符串来移动原始的Windows 更新二进制文件，然后运行原始的Windows更新二进制文件(使用winExec)，此时原始Windows更新二进制文件被保存到用户临时目录中。如果用户想要执行一个Windows更新，似平一切正常运转，原始的Windows更新文件也将会运行。

接下来，在IDA Pro中，我们在0x4010C3开始看到构建的字符串:\indows\system32\wupdmgrd.exe，它被存储在局部变量Dest中。除了文件名中的d，这个字符串与原始Windows更新二进制文件的文件名十分接近。

004010FA处的参数sZURL被设置为<http://www,practicalmalwareanalysis.com/updater.exe>。004010F3处的参数szFileName被置为Dest(C:\Windows\system32\wupdmgrd.exe)。恶意代码自己完成更新，并下载更多的恶意代码。下载的文件updater.exe被保存到wupdmgrd.exe中。恶意代码将URLDownloadToFileA的返回值与0比较，查看这个函数调用是否失败。如果返回值不等于0，则恶意代码会运行新创建的文件，然后二进制文件返回并且退出。



第二部分：YARA规则：

Lab12-01.exe的YARA规则

import "pe"
  
rule IsPE{
  
 meta:
  
 description = "检查文件是否为PE文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 uint16(0) == 0x5A4D and //“MZ”头
  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头
  
  
}
  
  
rule smallFileSize{
  
 meta:
  
 description = "检查文件小是否大概率为exe文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 filesize<500KB
  
}
  
  
rule mayBeMaleware{
  
 meta:
  
 description = "检查文件小是否大概率为maleware文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = { E8 00 00 00 00 }
  
 $b = "malware" nocase
  
 condition:
  
 $a at pe.entry\_point or $b
  
}
  
  
rule Lab1201exe{
  
 meta:
  
 description = "检查改文件是否大概率与Lab012-01exe的文件属于同一类型的恶意代码"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = "EnumProcessModule"
  
 $b = "psapi.dll"
  
 $c = "EnumProcesses"
  
 $d = "explorer.exe"
  
 $e = "VirtualAlloc"
  
 $f = "GETACP"
  
 $g = "GETOEMCP"
  
 $h = "Lab12-01.dll"
  
 $i = "LoadLibraryA"
  
 $j = "GetProcAddress"
  
 condition:
  
 (7 of them) and (IsPE and smallFileSize) or (mayBeMaleware)
  
}

Lab12-02.DLL的YARA规则

import "pe"
  
rule IsPE{
  
 meta:
  
 description = "检查文件是否为PE文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 uint16(0) == 0x5A4D and //“MZ”头
  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头
  
  
}
  
  
rule smallFileSize{
  
 meta:
  
 description = "检查文件小是否大概率为exe文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 filesize<500KB
  
}
  
  
rule mayBeMaleware{
  
 meta:
  
 description = "检查文件小是否大概率为maleware文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = { E8 00 00 00 00 }
  
 $b = "malware" nocase
  
 condition:
  
 $a at pe.entry\_point or $b
  
}
  
  
rule Lab1201dll{
  
 meta:
  
 description = "检查改文件是否大概率与Lab012-01dll的文件属于同一类型的恶意代码"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = "Practical Malware Analysis %d"
  
 $b = "Press OK to reboot"
  
 $c = "WriteFile"
  
 $d = "InitiallizeCriticalSection"
  
 $e = "MessageBoxA"
  
 $f = "GETACP"
  
 $g = "GETOEMCP"
  
 $h = "H:mm:ss"
  
 $i = "dddd, MMMM dd, yyyy"
  
 $j = "SunMonTueWedThuFriSat"
  
 $k = "JanFebMarAprMayJunJulAugSepOctNovDec"
  
 $l = "Sleep"
  
 $m = "user32.dll" nocase
  
 condition:
  
 (9 of them) and (IsPE and smallFileSize) or (mayBeMaleware)
  
}

Lab12-02.exe的YARA规则

import "pe"
  
rule IsPE{
  
 meta:
  
 description = "检查文件是否为PE文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 uint16(0) == 0x5A4D and //“MZ”头
  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头
  
  
}
  
  
rule smallFileSize{
  
 meta:
  
 description = "检查文件小是否大概率为exe文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 filesize<500KB
  
}
  
  
rule mayBeMaleware{
  
 meta:
  
 description = "检查文件小是否大概率为maleware文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = { E8 00 00 00 00 }
  
 $b = "malware" nocase
  
 condition:
  
 $a at pe.entry\_point or $b
  
}
  
  
rule Lab1202exe{
  
 meta:
  
 description = "检查改文件是否大概率与Lab012-02exe的文件属于同一类型的恶意代码"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = {AA [0-400] AA}
  
 $b = "LOCALIZATION"
  
 $c = "UNICODE"
  
 $d = "NTDLL.DLL" nocase
  
 $e = "\svchost.exe"
  
 $f = "NtUnmapViewOfSection"
  
 $g = "LoadLibraryA"
  
 $h = "GetACP"
  
 $i = "Sleep"
  
 $j = "FindResourceA"
  
 $k = "LoadResource"
  
 $l = "LockResource"
  
 $m = "SizeofResource"
  
 $n = "WriteProcessMemory"
  
 $o = "SetThreadContext"
  
 $p = "ResumeThread"
  
 $q = "VirtualAllocEx"
  
 condition:
  
 (11 of them) and (IsPE and smallFileSize) or (mayBeMaleware)
  
}

Lab12-03.exe的YARA规则

import "pe"
  
rule IsPE{
  
 meta:
  
 description = "检查文件是否为PE文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 uint16(0) == 0x5A4D and //“MZ”头
  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头
  
  
}
  
  
rule smallFileSize{
  
 meta:
  
 description = "检查文件小是否大概率为exe文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 filesize<500KB
  
}
  
  
rule mayBeMaleware{
  
 meta:
  
 description = "检查文件小是否大概率为maleware文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = { E8 00 00 00 00 }
  
 $b = "malware" nocase
  
 condition:
  
 $a at pe.entry\_point or $b
  
}
  
  
rule Lab1203exe{
  
 meta:
  
 description = "检查改文件是否大概率与Lab012-03exe的文件属于同一类型的恶意代码"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = "[CAPS LOCK]"
  
 $b = "[DEL]"
  
 $c = "[TAB]"
  
 $d = "[CTRL]"
  
 $e = "[BACKSPACE]"
  
 $f = "practicalmalwareanalysis.log"
  
 $g = "ConsoleWindowClass"
  
 $h = "LoadLibraryA"
  
 $i = "GetProcAddress"
  
 $j = "VirtualAlloc"
  
 $k = "GetCurrentProcess"
  
 $l = "USER32.DLL" nocase
  
 $m = "CallNextHookEx"
  
 $n = "UnhookWindowsHookEx"
  
 $o = "SetWindowsHookExA"
  
 condition:
  
 (10 of them) and (IsPE and smallFileSize) or (mayBeMaleware)
  
}

Lab12-04.exe的YARA规则

import "pe"
  
rule IsPE{
  
 meta:
  
 description = "检查文件是否为PE文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 uint16(0) == 0x5A4D and //“MZ”头
  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头
  
  
}
  
  
rule smallFileSize{
  
 meta:
  
 description = "检查文件小是否大概率为exe文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 condition:
  
 filesize<500KB
  
}
  
  
rule mayBeMaleware{
  
 meta:
  
 description = "检查文件小是否大概率为maleware文件"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = { E8 00 00 00 00 }
  
 $b = "malware" nocase
  
 condition:
  
 $a at pe.entry\_point or $b
  
}
  
  
rule Lab1204exe{
  
 meta:
  
 description = "检查改文件是否大概率与Lab012-04exe的文件属于同一类型的恶意代码"
  
 author="wbf"
  
 date="2022-12-3"
  
 strings:
  
 $a = "http://www.practicalmalwareanalysis.com/updater.exe"
  
 $b = "\system32\wupdmgrd.exe"
  
 $c = "\winup.exe"
  
 $d = "exit"
  
 $e = "urlmon.dll"
  
 $f = "WinExec"
  
 $g = "URLDownloadToFileA"
  
 $h = "GetTempPathA"
  
 $i = "<not real>"
  
 $j = "SeDebugPrivilege"
  
 $k = "sfc\_os.dll"
  
 $l = "BIN"
  
 $m = "#101"
  
 $n = "winlogon.exe"
  
 $o = "EnumProcessModules"
  
 $p = "GetModuleBaseNameA"
  
 $q = "EnumProcesses"
  
 condition:
  
 (12 of them) and (IsPE and smallFileSize) or (mayBeMaleware)
  
}

第三部分：IDAPython脚本自动化编写

代码1：

def find\_function\_arg(addr):
  
 while True:
  
 addr = idc.PrevHead(addr)
  
 if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
  
 print “We found it at 0x%x” % GetOperandValue(addr, 1)
  
 break

代码2：

def get\_string(addr):
  
 out = ""
  
 while True:
  
 if Byte(addr) != 0:
  
 out += chr(Byte(addr))
  
 else:
  
 break
  
 addr += 1
  
 return out

代码3：

"""
  
summary: decompile & print current function.
  
"""
  
  
from \_\_future\_\_ import print\_function
  
  
import ida\_hexrays
  
import ida\_lines
  
import ida\_funcs
  
import ida\_kernwin
  
  
def main():
  
 if not ida\_hexrays.init\_hexrays\_plugin():
  
 return False
  
  
 print("Hex-rays version %s has been detected" % ida\_hexrays.get\_hexrays\_version())
  
  
 f = ida\_funcs.get\_func(ida\_kernwin.get\_screen\_ea());
  
 if f is None:
  
 print("Please position the cursor within a function")
  
 return True
  
  
 cfunc = ida\_hexrays.decompile(f);
  
 if cfunc is None:
  
 print("Failed to decompile!")
  
 return True
  
  
 sv = cfunc.get\_pseudocode();
  
 for sline in sv:
  
 print(ida\_lines.tag\_remove(sline.line));
  
  
 return True
  
  
main()

代码4：

"""
  
summary: a focus on the 'curpos' hook, printing additional details about user input
  
  
description:
  
 Shows how user input information can be retrieved during
  
 processing of a notification triggered by that input
  
  
see\_also: vds\_hooks
  
"""
  
  
import ida\_hexrays
  
import ida\_kernwin
  
  
class curpos\_details\_t(ida\_hexrays.Hexrays\_Hooks):
  
 def curpos(self, v):
  
 parts = ["cpos={lnnum=%d, x=%d, y=%d}" % (v.cpos.lnnum, v.cpos.x, v.cpos.y)]
  
 uie = ida\_kernwin.input\_event\_t()
  
 if ida\_kernwin.get\_user\_input\_event(uie):
  
 kind\_str = {
  
 ida\_kernwin.iek\_shortcut : "shortcut",
  
 ida\_kernwin.iek\_key\_press : "key\_press",
  
 ida\_kernwin.iek\_key\_release : "key\_release",
  
 ida\_kernwin.iek\_mouse\_button\_press : "mouse\_button\_press",
  
 ida\_kernwin.iek\_mouse\_button\_release : "mouse\_button\_release",
  
 ida\_kernwin.iek\_mouse\_wheel : "mouse\_wheel",
  
 }[uie.kind]
  
  
 #
  
 # Retrieve input kind-specific information
  
 #
  
 if uie.kind == ida\_kernwin.iek\_shortcut:
  
 payload\_str = "shortcut={action\_name=%s}" % uie.shortcut.action\_name
  
 elif uie.kind in [
  
 ida\_kernwin.iek\_key\_press,
  
 ida\_kernwin.iek\_key\_release]:
  
 payload\_str = "keyboard={key=%d, text=%s}" % (uie.keyboard.key, uie.keyboard.text)
  
 else:
  
 payload\_str = "mouse={x=%d, y=%d, button=%d}" % (
  
 uie.mouse.x,
  
 uie.mouse.y,
  
 uie.mouse.button)
  
  
 #
  
 # And while at it, retrieve a few extra bits from the
  
 # source QEvent as well, why not
  
 #
  
 qevent = uie.get\_source\_QEvent()
  
 qevent\_str = str(qevent)
  
 from PyQt5 import QtCore
  
 if qevent.type() in [
  
 QtCore.QEvent.KeyPress,
  
 QtCore.QEvent.KeyRelease]:
  
 qevent\_str="{count=%d}" % qevent.count()
  
 elif qevent.type() in [
  
 QtCore.QEvent.MouseButtonPress,
  
 QtCore.QEvent.MouseButtonRelease]:
  
 qevent\_str="{globalX=%d, globalY=%d, flags=%s}" % (
  
 qevent.globalX(),
  
 qevent.globalY(),
  
 qevent.flags())
  
 elif qevent.type() == QtCore.QEvent.Wheel:
  
 qevent\_str="{angleDelta={x=%s, y=%s}, phase=%s}" % (
  
 qevent.angleDelta().x(),
  
 qevent.angleDelta().y(),
  
 qevent.phase())
  
  
 #
  
 # If the target QWidget is a scroll area's viewport,
  
 # pick up the parent
  
 #
  
 from PyQt5 import QtWidgets
  
 qwidget = uie.get\_target\_QWidget()
  
 if qwidget:
  
 parent = qwidget.parentWidget()
  
 if parent and isinstance(parent, QtWidgets.QAbstractScrollArea):
  
 qwidget = parent
  
  
 parts.append("user\_input\_event={kind=%s, modifiers=0x%x, target={metaObject={className=%s}, windowTitle=%s}, source=%s, %s, source-as-qevent=%s}" % (
  
 kind\_str,
  
 uie.modifiers,
  
 qwidget.metaObject().className(),
  
 qwidget.windowTitle(),
  
 uie.source,
  
 payload\_str,
  
 qevent\_str))
  
 print("### curpos: %s" % ", ".join(parts))
  
 return 0
  
  
curpos\_details = curpos\_details\_t()
  
curpos\_details.hook()