

恶意代码分析与防治技术实验报告

Lab5

网络空间安全学院 信息安全专业

2112492 刘修铭 1063

<https://github.com/lxmliu2002/Malware Analysis and Prevention Techniques>

一、实验目的

1. 使用动态分析技术完成对给定病毒样本的分析；
2. 熟悉IDA的操作。

二、实验环境

为了保护本机免受恶意代码攻击，本次实验主体在虚拟机上完成，以下为相关环境：

1. 已关闭病毒防护的Windows11
2. 在VMware上部署的Windows XP虚拟机
 - 在进行动态分析时，需对虚拟机做如下处理：
 - 对VMware进行快照，便于恢复到运行前的状态
 - 启动ApateDNS，将DNS Reply IP设置为127.0.0.1
 - 启动Process Monitor，并按照实验要求设置过滤条件
 - 启动Process Explorer
 - 启动netcat：nc-l -p XXX
 - 启动wireShark抓取数据包

三、实验工具

1. 待分析病毒样本（解压缩于XP虚拟机）
2. 相关病毒分析工具，如PETools、PEiD、Strings等
3. Yara检测引擎

四、实验过程

（一）问题解答

1. DllMain的地址是什么?

使用IDA打开文件，定位DllMain，可以看到其地址为.text 0x1000D02E。

```

.text:1000D02A C9          leave     eax
.text:1000D02B C2 08 00    ret      8
ServiceMain
.text:1000D02B          endp
; ===== S U B R O U T I N E =
.text:1000D02E          ; BOOL __stdcall DllMain(HINSTANCE hinst
.text:1000D02E          _DllMain@12 proc near
.text:1000D02E
.text:1000D02E          hinstDLL      = dword ptr  4
.text:1000D02E          fdwReason     = dword ptr  8
.text:1000D02E          lpvReserved  = dword ptr 0Ch
.text:1000D02E
.text:1000D02E 8B 44 24 08    mov     eax, [esp+fdwReason]
.text:1000D032 48            dec     eax
.text:1000D033 0F 85 CE 00 00 00    jnz     loc_1000D107
.text:1000D039 8B 44 24 04    mov     eax, [esp+hinstDLL]
.text:1000D03D 53            push    ebx

```

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

打开Imports窗口，查找搜索定位该函数，然后双击查看，可以看到其地址为.idata 0x100163CC。

```

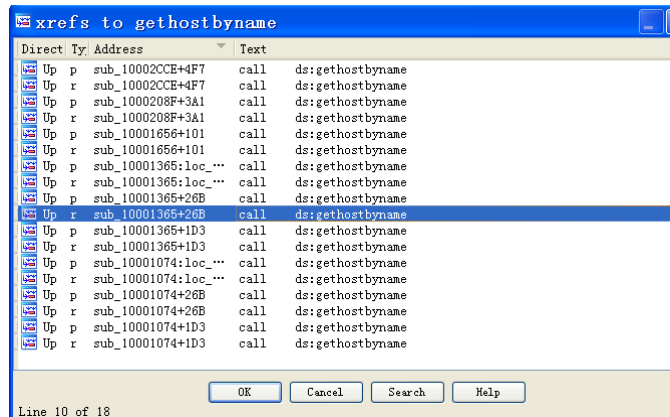
1000C3B 1001639C GetUserObjectInformationA USER32
1000C46 100163A4 waveInReset WINMM
1000C51 100163A8 waveInOpen WINMM
1000C56 100163AC waveInClose WINMM
1000C62 100163B0 waveInUnprepareHeader WINMM
1000C73 100163B4 waveInPrepareHeader WINMM
1000C8E 100163B8 waveInAddBuffer WINMM
1000C9D 100163BC waveInStart WINMM
1000CA5 100163C4 18 select WS2_32
1000CC0 100163C8 11 inet_addr WS2_32
1000CF3 100163CC 52 gethostbyname WS2_32
1000D00 100163D0 12 inet_ntoa WS2_32
1000D10 100163D4 16 recv WS2_32
1000D1D 100163D8 19 send WS2_32
1000D26 100163DC 4 connect WS2_32
1000D2F 100163E0 15 ntohs WS2_32
1000D3D 100163E4 9 htons WS2_32
1000D5B 100163E8 21 setsockopt WS2_32
1000D84 100163EC 116 WSACleanup WS2_32
1000D8A 100163F0 115 WSAStartup WS2_32
1000D92 100163F4 3 closesocket WS2_32
1000D9C 100163F8 20 WSACleanup WS2_32

.idata:100163C4 ; CODE XREF:
.idata:100163C8 ; unsigned __int32 __stdcall inet_addr(const char *c
.idata:100163C8 ?? ?? ?? ?? extrn inet_addr:dword ; CODE XREF:
.idata:100163C8 ; sub_100010
.idata:100163CC ; struct hostent * __stdcall gethostbyname(const char
.idata:100163CC ?? ?? ?? ?? extrn gethostbyname:dword
.idata:100163CC ; CODE XREF:
.idata:100163CC ; sub_100010
.idata:100163D0 ; char * __stdcall inet_ntoa(struct in_addr in)
.idata:100163D0 ?? ?? ?? ?? extrn inet_ntoa:dword ; CODE XREF:
.idata:100163D0 ; sub_100013
.idata:100163D4 ; int __stdcall recv(SOCKET s, char *buf, int len, i
.idata:100163D4 ?? ?? ?? ?? extrn recv:dword ; CODE XREF:
.idata:100163D4 ; sub_100016
.idata:100163D8 ; int __stdcall send(SOCKET s, const char *buf, int
.idata:100163D8 ?? ?? ?? ?? extrn send:dword ; CODE XREF:
.idata:100163D8 ; sub_100016
.idata:100163DC ; int __stdcall connect(SOCKET s, const struct socka
.idata:100163DC ?? ?? ?? ?? extrn connect:dword ; CODE XREF:
.idata:100163DC ; sub_100018

```

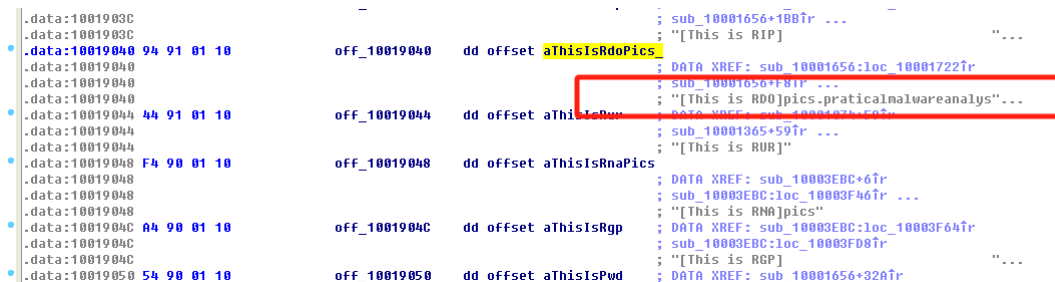
3. 有多少函数调用了gethostbyname?

使用 CTRL + X 查看其交叉引用，可以看到有18行记录。仔细查看该记录，可以看到IDA将p（被调用的引用）与r（被读取的引用）都予以计算，故而是9次引用。而地址栏中的+与. 都是表示地址偏移，故属于同一函数，故而共计有5个函数调用了该函数。



4. 将精力集中在位于0x1001757处的对gethostbyname的调用，你能找出哪个DNS 请求将被触发吗？

跳转到该地址，可以看到，该函数使用了一个参数。双击查看，可以发现，该地址存储了字符串[This is PDO]pics.practicalmalwareanalysis.com。其放入eax寄存器后，又增加了0Dh，经过计算可以发现，增加之后，该地址正好指向前面字符串中的p，即eax此时存储的为该url。



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

跳转到该地址后，可以看到许多绿色高亮的代码，这些即为局部变量，经过计算以及删除 arg_0（参数），可以知道，总共有23个。

```

.text:10001656          SUB_10001656      proc near                                ; UH1H
.text:10001656
.text:10001656          var_675          = byte ptr -675h
.text:10001656          var_674          = dword ptr -674h
.text:10001656          hLibModule      = 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          Parameter    = byte ptr -644h
.text:10001656          var_640       = byte ptr -640h
.text:10001656          CommandLine  = byte ptr -63Fh
.text:10001656          Source       = byte ptr -63Dh
.text:10001656          Data         = byte ptr -638h
.text:10001656          var_637       = byte ptr -637h
.text:10001656          var_544       = dword ptr -544h
.text:10001656          var_50C       = dword ptr -50Ch
.text:10001656          var_500       = dword ptr -500h
.text:10001656          Buf2         = byte ptr -4FCh
.text:10001656          readfds      = fd_set ptr -48Ch
.text:10001656          phkResult    = 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          arg_0        = dword ptr 4

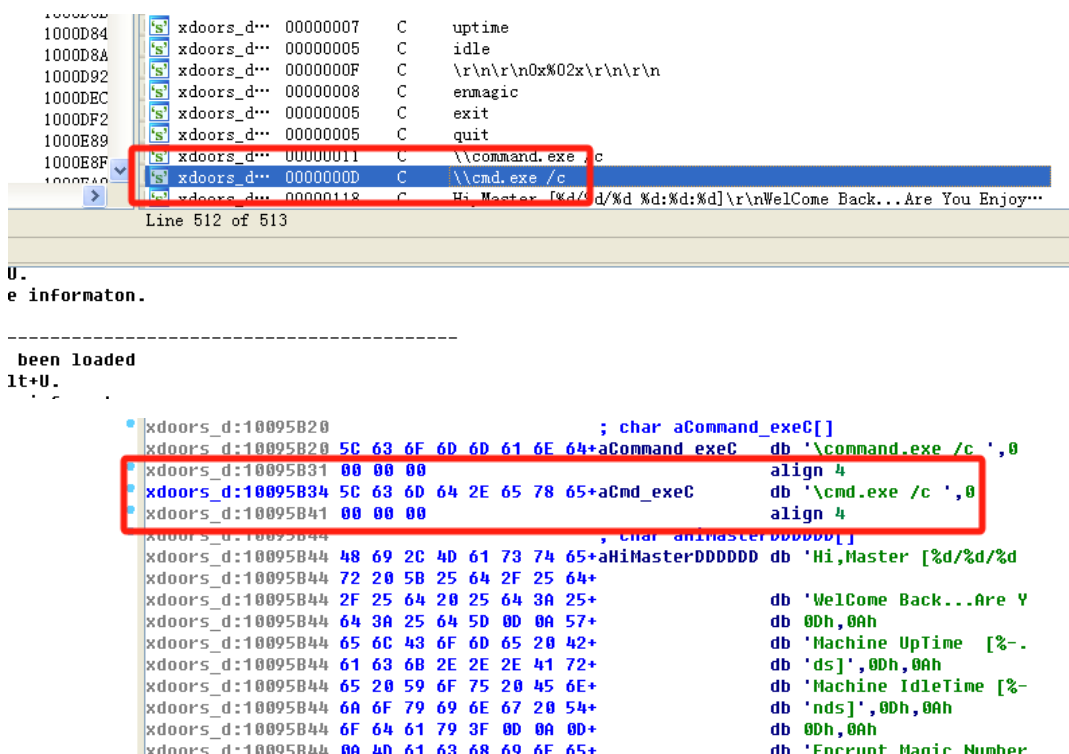
```

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

由上面图中分析可知，其识别了1个参数 arg_0。

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

在Strings窗口中搜索然后双击，即可看到其相关情况。可以看到，其位于xdoors_d 0x10095B34处。



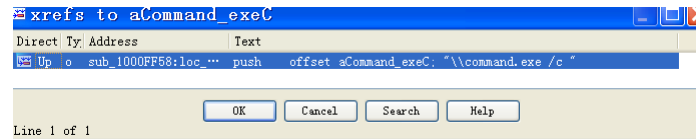
```

1000D84  xdoors_d... 00000007 C uptime
1000D8A  xdoors_d... 00000005 C idle
1000D92  xdoors_d... 0000000F C \r\n\r\n0x%02x\r\n\r\n
1000DEC  xdoors_d... 00000008 C ermagic
1000DF2  xdoors_d... 00000005 C exit
1000E89  xdoors_d... 00000005 C quit
1000E8F  xdoors_d... 00000011 C \\command.exe /c
1000E8F  xdoors_d... 00000000 C \\cmd.exe /c
1000E8F  xdoors_d... 00000118 C Hi,Master [%d/%d/%d\r\nWelCome Back...Are You Enjoy...
Line 512 of 513
U.
e informaton.
-----
been loaded
lt+U.
.
xdoors_d:10095B20          ; char aCommand_exeC[]
xdoors_d:10095B20 5C 63 6F 6D 6D 61 6E 64+aCommand_exeC db '\command.exe /c ',0
xdoors_d:10095B31 00 00 00 align 4
xdoors_d:10095B34 5C 63 6D 64 2E 65 78 65+aCmd_exeC db '\cmd.exe /c ',0
xdoors_d:10095B41 00 00 00 align 4
xdoors_d:10095B44          ; char aHiMasterDDDDDD[]
xdoors_d:10095B44 48 69 2C 4D 61 73 74 65+aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d
xdoors_d:10095B44 72 20 58 25 64 2F 25 64+
xdoors_d:10095B44 2F 25 64 20 25 64 3A 25+
xdoors_d:10095B44 64 3A 25 64 5D 0D 0A 57+
xdoors_d:10095B44 65 6C 43 6F 6D 65 20 42+
xdoors_d:10095B44 61 63 6B 2E 2E 2E 41 72+
xdoors_d:10095B44 65 20 59 6F 75 20 45 6E+
xdoors_d:10095B44 6A 6F 79 69 6E 67 20 54+
xdoors_d:10095B44 6F 64 61 79 3F 0D 0A 0D+
xdoors_d:10095B44 0A 4D 61 63 68 69 6F 65+

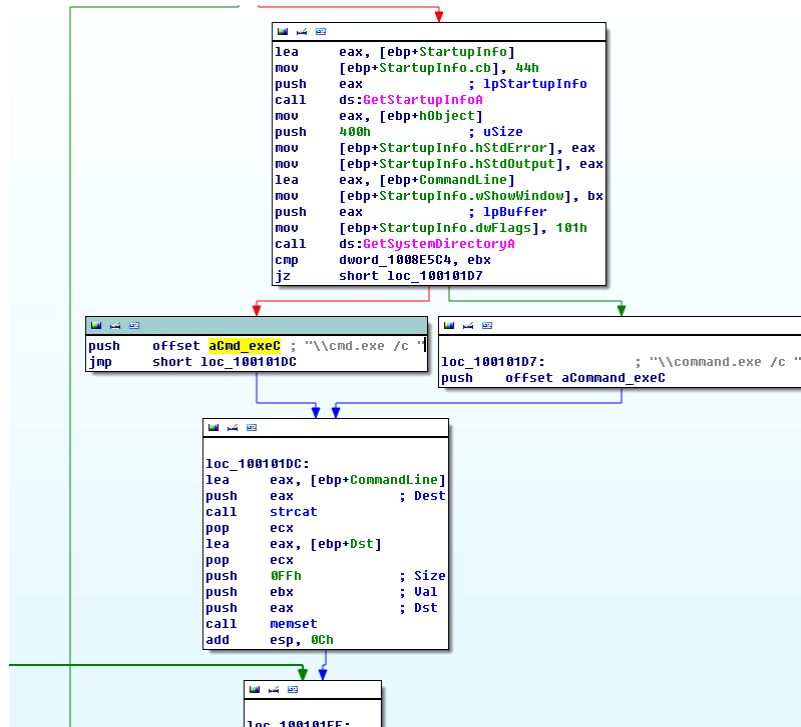
```

8. 在引用\cmd.exe /c 的代码所在区域发生了什么？

查看交叉引用，可以看到该字符串被压栈。



点击 **OK** 跳转到其被引用位置并切换视图，可以看到后面会有诸如recv、quit、exit、cd等指令，以及“This Remote Shell Session”字符串，推测是一个远程会话函数。

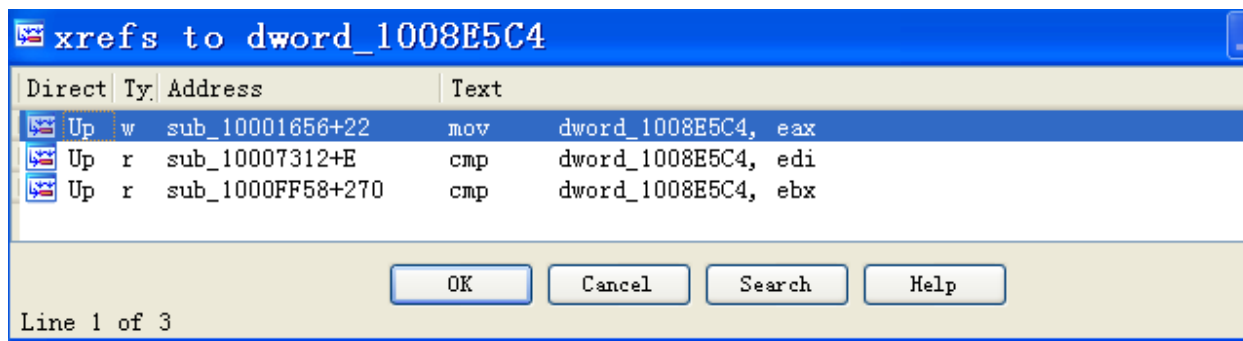


9. 在同样的区域，在0x100101C8处，看起来好像dword_1008E5C4是一个全局变量，它帮助决定走哪条路径。那恶意代码是如何设置dword_1008E5C4的呢？（提示：使用dword_1008E5C4的交叉引用）

跳转到该地址，可以看到 `cmp dword_1008E5C4, ebx` 的指令，即将ebx与该全局变量进行比较。



查看其交叉引用，可以看到，只有 `mov` 改变了其值。



跳转到该位置可以看到，eax是上面调用函数的返回值。

```

.text:1000166B 89 5C 24 14      mov     [esp+688h+var_674], ebx
.text:1000166F 89 5C 24 18      mov     [esp+688h+hLibModule], ebx
.text:10001673 E8 1D 20 00 00    call    sub_10003695
.text:10001678 A3 C4 E5 08 10    mov     dword_1008E5C4, eax
.text:1000167D E8 41 20 00 00    call    sub_100036C3
.text:10001682 68 98 3A 00 00    push    3A98h          ; dwMilliseconds
.text:10001687 A3 C8 E5 08 10    mov     dword_1008E5C8, eax
.text:1000168C FF 15 1C 62 01 10 call    ds:Sleep
.text:10001692 E8 68 FA 00 00    call    sub_100110FF
.text:10001697 8D 84 24 F8 04 00 00 lea     eax, [esp+688h+WSAData]

```

双击查看该函数，可以看到其调用了GetVersionEx，获取当前操作系统的信息。xor eax, eax语句则将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。

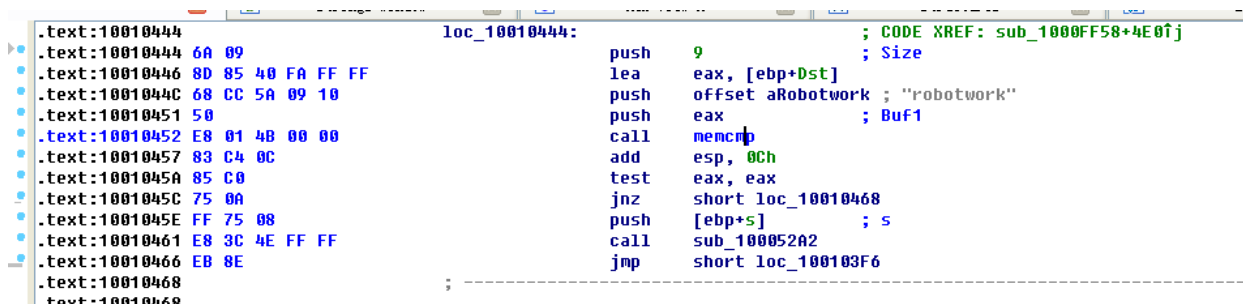
```

.text:10003695                                     VersionInformation= _OSVERSIONINFOA ptr -94h
.text:10003695                                     .text:10003695
.text:10003695 55                                     push    ebp
.text:10003696 8B EC                                     mov     ebp, esp
.text:10003698 81 EC 94 00 00 00                         sub     esp, 94h
.text:1000369E 8D 85 6C FF FF FF                         lea     eax, [ebp+VersionInformation]
.text:100036A4 C7 85 6C FF FF FF 94 00+                 mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
.text:100036AE 50                                     push    eax          ; lpVersionInformation
.text:100036AF FF 15 D4 60 01 10                         call    ds:GetVersionExA
.text:100036B5 33 C0                                     xor     eax, eax
.text:100036B7 83 BD 7C FF FF FF 02                     cmp     [ebp+VersionInformation.dwPlatformId], 2
.text:100036BE 0F 94 C0                                     setz    al
.text:100036C1 C9                                     leave
.text:100036C2 C3                                     retn
.text:100036C2                                     sub_10003695 endp

```

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？

定位到该位置，可以看到与robotwork比较的memcmp，如果eax和robotwork相同，则memcmp的结果为0，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语句的含义是，若eax为0，那么test的结果为ZF=1。而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。



双击查看该代码，可以看到其参数为socket类型，即上面push的[ebp+s]。继续阅读可以发现，后面aSoftWareMicros处的值为“SOFTWARE\Microsoft\Windows\CurrentVersion”，然后调用RegOpenKeyEx函数读取该注册表值。

```
push     eax                ; phhResult
push     0F003Fh           ; samDesired
push     0                 ; uOptions
push     offset aSoftWareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe".
push     80000002h         ; hKey
call     ds:RegOpenKeyExA
test     eax, eax
jz       short loc_10005309
push     [ebp+hKey]        ; hKey
call     ds:RegCloseKey
```

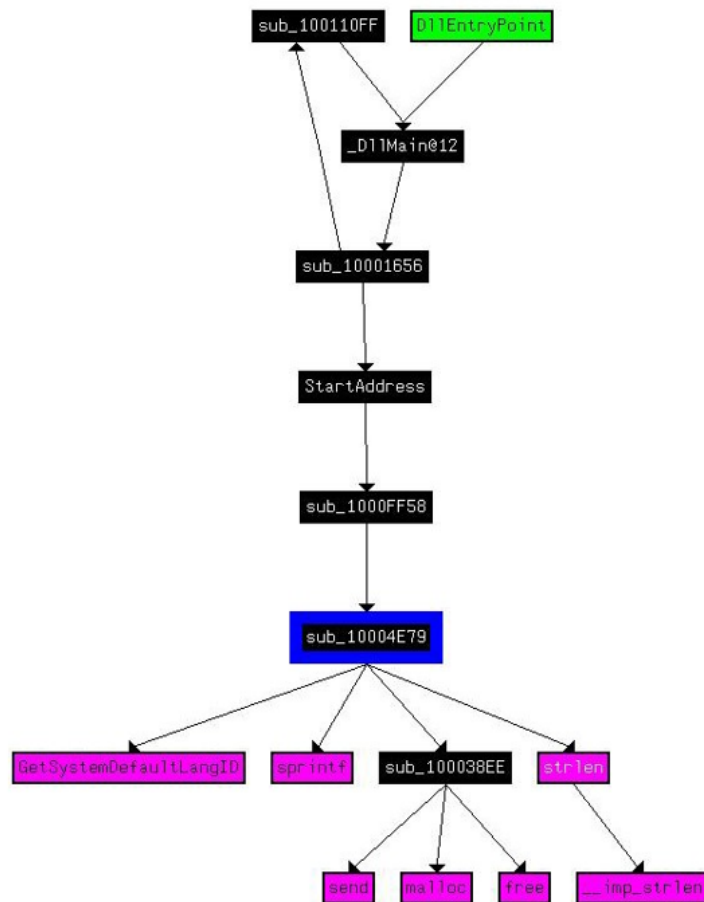
11. PSLIST导出函数做了什么？

在Exports窗口找到PSLIST，双击查看其情况。可以看到首先调用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:100036C3                                     VersionInformation= _OSVERSIONINFOA ptr -94h
.text:100036C3
.text:100036C3 55                                     push     ebp
.text:100036C4 8B EC                                     mov      ebp, esp
.text:100036C6 81 EC 94 00 00 00                         sub      esp, 94h
.text:100036CC 8D 85 6C FF FF FF                         lea      eax, [ebp+VersionInformation]
.text:100036D2 C7 85 6C FF FF FF 94 00+                 mov      [ebp+VersionInformation.dwOSVersionInfoSize], 94h
.text:100036D0 50                                     push     eax                ; lpVersionInformation
.text:100036D0 FF 15 D4 60 01 10                         call     ds:GetVersionExA
.text:100036E3 83 BD 7C FF FF FF 02                     cmp      [ebp+VersionInformation.dwPlatformId], 2
.text:100036EA 75 0E                                     jnz      short loc_100036FA
.text:100036EC 83 BD 70 FF FF FF 05                     cmp      [ebp+VersionInformation.dwMajorVersion], 5
.text:100036F3 72 05                                     jnb      short loc_100036FA
.text:100036F5 6A 01                                     push     1
.text:100036F7 58                                     pop      eax
.text:100036F8 C9                                     leave
.text:100036F9 C3                                     retn
```

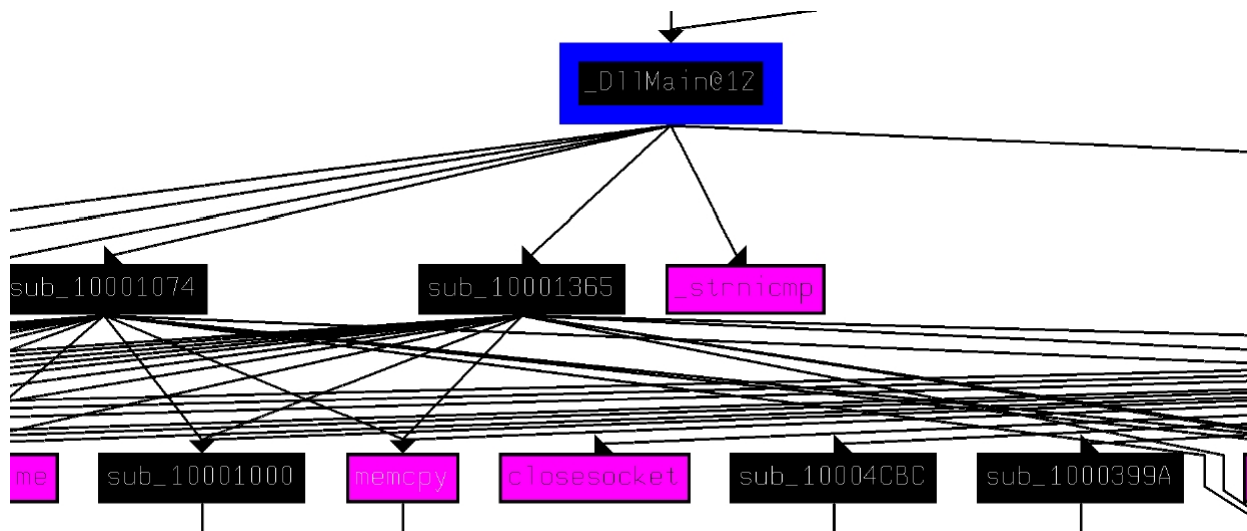
12. 使用图模式来绘制出对sub_10004E79的交叉引用图。当进入这个函数时，哪个API函数可能被调用？仅仅基于这些API函数，你会如何重命名这个函数？

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，因而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?

跳转到DllMain，右键Xrefs graph from，可以看到该图非常复杂，即DllMain调用了非常多函数。



观察可知，第一层调用了如sub_10001365、_strnicmp、strncpy、strlen等函数，接着在第二层又调用了如__imp_strlen、memcpy、gethostbyname等函数。

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，而eax的值又来自imul eax,3E8h的运算结果。再往上看，可以看到，eax是由atoi函数对Str运算得到的，也即字符串转整数。

```
; UINT __stdcall WinExec(LPCSTR lpCmdLine, UINT uCmdShow)
extrn _WinExec:dword ; CODE XREF: sub_10001074+70h↑
; void __stdcall Sleep(DWORD dwMilliseconds)
extrn Sleep:dword ; CODE XREF: sub_10001074+2E4h↑
; DWORD __stdcall GetCurrentProcessId()
extrn _GetCurrentProcessId:dword ; CODE XREF: sub_10001074+2E4h↑ ...

.text:10001332 83 25 CC E5 08 10 00 and dword_1000E5CC, 0
.text:10001339 EB 06 jmp short loc_10001341
; -----
.text:1000133B ;
.text:1000133B loc_1000133B: mov dword_1000E5CC, ebp ; CODE XREF: sub_10001074+71h↑
.text:10001341 loc_10001341: ; CODE XREF: sub_10001074+10Fh↑
; sub_10001074+180h↑ ...
.text:10001341 mov eax, off_10019020
.text:10001346 83 C0 00 add eax, 00h
.text:10001349 50 push eax ; Str
.text:1000134A FF 15 B4 62 01 10 call ds:atoi
.text:10001350 69 C0 E8 03 00 00 imul eax, 3E8h
.text:10001356 59 pop ecx
.text:10001357 50 push eax ; dwMilliseconds
.text:10001358 FF 15 1C 62 01 10 call ds:Sleep
.text:1000135E 33 ED xor ebp, ebp
.text:10001360 E9 4F FD FF jmp loc_100010B4
sub_10001074 endp
; ===== SUBROUTINE =====
```

继续回溯，可以看到，Str由off_10019020+0Dh位置的字符串得到，最终转换成数字30。所以睡眠的时间应为30*1000 = 30000（毫秒），即30秒。

```
db 0
db 0
off_10019020 dd offset aThisIsCTI30 ; DATA XREF: sub_10001074:loc_10001341↑tr
; sub_10001365:loc_10001632↑tr ...
; "[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？

跳转到该地址，可以看到三个参数名：af、type、protocol。

```
loc_10001701: ; CODE XREF: sub_10001074+6A↑
; sub_10001074+6A↑
push 6 ; protocol
push 1 ; type
push 2 ; af
call ds:socket
mov edi, eax
```

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？

查阅socket的官方文档，可以确认，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，分别替换成如图所示的实际的常量名。

```
1165B: ; CODE XREF: sub_10001701↑tr
; sub_10001701↑tr
push IPPROTO_TCP ; protocol
push SOCK_STREAM ; type
push AF_INET ; af
call ds:socket
```

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？

搜索in指令，可以发现，该指令只在.text 0x199G61DB处的in eax, dx处进行使用。

```
.text:100061D6 BA 58 56 00 00      mov     edi, 5658h
.text:100061DB ED                in      eax, dx
.text:100061DC 81 FB 68 58 4D 56      cmp     ebx, 564D5868h
.text:100061E2 0F 94 45 E4            setz    [ebp+var_1C]
.text:100061EA CB                    nop     ahv
```

双击查看其相关内容，可以看到，eax中存储了字符串“VMXh”，即反虚拟机技术。继续查找，可以看到其入口，查看交叉引用，可以看到字符串“Found Virtual Machine,Install Cancel.”，确认其使用反虚拟机技术。

```
.text:10000870
.text:10000870 loc_10000870: push    offset unk_1000E5F0 ; CODE XREF: InstallRT+1E1j
.text:10000875 68 F0 E5 08 10      call   sub_10003592
.text:1000087A C7 04 24 88 4F 10      mov     [esp+8+var_8], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:10000881 E8 0C 5D FF FF      call   sub_10003592
.text:10000886 59                pop     ecx
.text:10000887 E8 DB 7C FF FF      call   sub_10005567
.text:1000088C EB 16                jmp     short loc_100008A4
```

18. 将你的光标跳转到0x1001D988处，你发现了什么？

看到一串乱码。

```
.data:1001D980 00
.data:1001D987 00
.data:1001D988 20
.data:1001D989 31
.data:1001D98A 3A
.data:1001D98B 3A
.data:1001D98C 27
.data:1001D98D 75
.data:1001D98E 3C
.data:1001D98F 26
.data:1001D990 75
.data:1001D991 21
.data:1001D992 3D
.data:1001D993 3C
.data:1001D994 26
.data:1001D995 75
.data:1001D996 37
.data:1001D997 34
.data:1001D998 36
.data:1001D999 3E
.data:1001D99A 31
.data:1001D99B 3A
.data:1001D99C 3A
.data:1001D99D 27
.data:1001D99E 79
.data:1001D99F 75
.data:1001D9A0 26
.data:1001D9A1 24
```

```
00 0
db 20h ; -
db 31h ; 1
db 3Ah ; :
db 3Ah ; :
db 27h ; '
db 75h ; u
db 3Ch ; <
db 26h ; &
db 75h ; u
db 21h ; ?
db 3Dh ; =
db 3Ch ; <
db 26h ; &
db 75h ; u
db 37h ; 7
db 34h ; 4
db 36h ; 6
db 3Eh ; >
db 31h ; 1
db 3Ah ; :
db 3Ah ; :
db 27h ; '
db 79h ; y
db 75h ; u
db 26h ; &
db 24h ; 4
```

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？

可以看到文件被解密。

```

.data:1001D985 00
.data:1001D986 00
.data:1001D987 00
.data:1001D988 78
.data:1001D989 64
.data:1001D98A 6F
.data:1001D98B 6F
.data:1001D98C 72
.data:1001D98D 20
.data:1001D98E 69
.data:1001D98F 73
.data:1001D990 20
.data:1001D991 74
.data:1001D992 68
.data:1001D993 69
.data:1001D994 73
.data:1001D995 20
.data:1001D996 62
.data:1001D997 61
.data:1001D998 63
.data:1001D999 68
.data:1001D99A 64
.data:1001D99B 6F
.data:1001D99C 6F
.data:1001D99D 72
.data:1001D99E 2C
.data:1001D99F 20
.data:1001D9A0 73

```

```

uu 0
db 0
db 0
db 78h ; x
db 64h ; d
db 6Fh ; o
db 6Fh ; o
db 72h ; r
db 20h
db 69h ; i
db 73h ; s
db 20h
db 74h ; t
db 68h ; h
db 69h ; i
db 73h ; s
db 20h
db 62h ; b
db 61h ; a
db 63h ; c
db 68h ; k
db 64h ; d
db 6Fh ; o
db 6Fh ; o
db 72h ; r
db 2Ch ; ,
db 20h
db 73h ; s

```

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？

按下键盘上的 A 键，即可转换成ASCII字符串，得到“xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234”。

```

.data:1001D987 00
.data:1001D988 78 64 6F 6F 72 20 69 73+aXdoorIsThisBac
.data:1001D988 20 74 68 69 73 20 62 61+
.data:1001D9D9 00
.data:1001D9DA 00
.data:1001D9DB 00
db 0
db 'xdoor is this backdoor, string decoded for Practical Malware Anal'
db 'ysis Lab :)1234',0
db 0
db 0
db 0

```

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。

```

Lab05-01.py - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
sea = ScreenEA()
for i in range(0x00,0x50):
    b = Byte(sea+i)
    decoded_byte = b ^ 0x55
    PatchByte(sea+i,decoded_byte)

```

(二) yara规则

使用Strings对文件进行字符串分析，可以看到如下字符串。

```

sethostinfo
socket() GetLastError reports %d
Plug_KeyLog_Restart
xkey.dll
WSAStartup() error: %d
wb+

```

```
GetDiskFreeSpaceEx
- MHz
HARDWARE\DESCRIPTION\System\CentralProcessor\0
default
GroupInfo
HostInfo
SOFTWARE\Microsoft\Windows\CurrentVersion
Fail To Create Snap Shot
(1) Enter Current Directory Error Update Failed
```

利用上述字符串进行yara规则的编写，得到如下规则：

```
1 rule lab5
2 {
3   strings:
4     $string1 = "socket() GetLastError reports %d"
5     $string2 = "WSAStartup() error: %d"
6     $string3 = "HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0"
7     $string4 = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion"
8     $string5 = "xkey.dll"
9   condition:
10    filesize < 150KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) ==
    0x00004550 and all of them
11 }
```

下面是运行结果图。

```
又表最新的 Powershell，了解新功能和改进：https://aka.ms/PSWindows
PS E:\刘修铭\南开大学\个人材料\课程\2023-2024 第1学期\恶意代码分析
n_Techniques\lab4\yara> .\yara64.exe .\lab4.y .\Lab05-01.dll
lab4 .\Lab05-01.dll
PS E:\刘修铭\南开大学\个人材料\课程\2023-2024 第1学期\恶意代码分析
```

下面测试其运行效率，得到如下运行结果。

```
人材料\课程\2023-2024 第1学期\恶意代码分析与防治技术 王志，邓琼
文件 ./yara\Chapter_17L\Lab17-02.dll 匹配的规则：[lab4]
文件 ./yara\Chapter_5L\Lab05-01.dll 匹配的规则：[lab4]
程序运行时间：0.030779123306274414 秒
PS E:\刘修铭\南开大学\个人材料\课程\2023-2024 第1学期\恶意代码分析
```

(三) IDA python脚本编写

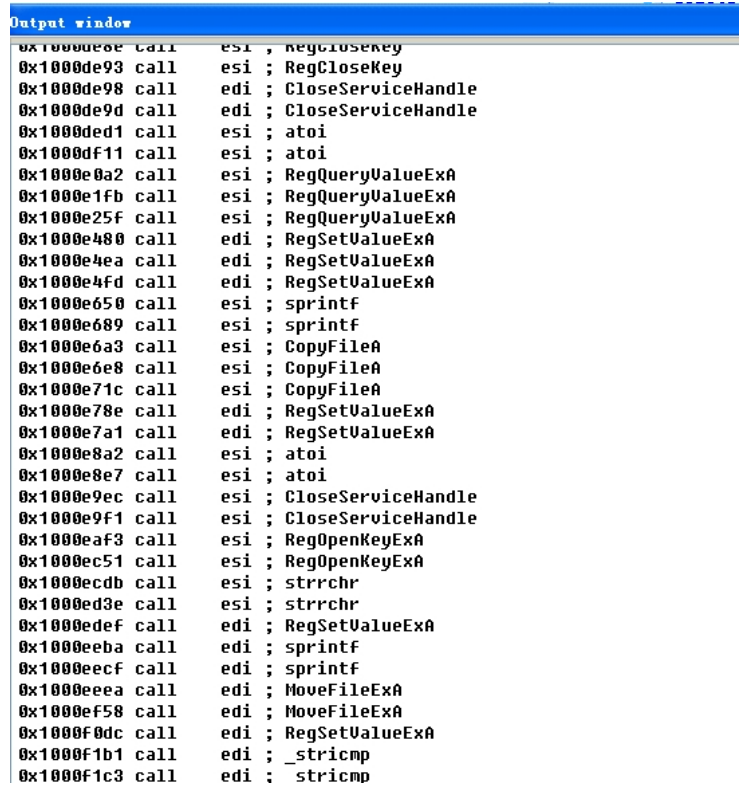
遍历所有函数，排除库函数或简单跳转函数，当反汇编的助记符为call或者jmp且操作数为寄存器类型时，输出该行反汇编指令。

```

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))

```

得到如下输出：



```

Output window
0x1000de92 call esi, RegCloseKey
0x1000de93 call esi; RegCloseKey
0x1000de98 call edi; CloseServiceHandle
0x1000de9d call edi; CloseServiceHandle
0x1000ded1 call esi; atoi
0x1000df11 call esi; atoi
0x1000e0a2 call esi; RegQueryValueExA
0x1000e1fb call esi; RegQueryValueExA
0x1000e25f call esi; RegQueryValueExA
0x1000e480 call edi; RegSetValueExA
0x1000e4ea call edi; RegSetValueExA
0x1000e4fd call edi; RegSetValueExA
0x1000e650 call esi; sprintf
0x1000e689 call esi; sprintf
0x1000e6a3 call esi; CopyFileA
0x1000e6e8 call esi; CopyFileA
0x1000e71c call esi; CopyFileA
0x1000e78e call edi; RegSetValueExA
0x1000e7a1 call edi; RegSetValueExA
0x1000e8a2 call esi; atoi
0x1000e8e7 call esi; atoi
0x1000e9ec call esi; CloseServiceHandle
0x1000e9f1 call esi; CloseServiceHandle
0x1000eaf3 call esi; RegOpenKeyExA
0x1000ec51 call esi; RegOpenKeyExA
0x1000ecdb call esi; strchr
0x1000ed3e call esi; strchr
0x1000edef call edi; RegSetValueExA
0x1000eeba call edi; sprintf
0x1000eecf call edi; sprintf
0x1000eeea call edi; MoveFileExA
0x1000ef58 call edi; MoveFileExA
0x1000f0dc call edi; RegSetValueExA
0x1000f1b1 call edi; _stricmp
0x1000f1c3 call edi; _stricmp

```

五、实验结论及心得

1. 了解并掌握了IDA分析功能；
2. python中版本更新对于API的影响较大，需适时更新。