

# 恶意代码分析与防治技术实验报告

## Lab12

学号：2011937 姓名：姜志凯 专业：信息安全

### 一、实验环境

- Windows10
- Windows xp

### 二、实验工具

- IDA Pro
- Resource Hacker
- WinHex

### 三、实验内容

#### Lab 12-1

Analyze the malware found in the file *Lab12-01.exe* and *Lab12-01.dll*. Make sure that these files are in the same directory when performing the analysis.

#### Questions

1. What happens when you run the malware executable?
2. What process is being injected?
3. How can you make the malware stop the pop-ups?
4. How does this malware operate?

还是先简单静态分析一波：

IDA 打开 exe 文件，看看 string：

```
00000000 .rdata:00... 00000000 C  MICROSOFT VISUAL C++ RUNTIME LIBRARY
00000000 .rdata:00... 0000001A C  Runtime Error!\n\nProgram:
00000000 .rdata:00... 00000017 C  <program name unknown>
00000000 .rdata:00... 00000013 C  GetLastActivePopup
00000000 .rdata:00... 00000010 C  GetActiveWindow
00000000 .rdata:00... 0000000C C  MessageBoxA
00000000 .rdata:00... 0000000B C  user32.dll
00000000 .rdata:00... 0000000D C  KERNEL32.dll
00000000 .data:004... 0000000D C  explorer.exe
00000000 .data:004... 0000000D C  LoadLibraryA
00000000 .data:004... 0000000D C  kernel32.dll
00000000 .data:004... 0000000D C  Lab12-01.dll
00000000 .data:004... 0000000E C  EnumProcesses
00000000 .data:004... 00000013 C  GetModuleBaseNameA
00000000 .data:004... 0000000A C  psapi.dll
00000000 .data:004... 00000013 C  EnumProcessModules
00000000 .data:004... 00000006 C  `y`!
```

有几个 dll 文件，尤其是 Lab12-01.dll，LoadLibraryA 函数，可能会注入；还有 explorer.exe，可能是一个进程，MessageBoxA 函数，弹窗。

再看看导入表：

Address	Ordinal	Name	Library
00405000		CloseHandle	KERNEL32
00405004		OpenProcess	KERNEL32
00405008		CreateRemoteThread	KERNEL32
0040500C		GetModuleHandleA	KERNEL32
00405010		WriteProcessMemory	KERNEL32
00405014		VirtualAllocEx	KERNEL32
00405018		lstrcmpA	KERNEL32
0040501C		GetCurrentDirectoryA	KERNEL32
00405020		GetProcAddress	KERNEL32
00405024		LoadLibraryA	KERNEL32
00405028		GetCommandLineA	KERNEL32
0040502C		GetVersion	KERNEL32

CreateRemoteThread、WriteProcessMemory、VirtualAllocEx、LoadLibraryA 等函数，怀疑是进程注入，重点关注一下。

1、运行恶意代码可执行文件时，会发生什么？

双击运行，使用 Process Monitor 和 Process Explorer 监控，均未发现问题；但是运行后会弹出一个对话框，之后每隔固定时间都会重复弹出（关掉上一个弹窗才会弹出），每次标题上的数字递增一。



2、哪个进程会被注入？

分析主函数：

```

.text:00401113          rep stosd
.text:00401115          mov    [ebp+var_118], 0
.text:0040111F          push   offset ProcName ; "EnumProcessModules"
.text:00401124          push   offset LibFileName ; "psapi.dll"
.text:00401129          call   ds:LoadLibraryA
.text:0040112F          push   eax      ; hModule
.text:00401130          call   ds:GetProcAddress
.text:00401136          mov    dword_408714, eax
.text:0040113B          push   offset aGetmodulebasen ; "GetModuleBaseNameA"
.text:00401140          push   offset LibFileName ; "psapi.dll"
.text:00401145          call   ds:LoadLibraryA
.text:0040114B          push   eax      ; hModule
.text:0040114C          call   ds:GetProcAddress
.text:00401152          mov    dword_40870C, eax
.text:00401157          push   offset aEnumprocesses ; "EnumProcesses"
.text:0040115C          push   offset LibFileName ; "psapi.dll"
.text:00401161          call   ds:LoadLibraryA
.text:00401167          push   eax      ; hModule
.text:00401168          call   ds:GetProcAddress
.text:0040116E          mov    dword_408710, eax
.text:00401173          lea    ecx, [ebp+Buffer]

```

程序解析"psapi.dll"中用于进程枚举的函数，使用了三组 LoadLibrary 和 GetProcAddress 函数组合来解析"EnumProcessModules", "GetModuleBaseNameA"和"EnumProcesses"这三个函数。然后将获取到的地址分别保存到 dword\_408714, dword\_40870C, dword\_408710 中，对这三个地址重命名，便于后续分析。

```

lea    edx, [ebp+Buffer]
push  edx      ; lpString1
call  ds:istrcatA
push  offset aLab1201_dll ; "Lab12-01.dll"
lea   eax, [ebp+Buffer]
push  eax      ; lpString1
call  ds:istrcatA
lea   ecx, [ebp+var_1120]
push  ecx
push  1000h
lea   edx, [ebp+dwProcessId]
push  edx
call  EnumProcesses ←
test  eax, eax
jnz   short loc_4011D0

```

调用 EnumProcesses 函数，获取系统中每一个进程的 PID 值。

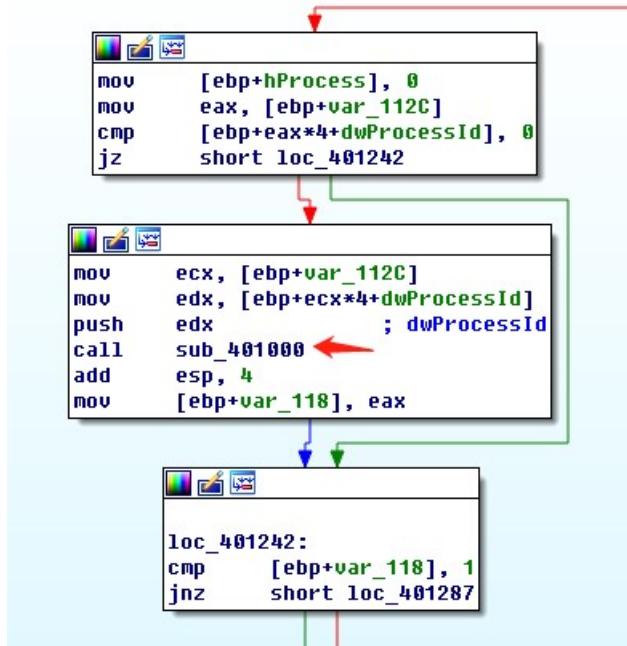
继续执行，之后是一个大的循环，循环的出口条件判断如下：

```

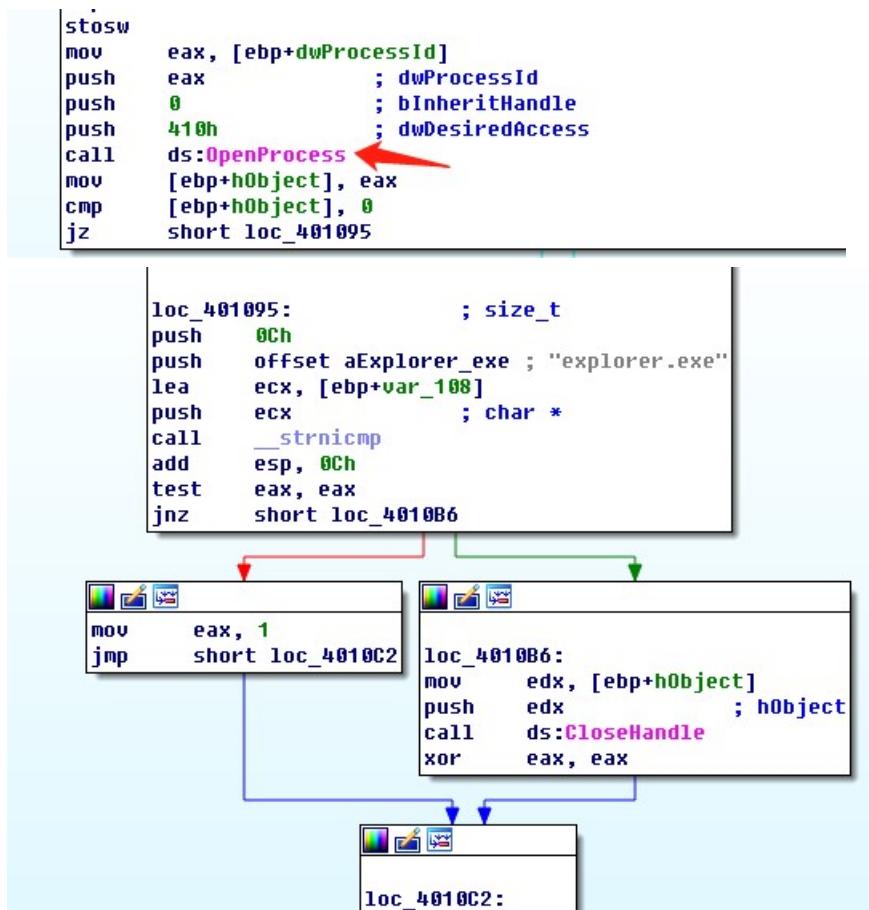
loc_4011FA:
mov   edx, [ebp+var_112C]
cmp   edx, [ebp+var_117C]
jnb   loc_40128C

```

继续看循环的内容：

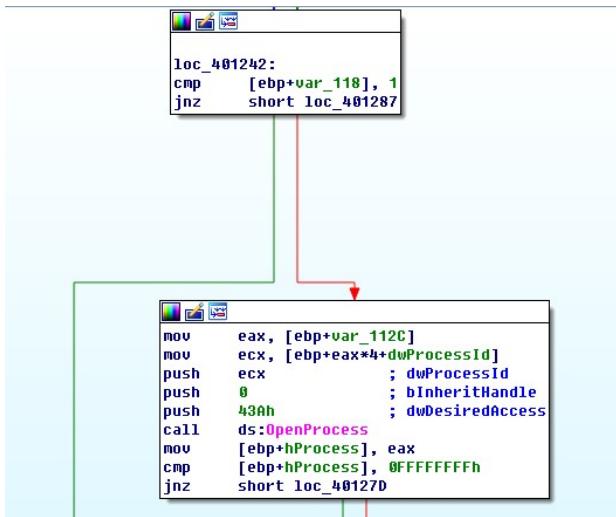


对每个 PID 值进行分析，分析函数为 sub\_401000，进入：

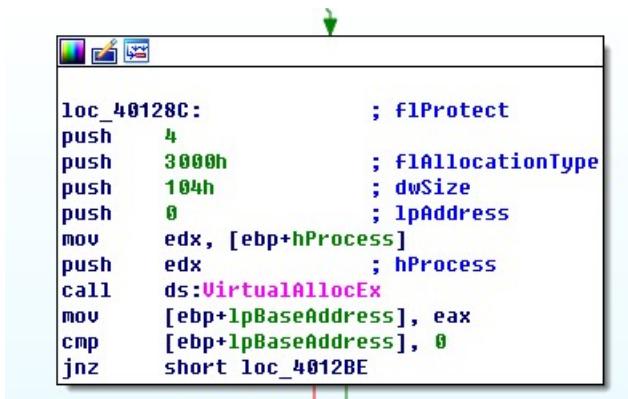


打开这个进程，然后与 explorer.exe 比较，如果一样，函数就返回 1，所以这个函数的目的就是比较当前 PID 值是不是 explorer.exe 的，如果是就返回 1，否则返回 0，就是查找进程中是否有"explorer.exe"这个进程。

继续：



判断返回值是不是 1，是 1 的话就是调用函数 OpenProcess 打开一个指向 explorer.exe 的句柄，如果成功得到句柄则循环结束，跳出循环执行以下程序：



调用函数 VirtualAllocEx，这个函数可以在 explorer.exe 这个进程里面分配空间，然后一个指向被分配内存的指针将被移入 lpParameter 中；

```
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
```

之后调用函数 WriteProcessMemory 来向 explorer.exe 中写入数据，写入的数据时 buffer，向上找看看 buffer 是啥：

```
push    104h          ; nBufferLength
call   ds:GetCurrentDirectoryA
push    offset String2 ; "\\"
lea     edx, [ebp+Buffer]
push    edx           ; lpString1
call   ds:lstrcmpA
push    offset aLab1201_dll ; "Lab12-01.dll"
lea     eax, [ebp+Buffer]
push    eax           ; lpString1
call   ds:lstrcmpA
```

先是获取一个路径，然后与"Lab12-01.dll"组合成一个新的路径，然后将这个组合好的路径保存到 buffer 里面，所以就是将 Lab12-01.dll 的路径写入 explorer.exe 进程。

```

push    , 00000000
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

```

调用函数 GetModuleHandle 和 GetProcAddress，获取 kernel32.dll 中的 LoadLibraryA 函数的地址。然后将这个地址保存到了 lpStartAddress 里面，lpStartAddress 作为 CreateRemoteThread 函数的参数，即在进程 explorer.exe 中创建了一个线程，这个线程执行 LoadLibraryA 函数，而 LoadLibraryA 函数的参数为 lpParameter，也就是 Lab12-01.dll。所以 explorer.exe 就一定会调用 LoadLibraryA 这个函数，而 LoadLibraryA 的参数是 Lab12-01.dll。

所以这个程序的作用就是，在一个远程进程也就是在 explorer.exe 里面创建一个线程，这个线程的功能就是调用 LoadLibraryA 这个函数，而这个函数的参数是 Lab12-01.dll，这个程序执行了 DLL 注入的操作，将 Lab12-01.dll 注入到了 explorer.exe 进程里面。  
所以 explorer.exe 进程被注入。

### 3、如何让恶意代码停止弹出窗口？

重启电脑

或者在调用弹窗函数之前找一个跳转语句，修改它，直接 jmp 到程序结束，就不会弹出了。

.text:004011BB	push   edx
.text:004011BC	call   EnumProcesses
.text:004011C2	test   eax, eax
<b>.text:004011C4</b>	<b>jnz   short loc_4011D0</b>
.text:004011C6	mov    eax, 1
.text:004011CB	jmp   loc_401342
.text:004011D0 ; -----	

比如在程序开始的位置，修改为 jmp 004011C6

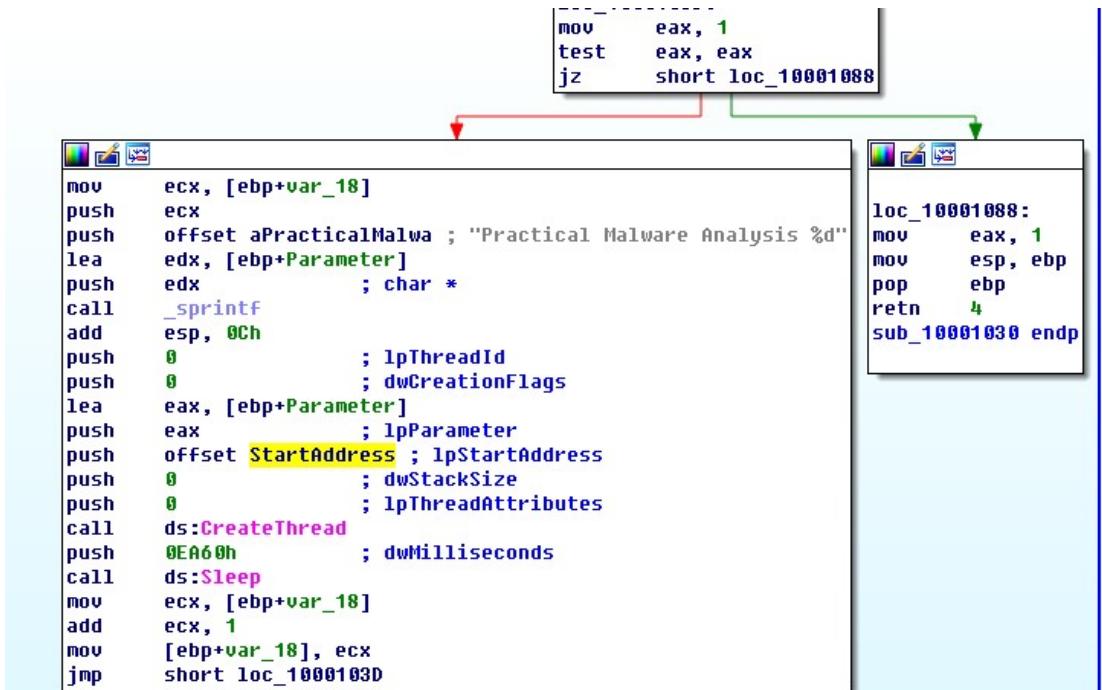
### 4、这个恶意代码如何工作？

由上述分析得知，Lab12-01.exe 将 Lab12-01.dll 注入到 explorer.exe 进程中，那么现在分析一下 dll 文件，开它的功能是什么：

IDA 打开：

```
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
```

程序很简单，就是执行 sub\_10001030 函数，然后创建一个线程，下面分析 sub\_10001030 函数：



发现这个函数就是一个循环，比较 eax 值是不是 1，即之前寻找 explorer.exe 的结果，找到了就是 1，eax 如果是 1 的话，就进入循环主体，这个主体每执行一次就休眠 60 秒。主体内部，StartAddress 作为参数传给 CreateThread 函数，创建一个线程，看看 StartAddress 是什么：

```
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
ret    4
StartAddress endp
```

调用了函数 MessageBoxA，参数是"Press OK to reboot"和 ecx，即弹出窗口，内容是"Press OK to reboot"，标题上的计数器为 ecx。

所以，这个 dll 文件就是每 60 秒弹出一个窗口。

## Lab 12-2

Analyze the malware found in the file *Lab12-02.exe*.

### Questions

1. What is the purpose of this program?
2. How does the launcher program hide execution?
3. Where is the malicious payload stored?
4. How is the malicious payload protected?
5. How are strings protected?

IDA 打开，直接静态分析：

先看看导入表：

Address	Ordinal	Name	Library
00000000...		CloseHandle	KERNEL32
00000000...		VirtualFree	KERNEL32
00000000...		ReadFile	KERNEL32
00000000...		VirtualAlloc	KERNEL32
00000000...		GetFileSize	KERNEL32
00000000...		CreateFileA	KERNEL32
00000000...		ResumeThread	KERNEL32
00000000...		SetThreadContext	KERNEL32
00000000...		WriteProcessMemory	KERNEL32
00000000...		VirtualAllocEx	KERNEL32
00000000...		GetProcAddress	KERNEL32
00000000...		GetModuleHandleA	KERNEL32
00000000...		ReadProcessMemory	KERNEL32
00000000...		GetThreadContext	KERNEL32
00000000...		CreateProcessA	KERNEL32
00000000...		FreeResource	KERNEL32
00000000...		SizeofResource	KERNEL32
00000000...		LockResource	KERNEL32
00000000...		LoadResource	KERNEL32
00000000...		FindResourceA	KERNEL32
00000000...		GetSystemDirectoryA	KERNEL32
00000000...		Sleep	KERNEL32
00000000...		GetCommandLineA	KERNEL32
00000000...		GetVersion	KERNEL32
00000000...		ExitProcess	KERNEL32
00000000...		TerminateProcess	KERNEL32
00000000...		GetCurrentProcess	KERNEL32
00000000		UnhandledExceptionFilter	KERNEL32

看到有 SetThreadContext，CreateProccessA 和 GetThreadContext 函数可以知道会创建新的进程并修改进程中的线程上下文，然后有 ReadProcessMemory 和 WriteProcessMemory 这两个函数知道恶意程序对进程的内存空间进行了读写操作；LockResource 和 SizeofResource 这两个函数让我们知道有可能会对资源节做一定的操作，有可能在资源里保存了一些比较重要的数据。

猜测这个程序从创建进程开始，找到 CreateProccessA 函数所在的位置：

发现是 sub\_4010EA 函数调用的：

```

mov    [ebp+var_4], eax
mov    ecx, [ebp+var_4]
xor    edx, edx
mov    dx, [ecx]
cmp    edx, 5A40h
jnz    loc_40131F

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

```

这个函数开头会检查 PE 和 MZ，说明目标文件是一个 PE 文件，可能创建进程

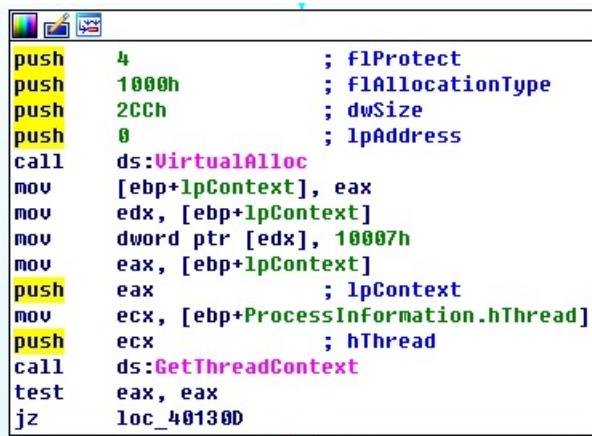
```

push    eax      ; lpStartupInfo
push    0         ; lpCurrentDirectory
push    0         ; lpEnvironment
push    4         ; dwCreationFlags
push    0         ; bInheritHandles
push    0         ; lpThreadAttributes
push    0         ; lpProcessAttributes
push    0         ; lpCommandLine
mov     ecx, [ebp+lpApplicationName]
push    ecx      ; lpApplicationName
call   ds:CreateProcessA
test   eax, eax
jz    loc_401313

```

CreateProcessA 的参数 dwCreationFlags 的值为 4, 表示的是 CREATE\_SUSPENDED, 允许进程被创建, 但是先不启动。只有当主进程调用 ResumeThread 函数时, 它才会被启动。

继续:

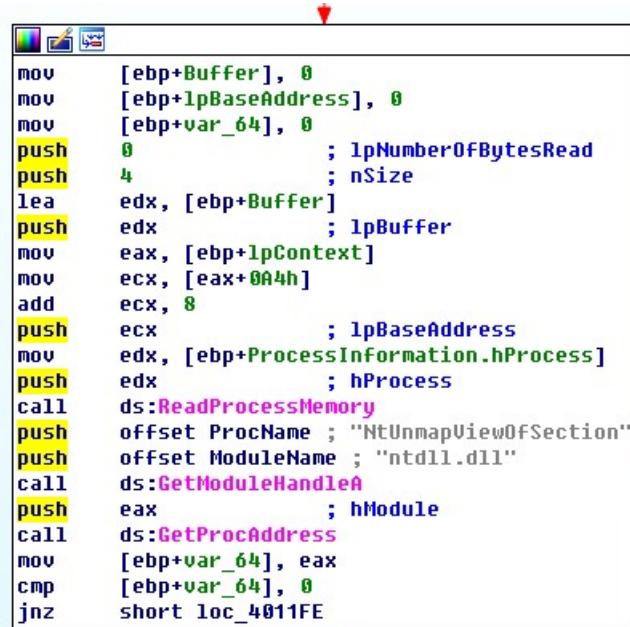


```

push    4          ; f1Protect
push    1000h      ; f1AllocationType
push    2CCh       ; dwSize
push    0          ; lpAddress
call   ds:VirtualAlloc
mov     [ebp+lpContext], eax
mov     edx, [ebp+lpContext]
mov     dword ptr [edx], 10007h
mov     eax, [ebp+lpContext]
push    eax        ; lpContext
mov     ecx, [ebp+ProcessInformation.hThread]
push    ecx        ; hThread
call   ds:GetThreadContext
test   eax, eax
jz    loc_40130D

```

GetThreadContext 函数作用为访问一个线程的上下文, 它的 hThread 参数与 CreateProcessA 函数的 lpProcessInformation 参数一样, 说明这个程序正在访问挂起的进程的上下文;



```

mov    [ebp+Buffer], 0
mov    [ebp+lpBaseAddress], 0
mov    [ebp+var_64], 0
push   0           ; lpNumberOfBytesRead
push   4           ; nSize
lea    edx, [ebp+Buffer]
push   edx, [ebp+lpBaseAddress]
mov    eax, [ebp+lpContext]
mov    ecx, [eax+0A4h]
add    ecx, 8
push   ecx, [ebp+lpBaseAddress]
mov    edx, [ebp+ProcessInformation.hProcess]
push   edx, [ebp+lpBaseAddress]
call   ds:ReadProcessMemory
push   offset ProcName ; "NtUnmapViewOfSection"
push   offset ModuleName ; "ntdll.dll"
call   ds:GetModuleHandleA
push   eax, [ebp+lpBaseAddress]
call   ds:GetProcAddress
mov    [ebp+var_64], eax
cmp    [ebp+var_64], 0
jnz   short loc_4011FE

```

然后调用了 ReadProcessMemory 函数, 参数为刚才得到的进程的上下文, 偏移量 0A4h 引用

了进程的 ebx 寄存器，然后对获取到的位置增加 8，将这个值压栈，作为读取内存的起始地址；

然后调用 GetProcAddress 函数获取 NtUnmapViewOfSection 函数的地址；

```
loc_4011FE:
mov    eax, [ebp+Buffer]
push   eax
mov    ecx, [ebp+ProcessInformation.hProcess]
push   ecx
call   [ebp+var_64]
push   40h          ; f1Protect
push   3000h        ; f1AllocationType
mov    edx, [ebp+var_8]
```

将 buffer 作为 NtUnmapViewOfSection 函数的一个参数入栈。这样的话 NtUnmapViewOfSection 就会将新创建的进程的内存空间释放掉；

之后就可以开始填充恶意代码了：

```
push   40h          ; flProtect
push   3000h        ; flAllocationType
mov    edx, [ebp+var_8]
mov    eax, [edx+50h]
push   eax          ; dwSize
mov    ecx, [ebp+var_8]
mov    edx, [ecx+34h]
push   edx          ; lpAddress
mov    eax, [ebp+ProcessInformation.hProcess]
push   eax          ; hProcess
call   ds:VirtualAllocEx
mov    [ebp+lpBaseAddress], eax
cmp    [ebp+lpBaseAddress], 0
jz    loc_401307
```

调用 VirtualAllocEx 函数，由参数 ProcessInformation.hProcess 可知，这个程序在被挂起的进程的地址空间中分配内存；

首先，flProtect 这个参数的值为 40h，表示的是 PAGE\_EXECUTE\_READWRITE，也就是该区域可以执行代码，应用程序可以读写该区域；然后将指向 PE 字符的指针位置加上 50h，这里指的是内存中映像的总尺寸，这个尺寸就是接下来所要分配的内存大小；之后程序会将指向 PE 字符的指针位置加上 34h，其实也就是指向了映像基址的位置，并将这个位置作为了 lpAddress 的参数，也就是 VirtualAllocEx 函数将要分配的空间的位置。

这样，内存被成功分配，继续：

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

调用 WriteProcessMemory，向内存中写入数据，一开始，将指向 PE 的指针加上 54h，也就是 SizeOfHeaders，程序会将这个地方的数据写入到被挂起的进程所分配的内存中。我们也知道了这个程序正在移动一个 PE 文件到另一个进程的地址空间。

```
loc_401269:
mov    ecx, [ebp+var_8]
xor    edx, edx
mov    dx, [ecx+6]
cmp    [ebp+var_70], edx
jge   short loc_4012B9
```

```
loc_401260:
mov    eax, [ebp+var_70]
add    eax, 1
mov    [ebp+var_70], eax
```

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

接下来是一个循环结构，意思就是将每一个节表复制到被挂起的进程中，到这，恶意程序已经完成了加载一个可执行文件到另一个进程地址空间的工作，循环完跳出循环，进入 loc\_4012b9：

```
loc_4012B9:          ; lpNumberOfBytesWritten
push    0
push    4          ; nSize
mov     edx, [ebp+var_8]
add    edx, 34h
push    edx          ; lpBuffer
mov     eax, [ebp+lpContext]
mov     ecx, [eax+0A4h]
add    ecx, 8
push    ecx          ; lpBaseAddress
mov     edx, [ebp+ProcessInformation.hProcess]
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
```

调用 SetThreadContext 函数，将 eax 加载到挂起的进程内存空间中的可执行文件的入口点，

然后执行 ResumeThread 函数，将之前 CreateProcessA 所创建的进程替换为另一个进程。

所以这个恶意代码就是实现了进程的替换，下面来分析，替换了哪个进程，被替换成什么了：

找到 CreateProcessA 函数的 lpApplicationName 参数，到主程序中找，看看传进来的是什么：

```
mov     edx, [ebp+lpAddress]
push    edx          ; lpBuffer
lea    eax, [ebp+ApplicationName]
push    eax          ; lpApplicationName
call    sub_4010EA
add    esp, 8
push    400h          ; size_t
push    0            ; int
lea    ecx, [ebp+ApplicationName]
push    ecx          ; void *
call    _memset
add    esp, 0Ch
push    8000h         ; dwFreeType
push    0            ; dwSize
mov     edx, [ebp+lpAddress]
push    edx          ; lpAddress
call    ds:VirtualAlloc
```

可以看到[ebp+ApplicationName]保存的就是进程的名称，向上找：

```

push    400h          ; uSize
lea     eax, [ebp+ApplicationName]
push    eax           ; lpBuffer
push    offset aSvchost_exe : "\\svchost.exe"
call    sub_40149D
add    esp, 0Ch
mov    ecx, [ebp+hModule]
push    ecx           ; hModule
call    sub_40132C
add    esp, 4
mov    [ebp+lpAddress], eax
cmp    [ebp+lpAddress], 0
jz     short loc_401573

```

查看 sub\_40149d 函数:

```

; int __cdecl sub_40149D(char *, LPSTR lpBu
sub_40149D proc near

arg_0= dword ptr  8
lpBuffer= dword ptr  0Ch
uSize= dword ptr  10h

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

```

就是整合系统目录，得到 svchost.exe 的路径，所以被替换的进程就是 svchost 进程，即创建一个 svchost 进程，然后替换掉。那么替换成什么呢？

```

; Attributes: bp-based frame

; int __cdecl sub_4010EA(LPCSTR lpApplicationName, LPCVOID lpBuffer)
sub_4010EA proc near

var_74= dword ptr -74h
var_70= dword ptr -70h
Buffer= dword ptr -6Ch
lpBaseAddress= dword ptr -68h
var_64= dword ptr -64h
lpContext= dword ptr -60h
StartupInfo= _STARTUPINFOA ptr -5Ch
ProcessInformation= _PROCESS_INFORMATION ptr -18h
var_8= dword ptr -8
var_4= dword ptr -4
lpApplicationName= dword ptr 8
lpBuffer= dword ptr 0Ch

push    ebp
mov     ebp, esp
sub    esp, 74h
mov     eax, [ebp+lpBuffer]
mov     [ebp+var_1], eax
mov     ecx, [ebp+var_4]
xor     edx, edx
mov     dx, [ecx]
cmp     edx, 5A4Dh
jnz    loc_40131F

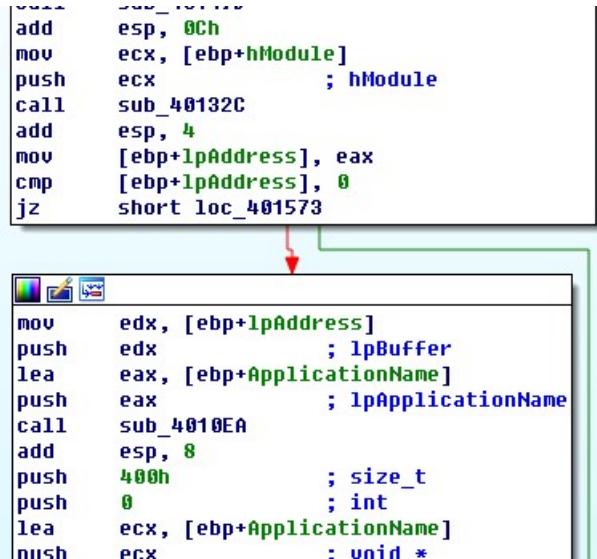
```

回到 4010ea，可知，应该分析参数 lpBuffer，再回到主程序，找到其对应的实参：

```

add    esp, 0Ch
mov    ecx, [ebp+hModule]
push   ecx          ; hModule
call   sub_40132C
add    esp, 4
mov    [ebp+lpAddress], eax
mov    [ebp+lpAddress], 0
cmp    [ebp+lpAddress], 0
jz    short loc_401573

```

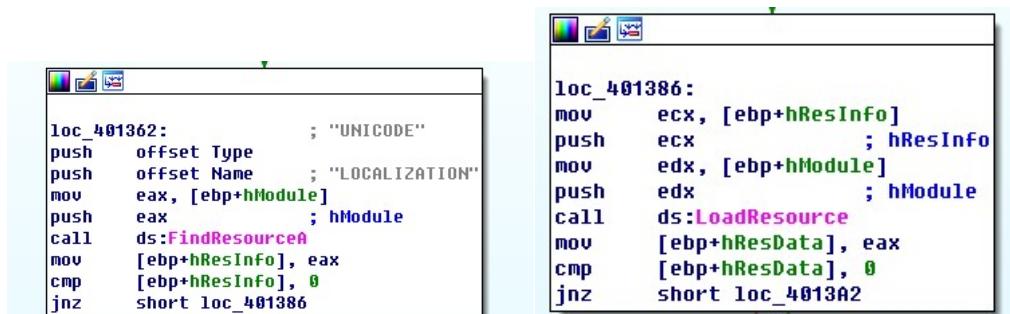


```

mov    edx, [ebp+lpAddress]
push   edx          ; lpBuffer
lea    eax, [ebp+ApplicationName]
push   eax          ; lpApplicationName
call   sub_4010EA
add    esp, 8
push   400h          ; size_t
push   0             ; int
lea    ecx, [ebp+ApplicationName]
push   ecx          : unid *

```

可以发现 lpBuffer 的实参是 sub\_40132C 函数的返回值，分析之：



```

loc_401362:           ; "UNICODE"
push    offset Type
push    offset Name   ; "LOCALIZATION"
mov    eax, [ebp+hModule]
push   eax          ; hModule
call   ds:FindResourceA
mov    [ebp+hResInfo], eax
cmp    [ebp+hResInfo], 0
jnz    short loc_401386

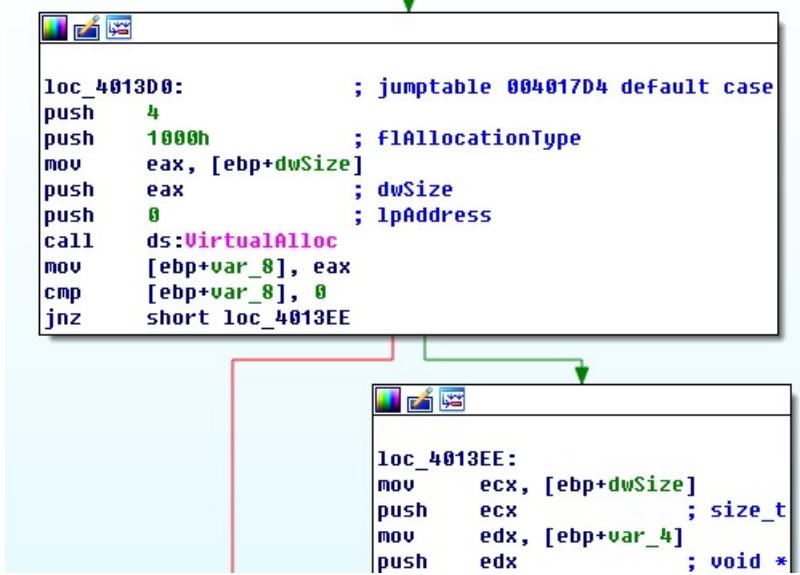
```

```

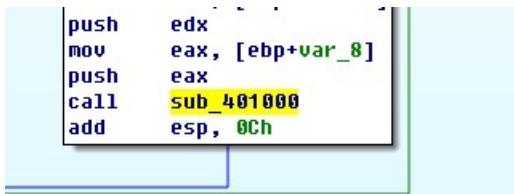
loc_401386:
mov    ecx, [ebp+hResInfo]
push   ecx          ; hResInfo
mov    edx, [ebp+hModule]
push   edx          ; hModule
call   ds:LoadResource
mov    [ebp+hResData], eax
cmp    [ebp+hResData], 0
jnz    short loc_4013A2

```

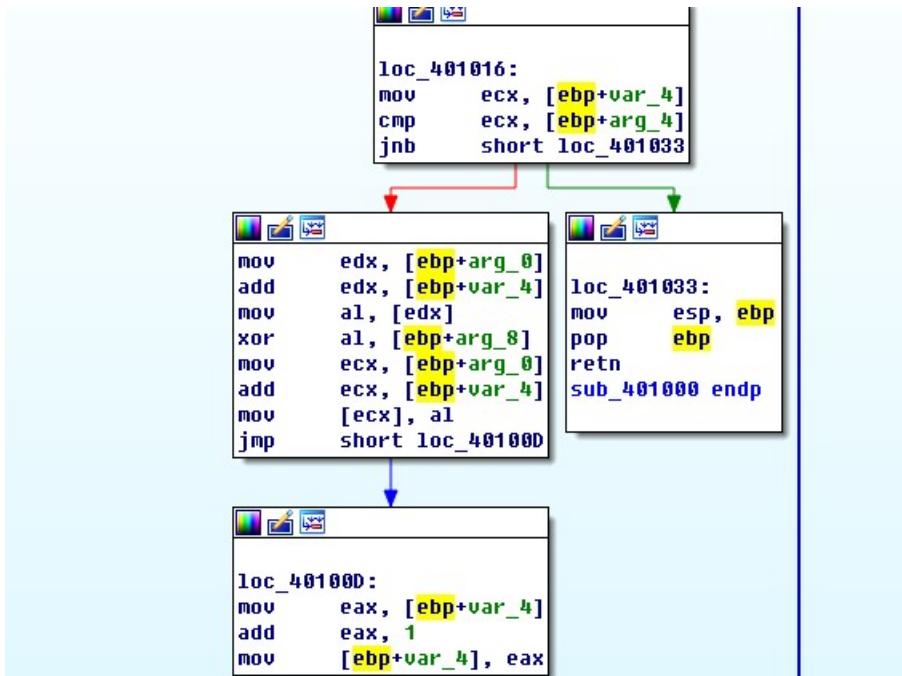
到当前可执行文件的资源段找到名为 LOCALIZATION 的资源，然后加载；



然后经过计算资源大小等准备工作，将资源加载到进程内存；



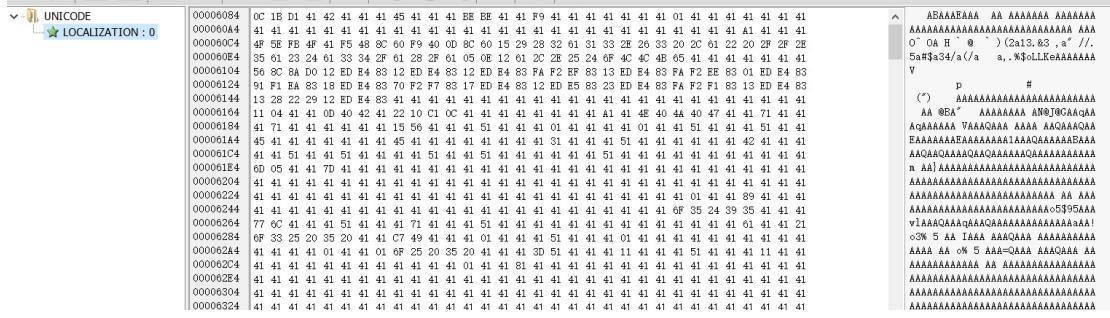
结果调用了这个函数，查看：



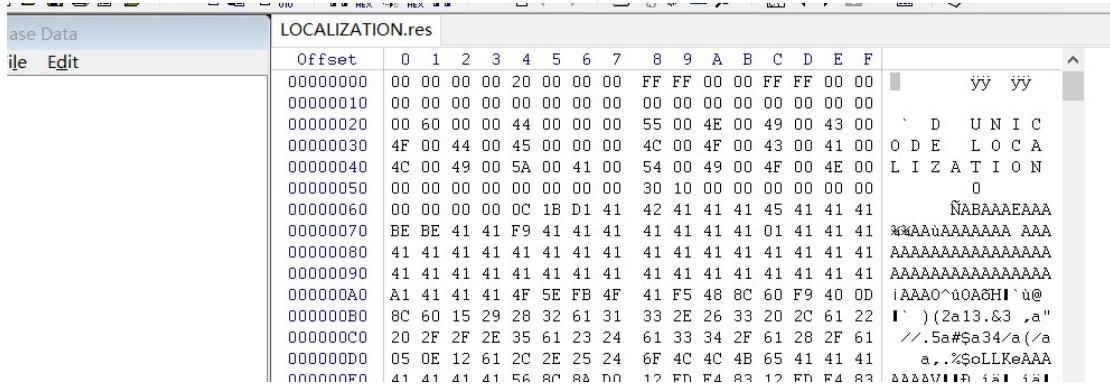
得知，是异或解密，xor 41h，资源可能是被加密了。

这样就知道了，这个恶意代码就是将新建的 svchost 进程替换了可执行文件中的资源，而这个资源经过解密，发现也是一个可执行程序，应该是隐藏的恶意程序。

## 用 Resource Hacker 提取文件:



提取资源文件，用 WinHex 打开:



经过 xor 41h 解密后:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	41	41	41	41	61	41	41	41	BE	BE	41	41	BE	BE	41	41
00000010	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000020	41	21	41	41	05	41	41	41	14	41	0F	41	08	41	02	41
00000030	0E	41	05	41	04	41	41	41	0D	41	0E	41	02	41	00	41
00000040	0D	41	08	41	1B	41	00	41	15	41	08	41	0E	41	0F	41
00000050	41	41	41	41	41	41	41	41	71	51	41	41	41	41	41	41
00000060	41	41	41	41	4D	5A	90	00	03	00	00	00	04	00	00	00
00000070	FF	FF	00	00	B8	00	00	00	00	00	00	40	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	E0	00	00	00	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C
000000B0	CD	21	54	68	69	73	20	70	72	6F	67	72	61	6D	20	63
000000C0	61	6E	6E	6F	74	20	62	65	20	72	75	6E	20	69	6E	20
000000D0	44	4F	53	20	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00
000000E0	00	00	00	00	17	CD	CB	91	53	AC	A5	C2	53	AC	A5	C2
000000F0	53	AC	A5	C2	BB	B3	AE	C2	52	AC	A5	C2	BB	B3	AF	C2
00000100	40	AC	A5	C2	DO	BO	AB	C2	59	AC	A5	C2	31	B3	B6	C2
00000110	56	AC	A5	C2	53	AC	A4	C2	62	AC	A5	C2	BB	B3	BO	C2
00000120	52	AC	A5	C2	52	69	63	68	53	AC	A5	C2	00	00	00	00
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

确实是可执行文件。

1、这个程序的目的是什么？

创建新的 svchost.exe 进程，然后用 exe 文件中隐藏的资源替换这个进程，也就是创建另一个恶意程序。

2、启动器代码如何隐藏执行？

使用替换技术，将新建的进程内存清空，然后重新装载资源。

3、恶意代码的负载存储在哪？

真实的功能代码被保存在这个程序的资源节中，且是经过加密的代码。

4、恶意负载是如何被保护的？

通过异或加密保护。

5、字符串列表如何被保护？

通过异或加密保护。

## Lab 12-3

Analyze the malware extracted during the analysis of Lab 12-2, or use the file *Lab12-03.exe*.

### Questions

1. What is the purpose of this malicious payload?
2. How does the malicious payload inject itself?
3. What filesystem residue does this program create?

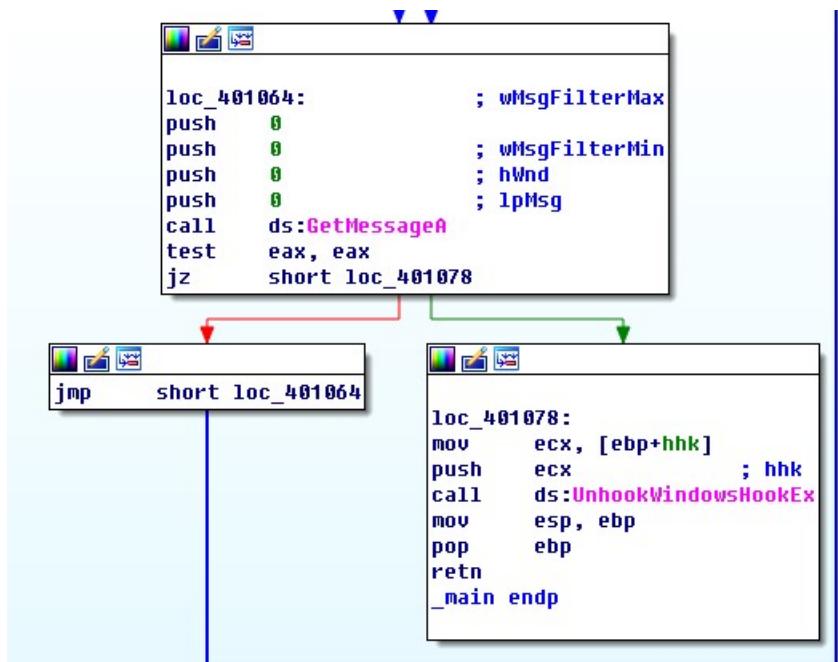
直接 IDA 分析：

The screenshot shows the assembly view of the IDA Pro debugger. The code is written in Intel x86 assembly language. It starts with a push of 400h onto the stack, followed by pushes of 1 and the offset of byte\_405350. A call to \_memset is made, followed by an add of esp, 0Ch. Then, two zeros are pushed onto the stack, followed by calls to ds:GetModuleHandleA and ds:SetWindowsHookExA. Finally, a mov instruction moves [ebp+hk] to eax.

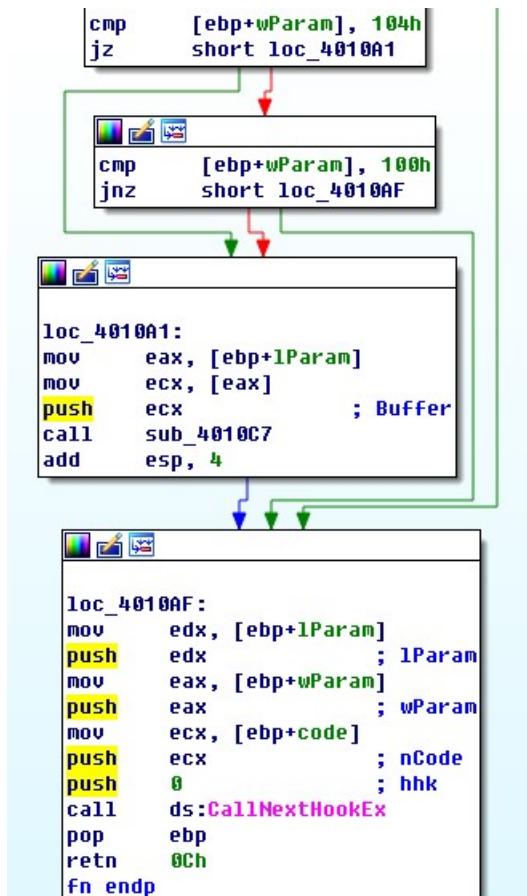
```
loc_401035:          ; size_t
push    400h
push    1             ; int
push    offset byte_405350 ; void *
call    _memset
add    esp, 0Ch
push    0              ; dwThreadId
push    0              ; lpModuleName
call    ds:GetModuleHandleA
push    eax            ; hmod
push    offset fn      ; lpfn
push    0Dh            ; idHook
call    ds:SetWindowsHookExA
mov     [ebp+hk], eax
```

首先通过 GetModuleHandleA 函数获得一个函数地址，其实是 fn 函数的地址，fn 函数启用键盘事件监控。然后调用 SetWindowsHookExA 函数，这个函数就是挂钩函数，用于监控 WinAPI 函数，参数 idhook 的值为 Odh 表示 WH\_KEYBOARD\_LL，作用是监控键盘消息，

参数 lpfn 表示 HOOK 函数地址，这里是前边获得的 fn 函数的地址。



之后是一个无限循环，反复调用 GetMessageA 函数，将消息发送到钩子函数，也就是前边的 fn 函数，即监控键盘，每次获取到消息，就将消息发送给 fn 函数，下面来看看 fn 函数：



刚开始的两个比较判断按键类型，然后将虚拟按键码传递给 sub\_4010c7 函数，进入：

```

push    2          ; dwShareMode
push    40000000h   ; dwDesiredAccess
push    offset FileName ; "practicalmalwareanalysis.log"
call    ds>CreateFileA
mov     [ebp+hFile], eax
cmp     [ebp+hFile], 0FFFFFFFh
jnz    short loc_4010FF

```

创建了一个名为 practicalmalwareanalysis.log 的文件；

```

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

```

调用了 GetForegroundWindow 和 GetWindowTextA 函数来获取按键的来源；

如果程序将窗口标题写入了日志文件就会执行 loc\_401202 函数：

```

loc_401202:
mov     edx, [ebp+Buffer]
mov     [ebp+var_C], edx
mov     eax, [ebp+var_C]
sub     eax, 8
mov     [ebp+var_C], eax
cmp     [ebp+var_C], 61h ; switch 98 cases
ja     loc_40142C        ; jmp table 00401226 default case

```

如果键盘输入 0x10，即 16，这里减 8 还是 8 然后进入一个跳表：

```

mov     edx, [ebp+var_C]
xor     ecx, ecx
mov     cl, ds:byte_40148D[edx]
jmp     ds:off_401441[ecx*4] ; switch jump

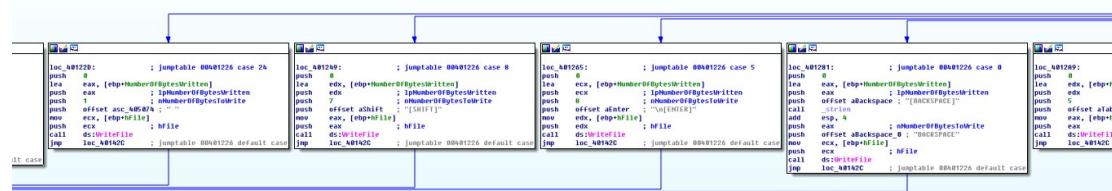
```

即到 byte\_40148d 中去寻找 8：

Offset	Value	Description
0	0	
1	1	
2	2	
3	3	highlighted
4	4	
5	5	
12h	12h	
12h	12h	

找到了为 3，然后 3 再乘 4，即 12，作为 off\_401441 的偏移，跳到 loc\_401249 的位置：

然后将字符串[SHIFT]写入日志文件，这只是一个例子，若键盘输入别的也会有对应的输出到日志文件：



1、这个恶意负载的目的是什么？

键盘记录器。

## 2、恶意负载如何注入自身？

SetWindowsHookEx 函数进行挂钩函数注入。

3、这个程序还创建了哪些其他文件？

创建了 practicalmalwareanalysis.log 日志文件，保存键盘记录。

## **Lab 12-4**

Analyze the malware found in the file *Lab12-04.exe*.

## Questions

1. What does the code at 0x401000 accomplish?
2. Which process has code injected?
3. What DLL is loaded using LoadLibraryA?
4. What is the fourth argument passed to the CreateRemoteThread call?
5. What malware is dropped by the main executable?
6. What is the purpose of this and the dropped malware?

查看导入表：

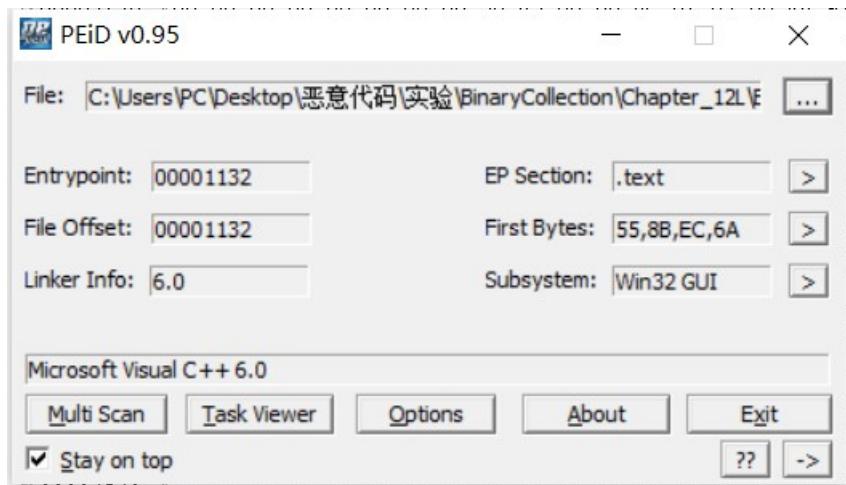
00000000...	WriteFile	KERNEL32
00000000...	CreateFileA	KERNEL32
00000000...	SizeofResource	KERNEL32
00000000...	CreateRemoteThread	KERNEL32
00000000...	FindResourceA	KERNEL32
00000000...	GetModuleHandleA	KERNEL32
00000000...	GetWindowsDirectoryA	KERNEL32
00000000...	MoveFileA	KERNEL32
00000000...	GetTempPathA	KERNEL32
00000000...	GetCurrentProcess	KERNEL32
00000000...	OpenProcess	KERNEL32
00000000...	CloseHandle	KERNEL32
00000000...	LoadResource	KERNEL32
00000000...	snprintf	MSVCRT

看到函数 CreateRemoteThread，用于创建远程线程，还发现了 FindResource 和 LoadResource

这两个关于资源操作的函数，所以资源节有东西，查看：



看到了 4d5a，是 PE 文件，提取出来，用 PEiD 查看，发现是 exe 文件：



动态分析：

运行程序，用 Wireshark 监控，可以发现恶意程序想打开一个网址，用 Process Monitor 监控，发现程序在系统目录下创建了一个名为 wupdmg.exe 的文件，经过 MD5 值比较，这跟之前提取出来的 exe 文件一样

SetEndOfFileInformat... C:\WINDOWS\system32\config\software.LOG	SUCCESS
CreateFile C:\WINDOWS\system32\wupdmg.exe	SUCCESS
QueryAttributeTagFile C:\WINDOWS\system32\wupdmg.exe	SUCCESS
<hr/>	
d41d8cd98f00b204e9800998ecf8427e 12-04.exe	
d41d8cd98f00b204e9800998ecf8427e wupdmg.exe	
<hr/>	
.a...GE T /updat er.exe H TTP/1.1. Accept: */*.Ac cept-Enc oding: g zip, def late..U er-Agent : Mozill a/4.0 (c ompatib e; MSIE 6.0; wi ndows NT 5.1; SV1 .NET CLR 2.0.5 0727; .NET CLR 3 0.4506. 2152; .N ET CLR 3 .5.3072 .Host: www.pra cticalmalwareana lysis.co m..Conn	.a...GE T /upda er.exe H TTP/1.1. .Accept: */*.A cept-Enc oding: zip, def late..U er-Agent : Mozil a/4.0 (c ompatib e; MSIE 6.0; wi ndows NT 5.1; SV .NET CLR 2.0. 0727; .NET CLR 3 0.4506. 2152; .NET CLR 3 .5.3072 .Host: www.pr acticalmalwarean alysis.co m..Conn

可以看到恶意程序试图从 www.practicalmalwareanalysis.com 中下载 updater.exe

静态分析：

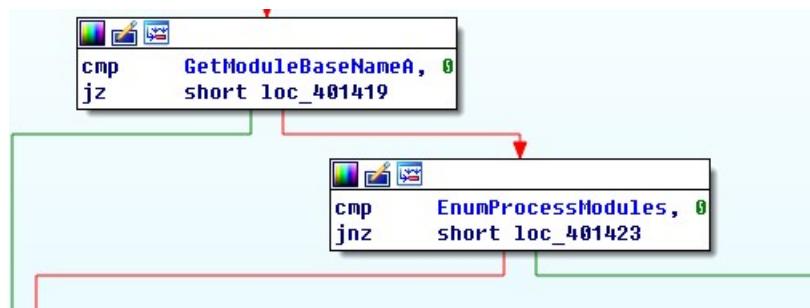
这个程序开始和 12-01 很像

```

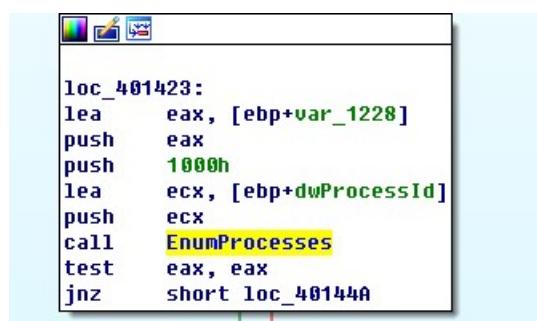
stosb
mov    [ebp+var_1234], 0
mov    [ebp+var_122C], 0
push   offset ProcName ; "EnumProcessModules"
push   offset aPsapi_dll ; "psapi.dll"
call   ds:LoadLibraryA
push   eax                ; hModule
call   ds:GetProcAddress
mov    dword_40312C, eax
push   offset aGetmodulebasen ; "GetModuleBaseNameA"
push   offset aPsapi_dll_0 ; "psapi.dll"
call   ds:LoadLibraryA
push   eax                ; hModule
call   ds:GetProcAddress
mov    dword_403128, eax
push   offset aEnumprocesses ; "EnumProcesses"
push   offset aPsapi_dll_1 ; "psapi.dll"
call   ds:LoadLibraryA
push   eax                ; hModule
call   ds:GetProcAddress
mov    dword_403124, eax
cmp   dword_403124, 0
jz    short loc_401419

```

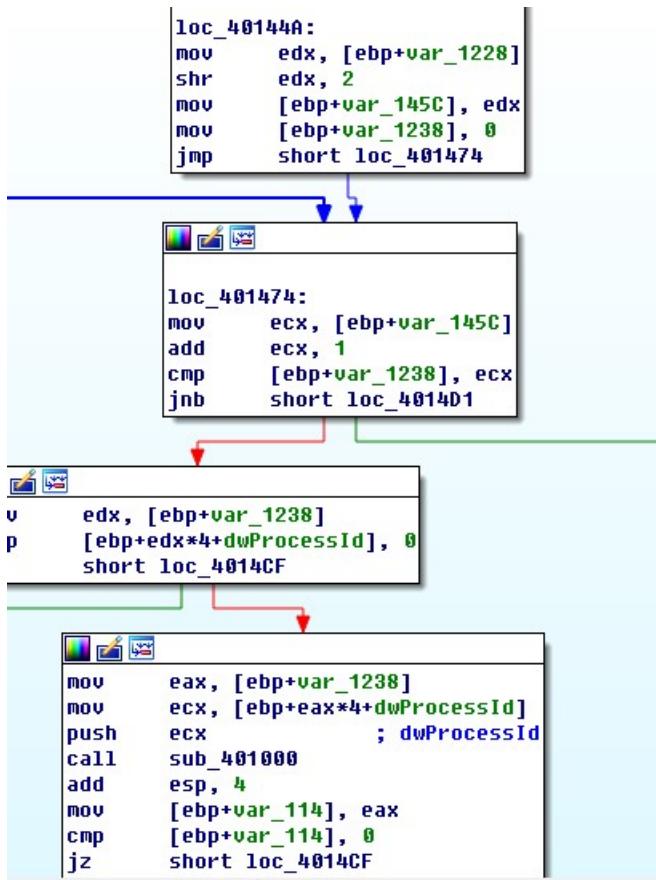
程序解析"psapi.dll"中用于进程枚举的函数，使用了三组 LoadLibrary 和 GetProcAddress 函数组合来解析"EnumProcessModules", "GetModuleBaseNameA"和"EnumProcesses"这三个函数。然后将获取到的地址分别保存到 dword\_40312C, dword\_403128, dword\_403124 中，对这三个地址重命名，便于后续分析。



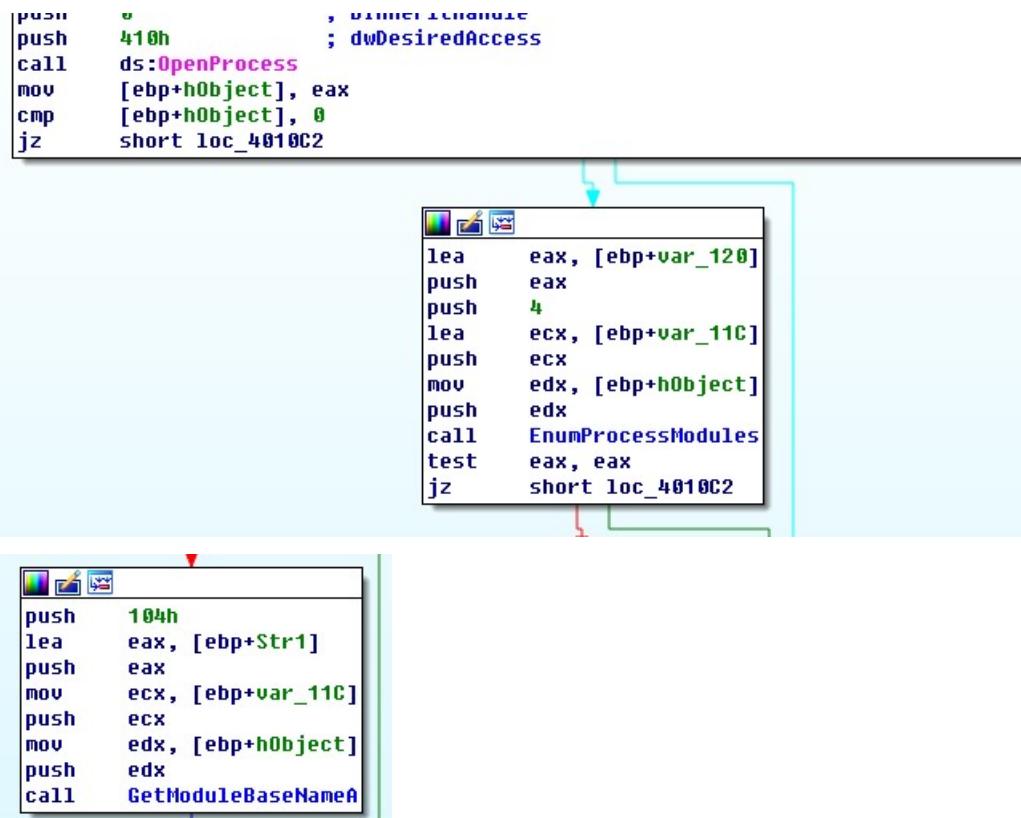
然后检查这几个指针的值：



调用 EnumProcessModules 函数枚举当前的进程，返回进程的 PID，并且保存在 dwProcessId 中。



接着程序进入循环，将每个 PID 值经过运算，作为参数传递给 sub\_401000，进入这个函数：



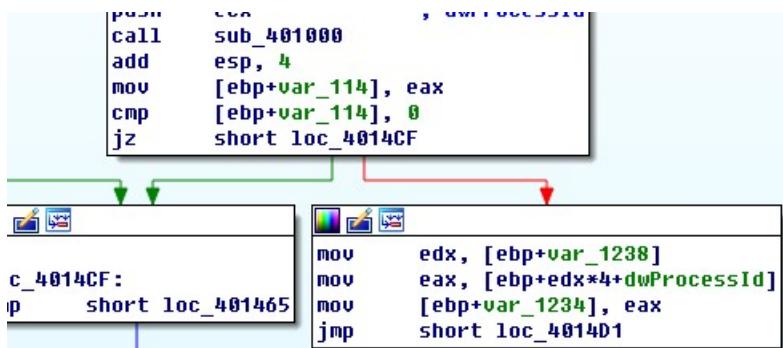
挨个获取进程 id，并把名赋值给，Str1;；

```

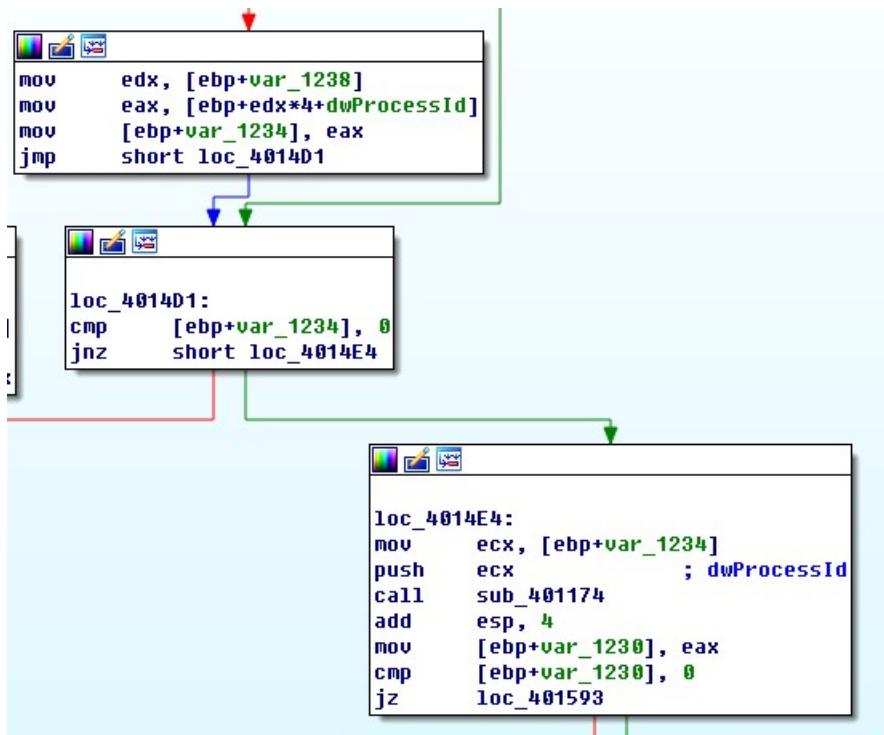
loc_4010C2:
    lea     eax, [ebp+Str2]
    push    eax           ; Str2
    lea     ecx, [ebp+Str1]
    push    ecx           ; Str1
    call    ds:_strcmp
    add    esp, 8
    test   eax, eax
    jnz    short loc_4010EB

```

然后将 Str1 和 Str2 进行比较，Str2 就是字符串 winlogon.exe，若两个相同则 eax 赋值为 1 返回。 (Q1-ANS)



这里会对 sub\_401000 的返回值与 0 进行比较，如果相等就左边，继续寻找，不等，右边执行：



dwProcessId 作为参数，传递给 sub\_401174，进入：

```

push    ebp
mov     ebp, esp
sub    esp, 0Ch
mov    [ebp+var_4], 0
mov    [ebp+hProcess], 0
mov    [ebp+var_C], 0
push    offset aSedebugprivile ; "SeDebugPrivilege"
call    sub_4010FC
test   eax, eax
jz     short loc_4011A1

```

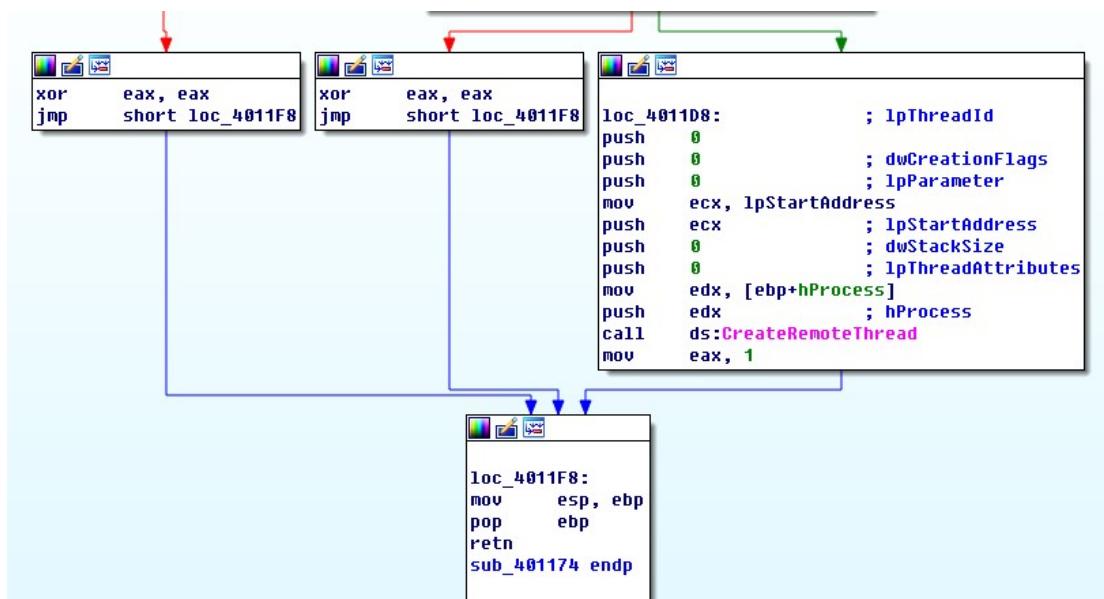
调用 sub\_4010fc，由参数和内部分析可知，作用为权限提升；

```

loc_4011A1:          ; lpProcName
push    2
push    offset LibFileName ; "sfc_os.dll"
call    ds:LoadLibraryA
push    eax           ; hModule
call    ds:GetProcAddress
mov    lpStartAddress, eax
mov    eax, [ebp+dwProcessId]
push    eax           ; dwProcessId
push    0              ; bInheritHandle
push    1F0FFFh        ; dwDesiredAccess
call    ds:OpenProcess
mov    [ebp+hProcess], eax
cmp    [ebp+hProcess], 0
jnz    short loc_4011D8

```

调用了函数 LoadLibraryA，这个函数的目的就是装载 sfc\_os.dll 这个动态链接库，并且利用函数 GetProcAddress 来获取 sfc\_os.dll 这个动态链接库中编号为 2 的地址。并将这个地址保存到 lpStartAddress 里面，然后程序会调用 OpenProcess 函数，它会打开 winlogon.exe 进程，并将它的句柄保存在 hProcess 里面；



调用了 CreateRemoteThread 函数，参数之一为 hProcess，也就是 winlogon.exe 的句柄，另一个参数为 lpStartAddress，即为 sfc\_os.dll 中序号为 2 的函数的地址，负责向 winlogon.exe 中

进行远程注入，所以 sfc\_os.dll 会被系统自动载入到 winlogon.exe 里面，实现 DLL 注入。

(Q2-ANS)

1、位置 0x401000 处的代码完成了什么功能？

检查传入的 PID 值是否为 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？

```
push    z
push    offset LibFileName ; "sfc_os.dll"
call    ds:LoadLibraryA
push    eax                ; hModule
sfc_os.dll
```

4、传递给 CreateRemoteThread 函数的第 4 个参数是什么？

```
loc_4011D8:          ; lpThreadId
push    0
push    0           ; dwCreationFlags
push    0           ; lpParameter
mov    ecx, lpStartAddress
push    ecx          ; lpStartAddress
push    0           ; dwStackSize
push    0           ; lpThreadAttributes
mov    edx, [ebp+hProcess]
push    edx          ; hProcess
call    ds>CreateRemoteThread
mov    eax, 1
```

lpStartAddress

5、可执行文件主程序释放了哪个恶意代码？

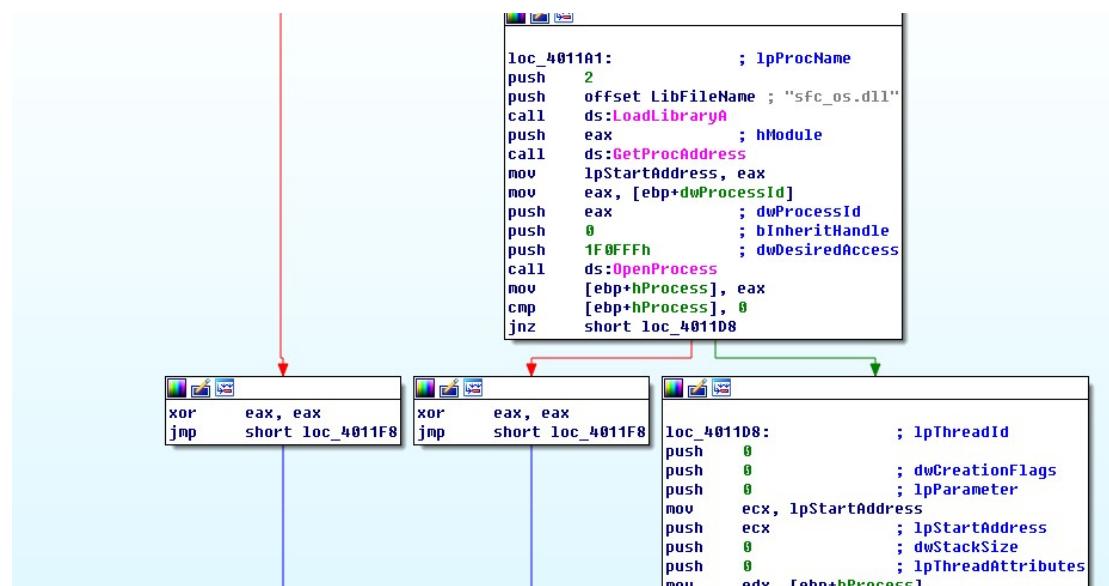
回到主程序继续看（401174 之后）

```

push 10Eh           ; uSize
lea   edx, [ebp+Buffer]
push  edx           ; lpBuffer
call ds:GetWindowsDirectoryA
push offset aSystem32Wupd_0 ; "\\system32\\wupdmgmgr.exe"
lea   eax, [ebp+Buffer]
push  eax
push offset aSS_0    ; "%5%5"
push  10Eh           ; Count
lea   ecx, [ebp+Dest]
push  ecx           ; Dest
call ds:_snprintf
add  esp, 14h
lea   edx, [ebp+var_110]
push  edx           ; lpBuffer
push  10Eh           ; nBufferLength
call ds:GetTempPathA
push offset aWinup_exe ; "\\winup.exe"
lea   eax, [ebp+var_110]
push  eax
push offset aSS_1    ; "%5%5"
push  10Eh           ; Count
lea   ecx, [ebp+NewFileName]
push  ecx           ; Dest
call ds:_snprintf
add  esp, 14h
lea   edx, [ebp+NewFileName]
push  edx           ; lpNewFileName
lea   eax, [ebp+Dest]
push  eax           ; lpExistingFileName
call ds:MoveFileA
call sub_4011FC
xor  eax, eax
jmp  short loc_401598

```

调用函数 GetWindowsDirectoryA，获取 Windows 目录的路径，然后将其与 \\system32\\wupdmgmgr.exe 进行组合，然后调用 GetTempPathA 来获取临时目录的路径，并且与 “\\winup.exe” 进行组合，存储在了 NewFileName 里面。利用 MoveFileA，把 wupdmgmgr.exe 改名并且复制到了临时文件夹中，然后调用函数 sub\_4011FC，进入：



调用函数 GetWindowsDirectoryA 获取 Windows 的路径，然后与 system32\\wupdmgmgr.exe 进行组合。这个恶意程序正在提取它自身的资源节到硬盘上。然后程序会将提取出来的文件写入

到 wupdmg.exe 文件。

## 6、释放出恶意代码的目的是什么？

这个恶意程序自己会进行更新，把下载的 updater.exe 程序保存到 window 下的 system32 目录中，并且重命名为 wupdmg.exe.

## Yara 规则

Lab12-01.exe

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

```
rule Lab12_1_feature{
    meta:
        description = "Lab12-01.exe's features"
    strings:
        $s1 = "Lab12-01.dll"
        $s2 = "psapi.dll"
        $s3 = "EnumProcessModules"
    condition:
        $s1 and $s2 and $s3
}
```

### Lab12-02.exe

\$.rdata:004043F0 00000010	C	GetActiveWindow
\$.rdata:00404400 0000000C	C	MessageBoxA
\$.rdata:0040440C 0000000B	C	user32.dll
\$.rdata:004046B8 0000000D	C	KERNEL32.dll
\$.data:00405030 0000000D	C	\svchost.exe
\$.data:00405040 00000015	C	NtUnmapViewOfSection
\$.data:00405058 0000000A	C	ntdll.dll
\$.data:00405064 00000008	C	UNICODE
\$.data:0040506C 0000000D	C	LOCALIZATION
\$.data:004051CC 00000006	C	'Dy!'

---

rule Lab12\_2\_feature{

meta:

description = "Lab12-02.exe' s features"

strings:

\$s1 = "NtUnmapViewOfSection"

\$s2 = "LOCALIZATION"

\$s3 = "\svchost.exe"

condition:

\$s1 and \$s2 and \$s3

}

### Lab12-03.exe

\$.data:00405030 0000000C	C	\r\n[Window:
\$.data:00405040 00000013	C	ConsoleWindowClass
\$.data:00405054 0000001D	C	practicalmalwareanalysis.log
\$.data:00405078 00000008	C	[SHIFT]
\$.data:00405080 00000009	C	\n[ENTER]
\$.data:0040508C 0000000C	C	[BACKSPACE]
\$.data:00405098 0000000A	C	BACKSPACE
\$.data:004050A4 00000006	C	[TAB]
\$.data:004050AC 00000007	C	[CTRL]
\$.data:004050B4 00000006	C	[DEL]
\$.data:004050E4 0000000C	C	[CAPS LOCK]
\$.data:004050F0 0000000C	C	[CAPS LOCK]
\$.data:0040524C 00000006	C	'Dy!'

---

rule Lab12\_3\_feature{

meta:

description = "Lab12-03.exe' s features"

strings:

\$s1 = "practicalmalwareanalysis.log"

\$s2 = "[SHIFT]"

```

$S3 = "BACKSPACE"

condition:

$S1 and $S2 and $S3

}

```

### Lab12-04.exe

Address	Length	Type	String
'S' .rdata:0040228E	0000000D	C	KERNEL32.dll
'S' .rdata:004022E0	0000000D	C	ADVAPI32.dll
'S' .rdata:004022FA	0000000B	C	MSVCRT.dll
'S' .data:0040302C	00000011	C	SeDebugPrivilege
'S' .data:00403040	0000000B	C	sfc_os.dll
'S' .data:0040304C	00000016	C	\system32\wupdmgr.exe
'S' .data:00403064	00000005	C	%s%s
'S' .data:00403070	00000005	C	#101
'S' .data:00403078	00000013	C	EnumProcessModules
'S' .data:0040308C	0000000A	C	psapi.dll
'S' .data:00403098	00000013	C	GetModuleBaseNameA
'S' .data:004030AC	0000000A	C	psapi.dll
'S' .data:004030B8	0000000E	C	EnumProcesses
'S' .data:004030C8	0000000A	C	psapi.dll
'S' .data:004030D4	00000016	C	\system32\wupdmgr.exe
'S' .data:004030EC	00000005	C	%s%s
'S' .data:004030F4	0000000B	C	\winup.exe
'S' .data:00403100	00000005	C	%s%s

```

rule Lab12_4_feature{

meta:

```

description = "Lab12-04.exe's features"

strings:

\$S1 = "\winup.exe"

\$S2 = "\system32\wupdmgrd.exe"

\$S3 = "%s%s"

condition:

\$S1 and \$S2 and \$S3

}

测试结果:

```
C:\Users\PC\Desktop\恶意代码\实验\yara64>yara64 Lab12_rules.txt C:\Users\PC\Desktop\恶意代码\实验\BinaryCollection\Chapter_12L  
Lab12_1_feature C:\Users\PC\Desktop\恶意代码\实验\BinaryCollection\Chapter_12L\Lab12-01.exe  
Lab12_2_feature C:\Users\PC\Desktop\恶意代码\实验\BinaryCollection\Chapter_12L\Lab12-02.exe  
Lab12_4_feature C:\Users\PC\Desktop\恶意代码\实验\BinaryCollection\Chapter_12L\Lab12-04.exe  
Lab12_3_feature C:\Users\PC\Desktop\恶意代码\实验\BinaryCollection\Chapter_12L\Lab12-03.exe
```

## IDA Python 脚本

1、快速定位到 init\_array 函数：

```
def goInitarray(self):  
    # _get_modules 是 idc 提供的接口  
    for module in idc._get_modules():  
        # 遍历所有 module，找到 linker  
        module_name = module.name  
        if 'linker' in module_name:  
            print 'linker address is ' + str(hex(module.base + 0x2464))  
            # 0x2464 是 Android 某个版本的 init_array 的偏移地址，  
            # jumpto 可以直接跳转到目标地址  
            idc.jumpto(module.base + 0x2464)  
            # 在 init_array 上下个断点  
            idc.add_bpt(module.base + 0x2464, 1)  
            # makecode 更不用说了，相当于 C  
            idaapi.auto_make_code(module.base + 0x2464)
```

2、保存日志、函数名字

即保存某些寄存器的值或者某个函数名之类的，方便快速回到调试之前

```
# 通过起始地址，终止地址，以及偏移地址去保存日志
```

```
def saveDebugMessage(self):  
    # create file first  
    # 用个轻量级的存储 shelve  
    f = shelve.open(self.id)  
    # 保存日志的起始地址  
    addr_start = int(self.address_start, 16)  
    # 保存日志的终止地址
```

```
addr_end = int(self.address_end, 16)

log_dict = {}

log_dict_list = []

for num in range(addr_start, addr_end):

    # 获取我们当前地址的日志

    com = idc.GetCommentEx(num, True)

    if com != None:

        #获取函数名

        fun_name = idc.GetFunctionName(num)

        print fun_name

        if fun_name != None and not 'sub' in fun_name:

            log_dict = {'offset': str(num - addr_start), 'msg': str(com), 'function_name': str(fun_name)}

    else:

        log_dict = {'offset': str(num - addr_start), 'msg': str(com)}

    log_dict_list.append(log_dict)

    pass

print(log_dict_list)

# 保存日志

f['info'] = log_dict_list

f.close()

# 通过起始地址即可，会自动判断长度，并且获取偏移地址去设置日志

def loadDebugMessage(self):

    f = shelve.open(self.id)

    data = f['info']

    addr_start = int(self.address_start, 16)

    for num in range(0, len(data)):

        offset = data[num]['offset']

        msg = data[num]['msg']

        fun_name = data[num]['function_name']
```

```
idc.MakeRptCmt(addr_start + int(offset), msg)
if fun_name is not None and fun_name != "":
    idc.SetFunctionCmt(addr_start + int(offset), fun_name, False)
```

## 四、 实验心得

本次实验，再次进行深入静态、动态分析，代码注入、释放、进程隐藏等方法及分析方法；

了解了恶意代码的行为，从静态、动态等多方面了解恶意代码，资源文件、注册表、进程监控，对于各种工具的使用更娴熟，也复习了 yara 规则和 ida python 脚本编写，收获颇丰。