恶意代码分析与防治技术实验报告

Lab5

学号：　　姓名：　　专业：

一、 实验环境

1. 已关闭病毒防护的 Windows10

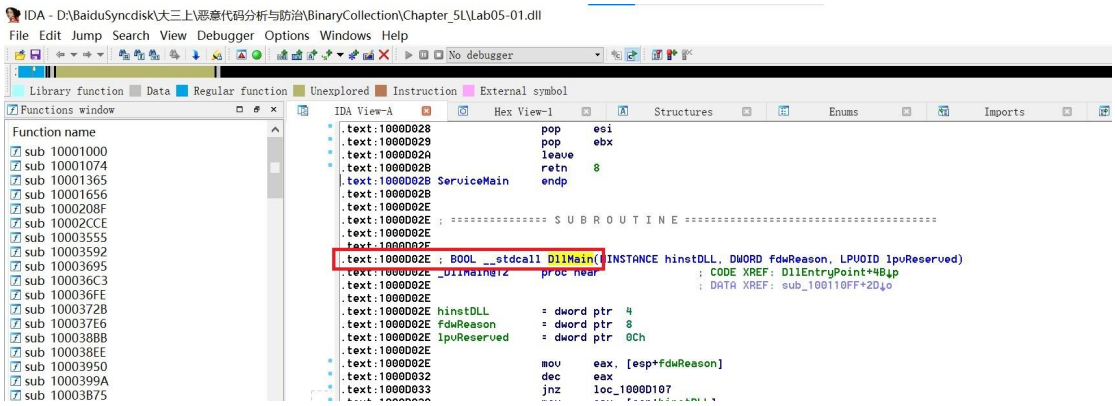
二、 实验工具

IDAPro, Pycharm, Strings

三、 实验内容

1. DllMain 的地址是什么?

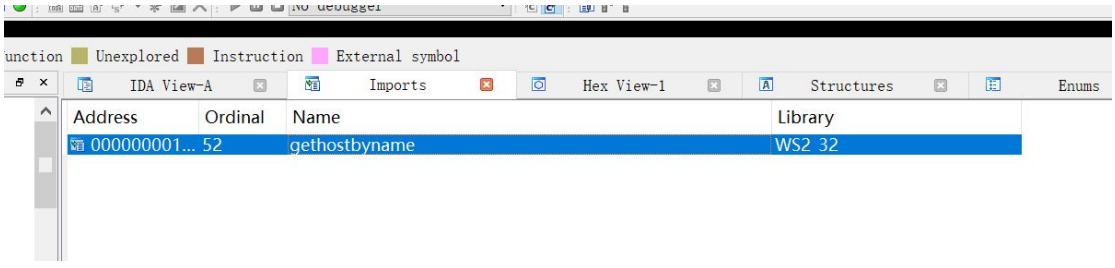
如图所示，地址是.text 节的 0x100163CC 处

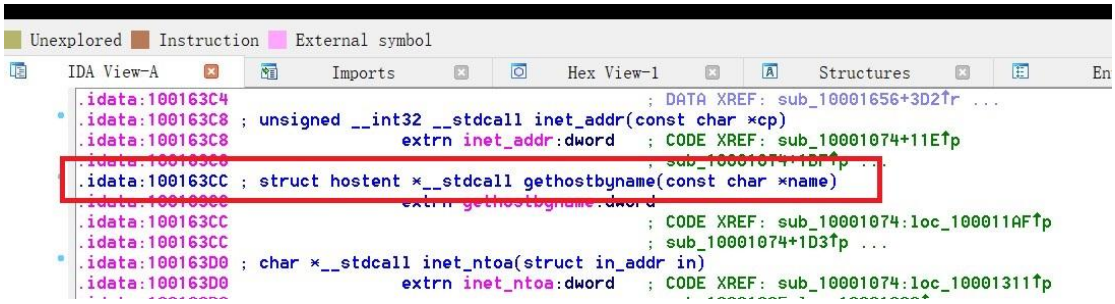


2. 使用 Imports 窗口并浏览到 gethostbyname，导入函数定位到什么地址?

.idata 节的 100163CC 处

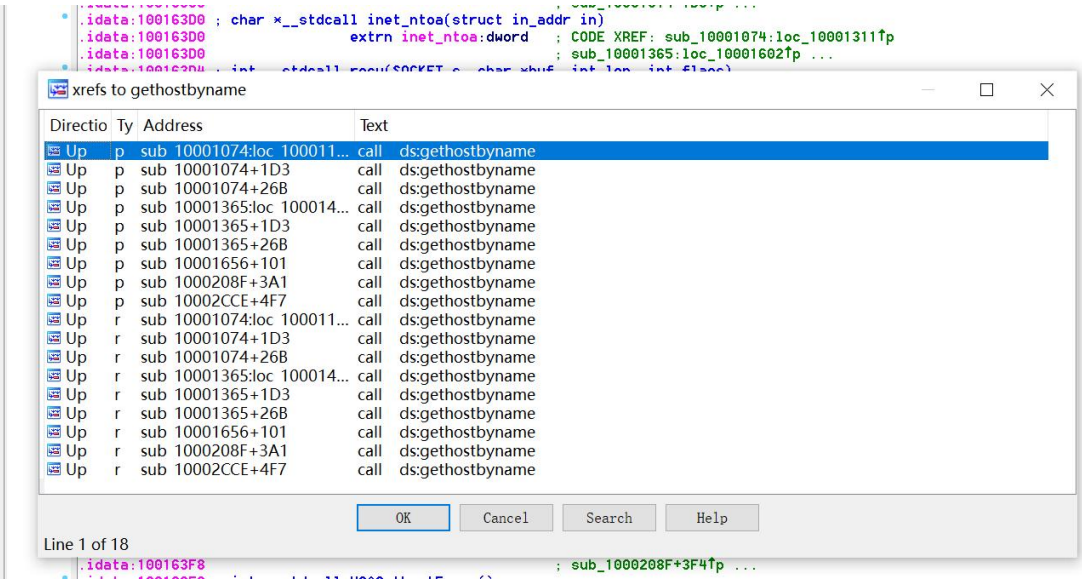
通过 View ->Open Subviews -> Imports 来查看这个 DLL 程序的导入表，直接搜索函数名 gethostbyname 并双击进入：





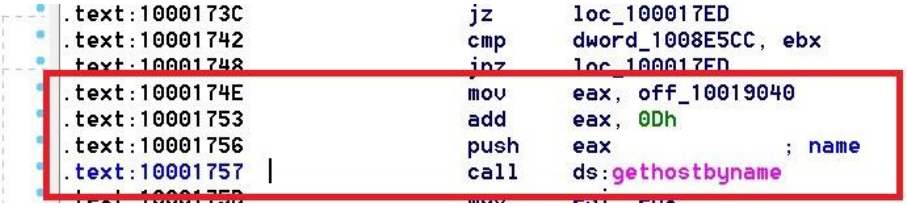
3. 有多少函数调用了 gethostbyname?

Ctrl + X 查看交叉引用，看上去有 18 行，实际上 IDA Pro 6.8 计算了两次交叉引用，一次是类型 p（被调用的引用），另一次是 r（被读取的引用），所以是 9 次交叉引用。按照 Address 排序，仔细数一下，一共被 5 个函数调用。后面 + 号和 : 号都是偏移地址，都属于同一个函数。



4. 将精力集中在位于 0x10001757 处的对 gethostbyname 的调用，你能找出哪个 DNS 请求将被触发吗?

按 G 输入跳转的地址 0x10001757，其中 gethostbyname 方法调用了一个参数——应该是一个包含了域名的字符串，往前看发现 EAX 中存储的应该是该参数



在 off_10019040 处。双击偏移地址，可以发现该地址开始储存了字符串 [This is RDO\]pics.practicalmalwareanalysis.com。在将 0x10001757 放入 eax 中以后，又增加了 0Dh，经过计数可以发现，在域名前面的[This is RDO]刚好是 13 个字符的长度，也就是说在增加了 0Dh 以后，这个地址就正好指向字符“p”，也就是说在增加了 0Dh 以后这个 eax 中存放的就是真正需要访问的域名的字符串了。

[This is RDO]pics.practicalmalwareanalysis.com

↑

0123456789ABCD

```
.data:1001903C                                  ; sub_10001656+1BB↑r ...
.data:1001903C                                  ; "[This is RIP]                          "...
.data:10019040 off_10019040    dd offset aThisIsRdoPics_
.data:10019040                                  ; DATA XREF: sub_10001656:loc_10001722↑r
.data:10019040                                  ; sub_10001656+F8↑r ...
.data:10019040                                  ; "[This is RDO]pics.praticalmalwareanalys"...
.data:10019044 off_10019044    dd offset aThisIsRur ; DATA XREF: sub_10001074+59↑r
.data:10019044                                  ; sub_10001365+59↑r ...
```

5. IDA Pro 识别了在 0x10001656 处的子过程中的多少个局部变量?

按 G 跳转到 0x10001656。看到绿色的部分 IDA Pro 识别出来的局部变量。一共是 23 个，不包含最后一行的 lpThreadParameter，因为是传入的参数

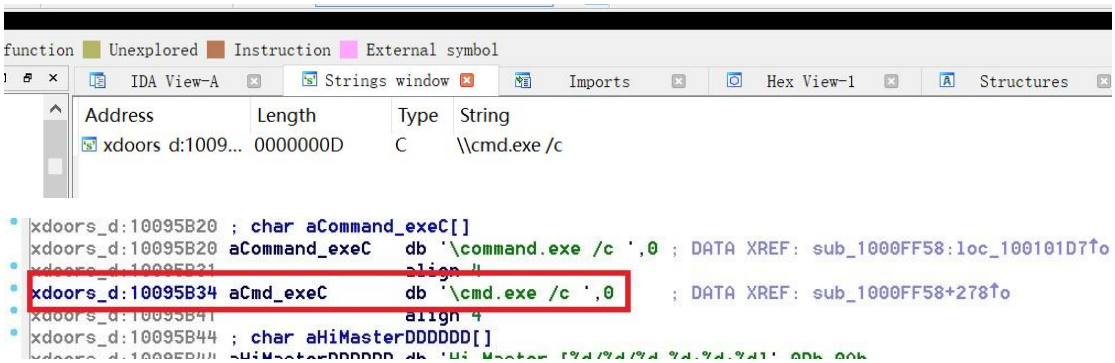
```
.text:10001656
.text:10001656 ; =============== S U B R O U T I N E =======================================
.text:10001656
.text:10001656
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
.text:10001656 sub_10001656    proc near               ; DATA XREF: DllMain(x,x,x)+C8↓o
.text:10001656
.text:10001656 var_675         = byte ptr -675h
.text:10001656 var_674         = dword ptr -674h
.text:10001656 hModule         = dword ptr -670h
.text:10001656 timeout         = timeval ptr -66Ch
.text:10001656 name            = sockaddr ptr -664h
.text:10001656 var_654         = word ptr -654h
.text:10001656 Dst             = dword ptr -650h
.text:10001656 Str1            = byte ptr -644h
.text:10001656 var_640         = byte ptr -640h
.text:10001656 CommandLine     = byte ptr -63Fh
.text:10001656 Str             = byte ptr -63Dh
.text:10001656 var_638         = byte ptr -638h
.text:10001656 var_637         = byte ptr -637h
.text:10001656 var_544         = byte ptr -544h
.text:10001656 var_50C         = dword ptr -50Ch
.text:10001656 var_500         = byte ptr -500h
.text:10001656 Buf2            = byte ptr -4FCh
.text:10001656 readfds         = fd_set ptr -4BCh
.text:10001656 buf             = byte ptr -3B8h
.text:10001656 var_3B0         = dword ptr -3B0h
.text:10001656 var_1A4         = dword ptr -1A4h
.text:10001656 var_194         = dword ptr -194h
.text:10001656 WSAData         = WSAData ptr -190h
.text:10001656 lpThreadParameter= dword ptr  4
.text:10001656
```

6. IDA Pro 识别了在 0x10001656 处的子过程中的多少个参数?

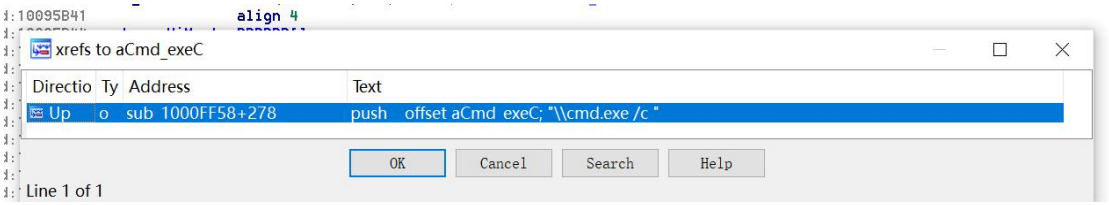
根据参数被标记和引用为正的偏移值，因此上图中可以观察到传入的参数为 lpThreadParameter。所以 IDA Pro 识别了子过程中的 1 个参数

7. 使用 Strings 窗口，在反汇编中定位字符串\cmd.exe /c。它位于哪?

通过 View -> Open Subviews -> Strings，直接搜索\cmd.exe /c，找到后双击，可以观察到该字符串位于 PE 文件 xdoors_d 节中的 0x10095B34 处

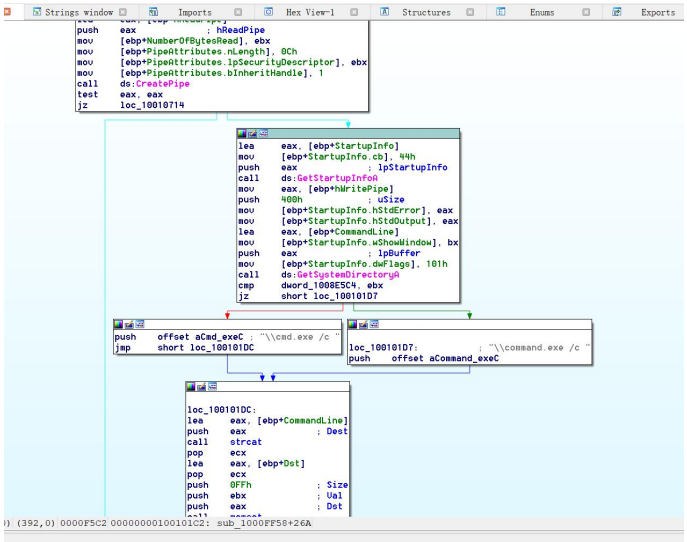


按下 Ctrl + X 查看交叉引用，有且仅有 sub_1000FF58+278 一处,此时，字符串被压到栈上：



8. 在引用\cmd.exe /c 的代码所在区域发生了什么?

在上图中，点击 OK，跳到 cmd.exe 被交叉引用的地方，也即 .text:100101D0 处，该命令被压栈。在命令的后面还能看到用 memcmp 比较 recv、quit、exit、cd、uptime 等指令字符串。在青框中的注释里，也出现了字符串 “This Remote Shell Session”。因此猜测这是一个远程 Shell 会话函数。

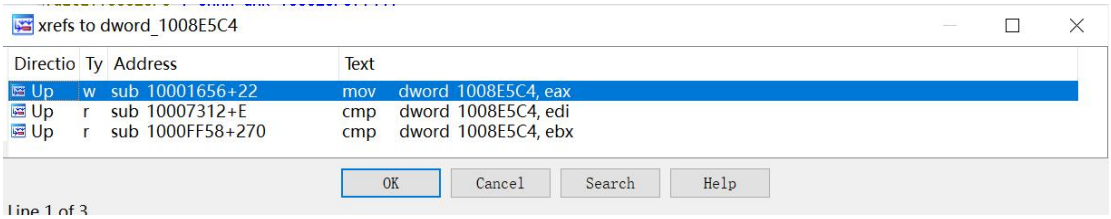


9. 在同样的区域，在 0x100101C8 处，看起来好像 dword_1008E5C4 是一个全局变量，它帮助决定走哪条路径。那恶意代码是如何设置 dword_1008E5C4 的呢?（提示：使用 dword_1008E5C4 的交叉引用）

跳到 100101c8，cmp dword_1008E5C4, ebx，也即将 ebx 同该全局变量比较。

```
.text:100101C2                call    ds:GetSystemDirectoryA
.text:100101C8                cmp     dword_1008E5C4, ebx
.text:100101CE                jz      short loc_100101D7
```

双击跳到定义位置（.data:1008E5C4），Ctrl+X 查看交叉引用，可以观察到它被引用了三次，只有 sub 10001656+22 处的 mov dword 1008E5C4, eax 语句修改了该值。选中，点击 OK 跳到对应位置。



其中 eax 是上一行调用的函数 sub_10003695 的返回值，双击进入该函数：

```
.text:1000166F                mov     [esp+688h+hModule], ebx
.text:10001673                call    sub_10003695
.text:10001678                mov     dword_1008E5C4, eax
.text:1000167D                call    sub_100036C3
.text:10001682                push    3A98h           ; dwMilliseconds
.text:10001687                mov     dword_1008E5C8, eax
```

可以看到该函数调用了 GetVersionEx，也即获取当前操作系统的信息，xor eax, eax 语句将 eax 置 0，并且，cmp [ebp+VersionInformation.dwPlatformId],2 语句将平台类型同 2 相比。这里只是简单的判断当前操作系统是否为 Windows 2000 或更高版本，根据微软的文档，我们得知通常情况下 dwPlatformId 的值为 2

```
.text:10003695 VersionInformation= _OSVERSIONINFOA ptr -94h
.text:10003695
.text:10003695                push    ebp
.text:10003696                mov     ebp, esp
.text:10003698                sub     esp, 94h
.text:1000369E                lea     eax, [ebp+VersionInformation]
.text:100036A4                mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
.text:100036AE                push    eax             ; lpVersionInformation
.text:100036AF                call    ds:GetVersionExA
.text:100036B5                xor     eax, eax
.text:100036B7                cmp     [ebp+VersionInformation.dwPlatformId], 2
.text:100036BE                setz    al
.text:100036C1                leave
.text:100036C2                retn
```

10. 在位于 0x1000FF58 处的子过程中的几百行指令中，有一系列使用 memcmp 来比较字符串的指令。如果对 rotbotwork 的字符串比较是成功的（memcmp 返回 0），会发生什么？

通过定位到 0x1000FF58 之后观察到一系列 memcmp，往下看在 0x10010452 可以看到与 robotwork 的 memcmp，如果 eax 和 robotwork 相同，则 memcmp 的结果为 0，也即 eax 为 0。test 的作用 和 and 类似，只是不修改寄存器操作数，只修改标志寄存器，因此 test eax,eax 的含义是，若 eax 为 0，那么 test 的结果为 ZF=1。jnz 检验的标志位就是 ZF，若 ZF=1，则不会跳转，继续向下执行，直到 call sub_100052A2。

```
.text:10010444 ; ---------------------------------------------------------------------------
.text:10010444
.text:10010444 loc_10010444:                           ; CODE XREF: sub_1000FF58+4E0↑j
.text:10010444                 push    9               ; Size
.text:10010446                 lea     eax, [ebp+Dst]
.text:1001044C                 push    offset aRobotwork ; "robotwork"
.text:10010451                 push    eax             ; Buf1
.text:10010452                 call    memcmp
.text:10010457                 add     esp, 0Ch
.text:1001045A                 test    eax, eax
.text:1001045C                 jnz     short loc_10010468
.text:1001045E                 push    [ebp+s]         ; s
.text:10010461                 call    sub_100052A2
.text:10010466                 jmp     short loc_100103F6
.text:10010468 ; ---------------------------------------------------------------------------
```

双击查看 sub_100052A2 的代码。其参数为 socket 类型。也就是上图中 1001045E 处 push 进去的 [ebp+s]。其他可以忽略，往下一直翻到 100052E7 处的 aSoftWareMicros，其值为 “SOFTWARE\Microsoft\Windows\CurrentVersion”。最后调用 RegOpenKeyEx 函数，读取该注册表值。

```
.text:100052A2                 push    ebp
.text:100052A3                 mov     ebp, esp
.text:100052A5                 sub     esp, 60Ch
.text:100052AB                 and     [ebp+Dest], 0
.text:100052B2                 push    edi
.text:100052B3                 mov     ecx, 0FFh
.text:100052B8                 xor     eax, eax
.text:100052BA                 lea     edi, [ebp+var_60B]
.text:100052C0                 and     [ebp+Data], 0
.text:100052C7                 rep stosd
.text:100052C9                 stosw
.text:100052CB                 stosb
.text:100052CC                 push    7Fh
.text:100052CE                 xor     eax, eax
.text:100052D0                 pop     ecx
.text:100052D1                 lea     edi, [ebp+var_20B]
.text:100052D7                 rep stosd
.text:100052D9                 stosw
.text:100052DB                 stosb
.text:100052DC                 lea     eax, [ebp+phkResult]
.text:100052DF                 push    eax             ; phkResult
.text:100052E0                 push    0F003Fh         ; samDesired
.text:100052E5                 push    0               ; ulOptions
.text:100052E7                 push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe"...
.text:100052EC                 push    80000002h       ; hKey
.text:100052F1                 call    ds:RegOpenKeyExA
.text:100052F7                 test    eax, eax
.text:100052F9                 jz      short loc_10005309
.text:100052FB                 push    [ebp+phkResult] ; hKey
.text:100052FE                 call    ds:RegCloseKey
.text:10005304                 jmp     loc_100053F6
.text:10005309 ; ---------------------------------------------------------------------------
.text:10005309
```

11. PSLIST 导出函数做了什么?

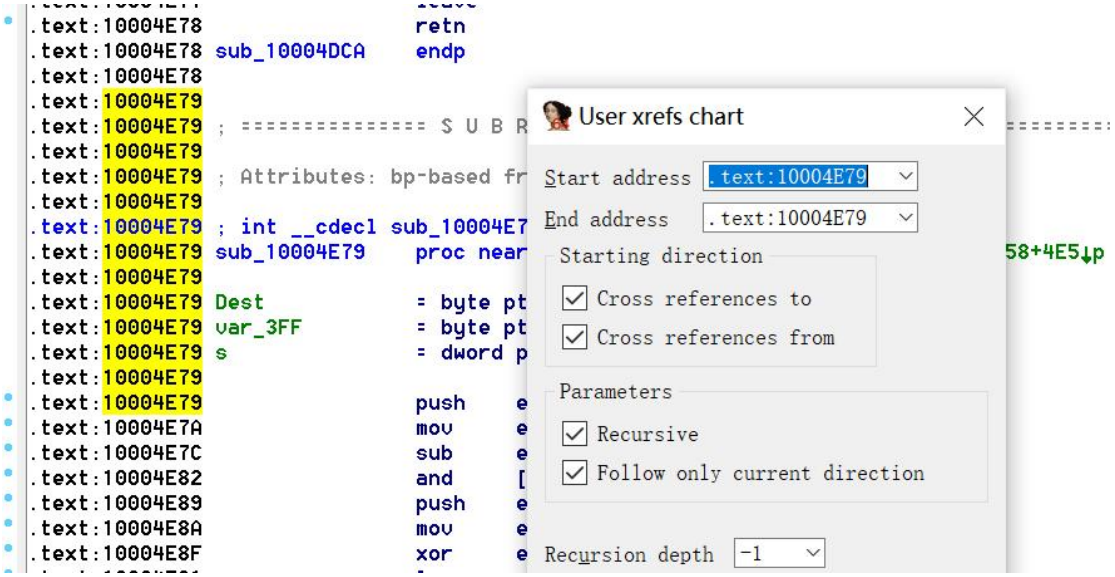
在 Exports 窗口找到 PSLIST，双击点开。首先调用 sub_100036C，这个函数检查操作系统的版本是 Windows Vista/7 或是 Windows XP/2003/2000。这两条代码都是用 CreateToolhelp32Snapshot 函数，从相关字符串和 API 调用来看，用于获得一个进程列表，这两条代码都通过 send 将进程列表通过 socket 发送。

```
.text:10007025
.text:10007025                 mov     dword_1008E5BC, 1
.text:1000702F                 call    sub_100036C3
.text:10007034                 test    eax, eax
.text:10007036                 jz      short l
.text:10007038                 push    [esp+St
.text:1000703C                 call    strlen
.text:10007041                 test    eax, ea
.text:10007043                 pop     ecx
.text:10007044                 jnz     short l
.text:10007046                 push    eax
.text:10007047                 call    sub_100
.text:1000704C                 jmp     short l
.text:1000704E ; ---------------------------------
.text:1000704E
```

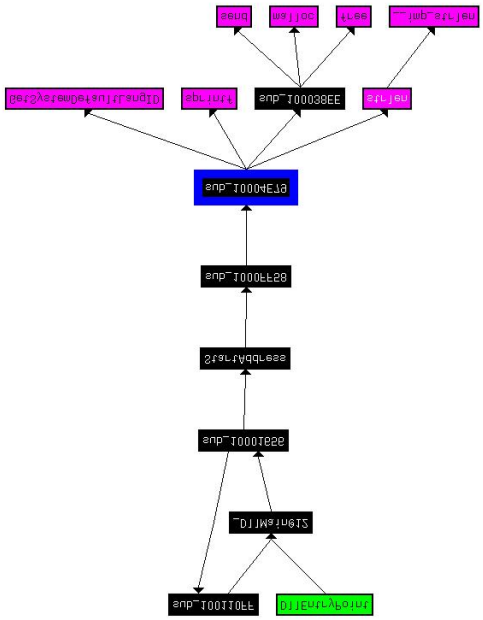
```
; Attributes: bp-based frame

sub_100036C3    proc near               ; CODE XREF: sub_10001656+27↑p
                                        ; PSLIST+A↓p

VersionInformation= _OSVERSIONINFOA ptr -94h

                push    ebp
                mov     ebp, esp
```

12. 使用图模式来绘制出对 sub_10004E79 的交叉引用图。当进入这个函数时，哪个 API 函数可能被调用？仅仅基于这些 API 函数，你会如何重命名这个函数?

先找到 sub_10004E79 函数位置，通过 View -> Graphs -> User xrefs chart 得到交叉引用图。

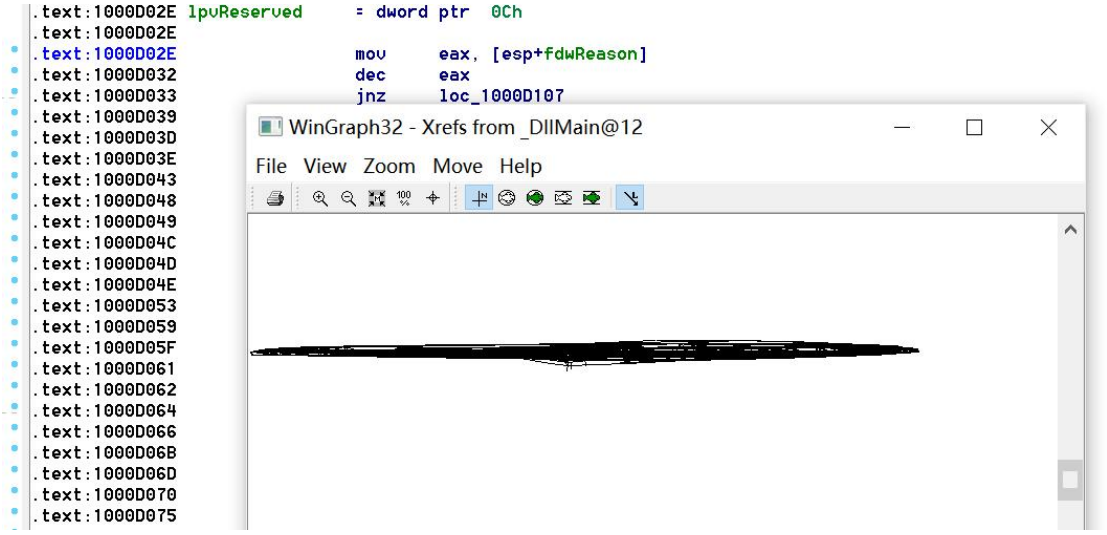


主要调用的 API 为 GetSystemDefaultLangID 和 send。因此推测可能是通过 socket 发送语言标志，可以直接在函数名处右键 Rename 重命名为 send_languageID。

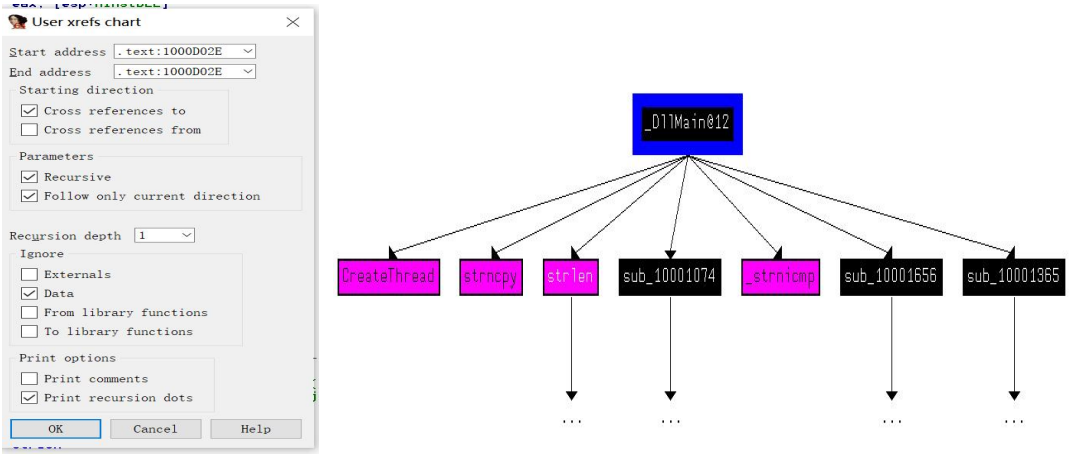


13. DllMain 直接调用了多少个 Windows API？多少个在深度为 2 时被调用?

按 G 跳到 Dllmain 的起始位置 1000D02E，然后右键 Xrefs graph from ，可以看到图中一团黑。



因此改为 View -> Graphs -> User xrefs chart 选项弹出对话框，曲线勾选 Cross references to，并且将 Recursion depth 从 -1 改为 1。可以看到调用的 Windows API 为 CreateThead、strncpy、strlen 以及 _strnicmp。



14. 在 0x10001358 处，有一个对 Sleep（一个使用包含要睡眠的毫秒数的参数的 API 函数）的调用。顺着代码向后看，如果这段代码执行，这个程序会睡眠多久?

调用的 sleep 的参数为上一行 push 的 eax，eax 的值又来自乘法 imul eax,3E8h 的运算结果。3E8h 的十进制为 1000。再往上，eax 是由 atoi 函数对 Str 运算得到的，也即字符串转整数。

Str 由 off_10019020+0Dh 位置的字符串得到，也即 “30”，最终转换成数字 30。所以睡眠的时间应为 30*1000 = 30000（毫秒），也即 30 秒

[This is CTI]30

↑

0123456789ABCD

```
.text:10001341                  ; sub_10001074+1B0↑j ...
.text:10001341                  mov     eax, off_10019020
.text:10001346                  add     eax, 0Dh
.text:10001349                  push    eax
.text:1000134A                  call    ds:atoi
.text:10001350                  imul    eax, 3E8h
off_10019020    dd offset aThisIsCti30  ; DATA XREF: sub_10001074:loc_10001341↑r
                                        ; sub_10001365:loc_10001632↑r ...
                                        ; "[This is CTI]30"
```

15. 在 0x10001701 处是一个对 socket 的调用。它的 3 个参数是什么?

IDA 中显示的三个参数名：af、type、protocol。

```
.text:100016FB
.text:100016FB loc_100016FB:                           ; CODE XREF: sub_10001656+374↓j
.text:100016FB                                         ; sub_10001656+A09↓j
.text:100016FB                  push    6              ; protocol
.text:100016FD                  push    1              ; type
.text:100016FF                  push    2              ; af
.text:10001701                  call    ds:socket
.text:10001707                  mov     edi, eax
.text:10001709                  cmp     edi, 0FFFFFFFFh
.text:1000170C                  jnz     short loc_10001722
.text:1000170E                  call    ds:WSAGetLastError
.text:10001714                  push    eax
.text:10001715                  push    offset aSocketGetlaste ; "socket() GetLastError reports %d\n"
.text:1000171A                  call    ds:__imp_printf
.text:10001720                  pop     ecx
.text:10001721                  pop     ecx
.text:10001722
.text:10001722 loc_10001722:                           ; CODE XREF: sub_10001656+B6↑j
```

16. 使用 MSDN 页面的 socket 和 IDA Pro 中的命名符号常量，你能使参数更加有意义吗？在你应用了修改以后，参数是什么？

根据 socket 官方文档，输入的参数含义为建立基于 IPv4 的 TCP 连接的 socket，通常在 HTTP 中使用。在数字上右键，Use standard symbolic constant，分别替换成实际的常量名，如图：

```
.text:100016FB loc_100016FB:                           ; CODE XREF: sub_10001656+374↓j
.text:100016FB                                         ; sub_10001656+A09↓j
.text:100016FB                  push    IPPROTO_TCP    ; protocol
.text:100016FD                  push    SOCK_STREAM    ; type
.text:100016FF                  push    AF_INET        ; af
.text:10001701                  call    ds:socket
.text:10001707                  mov     edi, eax
.text:10001709                  cmp     edi, 0FFFFFFFFh
.text:1000170C                  jnz     short loc_10001722
.text:1000170E                  call    ds:WSAGetLastError
.text:10001714                  push    eax
.text:10001715                  push    offset aSocketGetlaste ; "socket() GetLastError reports %d\n"
.text:1000171A                  call    ds:__imp_printf
.text:10001720                  pop     ecx
.text:10001721                  pop     ecx
.text:10001722
```

17. 搜索 in 指令（opcode 0xED）的使用。这个指令和一个魔术字符串 VMXh 用来进行 VMware 检测。这在这个恶意代码中被使用了吗？使用对执行 in 指令函数的交叉引用，能发现进一步检测 VMware 的证据吗?

Search -> Text（Alt+T），输入 in 或者 Search Sequence of Bytes（Alt+B），输入 ED。然后选择 Find All Occurences。搜索结果中只有在地址 100061DB 处的一条 in eax,dx 的指令符合要求。双击跳转到对应位置，可以看到 eax 中储存了字符串 “VMXh”，也就确认了这段代码采用了反虚拟机技巧,往上翻到所在函数的入口处，也即 sub_10006196，按下 Ctrl+X 打开交叉引用，选择第一个，点击 OK。可以看到该函数的后文中也出现了字符串 “Found Virtual Machine,Install Cancel.” 的字眼，印证了猜

测。

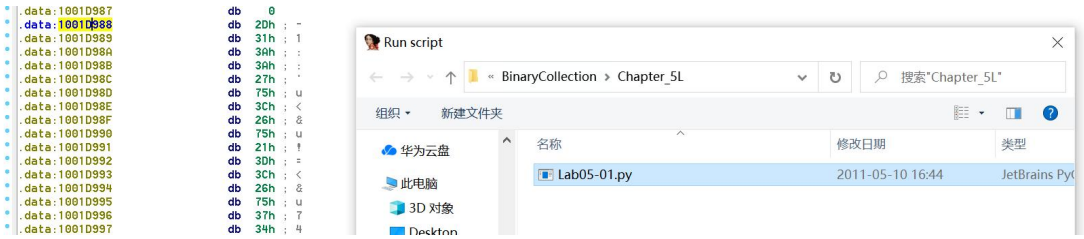
```
.text:1000D859                 test    eax, eax
.text:1000D85B                 pop     ecx
.text:1000D85C                 jz      short loc_1000D88E
.text:1000D85E                 call    sub_10006119
.text:1000D863                 test    al, al
.text:1000D865                 jnz     short loc_1000D870
.text:1000D867                 call    sub_10006196
.text:1000D86C                 test    al, al
.text:1000D86E                 jz      short loc_1000D88E
.text:1000D870
.text:1000D870 loc_1000D870:                           ; CODE XREF: InstallRT+1E↑j
.text:1000D870                 push    offset unk_1008E5F0 ; Format
.text:1000D875                 call    sub_10003592
.text:1000D87A                 mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:1000D881                 call    sub_10003592
.text:1000D886                 pop     ecx
.text:1000D887                 call    sub_10005567
.text:1000D88C                 jmp     short loc_1000D8A4
```

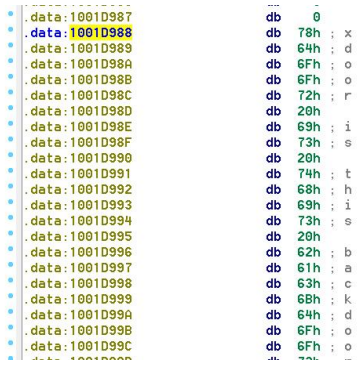
18. 将你的光标跳转到 0x1001D988 处，你发现了什么？

发现了一段看似随机的数据。据书中所述，需要运行一段 Python 脚本:

```
.data:1001D986                 db    0
.data:1001D987                 db    0
.data:1001D988                 db  2Dh ; -
.data:1001D989                 db  31h ; 1
.data:1001D98A                 db  3Ah ; :
.data:1001D98B                 db  3Ah ; :
.data:1001D98C                 db  27h ; '
.data:1001D98D                 db  75h ; u
.data:1001D98E                 db  3Ch ; <
.data:1001D98F                 db  26h ; &
.data:1001D990                 db  75h ; u
.data:1001D991                 db  21h ; !
.data:1001D992                 db  3Dh ; =
.data:1001D993                 db  3Ch ; <
.data:1001D994                 db  26h ; &
.data:1001D995                 db  75h ; u
.data:1001D996                 db  37h ; 7
.data:1001D997                 db  34h ; 4
.data:1001D998                 db  36h ; 6
.data:1001D999                 db  3Eh ; >
.data:1001D99A                 db  31h ; 1
.data:1001D99B                 db  3Ah ; :
.data:1001D99C                 db  3Ah ; :
.data:1001D99D                 db  27h ; '
.data:1001D99E                 db  79h ; y
```

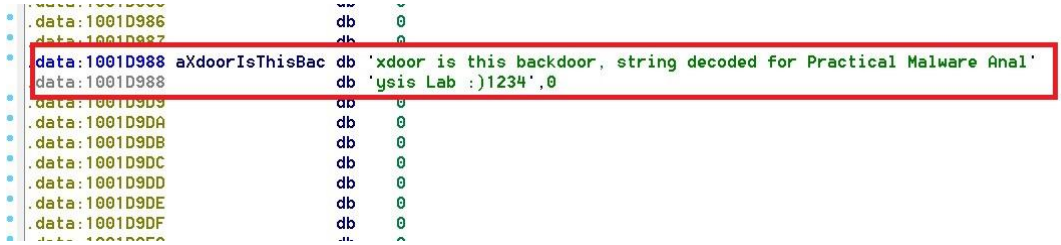
19. 如果你安装了 IDA Python 插件（包括 IDA Pro 的商业版本的插件），运行 Lab05-01.py，一个本书中随恶意代码提供的 IDA Pro Python 脚本，（确定光标是在 0x1001D988 处）在你运行这个脚本后发生了什么？





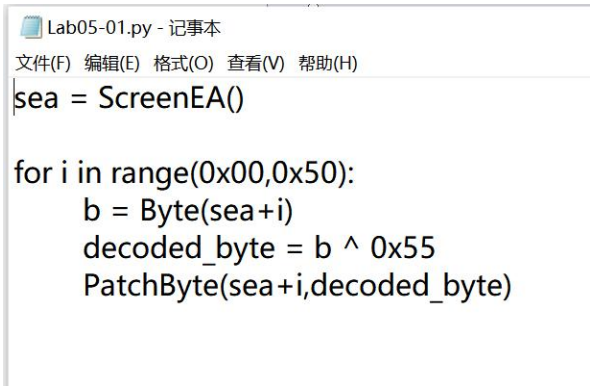
20. 将光标放在同一位置，你如何将这个数据转成一个单一的 ASCII 字符串？

在地址 0x1001D988 处，右键选择转换成 ASCII 字符串（或者按下 A 键），得到字符串 'xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234'：



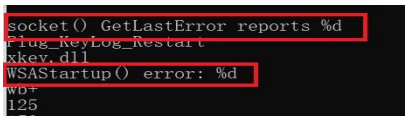
21. 使用一个文本编辑器打开这个脚本。它是如何工作的?

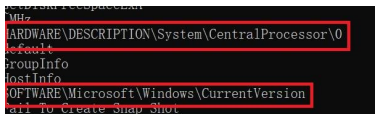
对长度为 0x50 字节的数据，用 0x55 分别与其进行异或，然后用 PatchByte 函数在 IDA Pro 中修改这些字节



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

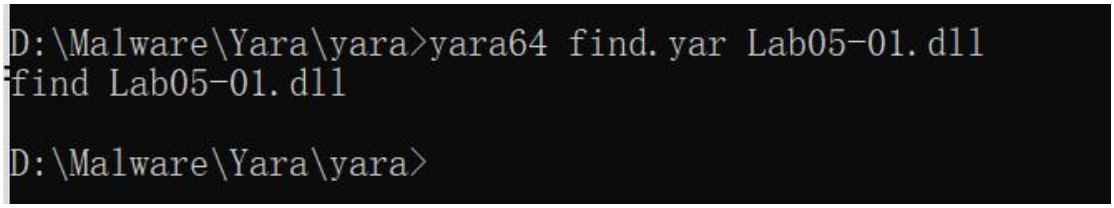




Yara 规则：

```
rule find
{
strings:
        $string1 = "socket() GetLastError reports %d"
        $string2 = "WSAStartup() error: %d"
        $string3 = "HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0"
        $string4 = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion"
condition:
        filesize<150KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550
    and all of them
}
```

运行结果：



五、 IDA Python 脚本编写

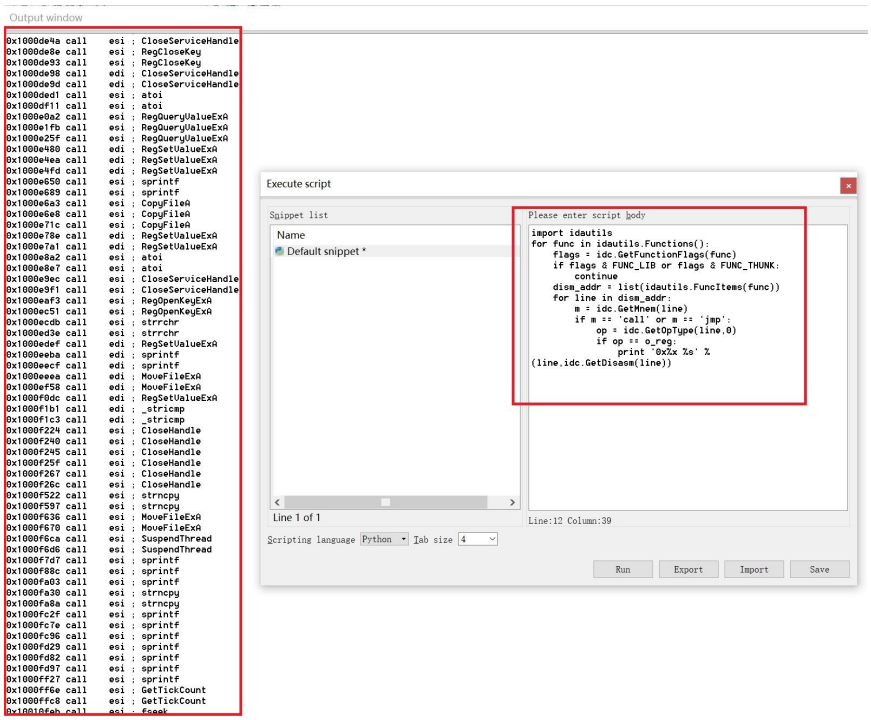
功能：

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 idautils
2.  for func in idautils.Functions():
3.      flags = idc.GetFunctionFlags(func)
4.      if flags & FUNC_LIB or flags & FUNC_THUNK:
5.          continue
6.      dism_addr = list(idautils.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：



六、 心得体会

1. 完成 IDA Python 代码编写时，一开始按照课上 PPT 的函数发现报错，通过查找资料发现可能因为 IDA 版本更新等原因 PPT 上某些函数已经不再使用，有新的函数进行替代。