

**《恶意代码分析与防治技术》课程实验报告**

**实验12**

****

学 院 网络空间安全学院

专 业 信息安全

学 号 2112060

姓 名 孙蕗

1. **实验目的**

对不同类型的恶意代码进行全面的静态和动态分析，包括 DLL、反调试、击键记录和 Windows 更新劫持。

学习使用 IDA Pro、Yara 规则、动态分析工具，提升对恶意代码的分析技能。

培养对恶意代码的深入理解和分析技能，提高对恶意行为的防范能力。

1. **实验原理**

IDA Pro 进行分析查看导入表、字符串和函数调用等

导入表分析关注反调试和反虚拟机相关的 API

使用 Procmon 进行动态分析，观察文件操作、注册表访问等行为

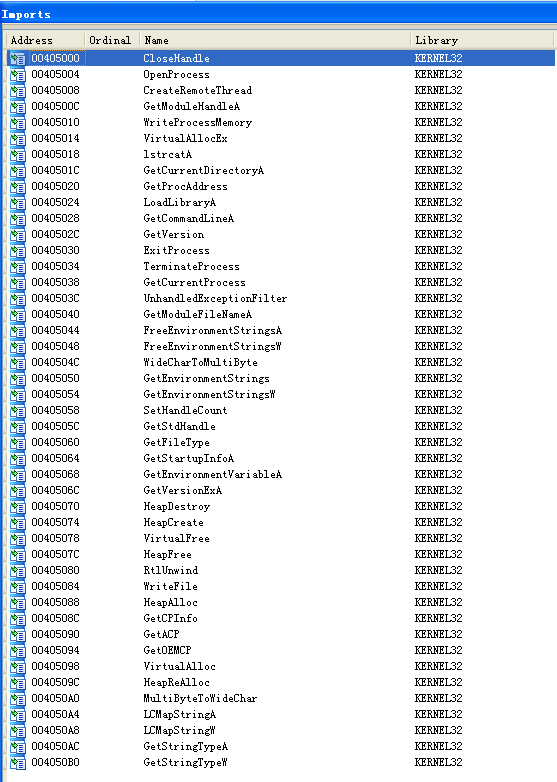
分析 LowLevelKeyboardProc 回调函数，查看击键记录的处理过程。

劫持 Windows 更新，禁用文件保护机制，替换系统文件，实现特洛伊木马功能。

恶意代码可能会尝试检测虚拟环境或调试器，以逃避分析和检测。

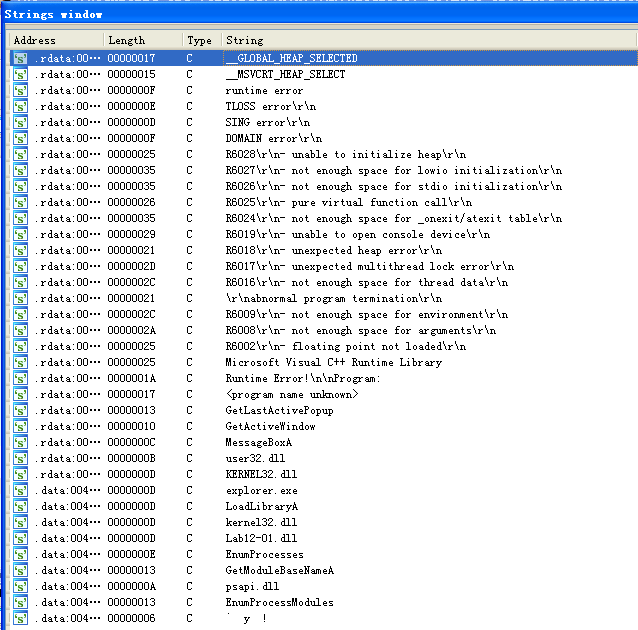
1. **实验过程**
2. **Lab12-1**

Ida pro打开Lab12-01.exe，查看导入函数



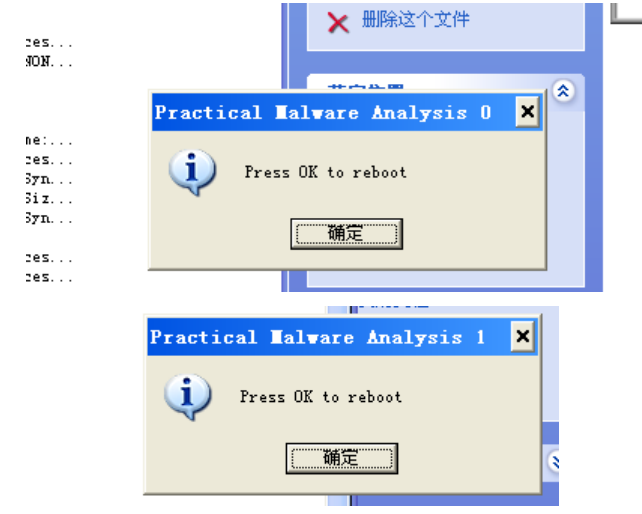
发现有CreateRemoteThread、WriteProcessMemory以及VirtualAllocEx，可能存在进程注入行为。

查看Lab12-01.exe的字符串

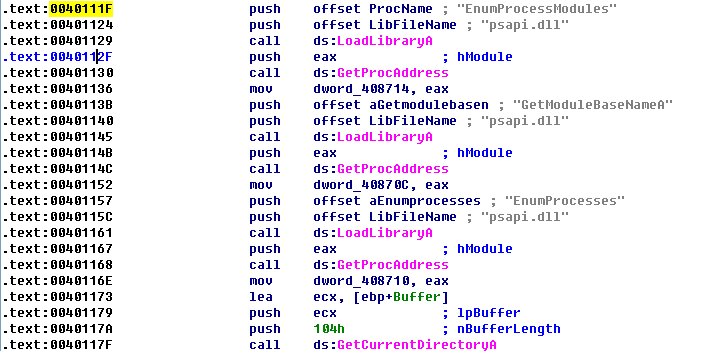


看到需要注意如下一些字符串，包括explorer.exe、Lab12-01.dll以及psapi.dll。

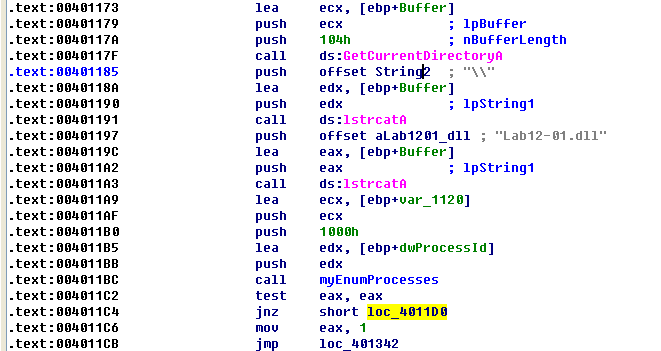
运行这个恶意代码时，它每分钟会创建一个消息框。同时可以看见这个提示框上面的数字在每次提示的时候都会增加。

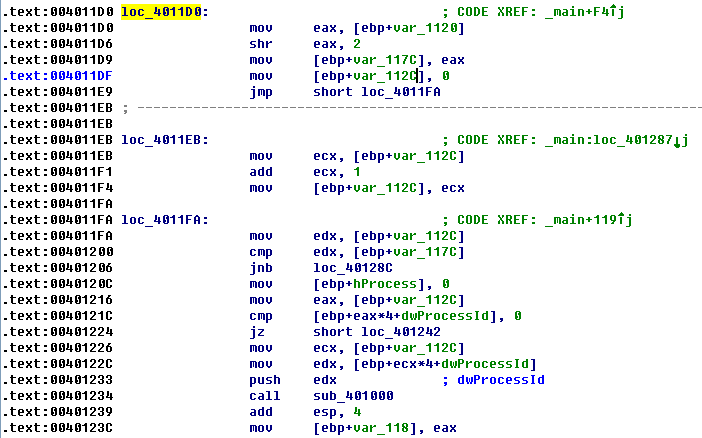


查看main函数

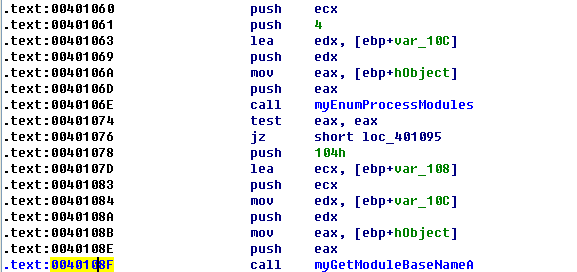


这个恶意代码保存指向dword\_408714、dword\_40870C以及dword\_408710的函数指针。重命名这些全局变量为myEnumProcessModules、myGetModuleBaseNameA,以及myEnumProcesses。这段代码调用dword\_408710,获取在系统中每一个进程对象的PID。EnumProcesses返回一个由局部变量dwProcessId引用的PID数组。dwProcessId被用来在一个循环中迭代进程列表，并对每一个PID调用sub\_401000。

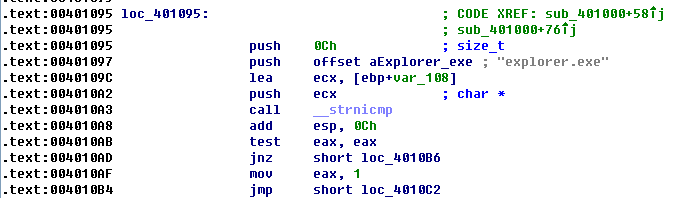




查看sub\_401000,这个动态解析的导入函数EnumProcessModules在PID被传给OpenProcess函数后被调用。接下来，0x0040108F处有一个对dword\_40870C(GetModuleBaseNameA)的调用。

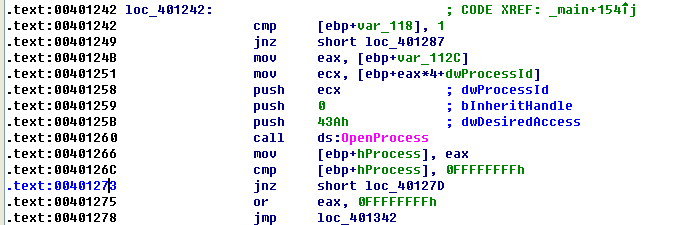


这个动态解析的函数GetModuleBaseNameA被用来将PID翻译为进程名。

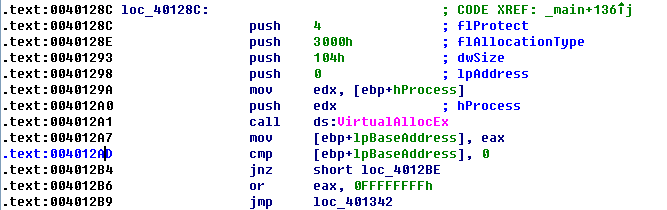


0x004010A3处看到通过GetModuleBaseNamA获得的字符串(Str1)与explorer.exe(Str2)之间的比较。可以看出，这个恶意代码在内存中查找explorer.exe进程。

如果explorer.ere被找到，sub\_401000函数将返回1,并且这个main函数将调用OpenProcess,来打开一个指向它的句柄。



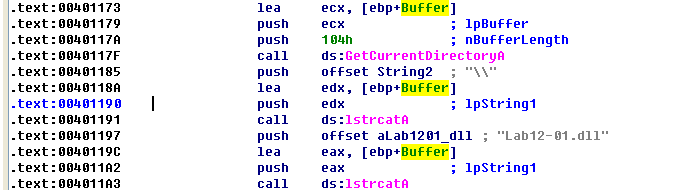
如果这个恶意代码成功地获取这个进程的句柄，下面的代码将会执行。句柄hProcess将被用来操纵这个进程。





VirtualAllocEx的调用动态地在explorer.exe中分配内存。0x104字节压入dwSize而被分配。如果VirtualAllocEx成功，一个指向被分配内存的指针被移动到lpParameter中，并在call ds:WriteProcessMemory处被和进程句柄一起传给WriteProcessMemory,以便向explorer.exe写入数据。写入这个进程的数据被0x004012C5处的Buffer参数所引用。

跟踪代码回溯到Buffer被设置的地方，发现它被设置成当前目录，并追加了Lab12-01.dll的一个路径。可以断定这个恶意代码将Lab12-01.dll的路径写入到explorer.exe进程中。



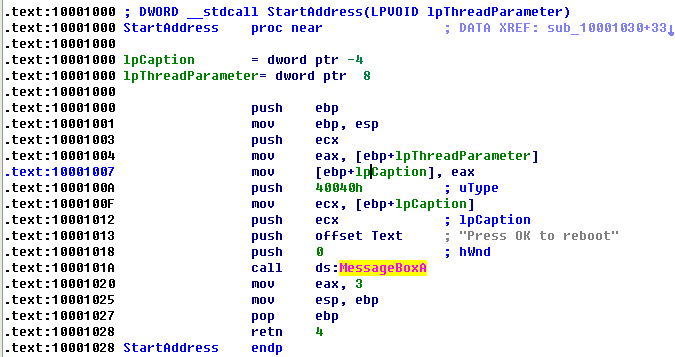
如果这个恶意代码成功地将这个DLL的路径写入到explorer.exe中,下面的代码将会执行



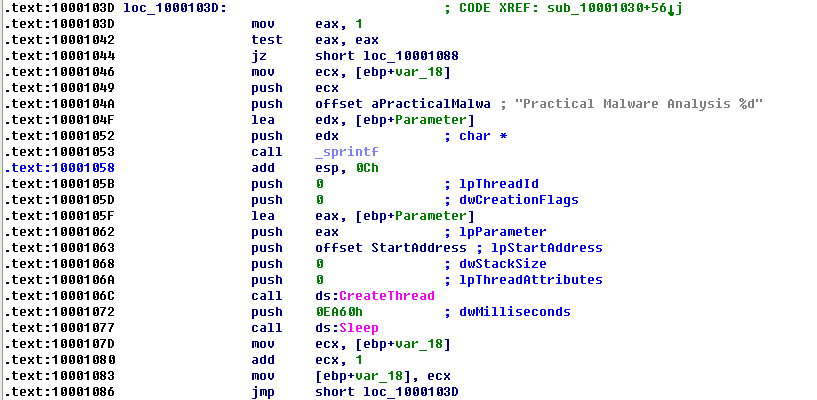
对GetModuleHandleA以及GetProcAddress的调用，用来获取LoadLibraryA的地址。恶意代码中的LoadLibraryA的地址与explorer.exe中的地址是相同的。0x00401303处LoadLibraryA的地址被写入到lpStartAddress中。lpStartAddress提供给CreateRemoteThread,强制explorer.exe调用LoadLibraryA。LoadLibraryA的参数通过CreateRemoteThread中的1pParameter传递，也就是包含Lab12-01.dll的字符串。反过来，在远程进程中启动一个线程，这个线程以参数Lab12-01.dll来调用LoadLibraryA。这个恶意代码执行DLL注入，将Lab12-01.dll注入到explorer.exe中。

分析Lab12-01.dll

创建消息框并显示“Press OK to reboot”



每分钟（0xEA6o毫秒）创建一个线程的循环。



用sprintf拼接参数作为消息框的标题，“Pratical Malware Analysis %d”，%d用保存在var\_18变量的递增计数器替换。

1. **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

1. **哪个进程会被注入?**

被注入的进程是explorer.exe。

1. **你如何能够让恶意代码停止弹出窗口?**

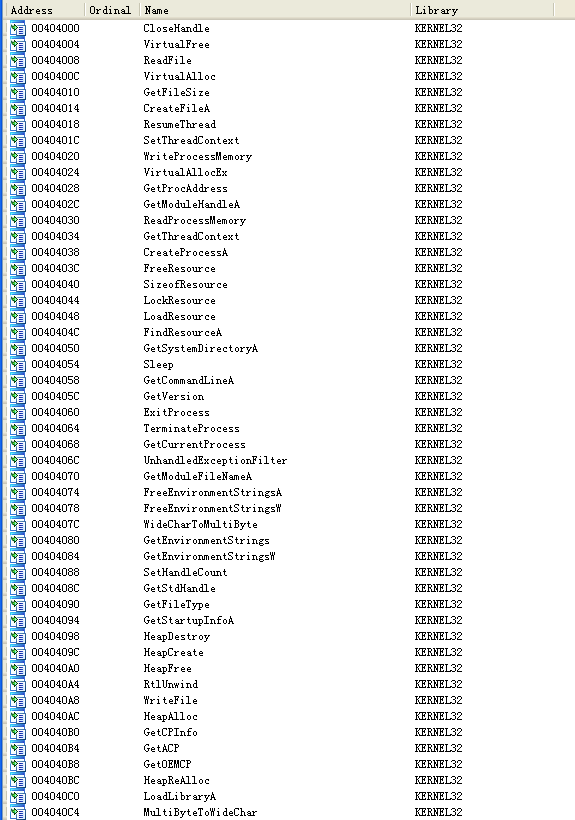
重新启动explorer.exe进程。

1. **这个恶意代码样本是如何工作的?**

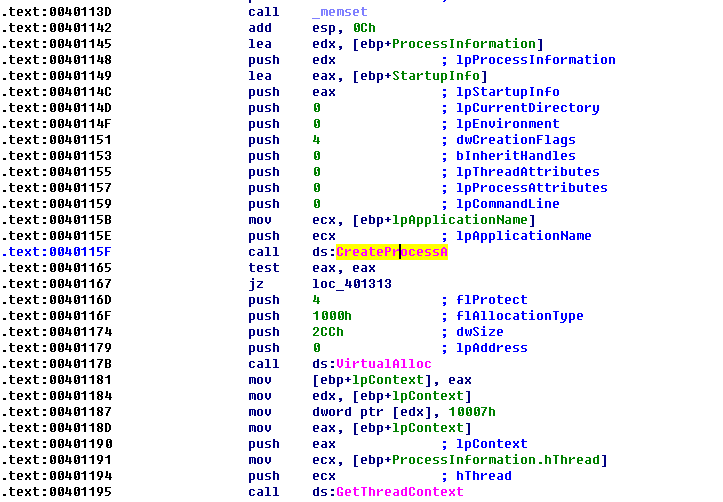
这个恶意代码执行DLL注入，来在explorer.exe中启动Lab12-01.dll。一旦Labl2-01.dll被注入，它在屏幕上每分钟显示一个消息框，并通过一个计数器，来显示已经过去了多少分钟。

1. **Lab12-2**

查看Lab12-2.exe的导入函数



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执行上下文。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。

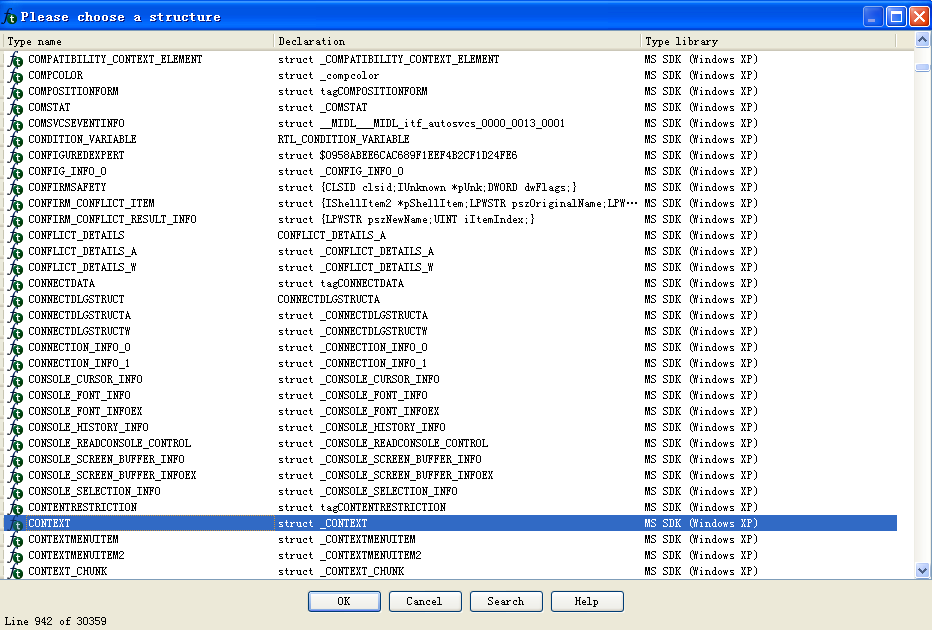


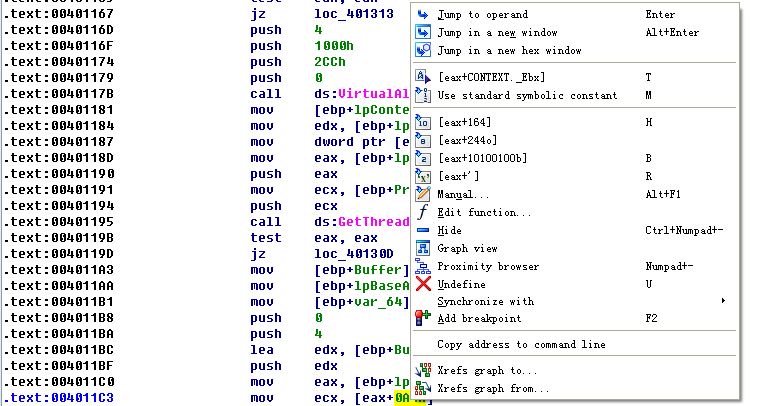


GetThreadContex的hThread参数与push edx传递给CreateProcessA的参数处于同一缓冲区，它告诉我们这个程序正在访问挂起进程的上下文。获取进程句柄非常重要，因为程序将使用这个进程句柄与挂起进程进行交互。

0x00401151处push 4指令，被IDA Pro标记为参数dwCreationFlags。 它允许进程被创建但并不启动。这个进程将不会执行，除非等这个主进程调用API ResumeThread函数时，它才会被启动。

调用GetThreadContext以后，看到这个进程被用于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，在IDA Pro中添加CONTEXT结构体。要添加这个标准结构体，需要单击Structures标签页，并按INS键。接着，单击Add Standard Structure按钮，并放置名为CONTEXT的结构体。右击位置0x004011C3来解析这个结构体的偏移，可以看到，偏移0xA4实际上通过[eax+CONTEXT.\_Ebx]来引用这个进程的EBX寄存器。



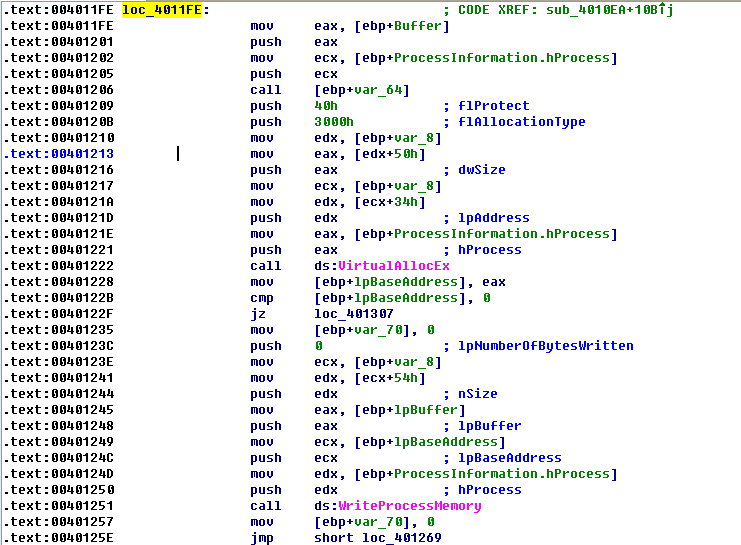


这个新创建就被挂起的进程EBX寄存器总是包含一个指向进程环境块(PEB)的数据结构。程序以8字节递增结构体，并将这个值压到栈上，作为要读取内存的起始地址。

PEB数据结构的8字节偏移处是一个指向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0x004011BA处读取4个字节，看到IDA Pro已标记为Buffer的变量将包含被挂起进程的ImageBase。

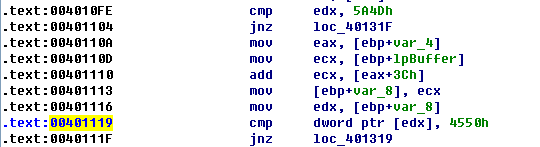


这个程序使用在0x004011E8处的GetProcAddress,手动解析导入函数NtUnMapViewofSection,并且在0x004011FE处，ImageBaseAddress作为NtUnMapViewOfSection的一个参数传入,然后保存在[ebp+var\_64]。UnMapViewOfSection的调用从内存中移除这个被挂起进程，此时这个程序将不再执行。

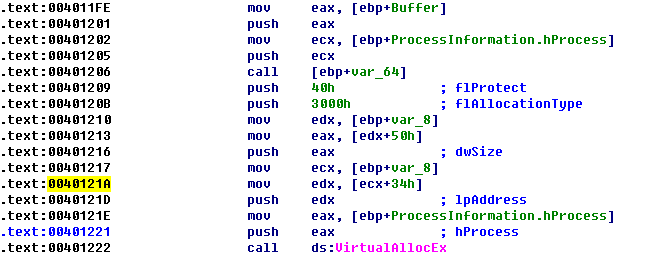


参数被压到栈上，然后调用VirtualAllocEx。在0x0040121E处，显示了这个程序在被挂起进程的地址空间中分配内存。这是需要进一步调查的行为。

在函数的开头，程序检查在0x004010FE处的魔术值MZ和在0x00401119处的魔术值PE。如果这个检查是有效的，var\_8是一个指针，指向加载到内存的PE头。

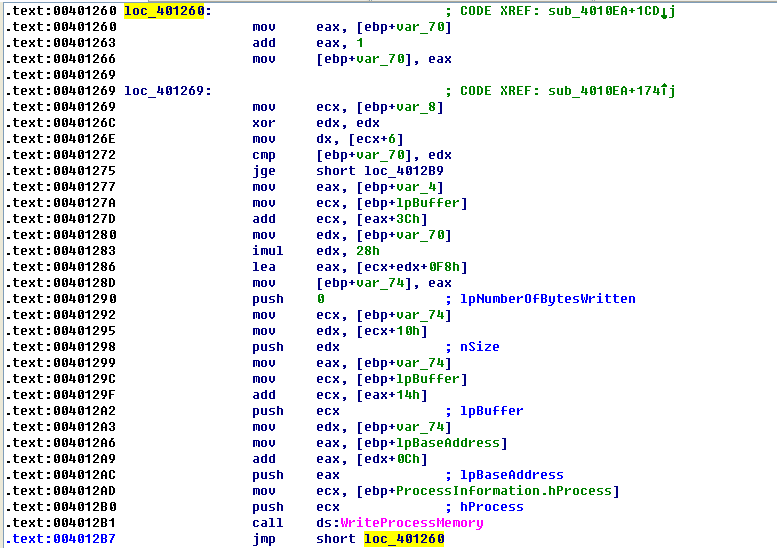


0x0040121A处，偏移量是34h，这个程序要求内存被分配在PE文件ImageBase的地址，它告诉Windows加载器这个可执行文件更倾向于被加载到哪个位置。在0x00401213处，这个程序请求由PE头ImageSize(偏移0x50)属性指定内存的大小。最终，在0x00401209处，查询MSDN文档判定这部分内存是以PAGE\_EXECUTE\_READWRITE权限分配的。



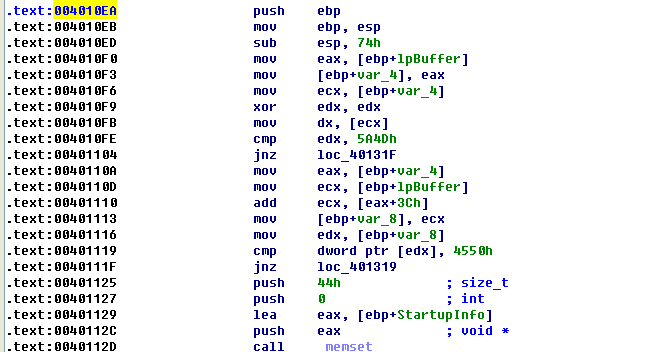
一旦这部分内存被分配后，在0x00401251处指令会调用WriteProcessMemory函数，从这个PE文件的开头写入数据到挂起进程时分配的内存中。要写入的字节数是从PE头的偏移0x54处取得的，也就是SizeOfHeaders。第一次WriteProcessMemory调用将PE文件头复制到被挂起的进程中，这表明了这个程序正移动一个PE文件到另一个进程的地址空间。

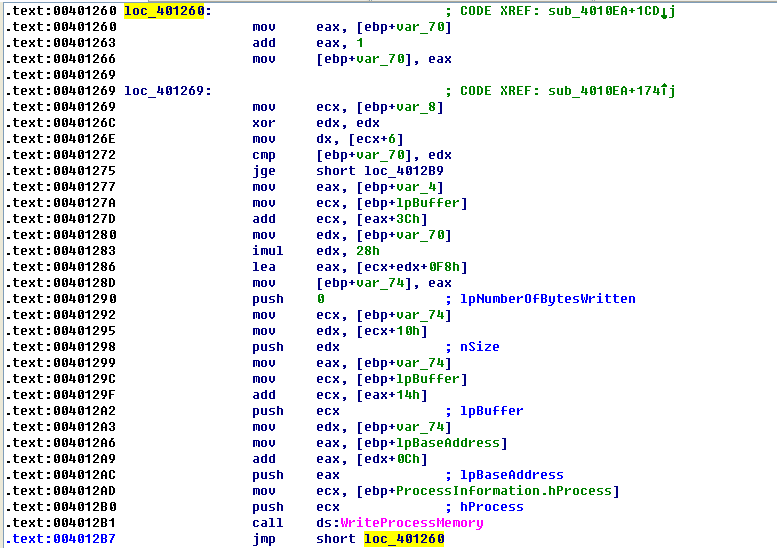
下面看到一个循环，这个循环的计数器var\_70在0x00401257处被初始化为0。



这个循环计数器用来与0x00401272处PE头部第6字节偏移处的值比较，也就是NumberOfSections。可执行代码包含一个可执行文件必需的数据——如代码、数据、重定位等,推测这个循环可能正在复制PE可执行端到这个挂起进程。

var\_4变量包含一个指向内存中MZ/PE文件的指针(被IDA Pro标记为lpBuffer),它在位置0x004010F3处被初始化。0x0040127D处看到这个程序正在将MZ头缓冲区的偏移量加上0x3C(到PE头的偏移),这使ECX指向PE头的起始位置。0x00401286处看到一个指针被获取到。EDX在这个循环中第一次迭代时的取值是0,所以可以在指针计算中将EDX移除，这样就只剩下ECX和0xF8了。



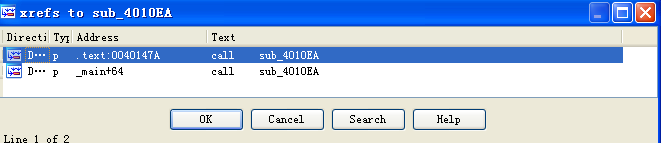


查看这个PE头部偏移，看到0xF8是IMAGE\_HEADER\_SECTION数组的起始位置。sizeof(IMAGE\_HEADER\_SECTION)告诉我们这个结构体的大小是40字节，与0x00401283处执行乘法的循环计数器的值相吻合。

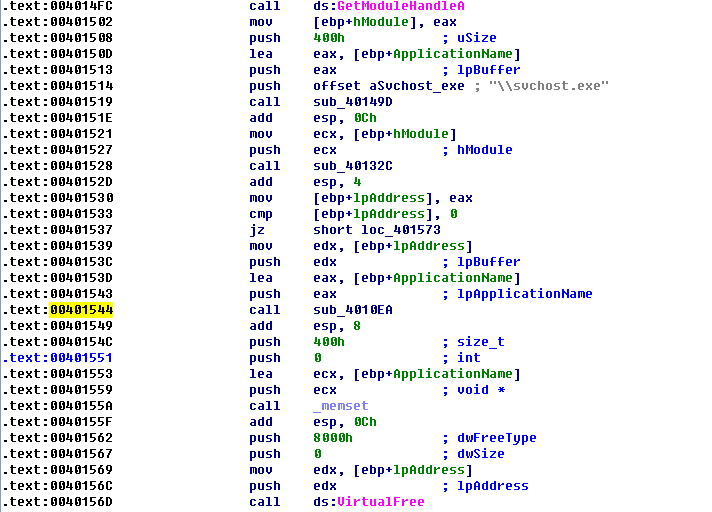


这个程序调用了SetThreadContext函数，这个函数设置在0x004012E7处的EAX寄存器为被加载到被挂起进程内存空间中可执行文件的入口点。一旦这个程序执行ResumeThread函数，它将会成功地达到将之前CreateProcessA创建的进程替换为另一个进程的目的。

将光标放在sub\_4010EA函数的开头，并按Ctrl+X,显示出所有的交叉引用，包括调用者sub\_40144B和main。

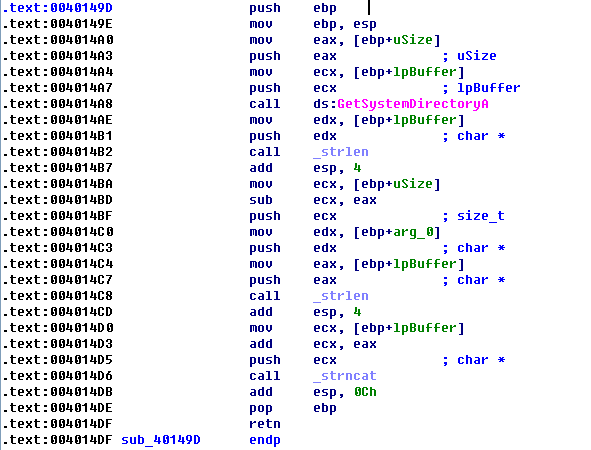


跟着main,可以找到0x00401544。



变量ApplicationName被载入到要传递给sub\_4010EA的寄存器中，作为CreateProcessA调用所准备的进程名。ApplicationName首次出现在0x0040105D处，被当做sub\_40149D函数的第二个参数。

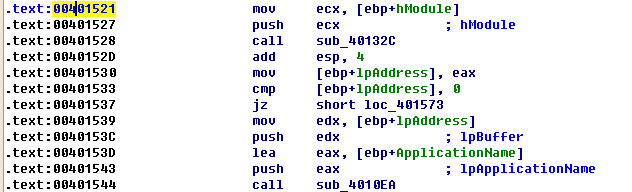
查看sub\_40109D



这个函数仅仅复制%SystemRoot%\System32\到第二个参数中，然后将第一个参数拼接到末尾。

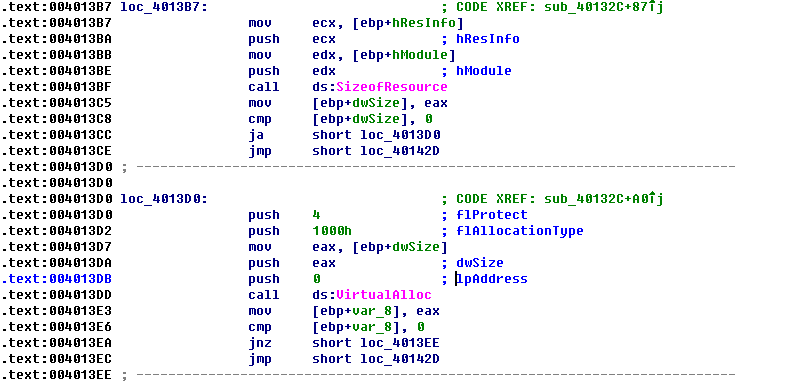
ApplicationName变量是第二个参数，这个函数接收新路径，0x00401514处是sub\_40149D函数的第一个参数，看到\\svchost.exe。这告诉被替换进程是%SystemRoot%\System32\svchost.exe。

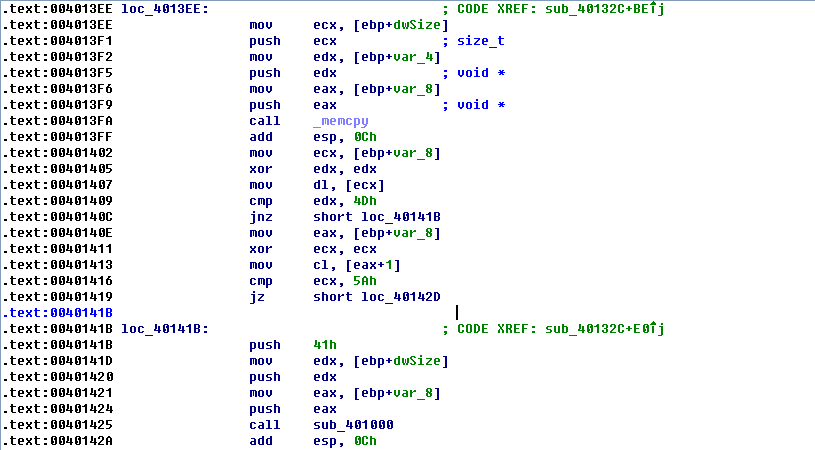
判断替换svchost.exe的进程。跟随0x00401539处的lpAddress,来跟踪传递给sub\_4010EA的PE缓冲区。lpAddress在0x00401530处接收eax。Eax用作一个函数的返回值，这个缓冲区来自函数sub\_40132C,它使用变量hModule，一个指向程序本身Lab12-02.exe的内存指针。

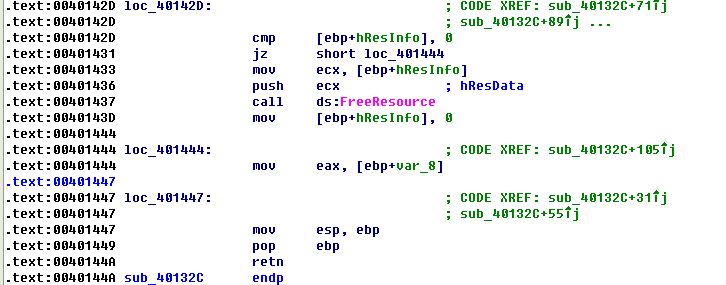


Sub\_40132C调用函数FindResource、LoadResource、LockResource、SizeOfResource、VirtualAlloc以及memcpy。这个程序从可执行文件的资源段复制数据到内存中。这个恶意代码的有效载荷(payload)被保存在这个程序的资源节中。这个资源节的类型是UNICODE,且名字是LOCALIZATION。



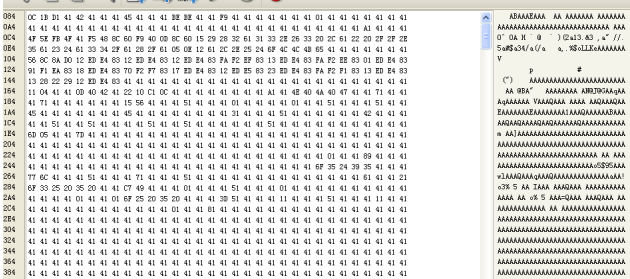




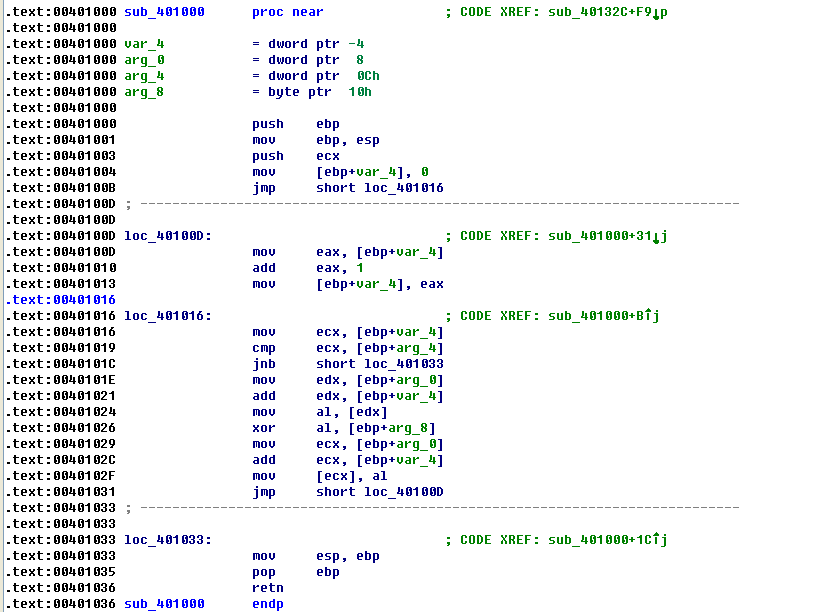


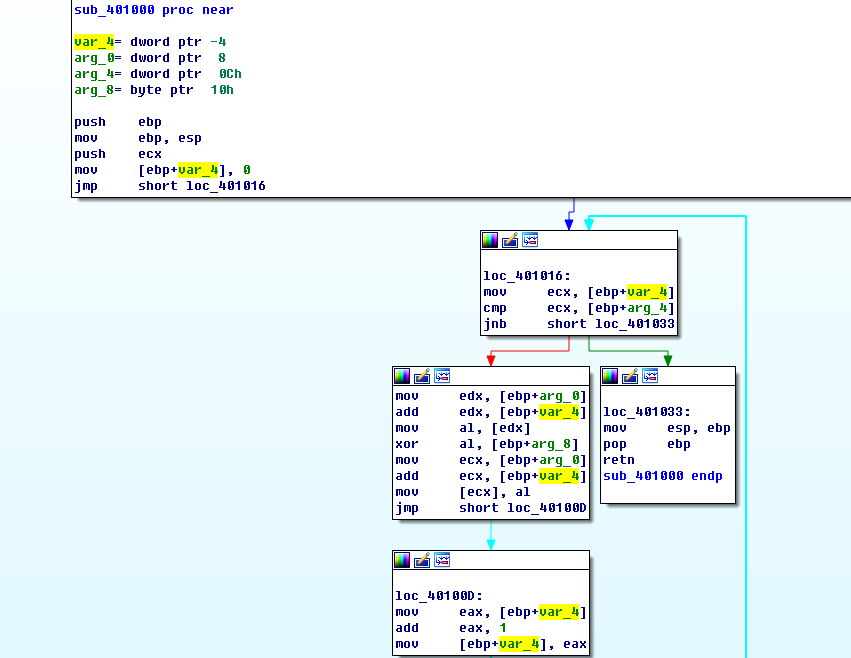
使用Resource Hacker来导出这个二进制文件。

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。猜想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。



继续检查这段反汇编，来判断这个可执行文件是如何被编码的。在0x00401425处，看到这个缓冲区被传递给函数sub\_401000,这个函数看起来像一个XOR例程。回顾传递给在0x0040141B处的函数的第三个参数，看到了0x41。利用WinHex,选择Edit Modify Data XOR,然后输入0x41,可以快速XOR之前Resource Hacker导出的文件。在执行这个转换以后，得到了一个有效的PE可执行文件，这个执行文件后续被用来替换svchost.exe进程实例。





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的41h。

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。

1. **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。

1. **恶意代码的负载存储在哪里?**

这个恶意代码的有效载荷(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。

1. **恶意负载是如何被保护的?**

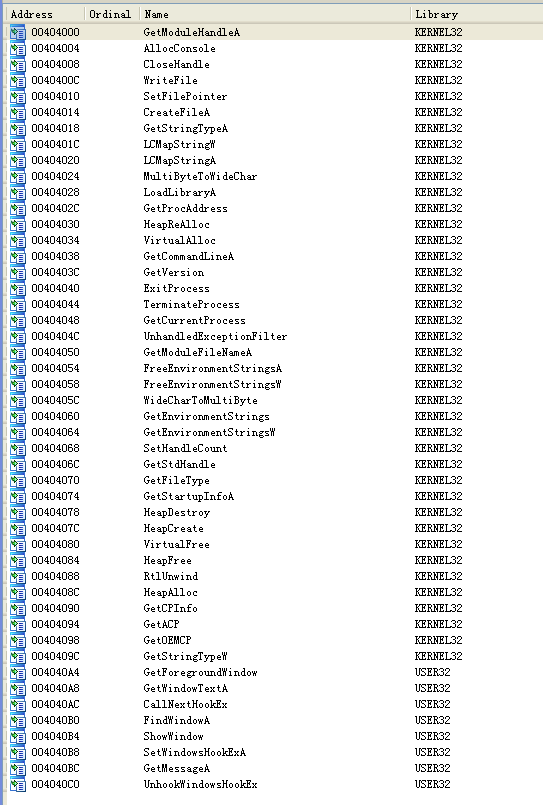
保存在这个程序资源节中的恶意有效载荷是经过XOR编码过的。这个解码例程可以在sub\_40132C处找到，而XOR字节在0x0040141B处可以找到。

1. **字符串列表是如何被保护的?**

这些字符串是使用在sub\_401000处的函数，来进行XOR编码的。

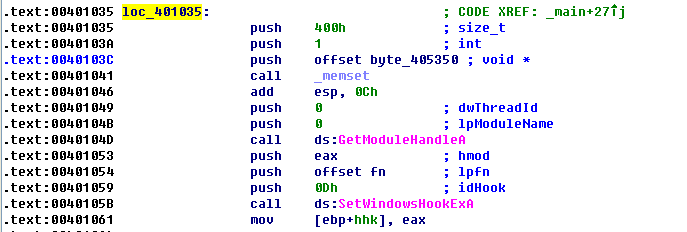
1. **Lab12-3**

Ida pro查看Lab12-03.exe的导入函数

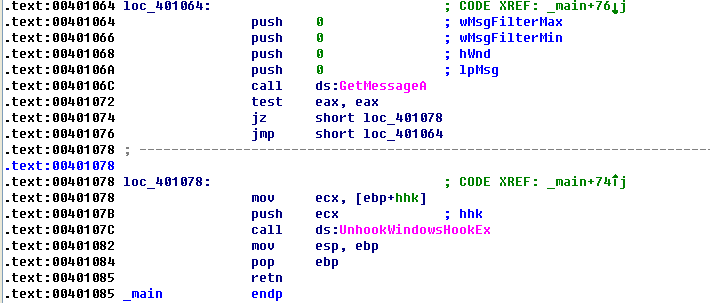


导入函数中看到了SetWindowsHookExA,这是一个允许应用程序挂钩或监控微软Windows内部事件的API。





SetWindowsHookExA在0x0040105B处从main中被调用。第一个参数为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。这个程序可能对键击消息做某些手脚，而这个fn函数正在接收这些击键记录。



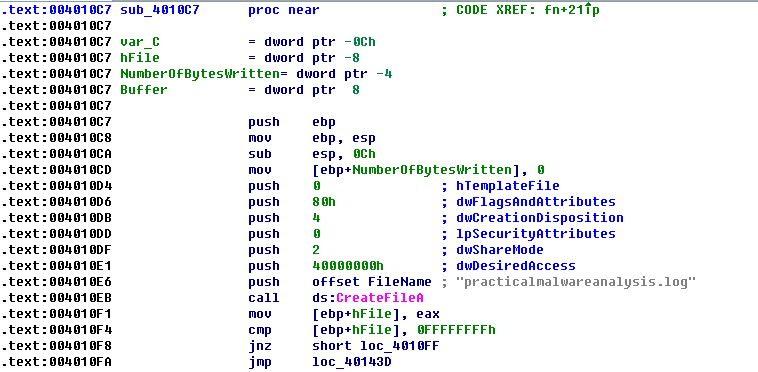
注册接收键盘事件的钩子之后，程序在0x00401076处的循环中调用了GetMessageA。程序调用GetMessageA,因为Windows不会将消息发送到程序进程的钩子函数中，直到循环产生错误，它才会终止。

移动到fn函数，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3个参数，它有一个定义为HOOKPROC的原型。WH\_KEYBOARD\_LL回调函数实际上是LowLevelKeyboardProc回调函数。



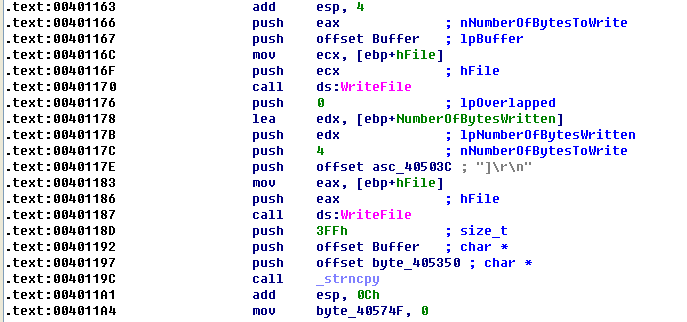
为了处理每次按键，程序用cmp指令在0x0040108F和0x00401098处检查按键的类型。在0x004010A4处，程序将虚拟按键码传递到sub\_4010C7函数中。

查看sub\_4010C7



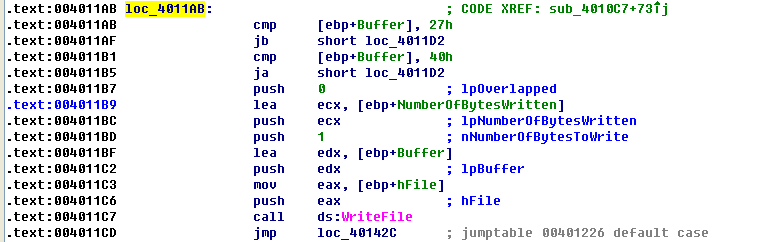
程序首先打开一个文件(practicalmalwareanalysislog)。

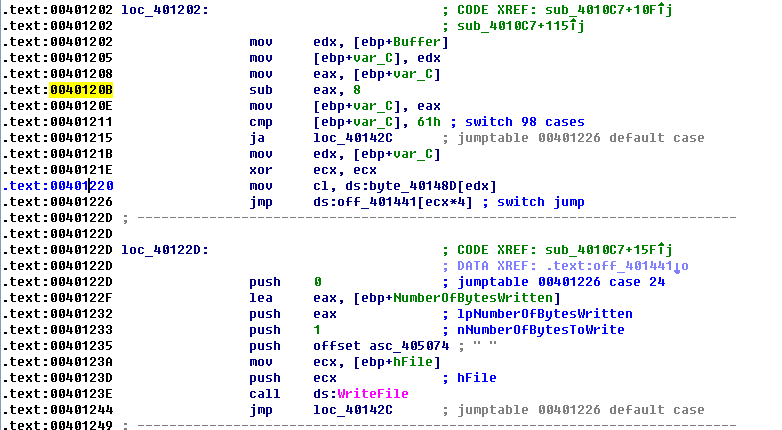


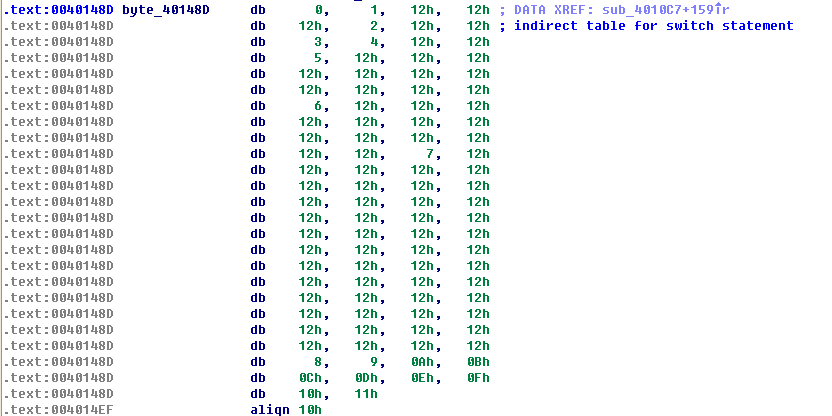


打开文件之后，恶意代码调用GetForegroundwindow和GetWindowTextA。GetForegroundWindow选择按键按下时的活动窗口，然后使用GetWindowTextA获得窗口的标题。通过这些操作，帮助程序提供按键来源的上下文。

程序将窗口标题写入到日志文件，进入0x00401226处的跳转表。确定变量var\_C包含传入函数sub\_4010C7的虚拟按键码后，看到虚拟按键码作为一个查询表的索引0x00401220。查询表得到的值作为跳转表off\_401441的一个索引0。







选择VK\_SHIFT(0x10)的值，跟随查询过程。在0x40120B处，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。到得值3,3被存入到ECX中。然后，0x00401226处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1oc\_401249,发现字符串[SHIFT]被写入到日志文件中。

推断这个恶意代码是一个击键记录器，它将击键记录到文件practicalmalwareanalysis.log。这个击键记录器使用SetWindowsHookEx,来实现它的击键记录功能。

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。

1. **恶意负载是如何注入自身的?**

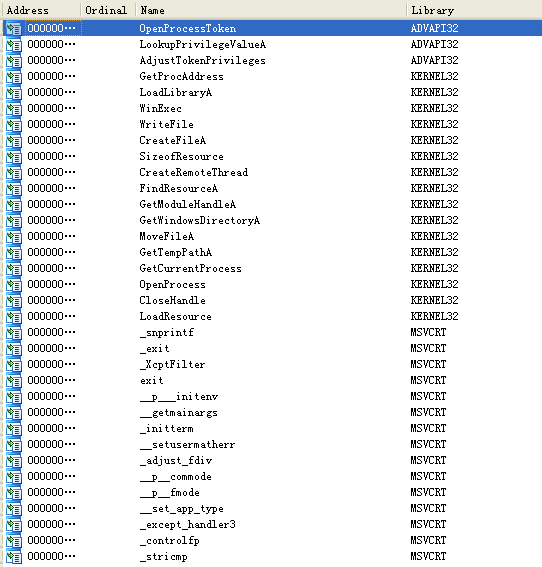
这个程序使用挂钩注入，来偷取击键记录。

1. **这个程序还创建了哪些其他文件?**

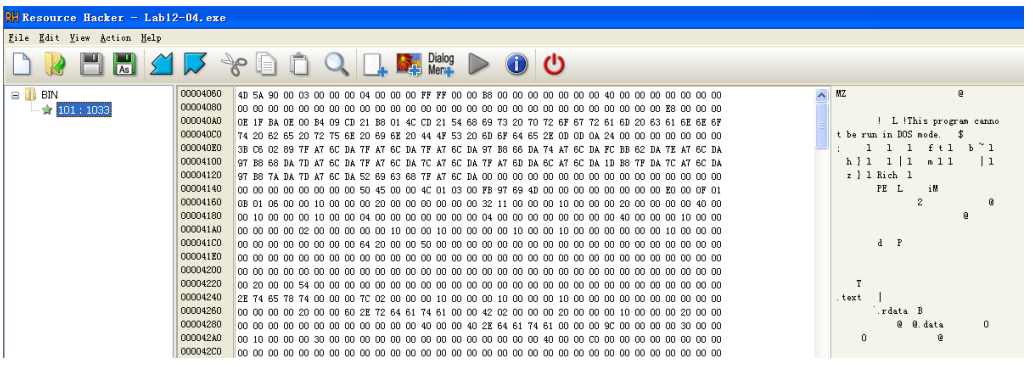
这个程序创建文件praticalmalwareanalysis.log,来保存击键记录。

1. **Lab12-4**

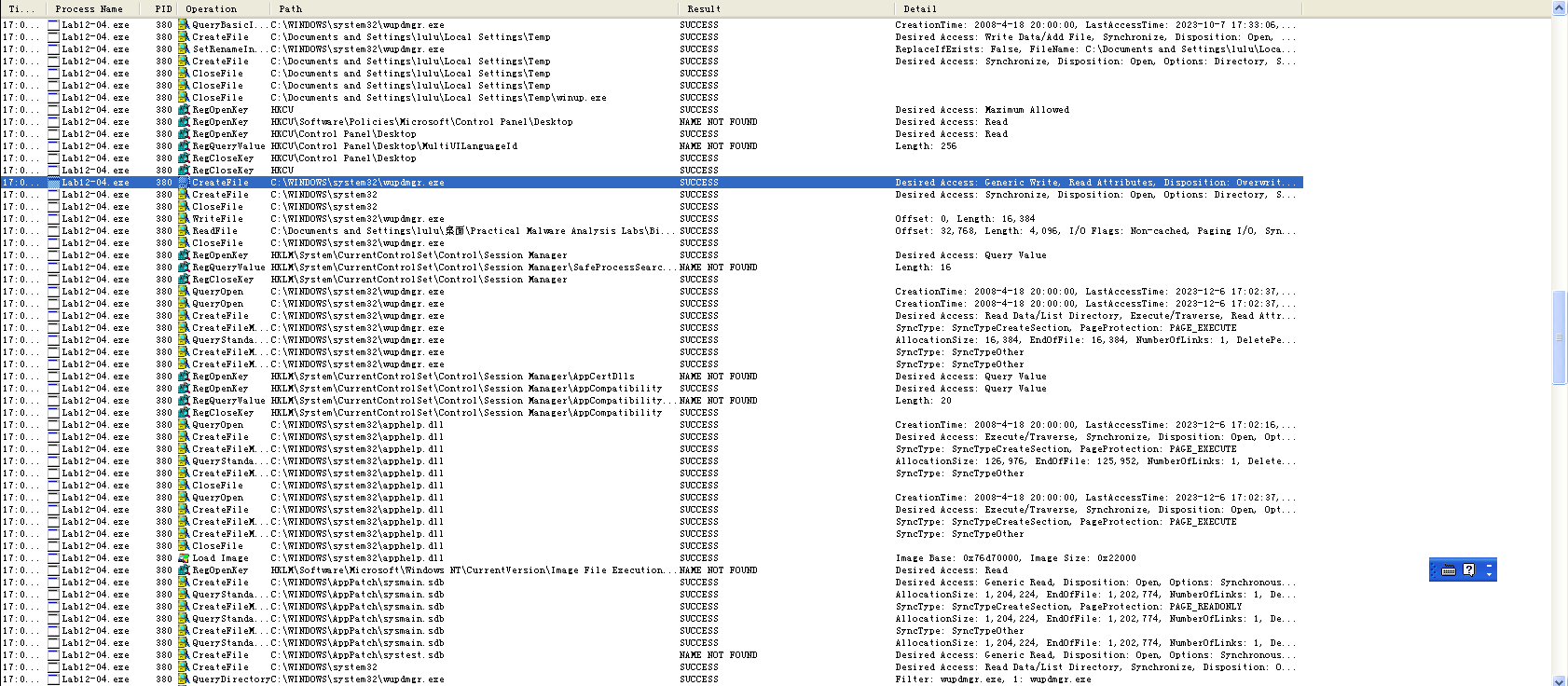
查看Lab12-04.exe的导入表



看到CreateRemoteThread,也看到有资源操作函数的导入，例如LoadResource和FindResourceA等，还有一些与特权级相关的函数LookupPrivilegeValueA和AdjustTokenPrivileges。用 Resource Hacker查看该程序，在资源节中发现了一个名为BIN的程序。



使用Procmon动态分析，恶意代码创建一个文件(%TEMP%\winup.exe),并且覆盖了位于%SystemRoot%\system32\wupdmgr.exe的Windows更新二进制文件。可以看到有对临时文件夹Temp的操作，在该文件夹下还看到了winup.exe之后有对系统目录下wupdmgr.exe的windows更新二进制文件的访问。比较恶意代码释放的wupdmgr.exe和恶意代码资源节中BIN文件，看到它们是相同的。

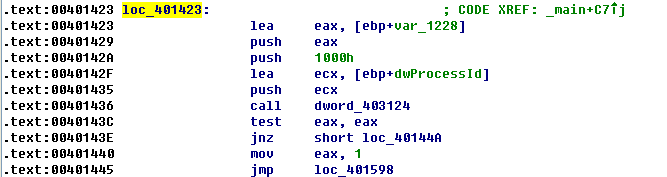


发现恶意代码试图从www.practicalmalwareanalysis.com中下载updater.exe。

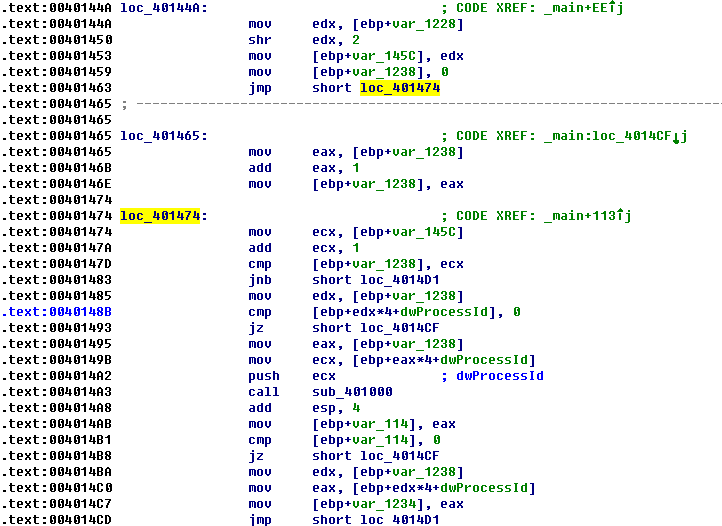
分析main函数，看到恶意代码从psapi.dll中解析Windows进程枚举的函数。恶意代码分别将函数指针保存到dword\_40312C、dword\_403128和dword\_403124中。



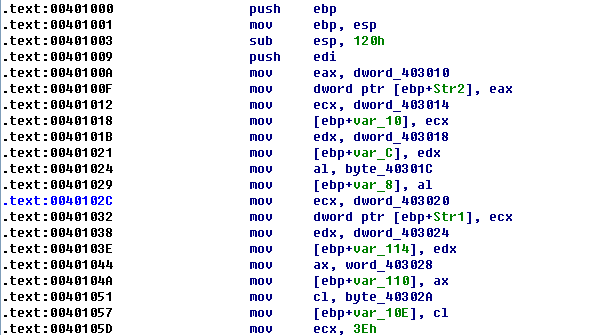
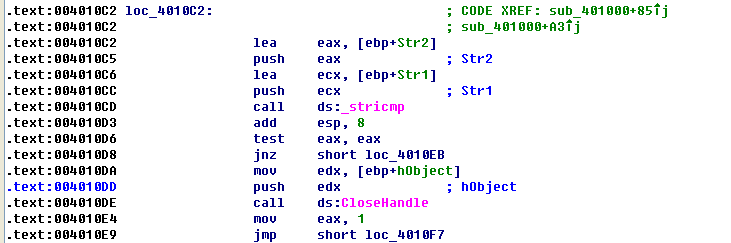
之后恶意代码检查这些函数指针的值，当达到0x00401423和myEnumProcesses调用时，返回系统进程PID的数组。数组被局部变量dwProcessId所引用。

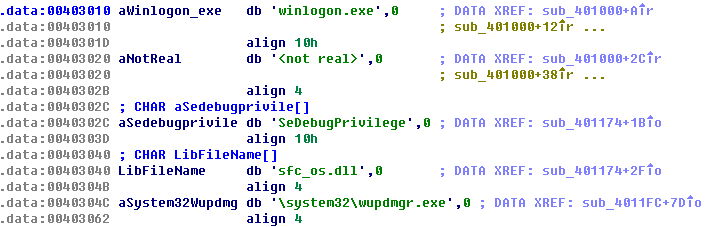


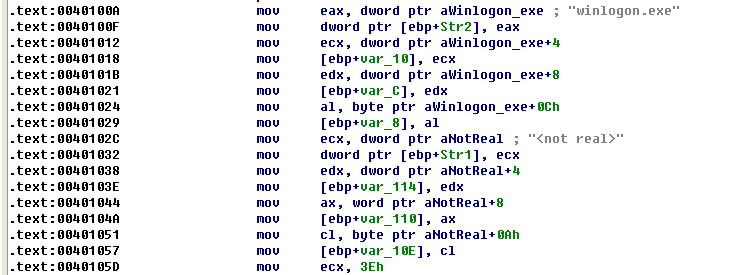
恶意代码开始循环遍历PID,并且将每个进程的PID传递给0x00401000的子例程。看到一个到dwProcessId引用数组的索引，在调用sub\_401000之前计算它。



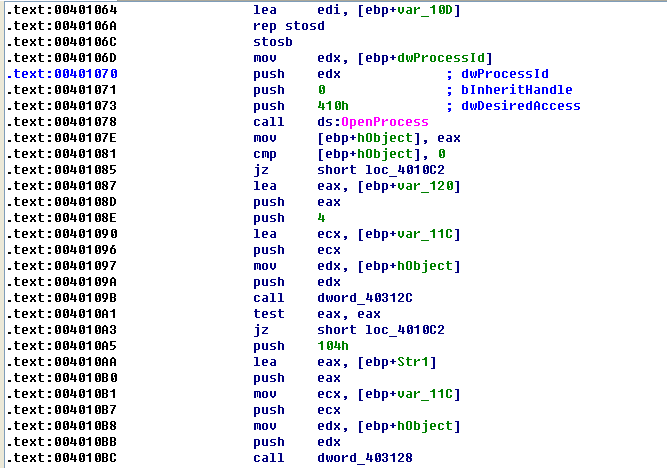
检查sub\_401000函数的内部，看到两个局部变量集合(Str1和Str2)。变量Str1包含字符串"<not real>",变量Str2包含字符串"winlogon.exe"。





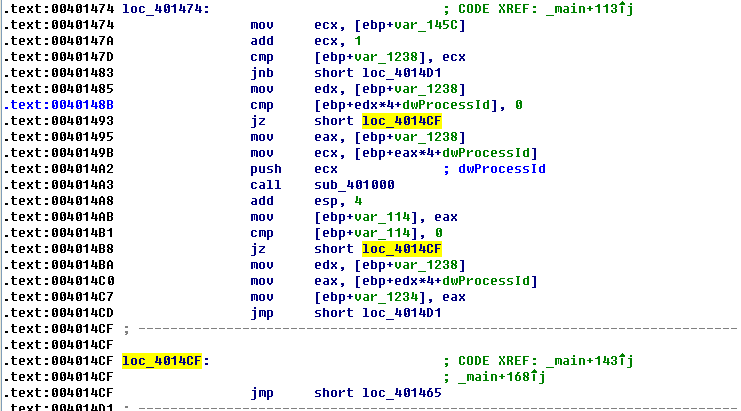


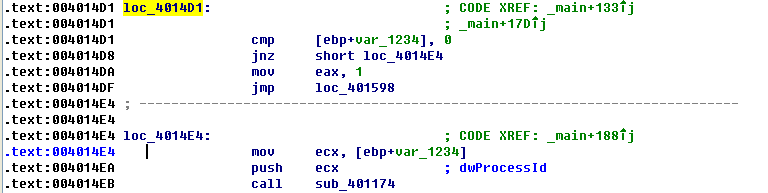
恶意代码将循环中的参数(dwProcessId)传递到OpenProcess,来获得进程的句柄。OpenProcess返回的句柄存入到EAX,并且传递给处的myEnumProcessModules函数，这个函数将返回加载到这个进程所有模块的句柄数组。恶意代码试图使用GetModuleBaseNameA,获得模块PID的基本名称。如果成功，Str1将会包含传递给子例程的模块PID的基本名称的字符串，如果失败，它会保持初始值"<not real>"。



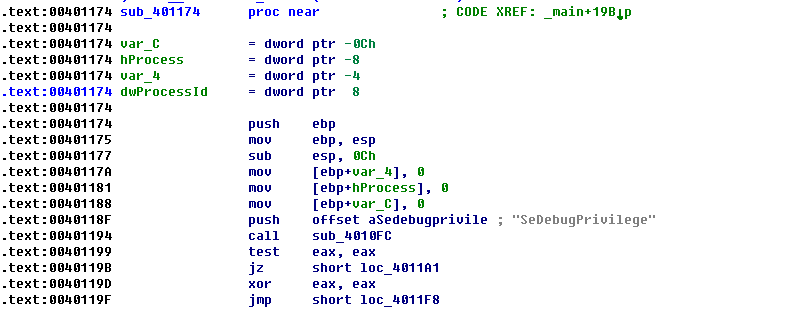
原有初始化字符串"<not real>"应该含有GetModuleBaseNameA返回的基本模块的名字。将这个字符串与字符串"winlogon.exe"比较，如果相同，EAX将会等于0,函数返回时EAX将会等于1。如果不相同，则函数返回值EAX等于0。sub\_401000函数正在确定与winlogon.exe关联的PID。

在0x004014B1处，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winlogon.exe相匹配，则PID将被传递给sub\_401174。

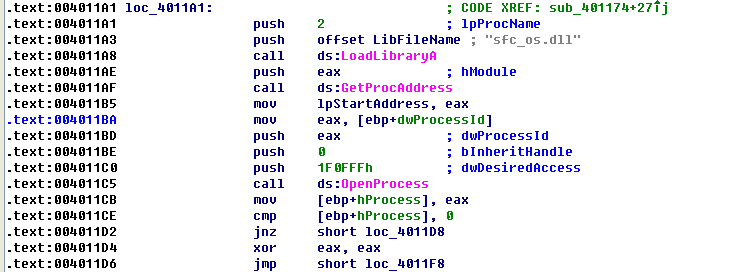




查看sub\_401174,看到它用参数SeDebugPrivilege立即调用SeDebugPrivilege提权过程。



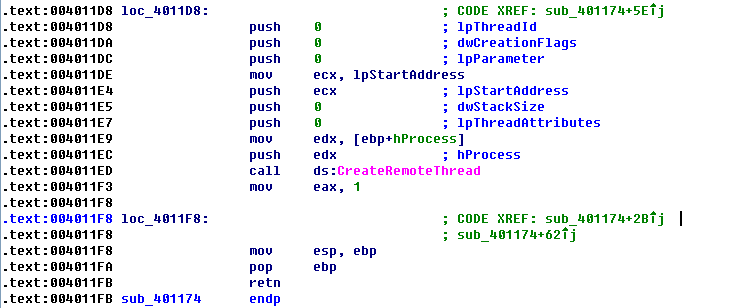
看到字符串sfc\_os.dll传递给LoadLibraryA。用sfc\_os.dll的句柄和序号2(未公开的Windows函数)调用GetProcAddress,在0x004011A1处序号2被压入到堆栈。在0x004011B5处序号2的函数指针被存入到lpStartAddress(IDA Pro提供的标签)。然后，恶意代码用winlogon.exe的PID和值为0x1FOFFF的dwDesiredAccess(符号常量为PROCESS\_ALL\_ACCESS),调用OpenProcess函数。在0x004011CB处winlogon.exe的句柄被保存到hProcess中。



调用CreateRemoteThread,检查CreateRemoteThread的参数，在

0x004011EC处看到hProcess参数是EDX,即winlogon.exe句柄。0x004011DE处传入的lpStartAddress是sfc\_os.dll中序号2函数的指针，它负责向winlogon.exe注入一个线程(因为sfc\_os.dll已经载入到winlogon.exe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sfc\_os.dll中序号为2的函数。

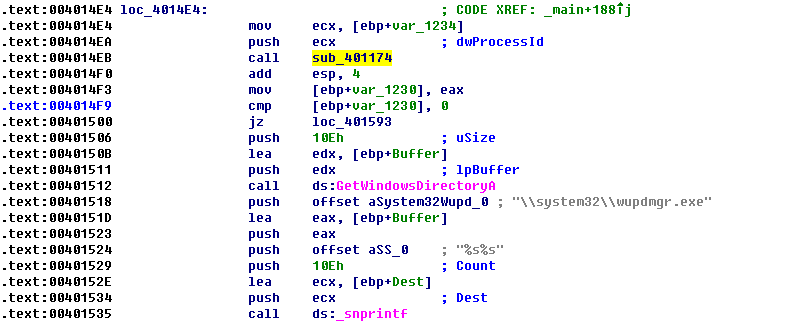
sfe\_os.dll的序号2函数是一个未公开的导出函数，它被称为SfcTerminateWatcherThread。



要成功地运行SfcTerminateWatcherThread函数，则必须将它运行在winlogon.exe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。

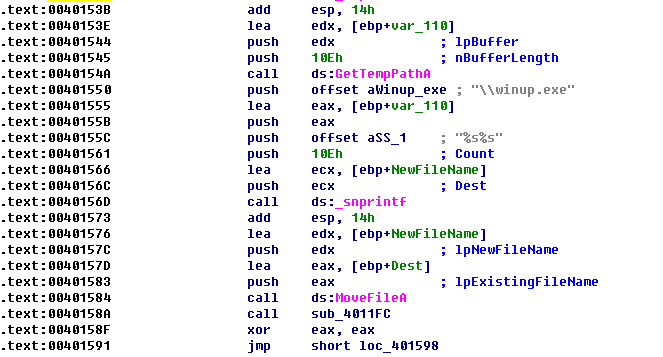
如果正确地注入线程，则运行并且创建一个字符串。0x00401512处的GetwindowsDirectoryA会返回当前Windows目录(通常是C:\Windows)的指针，同时，恶意代码也会将这个字符串和\system32\wupdmgr.exe传递给\_snprintf调用，如0x00401523和0x00401518所示。

这段代码在构建字符串"C:\Windows\system32\wupdmgr.exe",构建的字符串被存储在ExistingFileName中。在Windows XP系统下，wupdmgr.exe被用作Windows系统的更新。



0x0040154A处的GetTempPathA调用提供了当前用户临时目录(通常是C:\Documents and Settings\<username>\Local\Temp)的指针。随后，临时目录的路径和参数\\winup.exe一起传递给另外一个\_snprintf调用，如0x0040155B和0x00401550所示，这个\_snprintf调用创建了字符串"C:\Documents and Settings\username\Local\Temp\winup.exe"并且存储在NewFileName中。

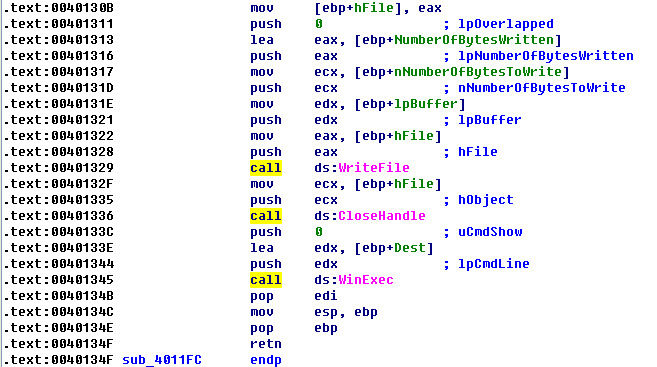
MoveFileA将把Windows更新二进制文件移到用户的临时目录中。



恶意代码在0x004012A1处调用GetModuleHandleA,GetModuleHandleA返回当前进程的模块句柄。随后，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,恶意代码正在提取它的资源节到硬盘上。

FindResourceA调用之后便调用LoadResource、SizeofResource、CreateFileA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:\Windows\System32\wupdmgr.exe中。





在本实验中，恶意代码是创建一个新的Windows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覆盖，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，是使用WinExec启动新的wupdmgr.exe。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，如0x0040133C所示。

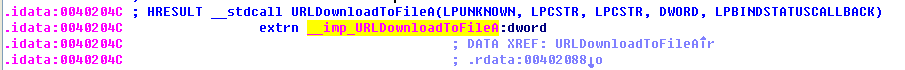
Ida pro打开新创建的wupdmgr.exe。

恶意代码创建个临时目录中的字符串来移动原始的Windows更新二进制文件(C:\Documents and Settings\username\Local\Temp\winup.exe),然后运行原始的Windows更新二进制文件(使用WinExec),此时原始的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，似乎一切正常运转，原始的Windows更新文件也将会运行。

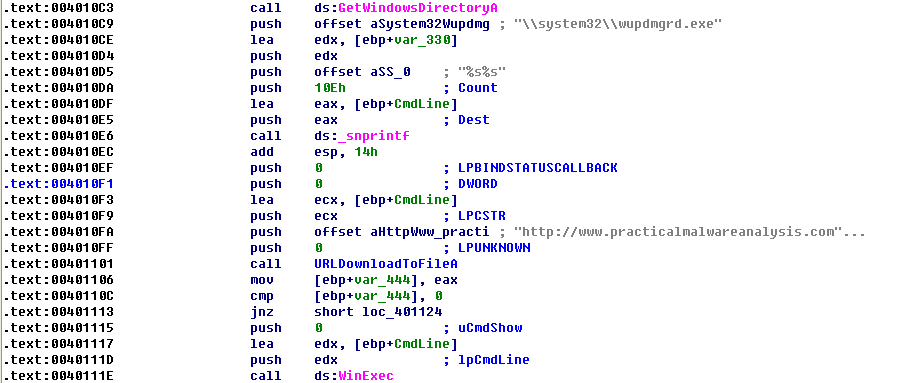
接下来，在IDA Pro中，在0x004010C3开始看到构建的字符串C:\Windows\system32\wupdmgrd.exe,它被存储在局部变量Dest中。除了文件名中的d,这个字符串与原始Windows更新二进制文件的文件名十分接近。



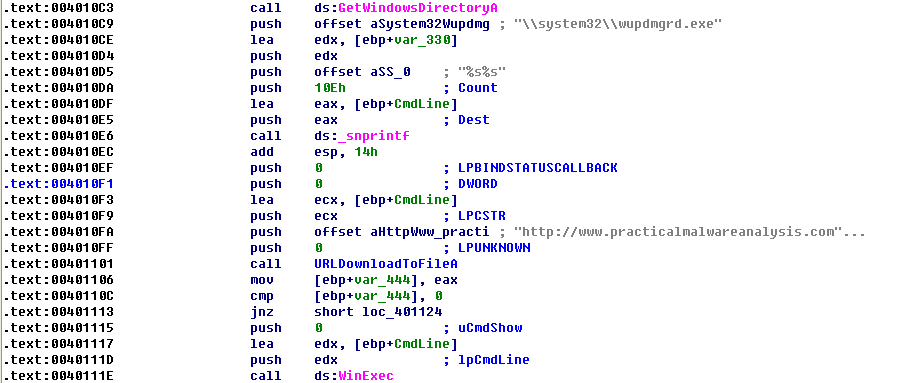
注意到API调用URLDownloadToFileA。这个调用带有一些需要进一步检查的有意义的参数。



0x004010FA处的参数szURL被设置为http://www.practicalmalwareanalysis.com/updater.exe. 0x004010F3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。



本实验中通过恶意代码分析介绍了恶意代码通过禁用Windows文件保护机制来修改Windows功能的一种通用方法。在本实验中，恶意代码特洛伊木马化Windows的更新进程，并且创建了它自己的更新程序。因为恶意代码没有完全破坏原始的Windows更新二进制文件，所以感染这个恶意代码机器的用户仍会看到正常的Windows更新功能。

1. **位置0x401000的代码完成了什么功能?**

恶意代码查看给定PID是否为winlogon.exe进程。

1. **代码注入了哪个进程?**

winlogon.exe是被注入的进程。

1. **使用LoadLibraryA装载了哪个DLL程序?**

DLL sfc\_os.dll用来禁用Windows的文件保护机制。

1. **传递给CreateRemoteThread调用的第4个参数是什么?**

传递给CreateRemoteThread的第4个参数是一个函数指针，指向sfc\_os.dll中一个未命名的序号为2的函数(SfcTerminateWatcherThread)。

1. **二进制主程序释放出了哪个恶意代码?**

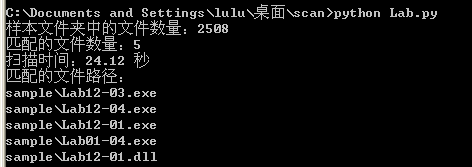
恶意代码从资源段中释放一个二进制文件，并且将这个二进制文件覆盖旧的Windows更新程序(wupdmgr.exe)。覆盖真实的wupdmgr.exe之前，恶意代码将它复制到%TEMP%目录，供以后使用。

1. **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，并且调用原始的二进制文件(位于%TEMP%目录)来特洛伊木马化wupdmgr.exe文件。

1. **Yara规则**

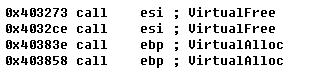
|  |
| --- |
| rule RuleforLab12\_01exe {  meta:  description = " Lab12-01.exe"  strings:  $s1 = "Lab12-01.dll" fullword ascii  $s2 = "\"WWSh T@" fullword ascii  $s3 = "YYh `@" fullword ascii  $s4 = "[Sh$T@" fullword ascii  $s5 = "RichLu" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and  all of them  } |
| rule RuleforLab12\_01dll {  meta:  description = "Lab12-01.dll"  strings:  $s1 = ":5:[:u:|:" fullword ascii  $s2 = "Press OK to reboot" fullword ascii  $s3 = ">.>4><>Z>`>q>" fullword ascii  $s4 = "9C9M9U9[9c9l9u9" fullword ascii  $s5 = "0<0Q0u0" fullword ascii  $s6 = "4b4h4v4" fullword ascii  $s7 = "1$1,14181@1" fullword ascii  $s8 = "0K0d0n0y0" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and  5 of them  } |
| rule RuleforLab12\_02 {  meta:  description = "Lab12-02.exe"  strings:  $s1 = "\\svchost.exe" fullword ascii  $s2 = "wqstLKla143$a7(354 -a'4/\"5(./a\" --LKAAA" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 200KB and  8 of them  } |
| rule RuleforLab12\_03{  meta:  description = "Lab12-03.exe"  strings:  $s1 = "practicalmalwareanalysis.log" fullword ascii  $s2 = "[Window: " fullword ascii  $s3 = "[BACKSPACE]" fullword ascii  $s4 = "BACKSPACE" fullword ascii  $s5 = "YYh P@" fullword ascii  $s6 = "[ENTER]" fullword ascii  $s7 = "[CTRL]" fullword ascii  $s8 = "0B=8S@" fullword ascii  $s9 = "[SHIFT]" fullword ascii  $s10 = "[CAPS LOCK]" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 70KB and  all of them  } |
| rule RuleforLab12\_04 {  meta:  description = "Lab12-04.exe"  strings:  $s1 = "\\system32\\wupdmgrd.exe" fullword ascii  $s2 = "\\system32\\wupdmgr.exe" fullword ascii  $s3 = "http://www.practicalmalwareanalysis.com/updater.exe" fullword ascii  $s4 = "\\winup.exe" fullword ascii  $s5 = "SeDebugPrivilege" fullword ascii  $s6 = "<not real>" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and  all of them  } |



1. **Ida-python**

查找并打印包含对寄存器的调用（call/jmp指令）的指令。

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import idaapi  import idautils  import idc  for func in idautils.Functions():  func\_flags = idaapi.get\_func(func).flags  if func\_flags & idaapi.FUNC\_LIB or func\_flags & idaapi.FUNC\_THUNK:  continue  dism\_addr = list(idautils.FuncItems(func))  for line in dism\_addr:  m = idc.GetMnem(line)  if m == "call" or m == "jmp":  op = idc.GetOpType(line, 0)  if op == idc.o\_reg:  print("0x%x %s" % (line, idc.GetDisasm(line))) |



1. **实验结论及心得体会**

Lab12-01.dll 可能是一个恶意 DLL，通过 Yara 规则进行了初步的特征检测。Lab12-02.exe 使用了反调试和反虚拟机技术，通过动态分析了解了这些技术的工作原理。Lab12-03.exe 是一个击键记录器，通过 Windows 挂钩技术捕获键盘事件，可能用于窃取敏感信息。Lab12-04.exe 劫持了系统更新过程，替换了 Windows 更新二进制文件，可能用于植入恶意代码。

实验深入了解了不同类型的恶意代码，提高了对安全威胁的感知能力。IDA Pro、Yara 规则等工具的灵活应用，对实际工作中的威胁分析有很大帮助。实验锻炼了分析和解决实际安全问题的能力，对于日后从事安全工作具有积极意义。了解不同类型的恶意代码，有助于建立更全面的威胁情报，提高系统的整体安全性。