南開大學

恶意代码分析与防治技术课程实验报告

实验六

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Lab6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

当我们分析恶意代码时，识别汇编代码中的高级语言结构是一项非常重要的技能。这样可以帮助我们更好地理解恶意代码的工作原理和它试图执行的操作。以下是一些基本的原理和提示，帮助我们在汇编代码中识别C语言的常见结构：

1. **全局和局部变量**：

- **全局变量**：全局变量通常存储在数据段或bss段中。在汇编代码中，我们可以通过查找对这些段的引用来识别它们。
- **局部变量**：局部变量通常存储在栈上。函数调用的开始通常会有指令来调整栈指针，为局部变量预留空间。

2. **循环结构**：

- **for循环**：通常可以通过固定的初始化、条件检查和迭代步骤来识别。在汇编中，这可能表现为初始化指令，接着是一个跳转指令进入条件检查，然后是循环体和迭代步骤。
- **while循环**：这种循环的特点是在循环开始之前进行条件检查。我们可以查找一个条件跳转指令，它决定是否进入循环体。

3. **条件结构**：

- **if-else语句**：在汇编代码中，这通常表示为一个条件跳转。如果条件为真，程序会跳转到一个代码块，否则它会跳转到另一个代码块或继续执行后面的指令。
- **switch语句**：这在汇编中可能更难识别，但通常会有一个跳转表，程序根据某个值来决定跳转到哪个代码块。

4. **数据结构**：

- **数组**：在汇编中，数组通常表示为基址加上一个索引乘以元素大小的形式。例如，对于一个整数数组，我们可能会看到类似`mov eax, [ebx+ecx*4]`这样的指令。
- **结构体**：结构体通常表示为基址加上固定偏移的形式。每个偏移对应结构中的一个字段。

5. **函数调用**：

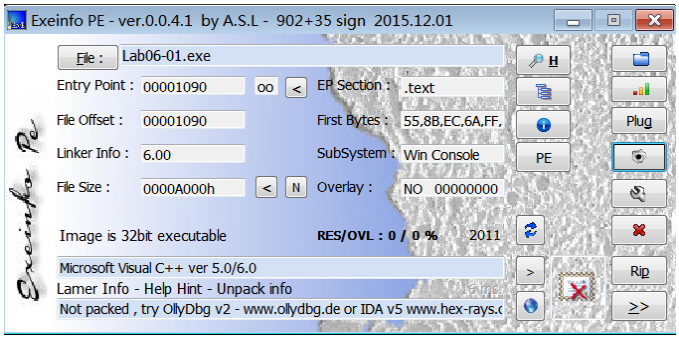
- 查找`call`指令。这通常表示一个函数被调用。返回地址被推送到栈上，然后程序跳转到函数的地址。

3 实验过程

3.1 Lab06-01.exe

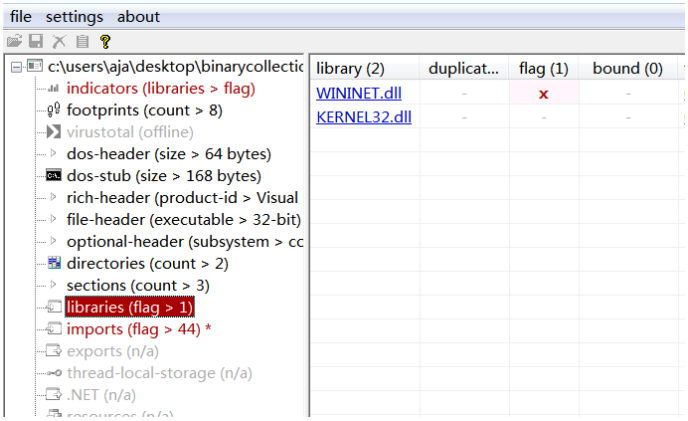
- **基本静态分析**

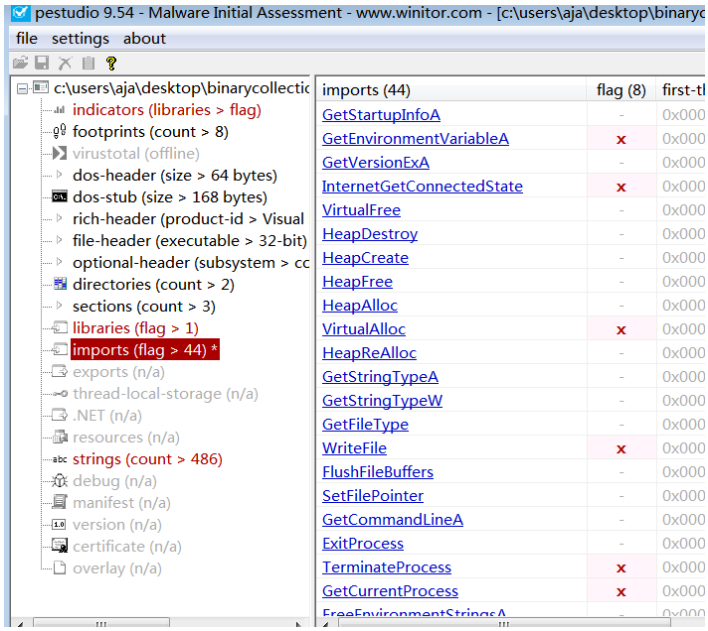
使用exeinfoPE查看加壳：

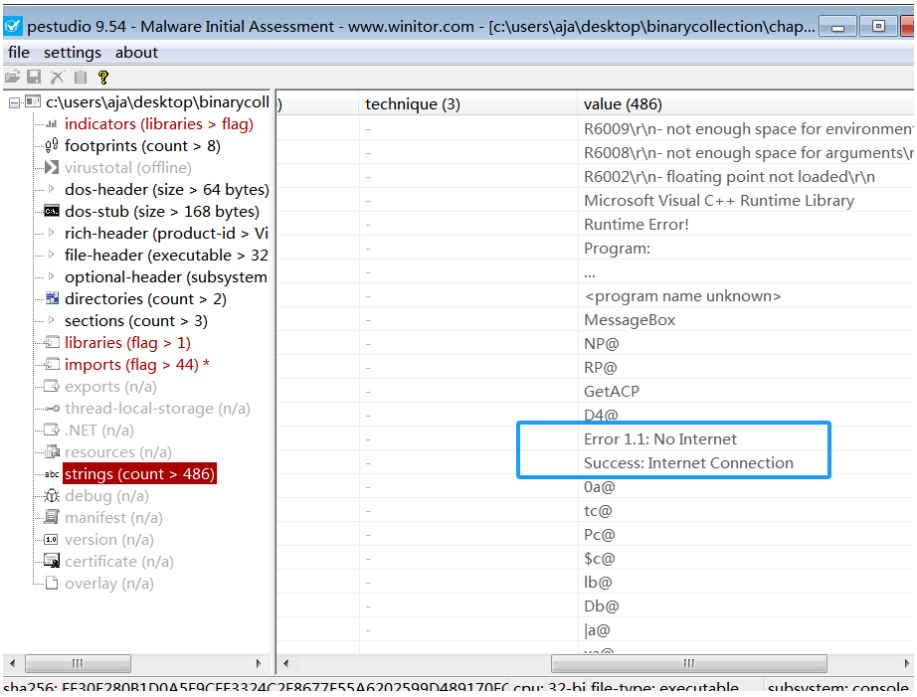


该代码未加壳，为VC++ 6.0编译。

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





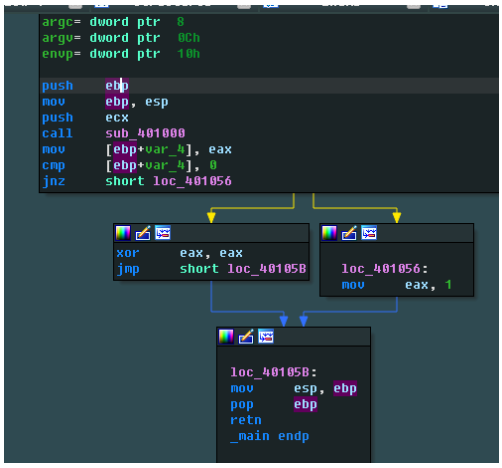


发现其导入了`WININET.dll`，且使用了网络函数`InternetGetConnectedState`。

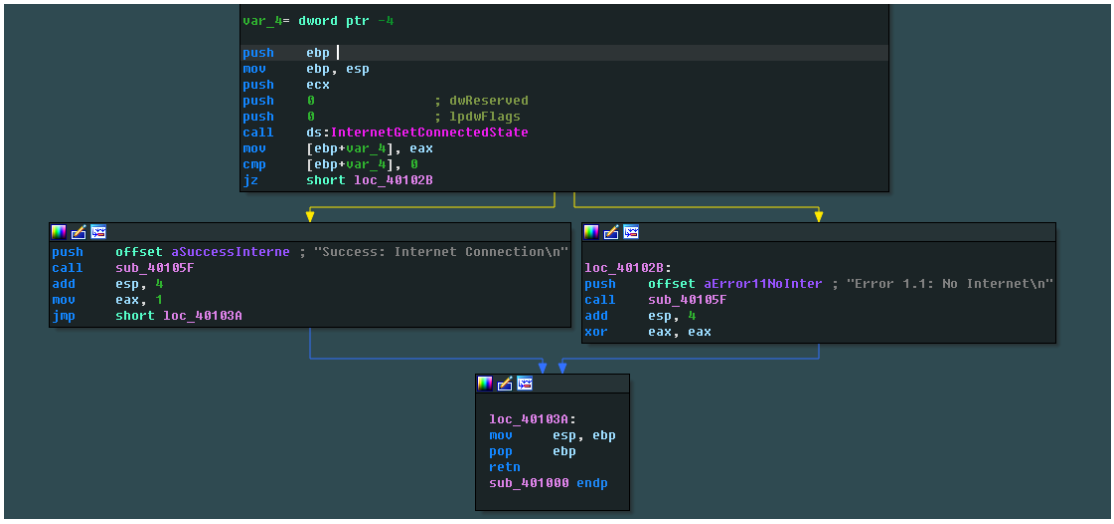
在字符串中，也发现了使用网络的迹象。

- **Q1：由main函数调用的唯一子过程中发现的主要代码结构是什么？**

打开IDA进入main函数如下：



发现main函数仅调用了一个子过程`sub_401000`，双击进入此函数查看其结构：



发现其中调用了函数`InternetGetConnectedState`，函数返回值位于eax，将其存入一个变量，然后把这个变量和0作对比，使用jz结合判断跳转。

因为其使用了**jz指令**控制程序流，使其分为**两个分支**，而且这两个分支最后又**汇聚为一个分支**，显然这是**C语言中的if代码结构**。

这段代码显然是在判断网络连接是否成功。

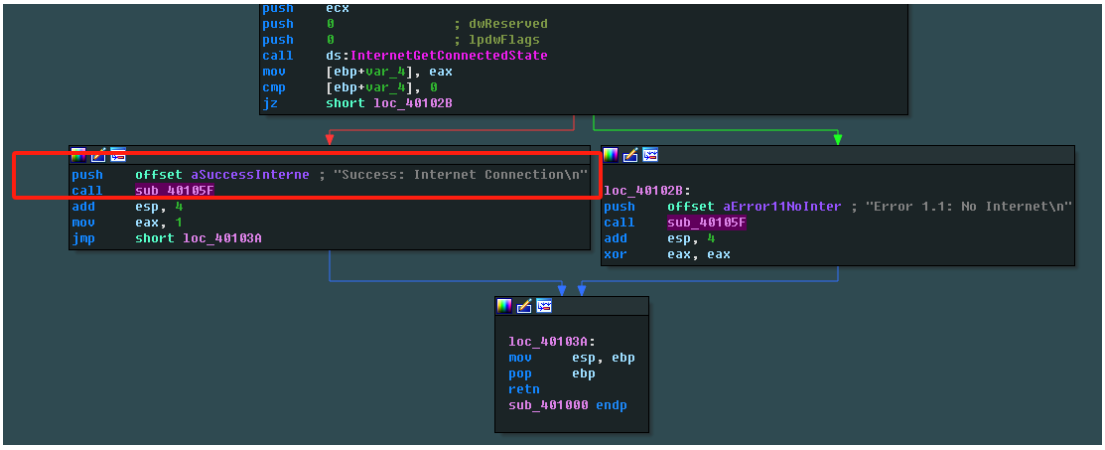
- **Q2：位于0x40105F的子过程是什么？**

在IDA中跳转到0x40105F处，反编译代码如下所示：

```
int __cdecl sub_40105F(int a1, int a2)
{
  int v2; // edi
  int v3; // ebx
  v2 = _stbuf(&File);
  v3 = sub_401282(&File, a1, (int)&a2);
  _ftbuf(v2, &File);
  return v3;
}
```

代码很短，用到了`stbuf`和`ftbuf`，但未清楚其功能。

查看sub_40105F的**交叉引用**，发现在sub_401000函数被调用：



push了一个字符串，为提示信息，然后调用了这个函数。显然这个函数**用来打印信息**，为`printf`。

- **Q3：这个程序的目的是什么？**

main函数仅调用了sub_401000函数，直接查看其反编译代码：

```
int sub_401000()
{
  BOOL ConnectedState; // [esp+0h] [ebp-4h]

  ConnectedState = InternetGetConnectedState(0, 0);
  if ( ConnectedState )
  {
    sub_40105F(aSuccessInterne, ConnectedState);
    return 1;
  }
  else
  {
    sub_40105F(aError11NoInter, 0);
    return 0;
  }
}
```

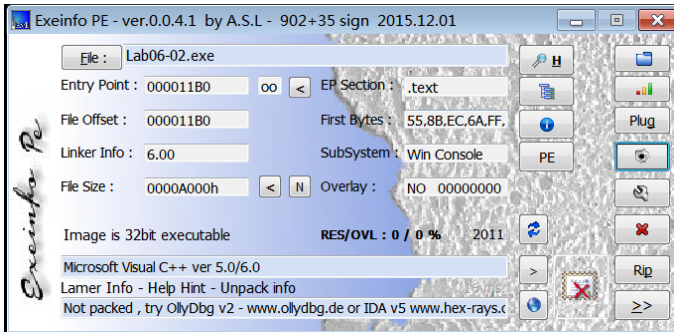
代码不长，**首先检查网络连接状态，然后判断这个状态是否为正常，分别调用sub_40105F函数来打印相应的提示信息，并且如果成功则返回1，失败返回0。**

总结来说，这是一个判断网络连接是否正常的代码。

3.2 Lab06-02.exe

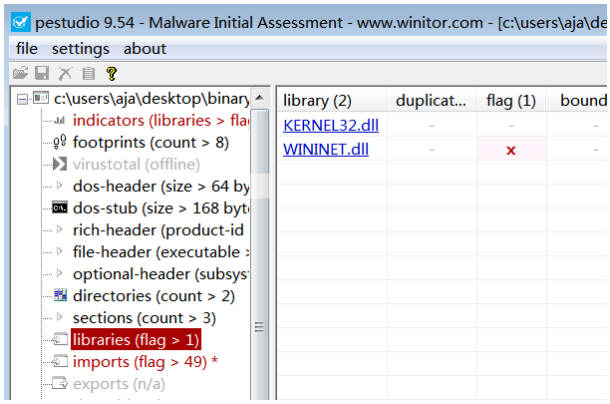
- **基本静态分析**

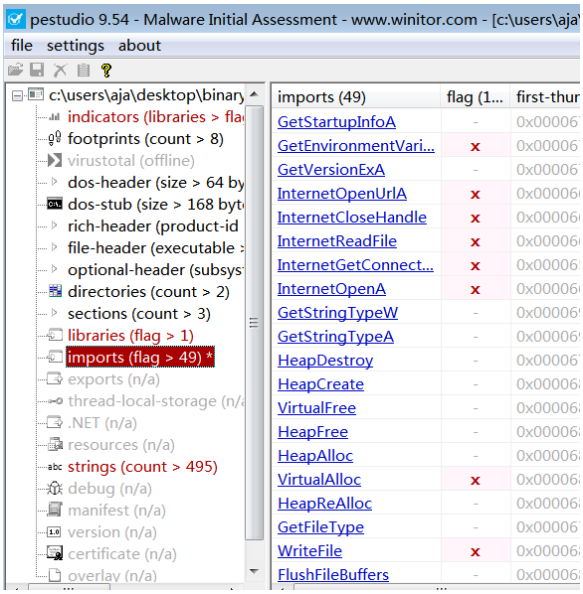
使用exeinfoPE查看加壳：

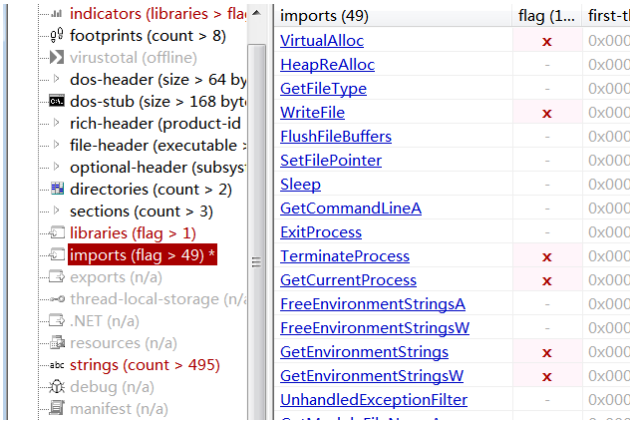


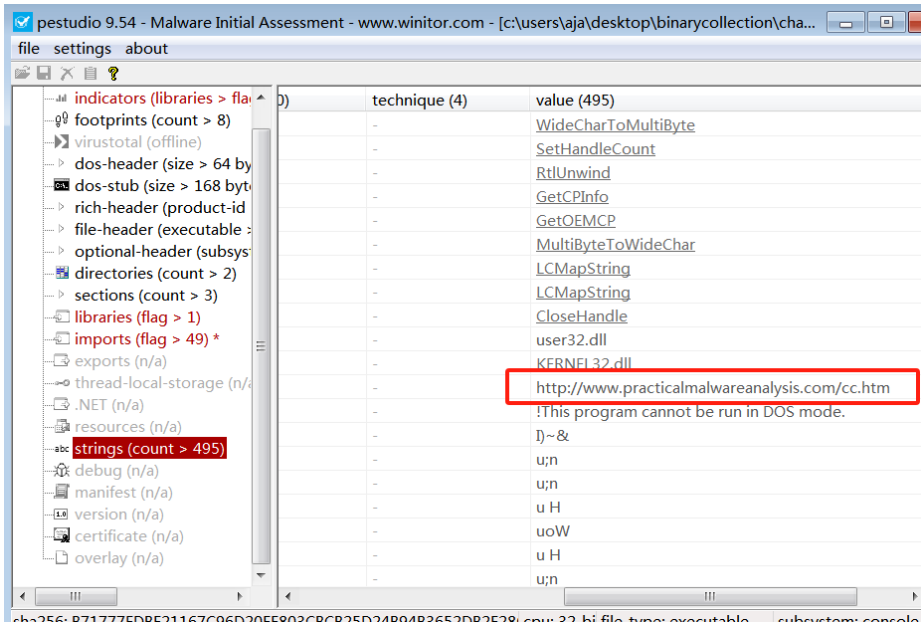
该代码未加壳，为VC++ 6.0编译。

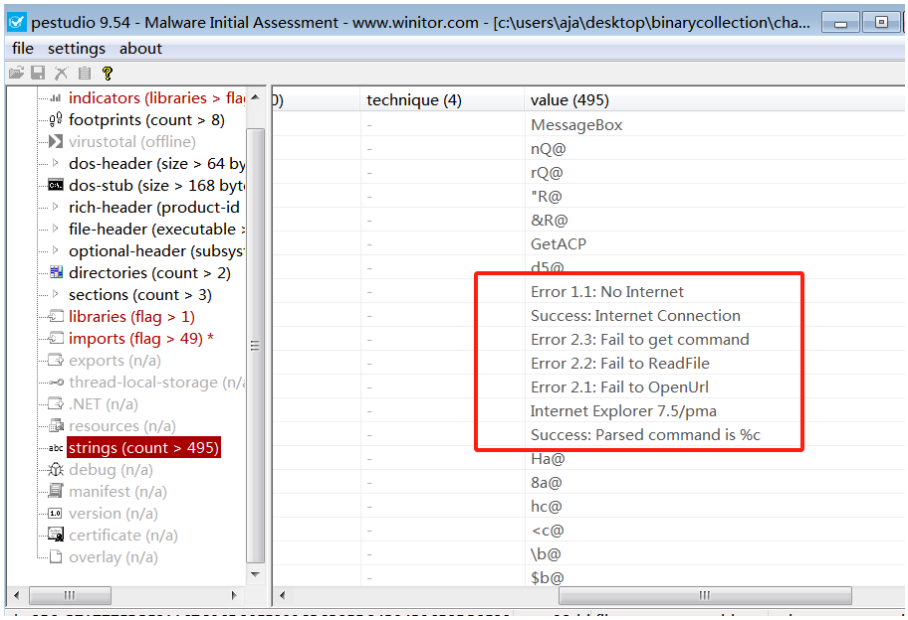
我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：











发现其也导入了`WININET.dll`，且也使用了网络函数`InternetGetConnectedState`，`InternetOpenUrlA`，`InternetReadFile`等，似乎存在从网络上下载文件的操作。

在字符串中，也发现了网络通信的迹象，并且发现了网址`http://www.practicalmalwareanalysis.com/cc.htm。

- **Q1：main函数调用的第一个子过程执行了什么操作?**

打开IDA进入main函数，进入**反编译**代码，如下：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [esp+0h] [ebp-8h]

  if ( !sub_401000() )
    return 0;
  v4 = sub_401040();
```

```
  if ( v4 )
  {
    sub_40117F("Success: Parsed command is %c\n", v4);
    Sleep(0xEA60u);
  }
  return 0;
}
```

显然，调用的第一个子过程是`sub_401000`，进入该函数的反汇编：

```
int sub_401000()
{
  if ( InternetGetConnectedState(0, 0) )
  {
    sub_40117F(aSuccessInterne);
    return 1;
  }
  else
  {
    sub_40117F(aError11NoInter);
    return 0;
  }
}
```

其中的`aSuccessInterne`和`aError11NoInter`分别是字符串提示信息：

```
.data:00407030 aError11NoInter db 'Error 1.1: No Internet',0Ah,0
.data:00407030                                         ; DATA XREF: sub_401000:loc_40102B↑o
.data:00407048 aSuccessInterne db 'Success: Internet Connection',0Ah,0
.data:00407048                                         ; DATA XREF: sub_401000+17↑o
.data:00407066                 align 4
```

因此和`Lab06-01.exe`的分析同理，**可知sub_40117F为打印信息的函数**。

同时`sub_401000`函数也使用了`InternetGetConnectedState`函数来判断网络状态，和Lab06-01.exe相似，因此这个**子过程的作用是检查是否存在可用的Internet连接**。

- **Q2：位于0x40117F的子过程是什么？**

由上述Q1分析知`sub_40117F`的**参数是一个字符串，功能是打印这个字符串**。

- **Q3：被main函数调用的第二个子过程做了什么？**

第二个子过程为`sub_401040`，进入该子过程的**反编译**代码：

```
char sub_401040()
{
  char Buffer[512]; // [esp+0h] [ebp-210h] BYREF
  HINTERNET hFile; // [esp+200h] [ebp-10h]
  HINTERNET hInternet; // [esp+204h] [ebp-Ch]
```

```
  DWORD dwNumberOfBytesRead; // [esp+208h] [ebp-8h] BYREF

  hInternet = InternetOpenA(szAgent, 0, 0, 0, 0);
  hFile = InternetOpenUrlA(hInternet, szUrl, 0, 0, 0, 0);
  if ( hFile )
  {
    if ( InternetReadFile(hFile, Buffer, 0x200u, &dwNumberOfBytesRead) )
    {
      if ( Buffer[0] == 60 && Buffer[1] == 33 && Buffer[2] == 45 &&
Buffer[3] == 45 )
      {
        return Buffer[4];
      }
      else
      {
        sub_40117F(aError23FailToG);
        return 0;
      }
    }
    else
    {
      sub_40117F(aError22FailToR);
      InternetCloseHandle(hInternet);
      InternetCloseHandle(hFile);
      return 0;
    }
  }
  else
  {
    sub_40117F(aError21FailToO);
    InternetCloseHandle(hInternet);
    return 0;
  }
}
```

其中几个重要的**常量**:

```
szAgent             db 'Internet Explorer 7.5/pma',0
szUrl               db 'http://www.practicalmalwareanalysis.com/cc.htm',0
aError23FailToG     db 'Error 2.3: Fail to get command',0Ah,0
aError22FailToR     db 'Error 2.2: Fail to ReadFile',0Ah,0
aError21FailToO     db 'Error 2.1: Fail to OpenUrl',0Ah,0
```

不难看出，代码将`User-Agent`字段设置为`Internet Explorer 7.5/pma`，并尝试**访问网址**`http://www.practicalmalwareanalysis.com/cc.htm`，

然后使用`InternetReadFile`函数来尝试**读取网页内容**，读取到Buffer中，**并将其前四个字节一一进行比较**，如果相等则返回第五个字节。

通过查阅ASCII码对照表可知，其希望前四个字符为`<!--`，**这显然是HTML注释的开始部分**。

那么可以总结这个**子过程的功能是访问指定网页，并检查网页内容是否从HTML注释开始**。

- **Q4：在这个子过程中使用了什么类型的代码结构?**

通过上述反编译代码可知，**主要的代码结构为if-else语句**。

- **Q5：在这个程序中有任何基于网络的指示吗?**

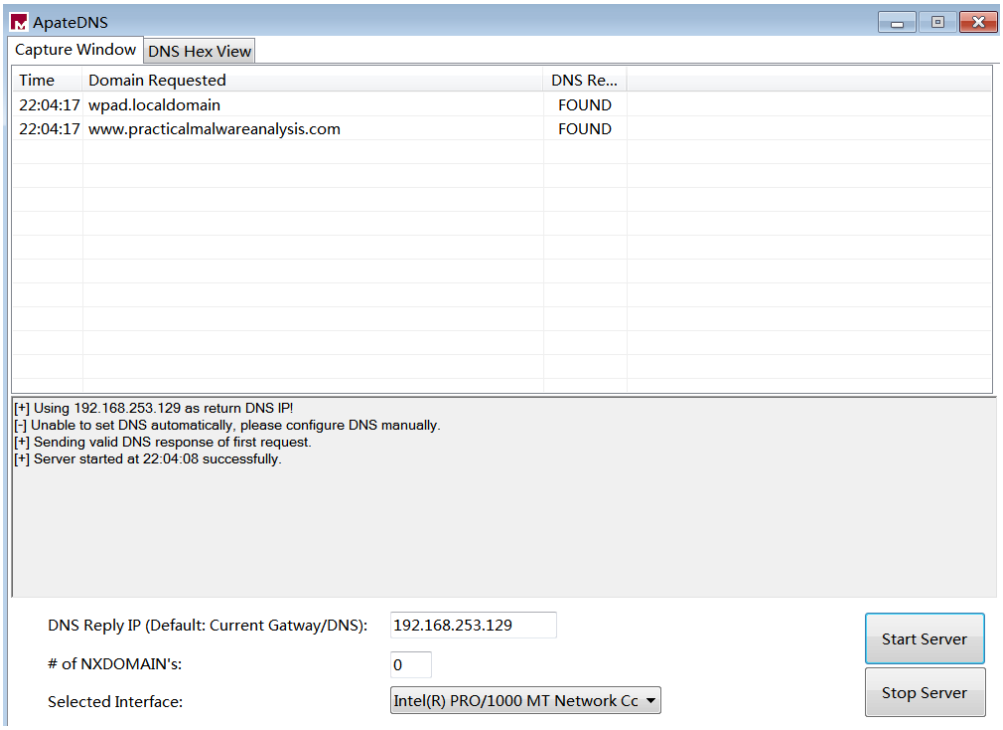
根据前面的分析，我们可以知道：

代码使用`Internet Explorer 7.5/pma`作为`User-Agent`。

代码下载了`http://www.practicalmalwareanalysis.com/cc.htm`这个网页内容。

那么我们可以通过**动态分析**加以确认。

准备好虚拟网络环境，伪DNS环境(使用kali虚拟机作为服务器)，然后打开`apateDNS`，运行Lab06-02.exe：



证实了代码确实访问了上述网页。

同时使用netcat来捕捉http请求，得到如下结果：

GET /cc.htm HTTP/1.1
User-Agent: Internet Explorer 7.5/pma
Host: www.practicalmalwareanalysis.com

- **Q6：这个恶意代码的目的是什么？**

在下载网页并判断后，sub_401040过程将返回HTML注释后的第一个字符。

返回到主函数查看接下来的代码：

```
if ( v4 ){
    sub_40117F("Success: Parsed command is %c\n", v4);
    Sleep(0xEA60u);
}
return 0;
```

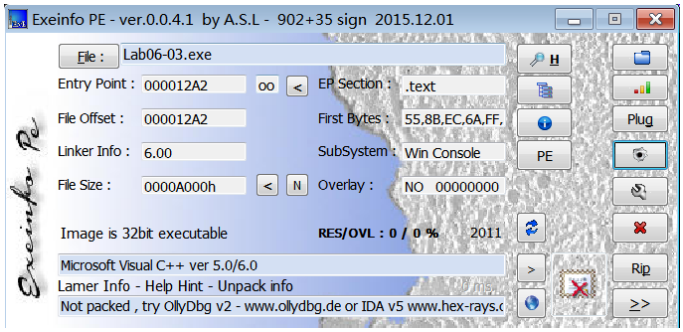
可知程序将**打印出这个字符**，并且**休眠一分钟**。

综合以上分析，可以得出程序的行为：首先判断是否存在可以使用的网络连接，如果不存在则报错停止运行。若存在网络，则更改User-Agent头然后尝试下载指定网页的内容，这个网页应该包括一段HTML注释，代码将提取注释后的第一个字符，然后输出到屏幕上，然后休眠1分钟，停止运行。

3.3 Lab06-03.exe

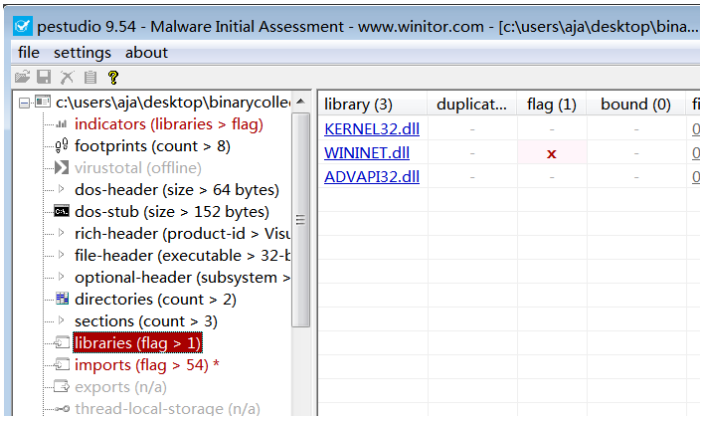
- **基本静态分析**

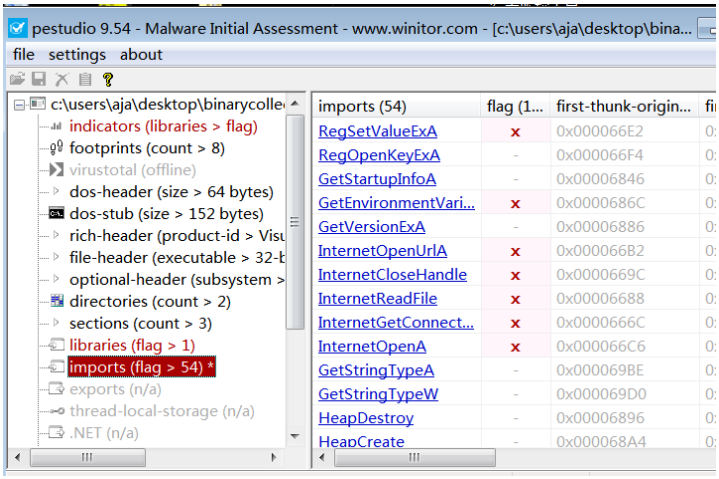
使用exeinfoPE查看加壳：

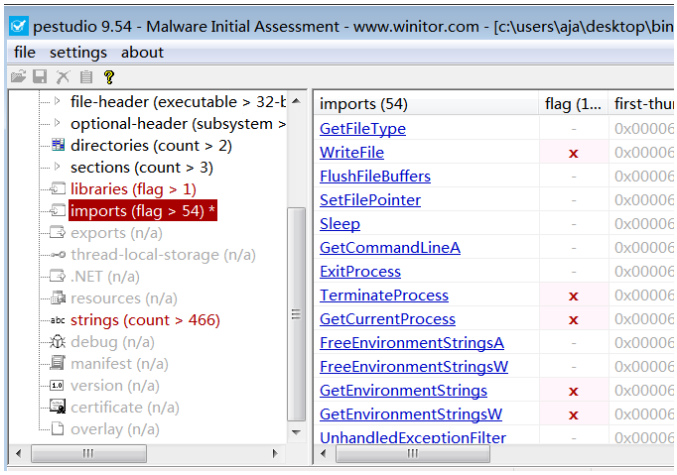


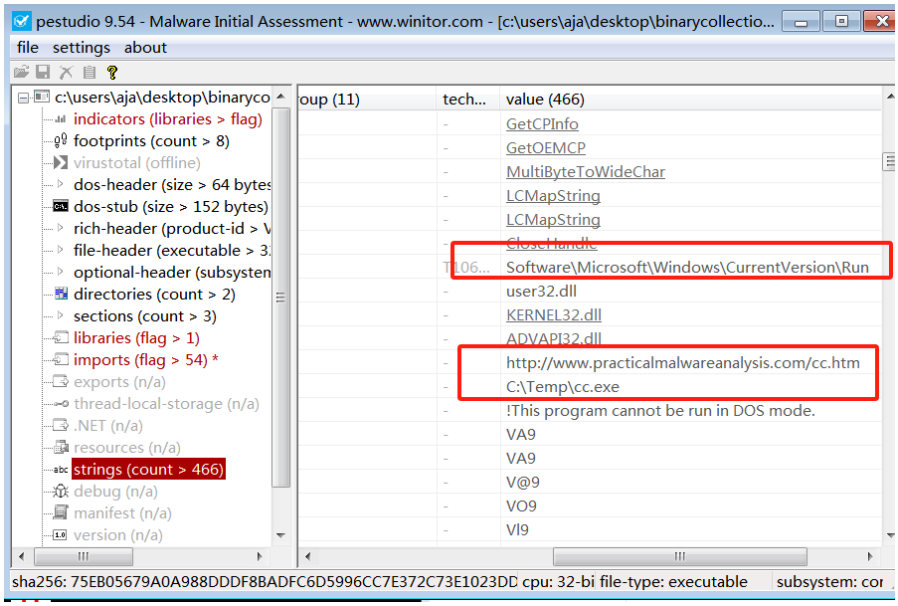
该代码未加壳，为VC++ 6.0编译。

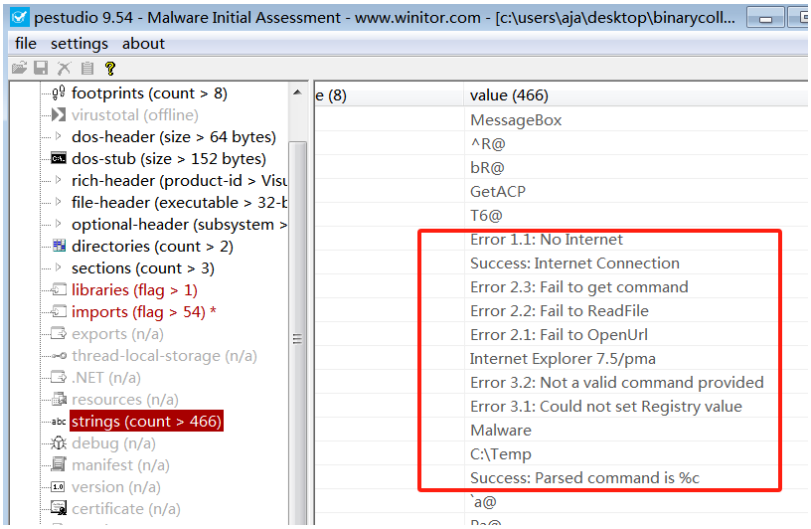
我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：


pestudio 9.54 - Malware Initial Assessment - www.winitor.com - [c:\users\aja\desktop\bina...
file settings about
c:\users\aja\desktop\binarycolle
indicators (libraries > flag)
footprints (count > 8)
virustotal (offline)
dos-header (size > 64 bytes)
dos-stub (size > 152 bytes)
rich-header (product-id > Visu
file-header (executable > 32-b
optional-header (subsystem >
directories (count > 2)
sections (count > 3)
libraries (flag > 1)
imports (flag > 54) *
exports (n/a)
thread-local-storage (n/a)
library (3) duplicat... flag (1) bound (0)
KERNEL32.dll
WININET.dll x
ADVAPI32.dll


pestudio 9.54 - Malware Initial Assessment - www.winitor.com - [c:\users\aja\desktop\bina...
file settings about
c:\users\aja\desktop\binarycolle
indicators (libraries > flag)
footprints (count > 8)
virustotal (offline)
dos-header (size > 64 bytes)
dos-stub (size > 152 bytes)
rich-header (product-id > Visu
file-header (executable > 32-b
optional-header (subsystem >
directories (count > 2)
sections (count > 3)
libraries (flag > 1)
imports (flag > 54) *
exports (n/a)
thread-local-storage (n/a)
.NET (n/a)
imports (54) flag (1... first-thunk-origin...
RegSetValueExA x 0x000066E2
RegOpenKeyExA 0x000066F4
GetStartupInfoA 0x00006846
GetEnvironmentVari... x 0x0000686C
GetVersionExA 0x00006886
InternetOpenUrlA x 0x000066B2
InternetCloseHandle x 0x0000669C
InternetReadFile x 0x00006688
InternetGetConnect... x 0x0000666C
InternetOpenA x 0x000066C6
GetStringTypeA 0x000069BE
GetStringTypeW 0x000069D0
HeapDestroy 0x00006896
HeapCreate 0x000068A4


pestudio 9.54 - Malware Initial Assessment - www.winitor.com - [c:\users\aja\desktop\bin
file settings about
file-header (executable > 32-b
optional-header (subsystem >
directories (count > 2)
sections (count > 3)
libraries (flag > 1)
imports (flag > 54) *
exports (n/a)
thread-local-storage (n/a)
.NET (n/a)
resources (n/a)
strings (count > 466)
debug (n/a)
manifest (n/a)
version (n/a)
certificate (n/a)
overlay (n/a)
imports (54) flag (1...
GetFileType
WriteFile x
FlushFileBuffers
SetFilePointer
Sleep
GetCommandLineA
ExitProcess
TerminateProcess x
GetCurrentProcess x
FreeEnvironmentStringsA
FreeEnvironmentStringsW
GetEnvironmentStrings x
GetEnvironmentStringsW x
UnhandledExceptionFilter





和Lab06-02.exe静态分析结果相似，均存在网络特征，以及文件操作.

另外发现路径`C:\Temp\cc.exe`，推测代码可能下载了文件然后存到这个路径。

也发现了注册表的路径，同时导入表有`RegSetValueExA`，推测代码存在**修改注册表**的行为。

- **Q1：比较在 main函数与实验6-2的main 函数的调用。从main 中调用的新的函数是什么?**

使用IDA打开Lab06-03.exe，得到主函数的**反编译**代码：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [esp+0h] [ebp-8h]

  if ( !sub_401000() )
    return 0;
  v4 = sub_401040();
  if ( v4 )
  {
```

```
    sub_401271("Success: Parsed command is %c\n", v4);
    sub_401130(v4, *argv);
    Sleep(0xEA60u);
  }
  return 0;
}
```

与6-2的exe反编译代码相对比，其多调用了一句：

```
sub_401130(v4, *argv);
```

其余函数，如sub_401000和sub_401040均和Lab06-02的相同，sub_401271是printf函数。

- **Q2：这个新的函数使用的参数是什么？**

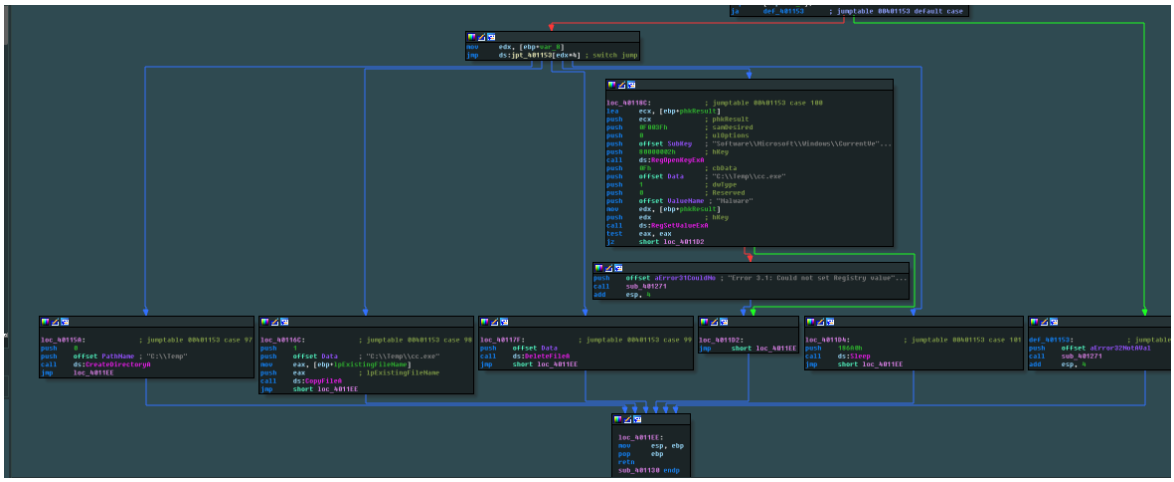
main调用该函数的时候传入了**两个参数**;

```
v4 ： 同理lab06-02的分析，这是解析出的HTML注释的第一个字符
*argv：即argv[0]，为该程序的名字
```

参数类型为：(char, char*)

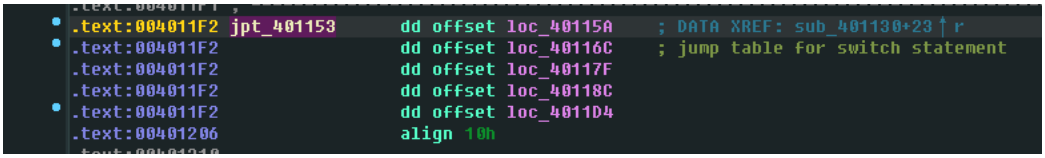
- **Q3：这个函数包含的主要代码结构是什么？**

查看新函数的反汇编代码图：



发现了多分支，最后汇聚到了一处。

同时找到了**跳转表**：



可以猜测为**Switch语句**。

查看新函数的**反编译**代码：

```
void __cdecl sub_401130(char a1, LPCSTR lpExistingFileName)
{
  HKEY phkResult; // [esp+4h] [ebp-4h] BYREF

  switch ( a1 )
  {
    case 'a':
      CreateDirectoryA(PathName, 0);
      break;
    case 'b':
      CopyFileA(lpExistingFileName, (LPCSTR)Data, 1);
      break;
    case 'c':
      DeleteFileA((LPCSTR)Data);
      break;
    case 'd':
      RegOpenKeyExA(HKEY_LOCAL_MACHINE, SubKey, 0, 0xF003Fu, &phkResult);
      if ( RegSetValueExA(phkResult, ValueName, 0, 1u, Data, 0xFu) )
        sub_401271(aError31CouldNo);
      break;
    case 'e':
      Sleep(0x186A0u);
      break;
    default:
      sub_401271(aError32NotAVal);
      break;
  }
```

印证了switch语句的猜想。

- **Q4：这个函数能够做什么？**

观察函数的反编译代码，不同分支有不同的操作，分述如下：

1. case ‘a’

   调用创建文件夹函数CreateDirectoryA来创建PathName目录：

   ```
   PathName        db 'C:\Temp',0
   ```

2. case ‘b’

   调用拷贝文件函数CopyFileA将恶意代码自身拷贝到下面的路径：

   ```
   Data            db 'C:\Temp\cc.exe',0
   ```

3. case ‘c’

调用删除文件函数DeleteFileA将下面路径的文件删除：

```
Data            db 'C:\Temp\cc.exe',0
```

4. case ‘d’

   调用注册表函数将注册表键`Software\Microsoft\Windows\CurrentVersion\Run`的值设置为`C:\Temp\cc.exe`，这样，恶意代码在每次开机时都会自动启动。

5. case ‘e’

   休眠100秒。

6. default

   打印字符串`Error 3.2: Not a valid command provided`。

总结如下：

case分支	操作
a	创建目录
b	复制自身
c	删除文件
d	篡改注册表
e	休眠
default	打印错误信息

- **Q5：在这个恶意代码中有什么本地特征吗？**

显然，上述的路径是定死的，可以根据**路径特征**，**注册表特征**来发现该恶意代码：

C:\Temp\cc.exe
Software\Microsoft\Windows\CurrentVersion\Run

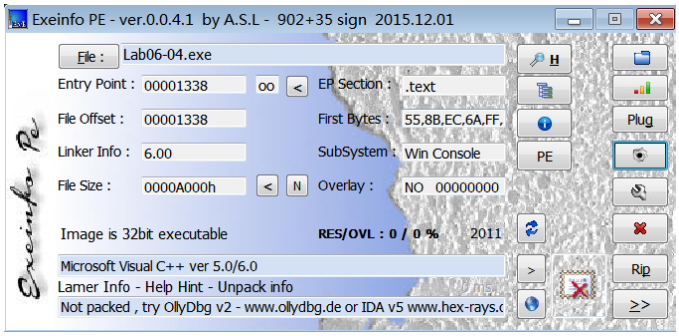
- **Q6：这个恶意代码的目的是什么？**

首先检查是否存在网络连接，如果存在，尝试获取指定网页内的HTML注释后的字符，作为“命令”，来决定在本地执行什么样的行为，如创建目录，复制自身，修改注册表等，可以说实现了**远程命令控制**。

3.4 Lab06-04.exe

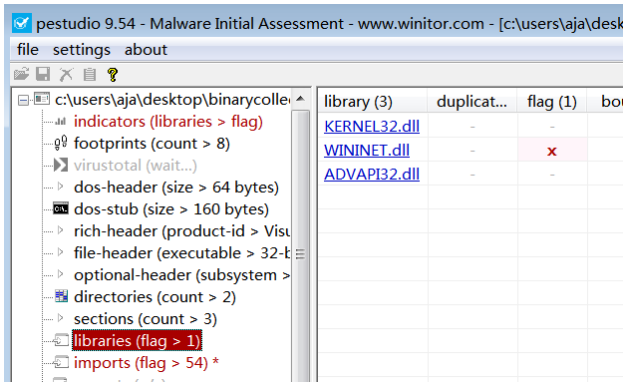
- **基本静态分析**

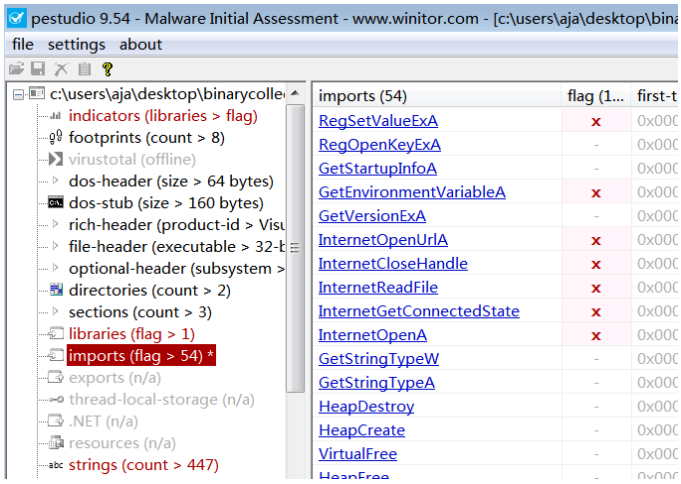
使用exeinfoPE查看加壳：

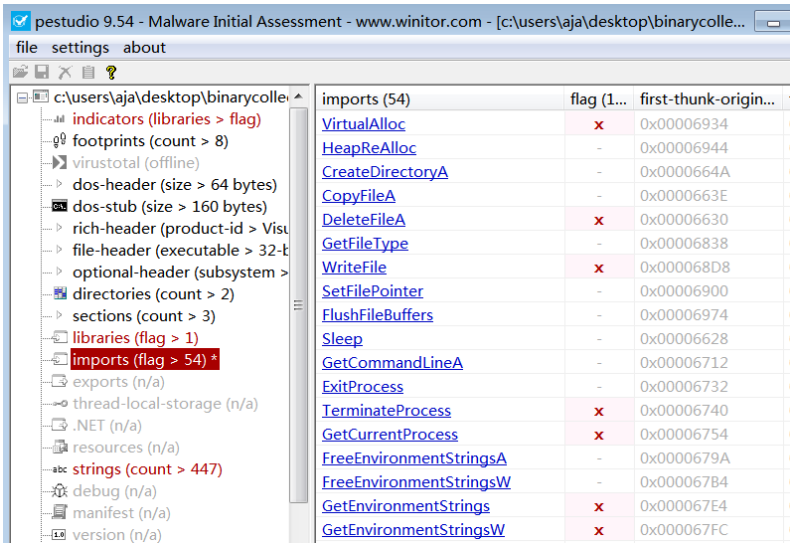


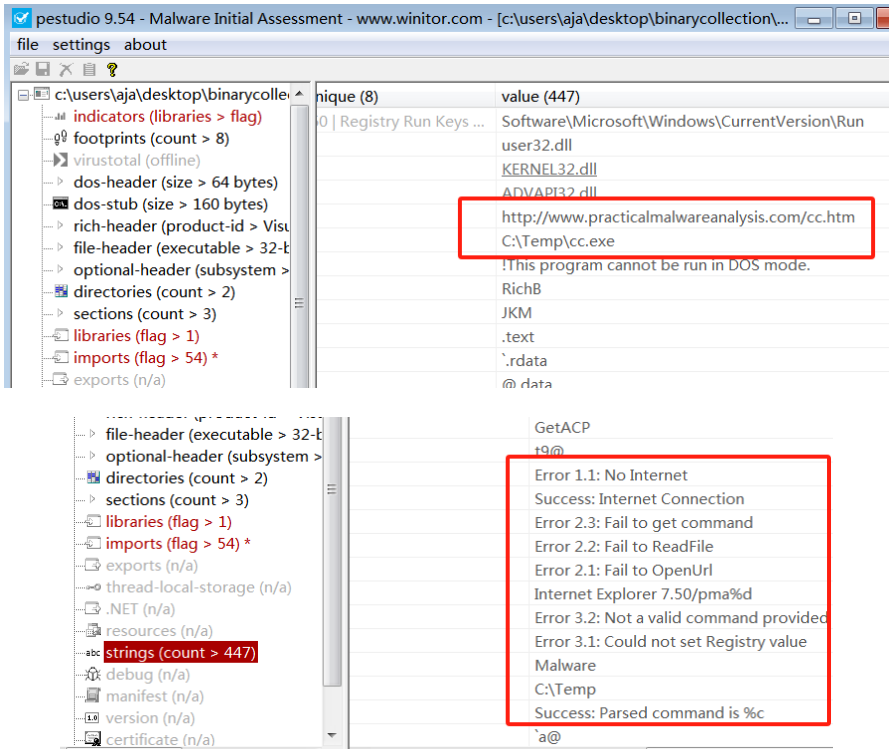
该代码未加壳，为VC++ 6.0编译。

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









和Lab06-03.exe静态分析结果相似，均存在网络特征，以及文件操作，以及对注册表的修改行为。

另外，`Internet Explorer 7.50/pma%d`字符串是Lab06-03没有的，它似乎可以**动态生成User-Agent**。

- **Q1：在实验6-3和6-4的main函数中的调用之间的区别是什么？**

打开IDA查看Lab06-04.exe的main函数**反编译**代码：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int i; // [esp+0h] [ebp-Ch]
  char v5; // [esp+4h] [ebp-8h]

  if ( !sub_401000() )
    return 0;
  for ( i = 0; i < 1440; ++i )
  {
    v5 = sub_401040(i);
    if ( !v5 )
      return 0;
    sub_4012B5("Success: Parsed command is %c\n", v5);
    sub_401150(v5, *argv);
    Sleep(0xEA60u);
  }
  return 0;
}
```

对比可知，`sub_401000`同样为**检查网络连接**，`sub_401040`用于**解析HTML内容**，`sub_4012B5`用于**格式化打印字符串**，`sub_401150`用于**switch分支决定命令**。

但`sub_401040`函数多了一个参数。

- **Q2：什么新的代码结构已经被添加到main中？**

由Q1可知新增了一个**for循环**结构。

- **Q3：这个实验的解析HTML的函数和前面实验中的那些有什么区别？**

该函数多了一个参数，进入函数反编译代码查看，发现多了一句代码；

```
sprintf(szAgent, "Internet Explorer 7.50/pma%d", a1);
```

显然，参数用于**动态修改User-Agent**字符串。结合main函数的for循环可知，

代码循环生成不同的User-Agent来访问指定网页。

- **Q4：这个程序会运行多久？（假设它已经连接到互联网。）**

程序主要时间花在Sleep休眠函数上，每次循环休眠1分钟，共循环1440次，因此共花费

$$1 * 1440 = 1440min = 24h$$

- **Q5：在这个恶意代码中有什么新的基于网络的迹象吗？**

由上述分析，恶意代码循环生成不同的User-Agent，特征是**后面跟的数字相当于运行的分钟数**，可以据此来进行网络迹象追踪。

- **Q6：这个恶意代码的目的是什么？**

综合以上，该程序首先判断是否有网络连接，若有，则访问指定网页，并解析网页内容，提取网页HTML注释后的字符。将这个字符当成“指令”，通过这个指令来决定本地执行的操作，如修改注册表等。程序将每分钟执行一次指令，运行24h后停止。**这意味着，运行该恶意代码的电脑将被非法利用24h，控制者可以利用网页的内容，来控制本机的行为。**

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
}
```

```
9
10 //Lab06-01
11 rule lab6_1
12 {
13 strings:
14     $s1 = "WININET.dll"
15     $s2 = "Success: Internet Connection"
16     $s3 = "InternetGetConnectedState"
17 condition:
18     IsPE and $s1 and $s2 and $s3
19 }
20
21 //Lab06-02
22 rule lab6_2
23 {
24 strings:
25     $s1 = "Internet Explorer"
26     $s2 = "Success: Parsed command is %c"
27     $s3 = "http://www.practicalmalwareanalysis.com/cc.htm"
28 condition:
29     IsPE and lab6_1 and $s1 and $s2 and $s3
30 }
31
32 //Lab06-03
33 rule lab6_3
34 {
35 strings:
36     $s1 = "C:\\Temp\\cc.exe"
37     $s2 = "RegSetValueExA"
38     $s3 = "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
39 condition:
40     IsPE and lab6_2 and $s1 and $s2 and $s3
41 }
42
43 //Lab06-04
44 rule lab6_4
45 {
46 strings:
47     $s1 = "Internet Explorer 7.50/pma%d"
48 condition:
49     IsPE and lab6_3 and $s1
50 }
```

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

def get_called_functions(start_ea, end_ea):
    """
    给定起始和结束地址，返回该范围内调用的所有函数。
    """
    called_functions = []
    for head in idautils.Heads(start_ea, end_ea):
        if idc.is_code(idc.get_full_flags(head)):
            mnemonic = idc.print_insn_mnem(head)
            if mnemonic in ['call']:
                operand_value = idc.get_operand_value(head, 0)
                called_functions.append(operand_value)
    return called_functions

def main():
    # 获取main函数的地址
    main_addr = idc.get_name_ea_simple('_main')
    if main_addr == idaapi.BADADDR:
        print("找不到 '_main' 函数。")
        return

    main_end_addr = idc.find_func_end(main_addr)

    # 列出main函数调用的所有函数
    main_called_functions = get_called_functions(main_addr, main_end_addr)

    print("被 '_main' 调用的函数:")
    for func_ea in main_called_functions:
        print(idc.get_func_name(func_ea))

        # 列出被main调用的函数内部调用的函数或API
        func_end_addr = idc.find_func_end(func_ea)
        called_by_func = get_called_functions(func_ea, func_end_addr)
        print("\t被 {} 调用的函数/APIs:
".format(idc.get_func_name(func_ea)))
        for sub_func_ea in called_by_func:
            print("\t\t{}".format(idc.get_func_name(sub_func_ea)))

if __name__ == "__main__":
```

```
    main()

```

这段脚本的作用是：找出main函数调用的所有子过程以及子过程调用的函数或API。

对四个恶意代码分别运行上述`IDA Python`脚本，结果如下：

- **Lab06-01.exe**

```
被 '_main' 调用的函数:
sub_401000
    被 sub_401000 调用的函数/APIs:
        sub_40105F
        sub_40105F
```

- **Lab06-02.exe**

```
被 '_main' 调用的函数:
sub_401000
    被 sub_401000 调用的函数/APIs:
        sub_40117F
        sub_40117F
sub_401040
    被 sub_401040 调用的函数/APIs:
        sub_40117F
        sub_40117F
        sub_40117F
sub_40117F
    被 sub_40117F 调用的函数/APIs:
        __stbuf
        sub_4013A2
        __ftbuf
```

- **Lab06-03.exe**

```
被 '_main' 调用的函数:
sub_401000
    被 sub_401000 调用的函数/APIs:
        sub_401271
        sub_401271
sub_401040
    被 sub_401040 调用的函数/APIs:
        sub_401271
        sub_401271
        sub_401271
sub_401271
    被 sub_401271 调用的函数/APIs:
        __stbuf
```

```
        sub_401494
        __ftbuf
sub_401130
    被 sub_401130 调用的函数/APIs:
        sub_401271
        sub_401271
```

- **Lab06-04.exe**

```
被 '_main' 调用的函数:
sub_401000
    被 sub_401000 调用的函数/APIs:
        sub_4012B5
        sub_4012B5
sub_401040
    被 sub_401040 调用的函数/APIs:
        _sprintf
        sub_4012B5
        sub_4012B5
        sub_4012B5
sub_4012B5
    被 sub_4012B5 调用的函数/APIs:
        __stbuf
        sub_40152A
        __ftbuf
sub_401150
    被 sub_401150 调用的函数/APIs:
        sub_4012B5
        sub_4012B5
```

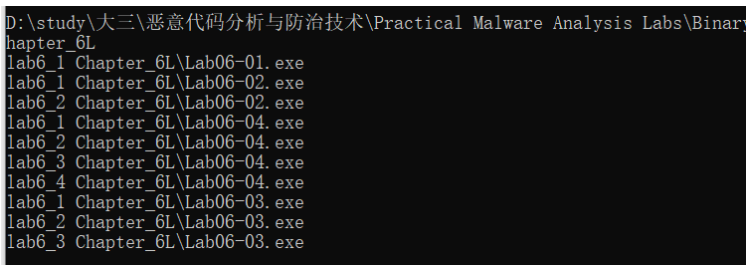
据此我们可以直观地看到main函数的**低层调用链**，这些函数将被我们主要进行分析。

4 实验结论及心得体会

把上述Yara规则保存为`rule_ex6.yar`，然后在Chapter_6L上一个目录输入以下命令:

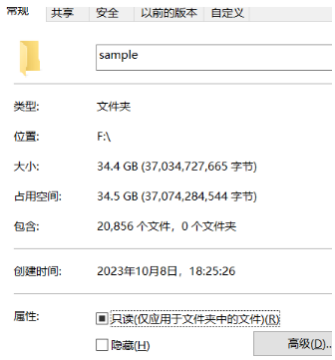
```
yara64 -r rule_ex6.yar Chapter_6L
```

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_ex6.yar'

# 定义要扫描的文件夹路径
folder_path = './sample/'

# 加载YARA规则
try:
    rules = yara.compile(rule_file)
except yara.SyntaxError as e:
    print(f"YARA规则语法错误: {e}")
    exit(1)

# 获取当前时间
start_time = datetime.datetime.now()

# 扫描文件夹内的所有文件
scan_results = []

for root, dirs, files in os.walk(folder_path):
    for file in files:
        file_path = os.path.join(root, file)
        try:
            matches = rules.match(file_path)
            if matches:
                scan_results.append({'file_path': file_path, 'matches':
[str(match) for match in matches]})
        except Exception as e:
            print(f"扫描文件时出现错误: {file_path} - {str(e)}")
```

```python

# 计算扫描时间
end_time = datetime.datetime.now()
scan_time = (end_time - start_time).seconds

# 将扫描结果写入文件
output_file = './scan_results.txt'

with open(output_file, 'w') as f:
    f.write(f"扫描开始时间: {start_time.strftime('%Y-%m-%d %H:%M:%S')}\n")
    f.write(f'扫描耗时: {scan_time}s\n')
    f.write("扫描结果:\n")
    for result in scan_results:
        f.write(f"文件路径: {result['file_path']}\n")
        f.write(f"匹配规则: {', '.join(result['matches'])}\n")
        f.write('\n')

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：

```
扫描开始时间: 2023-10-18 21:16:51
扫描耗时: 94s
扫描结果:
文件路径: ./sample/Lab06-01.exe
匹配规则: lab6_1

文件路径: ./sample/Lab06-02.exe
匹配规则: lab6_1, lab6_2

文件路径: ./sample/Lab06-03.exe
匹配规则: lab6_1, lab6_2, lab6_3

文件路径: ./sample/Lab06-04.exe
匹配规则: lab6_1, lab6_2, lab6_3, lab6_4
```

将几个实验样本扫描了出来，**共耗时94s**。

心得体会：通过此次实验，我学会了综合使用IDA来进行分析，对yara规则的编写更加熟练。同时也对IDA Python的使用有了更深的体会。另外，IDA的反编译功能十分强大，我们需要好好利用这个功能，它可以直接识别出恶意代码的C语言结构，帮助我们分析。