南開大學

《恶意代码分析与防治技术》课程实验报告

实验五

学　　院　网络空间安全学院

专　　业　　信息安全

学　　号　　2112060

姓　　名　　孙蕗

班　　级　信息安全 1 班

《恶意代码分析与防治技术》课程 Lab5 实验报告

一、 实验目的

分析一个 DLL 文件中的恶意代码，使用逆向工程技术来了解该恶意代码的行为和功能。

通过 IDA Pro 工具来分析 DLL 文件，识别并理解其中的恶意代码，掌握 IDA Pro 的基本功能，如查找函数、变量、字符串等，以及进行逆向分析，探索 DLL 文件的导出函数和内部函数的调用关系。

二、 实验原理

IDA Pro 工具：IDA Pro 是一种专业的逆向工程工具，用于分析二进制文件，特别是可执行文件和动态链接库。它提供了强大的反汇编和分析功能，允许分析人员查看和理解二进制代码。

导出函数和内部函数：分析 DLL 文件的导出函数可以帮助了解该库提供的功能。内部函数的调用关系和交叉引用可以揭示代码的逻辑和行为。

字符串和常量分析：分析恶意代码中的字符串和常量可以揭示其功能、特征或目的。这些信息有助于理解代码的作用。

使用 Yara 规则：Yara 是一种用于创建规则以检测恶意代码特征的工具。创建和应用 Yara 规则可以帮助自动检测和识别恶意代码。

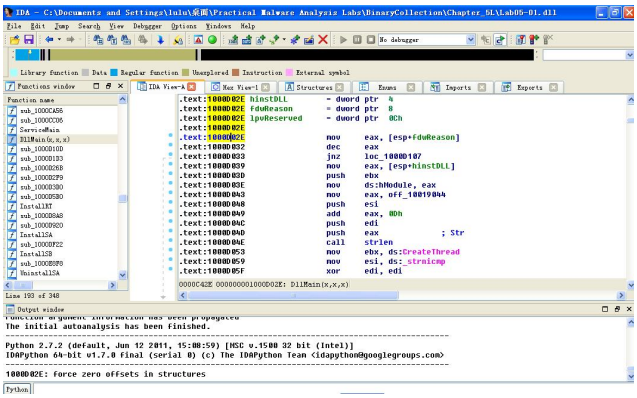
三、 实验过程

（一） Lab5-1

只用 IDA Pro 分析在文件 Lab05-01.dll 中发现的恶意代码。这个实验的目标是给你一个用 IDA Pro 动手的经验。如果你已经用 IDA Pro 工作过，你可以选择忽略这些问题，而将精力集中在逆向工程恶意代码上。

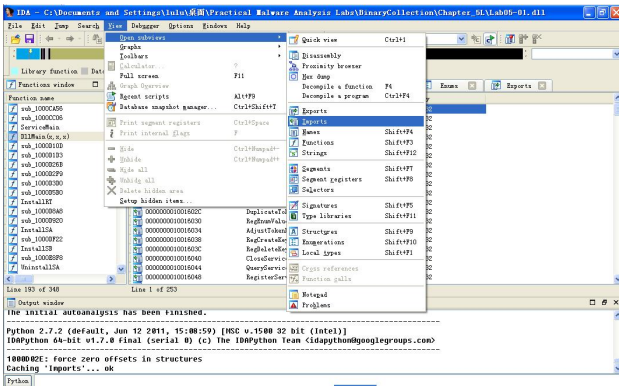
（1） DllMain 的地址是什么？

在 IDA Pro 中打开 Lab05-01.dll 文件，使用空格键从图形模式切换到汇编模式，在 function name 中选择 DllMain(x,x,x)，发现 DllMain 地址在.text 节的 0x1000D02E 处。



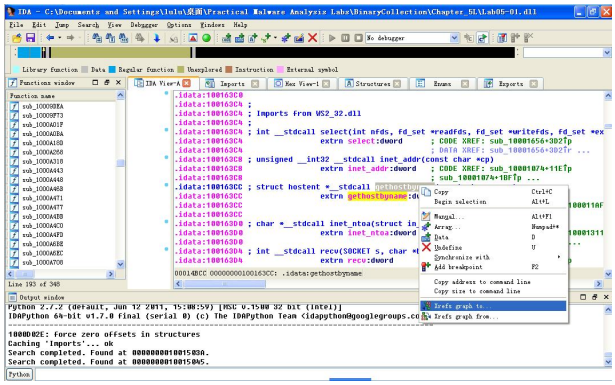
（2） 使用 Imports 窗口并测览到 gethostbyname,导入函数定位到什么地址?

通过 View->Open Subviews->Imports 查看该程序的导入表



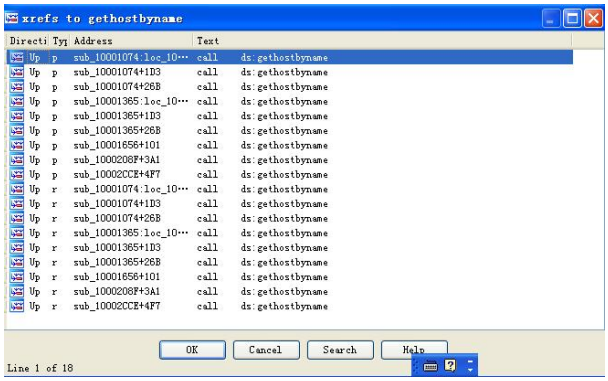
搜索 gethostbuname，地址定位到.idata 节的 0x100163CC 处。

（3） 有多少函数调用了 gethostbyname?



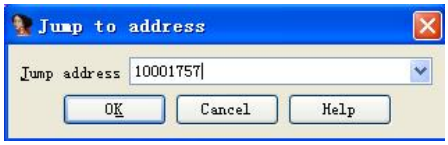
在汇编窗口中找到有 gethostbyname 函数名的语句，Ctrl+X 键来检查其交叉引用，其中 p 是引用，r 是读取，要先读取，再引用。

结合地址和偏移信息，发现 gethostbyname 在被 5 个不同的函数调用了 9 次。



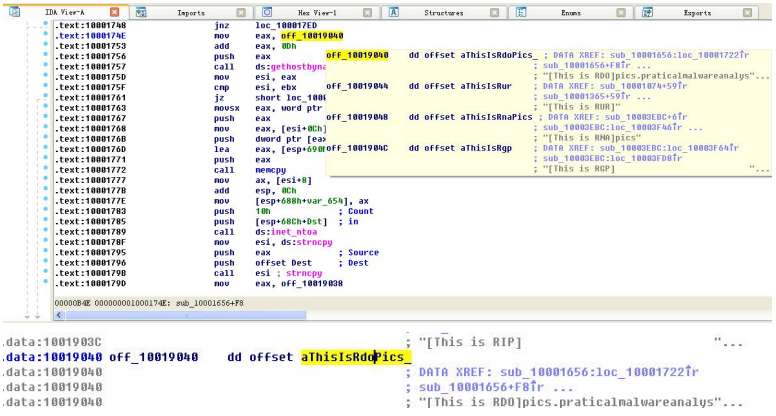
（4） 将精力集中在位于 0x10001757 处的对 gethostbyname 的调用，你能找出哪个 DNS 请求将被触发吗?

G 快速定位到 0x10001757，




```
.text:1000174E                  mov     eax, off_10019040
.text:10001753                  add     eax, 0Dh
.text:10001756                  push    eax             ; name
.text:10001757                  call    ds:gethostbyname
```

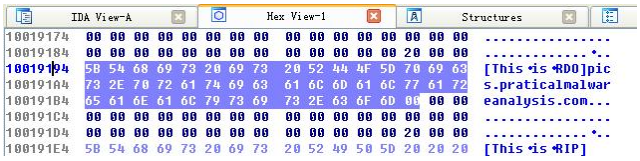
gethostbyname 方法用了一个参数——一个包含了域名的字符串，回看确定 gethostbyname 被调用时 off_10019040 被赋值给了 eax，在此处看到字符串 pics.practicalmalwareanalysis.com。



双击 aThisIsRdoPics_，发现了 praticalmalwareanalysis.com 的域名信息。

```
.data:10019193                  db    0
.data:10019194 aThisIsRdoPics_  db '[This is RDO]pics.praticalmalwareanalysis.com',0
.data:10019194                                  ; DATA XREF: .data:off_10019040↑o
.data:100191C2                  db    0
```

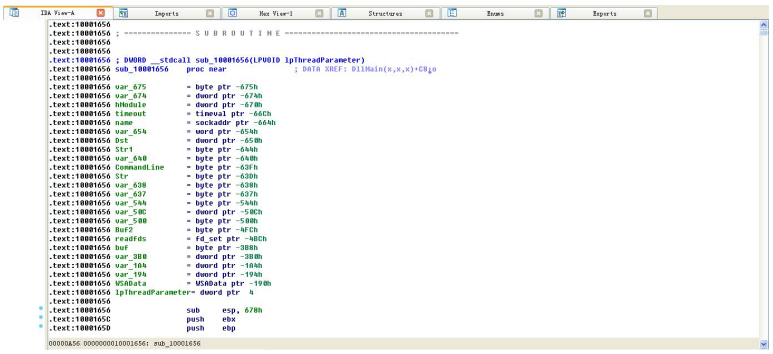
在 HEX-View1 窗口对 0x10019194 进行分析，前 13 字节就是[This is RDO]。0x10001753 处的代码令 eax 加上 0Dh，就是将 eax 指向了 praticalmalwareanalysis.com,然后再将 eax 压入栈作为 gethostbyname 的参数。



综上所述，恶意代码会发起对 pics.practicalmalwareanalysis.com 的 DNS 请求，以获得其 IP 地址。

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

G 快速定位到 0x10001656，被标记的局部变量与负偏移值相关，一共识别出 23 个局部变量。

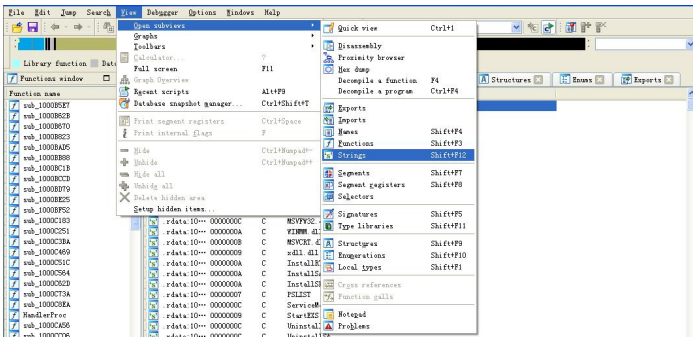


（6） IDA Pro 识别了在 0x10001656 处的子过程中的多少个参数?

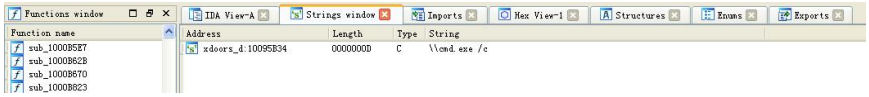
参数被标记和引用为正的偏移值，可以看到 IDA Pro 识别出函数的一个参数，标记为 lpThreadParameter，一共识别出 1 个参数。

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

View->Open Subviews->Strings，查看字符串

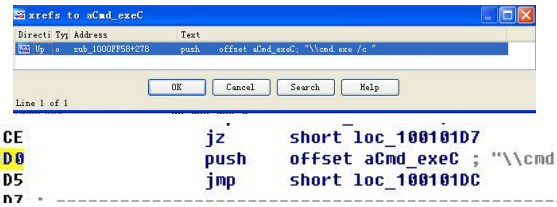


搜索字符串\cmd.exe /c，看到字符串\cmd.exe /c 出现在 xdoors_d 节的 0x10095B34 处。



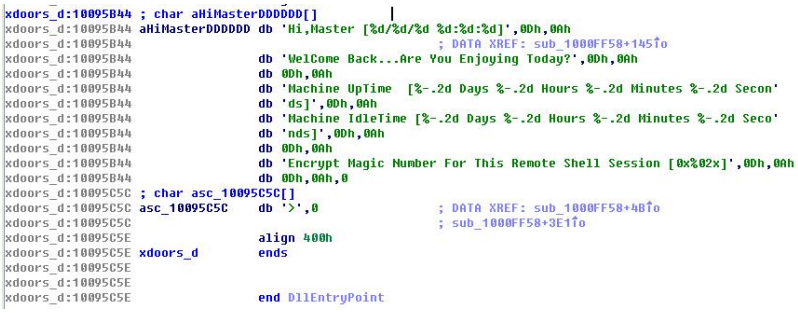
（8） 在引用\cmd.exe /c 的代码所在的区域发生了什么?

检查该字符串的交叉引用，只看到一处在 0x100101D0，字符串被压到栈上。

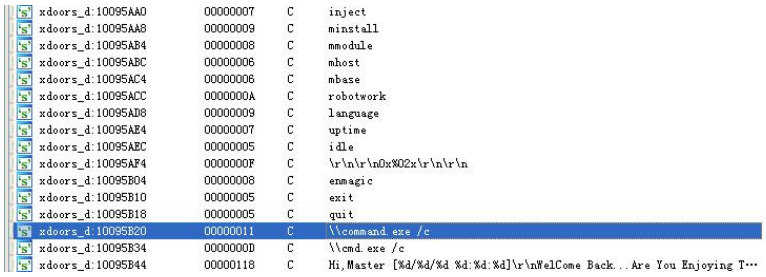


该字符串被引用之前的 0x1001009D 处，存在 aHiMasterDDDDDD，有一个字符串 This Remote Shell Session。

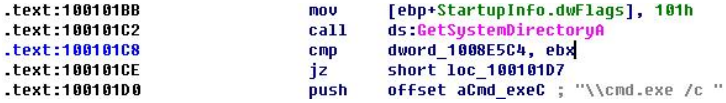
双击 aHiMasterDDDDDD 查看，



根据上图被选中区域的字符串，分析得出远程 shell 会话解密一段数字,我们正在查看的是一个远程 shell 会话函数,攻击者开启一个远程 shell 会话。



（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。)



dword_1008E5C4 函数是一个全局变量，在局部变量和参数以在 0x100101C8 处双击它，会跳转到内存中的 0x1008E5C4 处，位于 DLL 文件的.data 节。Ctrl+X 键来检查其交叉引用，可以看到被引用了三次，但只有第一处修改了 dword_1008E5C4。


```
.text:10001669                  xor     ebx, ebx
.text:1000166B                  mov     [esp+688h+var_674], ebx
.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
```

EAX 被赋给 dword_1008E5C4,而 EAX 是前一条指令函数调用的返回值。

双击 sub_10003695 查看 eax 的返回值，该函数包括了一个 GetVersionEx 的调用，可用于获取当前操作系统版本的信息。

```
.text:10003695
.text:10003695 ; =============== S U B R O U T I N E =======================================
.text:10003695
.text:10003695 ; Attributes: bp-based frame
.text:10003695
.text:10003695 sub_10003695    proc near               ; CODE XREF: sub_10001656+1D↑p
.text:10003695                                         ; sub_10003B75+7↓p ...
.text:10003695
.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
.text:100036C2 sub_10003695    endp
.text:100036C2
```

cmp [ebp+VerisonInformation.dwPlatformId],2 语句，将 VersionInformation.dwPlatformId 与数字 2 进行比较，来确定如何设置 AL 寄存器，2 代表 WIN32_NT 系统。如果 PlatformId 为 VER_PLATFORM_WIN32_NT,AL 会被置位。

这里的函数会检查一下当前操作系统是不是 win32 系统或更高版本来决定 eax 的值,之后再将得到的系统版本号存放入 dword_1008E5C4 中，该全局变量通常会被置为 1。

综上所述，操作系统版本号被保存在了 dword_1008E5C4 中。

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

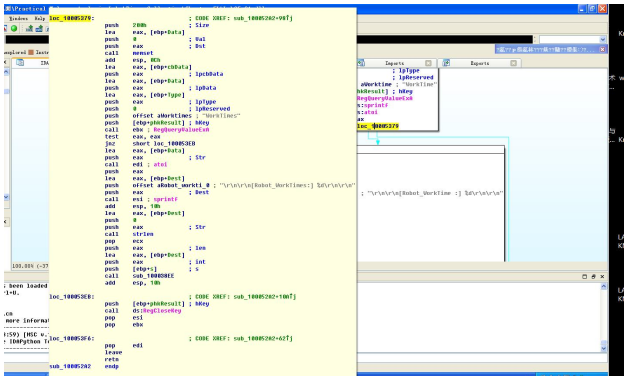
位于 0x1000FF58 处的远程 shell 函数从 0x1000FF58 开始包含了一系列的 memcmp 函数。其中，在 0x10010452,可以看到与 robotwork 的 memcmp。

```
.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 ; ---------------------------------------------------------------------------
```

如果该字符串为 robotwork,call sub_100052A2 处的代码会被调用。查看 sub_100052A2 的代码，可以看到它查询了注册表中 HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime 和 WorkTimes 键的值。

```
.text:100052A2
.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 ; ---------------------------------------------------------------------------
```

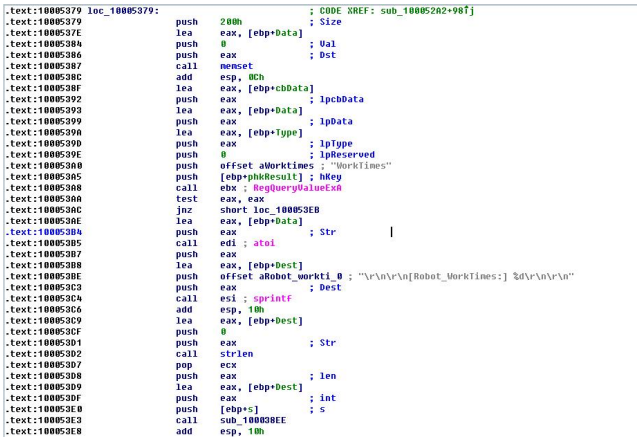
在 sub_100052A2 区域中发现调用了 sub_100038EE 函数。



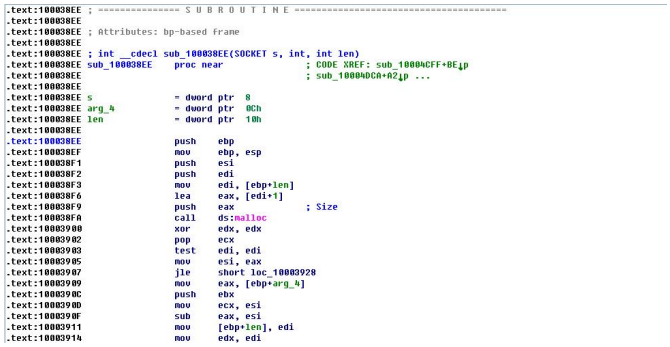
查看 loc_10005309，

```
.text:10005309 loc_10005309:                           ; CODE XREF: sub_100052A2+57↑j
.text:10005309                 push    ebx
.text:1000530A                 lea     eax, [ebp+cbData]
.text:1000530D                 push    esi
.text:1000530E                 push    eax             ; lpcbData
.text:1000530F                 lea     eax, [ebp+Data]
.text:10005315                 mov     ebx, ds:RegQueryValueExA
.text:1000531B                 push    eax             ; lpData
.text:1000531C                 lea     eax, [ebp+Type]
.text:1000531F                 push    eax             ; lpType
.text:10005320                 push    0               ; lpReserved
.text:10005322                 push    offset aWorktime ; "WorkTime"
.text:10005327                 push    [ebp+phkResult] ; hKey
.text:1000532A                 call    ebx ; RegQueryValueExA
.text:1000532C                 mov     esi, ds:sprintf
.text:10005332                 mov     edi, ds:atoi
.text:10005338                 test    eax, eax
.text:1000533A                 jnz     short loc_10005379
.text:1000533C                 lea     eax, [ebp+Data]
.text:100053A2                 push    eax             ; Str
.text:100053A3                 call    edi ; atoi
.text:100053A5                 push    eax
.text:100053A6                 lea     eax, [ebp+Dest]
.text:100053AC                 push    offset aRobot_worktime ; "\r\n\r\n[Robot_WorkTime :] %d\r\n\r\n"
```

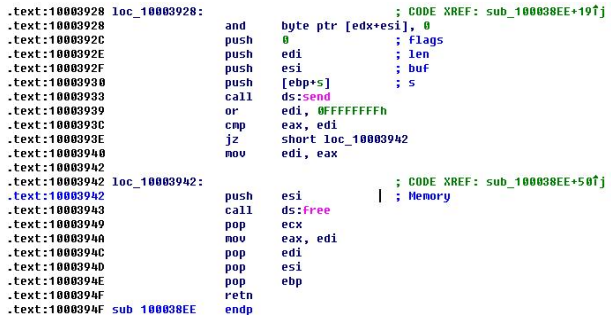
查看 loc_10005379，



查看 sub_100038EE 函数，



查看 loc_10003928 和 loc_10003942，

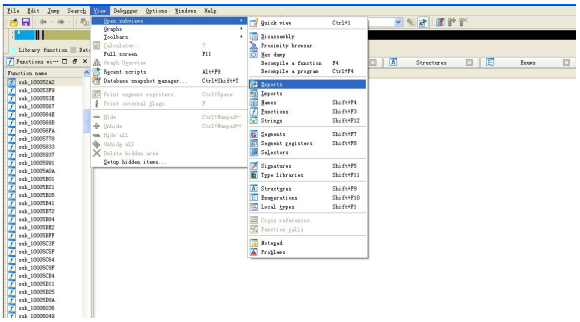


发现其调用了 malloc 函数创建了内存空间，然后又调用了 send 函数，最后调用了 free 函数释放内存空间。

综上所述，对注册表 SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime 和 Worktimes 键的值进行了查询操作，然后将这一信息返回给 0x1001045E 处的 push [ebp+s]传给该函数的网络 socket，通过远程 shell 连接发送出去。

（11） PSLIST 导出函数做了什么?

View→Open Subviews→Exports,查看该 DLL 的导出表。



Name	Address	Ordinal
InstallRT	0000000010000D847	1
InstallSA	0000000010000DEC1	2
InstallSB	0000000010000E892	3
PSLIST	0000000010007025	4
ServiceMain	0000000010000CF30	5
StartEXS	0000000010007ECB	6
UninstallRT	0000000010000F405	7
UninstallSA	0000000010000EA05	8
UninstallSB	0000000010000F138	9
DllEntryPoint	0000000010015160	

双击 PSLIST，跳转到 0x10007025，

```
.text:10007025 ; int __stdcall PSLIST(int, int, char *Str, int)
.text:10007025                 public PSLIST
.text:10007025 PSLIST          proc near               ; DATA XREF: .rdata:off_10017F78↓o
.text:10007025
.text:10007025 Str             = dword ptr  0Ch
.text:10007025
.text:10007025                 mov     dword_1008E5BC, 1
.text:1000702F                 call    sub_100036C3
.text:10007034                 test    eax, eax
.text:10007036                 jz      short loc_1000705B
.text:10007038                 push    [esp+Str]       ; Str
.text:1000703C                 call    strlen
.text:10007041                 test    eax, eax
.text:10007043                 pop     ecx
.text:10007044                 jnz     short loc_1000704E
.text:10007046                 push    eax
.text:10007047                 call    sub_10006518
.text:1000704C                 jmp     short loc_1000705A
.text:1000704E ; ---------------------------------------------------------------------------
.text:1000704E
.text:1000704E loc_1000704E:                           ; CODE XREF: PSLIST+1F↑j
.text:1000704E                 push    [esp+Str]
.text:10007052                 push    0
.text:10007054                 call    sub_1000664C
.text:10007059                 pop     ecx
.text:1000705A
.text:1000705A loc_1000705A:                           ; CODE XREF: PSLIST+27↑j
.text:1000705A                 pop     ecx
.text:1000705B
.text:1000705B loc_1000705B:                           ; CODE XREF: PSLIST+11↑j
.text:1000705B                 and     dword_1008E5BC, 0
.text:10007062                 retn    10h
.text:10007062 PSLIST          endp
```

这个函数选择两条路径之一执行，取决于 sub_100036C3 的结果。如果返回值为 1，则往下执行函数 sub_10006518 和 sub_1000664C。

sub_100036C3 函数可用于检查操作系统的版本是 Windows Vista/7，或是 Windows XP/2003/2000，如果是，则返回 1。

```
.text:100036C3
.text:100036C3 ; =============== S U B R O U T I N E =======================================
.text:100036C3
.text:100036C3 ; Attributes: bp-based frame
.text:100036C3
.text:100036C3 sub_100036C3    proc near               ; CODE XREF: sub_10001656+27↑p
.text:100036C3                                         ; PSLIST+A↓p
.text:100036C3
.text:100036C3 VersionInformation= _OSVERSIONINFOA ptr -94h
.text:100036C3
.text:100036C3                 push    ebp
.text:100036C4                 mov     ebp, esp
.text:100036C6                 sub     esp, 94h
.text:100036CC                 lea     eax, [ebp+VersionInformation]
.text:100036D2                 mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
.text:100036DC                 push    eax             ; lpVersionInformation
.text:100036DD                 call    ds:GetVersionExA
.text:100036E3                 cmp     [ebp+VersionInformation.dwPlatformId], 2
.text:100036EA                 jnz     short loc_100036FA
.text:100036EC                 cmp     [ebp+VersionInformation.dwMajorVersion], 5
.text:100036F3                 jb      short loc_100036FA
.text:100036F5                 push    1
.text:100036F7                 pop     eax
.text:100036F8                 leave
.text:100036F9                 retn
.text:100036FA ; ---------------------------------------------------------------------------
```

查看 sub_10006518 和 sub_1000664C，

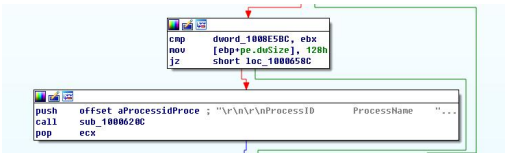
```
.text:10006518 ; =============== S U B R O U T I N E =======================================
.text:10006518
.text:10006518 ; Attributes: bp-based frame
.text:10006518
.text:10006518 sub_10006518    proc near               ; CODE XREF: PSLIST+22↓p
.text:10006518
.text:10006518 hModule         = dword ptr -1530h
.text:10006518 var_152C        = byte ptr -152Ch
.text:10006518 Dst             = byte ptr -530h
.text:10006518 var_52F         = byte ptr -52Fh
.text:10006518 pe              = PROCESSENTRY32 ptr -130h
.text:10006518 cbNeeded        = dword ptr -8
.text:10006518 hSnapshot       = dword ptr -4
.text:10006518
.text:10006518                 push    ebp
.text:10006519                 mov     ebp, esp
.text:1000651B                 mov     eax, 1530h
.text:10006520                 call    __alloca_probe
.text:10006525                 push    ebx
.text:10006526                 push    esi
.text:10006527                 push    edi             ; ArgList
.text:10006528                 xor     ebx, ebx
.text:1000652A                 mov     ecx, 3FFh
.text:1000652F                 xor     eax, eax
.text:10006531                 lea     edi, [ebp+var_152C]
.text:10006537                 mov     [ebp+hModule], ebx
.text:1000653D                 rep stosd
.text:1000653F                 mov     ecx, 0FFh
.text:10006544                 lea     edi, [ebp+var_52F]
.text:1000654A                 mov     [ebp+Dst], bl
.text:10006550                 push    ebx             ; th32ProcessID
.text:10006551                 rep stosd
.text:10006553                 stosw
.text:10006555                 push    2               ; dwFlags
.text:10006557                 stosb
.text:10006558                 call    CreateToolhelp32Snapshot
.text:1000655D                 mov     esi, ds:CloseHandle
.text:10006563                 cmp     eax, 0FFFFFFFFh
.text:10006566                 mov     [ebp+hSnapshot], eax
.text:10006569                 jz      loc_1000664A
.text:1000656F                 cmp     dword_1008E58C, ebx
.text:10006575                 mov     [ebp+pe.dwSize], 128h
.text:1000657F                 jz      short loc_1000658C
.text:10006581                 push    offset aProcessidProce ; "\r\n\r\nProcessID       ProcessName    "...
.text:10006586                 call    sub_1000620C
.text:1000658B                 pop     ecx
```

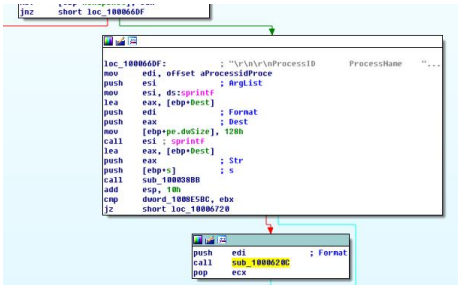
```
.text:1000664C Dst             = byte ptr -234h
.text:1000664C pe              = PROCESSENTRY32 ptr -130h
.text:1000664C cbNeeded        = dword ptr -8
.text:1000664C hSnapshot       = dword ptr -4
.text:1000664C s               = dword ptr  8
.text:1000664C Str             = dword ptr  0Ch
.text:1000664C
.text:1000664C                 push    ebp
.text:1000664D                 mov     ebp, esp
.text:1000664F                 mov     eax, 163Ah
.text:10006654                 call    __alloca_probe
.text:10006659                 and     [ebp+Dest], 0
.text:1000665D                 push    ebx
.text:1000665E                 push    edi
.text:1000665F                 mov     ecx, 0FFh
.text:10006664                 xor     eax, eax
.text:10006666                 lea     edi, [ebp+var_633]
.text:1000666C                 rep stosd
.text:1000666E                 stosw
.text:10006670                 stosb
.text:10006671                 push    49h
.text:10006673                 xor     ebx, ebx
.text:10006675                 pop     ecx
.text:10006676                 xor     eax, eax
.text:10006678                 lea     edi, [ebp+pe.cntUsage]
.text:1000667E                 mov     [ebp+pe.dwSize], ebx
.text:10006684                 rep stosd
.text:10006686                 mov     ecx, 3FFh
.text:1000668B                 lea     edi, [ebp+var_1630]
.text:10006691                 mov     [ebp+hModule], ebx
.text:10006697                 push    ebx             ; th32ProcessID
.text:10006698                 rep stosd
.text:1000669A                 push    2               ; dwFlags
.text:1000669C                 call    CreateToolhelp32Snapshot
.text:100066A1                 cmp     eax, 0FFFFFFFFh
.text:100066A4                 mov     [ebp+hSnapshot], eax
.text:100066A7                 jnz     short loc_100066DF
.text:100066A9                 call    ds:GetLastError
.text:100066AF                 push    eax
.text:100066B0                 lea     eax, [ebp+Dest]
.text:100066B6                 push    offset aCreatetoolhe_0 ; "\r\n\r\nCreateToolhelp32Snapshot Fail:E"...
.text:100066BB                 push    eax             ; Dest
.text:100066BC                 call    ds:sprintf
.text:100066C2                 lea     eax, [ebp+Dest]
.text:100066C8                 push    eax             ; Str
.text:100066C9                 push    [ebp+s]         ; s
.text:100066CC                 call    sub_100038BB
.text:100066D1                 add     esp, 14h
.text:100066D4                 push    1
.text:100066D6                 pop     eax
.text:100066D7                 jmp     loc_1000689B
```

sub_10006518 和 sub_1000664C 都使用 CreateToolhelp32Snapshot 函数，可用于获得主机的进程列表。

```
.text:100111C4 ; =============== S U B R O U T I N E =======================================
.text:100111C4
.text:100111C4 ; Attributes: thunk
.text:100111C4
.text:100111C4 ; HANDLE __stdcall CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID)
.text:100111C4 CreateToolhelp32Snapshot proc near      ; CODE XREF: sub_10003C0D+23↑p
.text:100111C4                                         ; sub_10004249+E7↑p ...
.text:100111C4
.text:100111C4 dwFlags         = dword ptr  4
.text:100111C4 th32ProcessID   = dword ptr  8
.text:100111C4
.text:100111C4                 jmp     ds:__imp_CreateToolhelp32Snapshot
.text:100111C4 CreateToolhelp32Snapshot endp
```

sub_10006518 和 sub_1000664C 还使用了 sub_1000620C 函数，应该是将查询到的进程信息写到一个文件中。



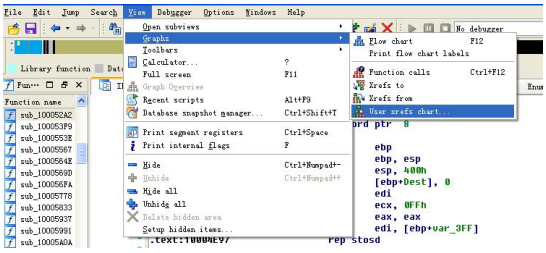


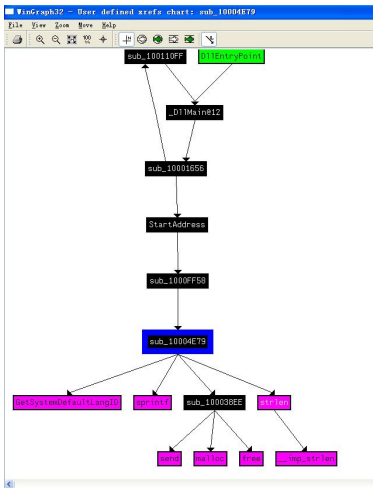
```
.text:1000620C ; ============== S U B R O U T I N E =======================================
.text:1000620C
.text:1000620C ; Attributes: bp-based frame
.text:1000620C
.text:1000620C ; int __cdecl sub_1000620C(char *Format, char ArgList)
.text:1000620C sub_1000620C    proc near               ; CODE XREF: sub_10006518+6E↓p
.text:1000620C                                         ; sub_10006518+109↓p ...
.text:1000620C
.text:1000620C DstBuf          = byte ptr -400h
.text:1000620C Format          = dword ptr  8
.text:1000620C ArgList         = byte ptr  0Ch
.text:1000620C
.text:1000620C                 push    ebp
.text:1000620D                 mov     ebp, esp
.text:1000620F                 sub     esp, 400h
.text:10006215                 lea     eax, [ebp+ArgList]
.text:10006218                 push    esi
.text:10006219                 push    eax             ; ArgList
.text:1000621A                 lea     eax, [ebp+DstBuf]
.text:10006220                 push    [ebp+Format]    ; Format
.text:10006223                 push    400h            ; MaxCount
.text:10006228                 push    eax             ; DstBuf
.text:10006229                 call    ds:_vsnprintf
.text:1000622F                 push    offset aA       ; "a"
.text:10006234                 push    offset aXinstall_dll ; "xinstall.dll"
.text:10006239                 call    ds:fopen
.text:1000623F                 mov     esi, eax
.text:10006241                 add     esp, 18h
.text:10006244                 test    esi, esi
.text:10006246                 jz      short loc_10006265
.text:10006248                 lea     eax, [ebp+DstBuf]
.text:1000624E                 push    eax
.text:1000624F                 push    offset aS_0     ; "%s\n"
.text:10006254                 push    esi             ; File
.text:10006255                 call    ds:fprintf
.text:1000625B                 push    esi             ; File
.text:1000625C                 call    ds:fclose
.text:10006262                 add     esp, 10h
.text:10006265
.text:10006265 loc_10006265:                           ; CODE XREF: sub_1000620C+3A↑j
.text:10006265                 pop     esi
.text:10006266                 leave
.text:10006267                 retn
.text:10006267 sub_1000620C    endp
```

综上所述，PSLIST 导出项可以寻找该列表中某个指定的进程名并获取其信息保存到文件中，并通过 send 将进程列表通过 socket 发送。

（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?

G 快速跳转到 0x10004E79，View→Graphs→User Xrefs Chart，得到交叉引用图。

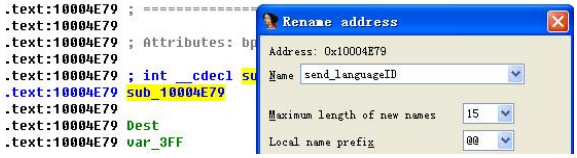




在 sub_10004E79 处的函数中调用了 GetSystemDefaultLangID、send 和 sprintf 这三个 API。该函数又调用了 sub_100038EE 函数，sub_100038EE 函数又调用了 send，malloc，free，__imp_strlen 函数。上面的信息告诉我们，该函数可能通过 socket 发送语言标志。

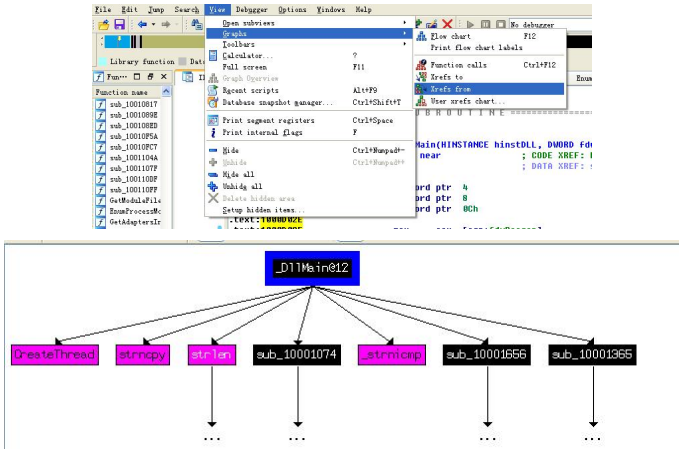
该函数调用了 GetSystemDefaultLangID 函数，又间接调用了 send 函数，猜测该函数会获取系统默认语言 ID 后，通过 send 函数发送，可将该函数取名 GetSystemLanguage。

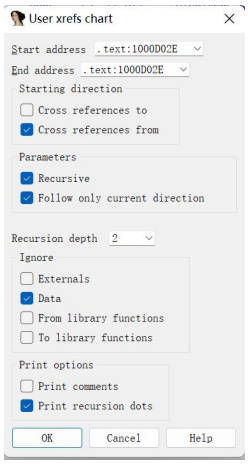
右击函数名，即可重命名为 GetSystemLanguage。



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

找到 DllMain 的起始地址 0x1000D02E，View→Graphs→Xrefs From，选取 Recursion depth 递归深度为 1，以显示其直接调用函数。





可以看到，DllMain 直接调用了 strncpy、strnicmp、CreateThread 和 strlen 这些 API。继续查看递归深度 2 所有被调用函数,可以看到非常多的 API(30 多个),包括 Sleep、WinExec、gethostbyname、CreateThread、ExitThread,以及许多其他网络函数调用。

（14） 在 0x10001358 处，有一个对 Sleep(一个使用一个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?

G 跳转到 0x10001358,该处有一个对 Sleep 的调用。Sleep 只使用休眠的毫秒数作为参数，eax 被压入栈上作为 Sleep 函数的参数。

```
.text:10001341 loc_10001341:                           ; CODE XREF: sub_10001074+10Fîj
.text:10001341                                         ; sub_10001074+1B0îj ...
.text:10001341                 mov     eax, off_10019020
.text:10001346                 add     eax, 0Dh
.text:10001349                 push    eax             ; Str
.text:1000134A                 call    ds:atoi
.text:10001350                 imul    eax, 3E8h
.text:10001356                 pop     ecx
.text:10001357                 push    eax             ; dwMilliseconds
.text:10001358                 call    ds:Sleep
.text:1000135E                 xor     ebp, ebp
.text:10001360                 jmp     loc_100010B4
.text:10001360 sub_10001074    endp
```

eax 被乘了 0x3E8(十进制是 1000),也就是说，对 atoi 调用的结果被乘以 1000,得到要休眠的毫秒数。

off_10019020 被赋给 eax,双击 off_10019020，指向了一个字符串[This is CTI]30。

```
.data:10019020 off_10019020    dd offset aThisIsCti30  ; DATA XREF: sub_10001074:loc_10001341îr
.data:10019020                                         ; sub_10001365:loc_10001632îr ...
.data:10019020                                         ; "[This is CTI]30"
```

0xD 被加到 eax 上作为偏移，也就是 13 字节，对应信息就是[This is CTI]，因此 eax 指向了 30。调用 atoi,把字符串 30 转成了数字 30。30 乘上 1000,得到 30000 毫秒(30 秒),也就是程序将休眠 30 秒。

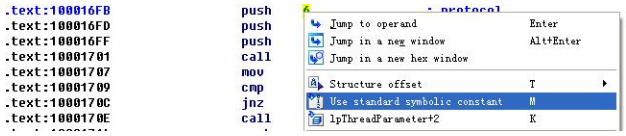
（15） 在 0x10001701 处是一个对 socket 的调用。它的 3 个参数是什么?

G 快速跳转到 0x10001701，看到 6、1、2 被压到栈上。因此三个参数为 6、1、2。

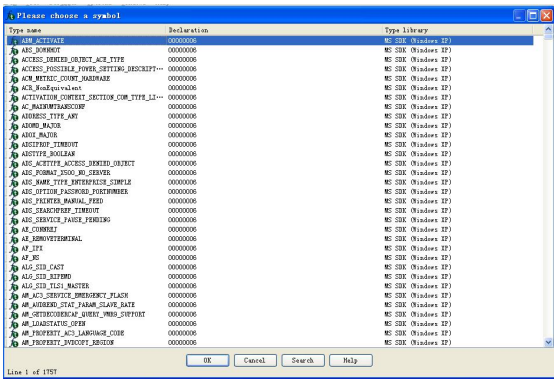
```
.text:100016FB loc_100016FB:                           ; CODE XREF: sub_10001656+374↓j
.text:100016FB                                         ; sub_10001656+A09↓j
.text:100016FB                 push    6               ; protocol
.text:100016FD                 push    1               ; type
.text:100016FF                 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
```

（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?

右键每个数字，选择 Use standard Symbolic constant。



弹出一个列举 IDA Pro 为这个特定值找到所有的对应常量的对话框。



3 个参数分别为 domain（协议域，或协议族 family）、type（socket 类型）、protocol（协议）。domain=2，指的是 AF_INET,用于设置一个 IPv4 socket，决定了要用 IPv4 地址（32 位）与端口号（16 位）的组合;type=1，指的是 SOCK_STREAM，即流式套接字。protocol=6，指的是 IPPROTO_TCP，即 TCP 协议。这个 socket 会被配置为基于 IPv4 的 TCP 连接(常被用于 HTTP)。

应用了修改后的参数如下：

```
.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
```

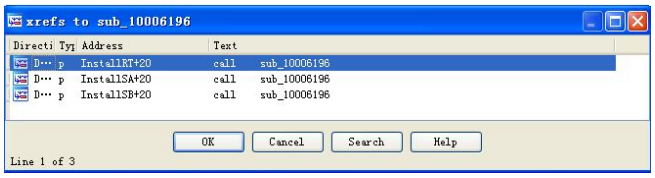
（17） 搜索 in 指令(opcode 0xED)的使用。这个指令和一个魔术字符串 VMXh 用来进行 VMware 检测。这在这个恶意代码中被使用了吗?使用对执行 in 指令函数的交叉引用，能发现进一步检测 VMware 的证据吗?

Search→Text，输入 in.

看到在 0x100061DB 处调用了 in 指令，在 0x100061C7 处的 mov 指令将 0x564D5868 赋给 EAX。这个 16 进制值相当于 ASCII 字符串 VMXh，该恶意代码采用了反虚拟机技巧。

```
.text:10006196 sub_10006196    proc near               ; CODE XREF: InstallRT+20↓p
.text:10006196                                         ; InstallSA+20↓p ...
.text:10006196
.text:10006196 var_1C          = byte ptr -1Ch
.text:10006196 ms_exc          = CPPEH_RECORD ptr -18h
.text:10006196
.text:10006196                 push    ebp
.text:10006197                 mov     ebp, esp
.text:10006199                 push    0FFFFFFFFh
.text:1000619B                 push    offset stru_10016438
.text:100061A0                 push    offset loc_10015050
.text:100061A5                 mov     eax, large fs:0
.text:100061AB                 push    eax
.text:100061AC                 mov     large fs:0, esp
.text:100061B3                 sub     esp, 0Ch
.text:100061B6                 push    ebx
.text:100061B7                 push    esi
.text:100061B8                 push    edi
.text:100061B9                 mov     [ebp+ms_exc.old_esp], esp
.text:100061BC                 mov     [ebp+var_1C], 1
.text:100061C0                 and     [ebp+ms_exc.registration.TryLevel], 0
.text:100061C4                 push    edx
.text:100061C5                 push    ecx
.text:100061C6                 push    ebx
.text:100061C7                 mov     eax, 564D5868h
.text:100061CC                 mov     ebx, 0
.text:100061D1                 mov     ecx, 0Ah
.text:100061D6                 mov     edx, 5658h
.text:100061DB                 in      eax, dx
.text:100061DC                 cmp     ebx, 564D5868h
.text:100061E2                 setz    [ebp+var_1C]
.text:100061E6                 pop     ebx
.text:100061E7                 pop     ecx
.text:100061E8                 pop     edx
.text:100061E9                 jmp     short loc_100061F6
```

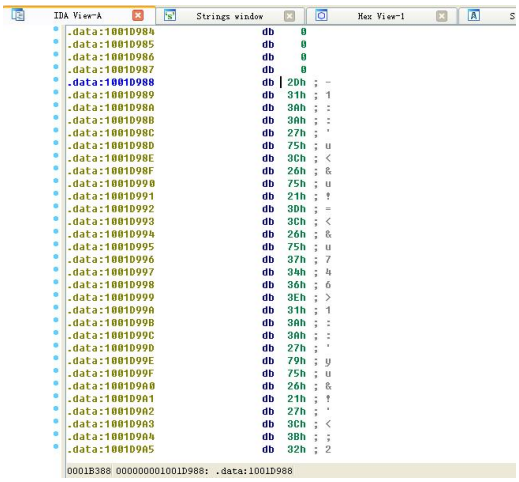
检查 sub_10006196 的交叉引用，在调用函数中均发现 Found Virtual Machine，Install Cancel 字符串。




```
.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

.text:1000DEE1                 call    sub_10006196
.text:1000DEE6                 test    al, al
.text:1000DEE8                 jz      short loc_1000DF08
.text:1000DEEA
.text:1000DEEA loc_1000DEEA:                           ; CODE XREF: InstallSA+1E↑j
.text:1000DEEA                 push    offset unk_1008E5F0 ; Format
.text:1000DEEF                 call    sub_10003592
.text:1000DEF4                 mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:1000DEFB                 call    sub_10003592
.text:1000DF00                 pop     ecx
.text:1000DF01                 call    sub_10005567
.text:1000DF06                 jmp     short loc_1000DF1E

.text:1000E8D2                 call    sub_10006196
.text:1000E8D7                 test    al, al
.text:1000E8D9                 jz      short loc_1000E8D9
.text:1000E8DB
.text:1000E8DB loc_1000E8DB:                           ; CODE XREF: InstallSB+1E↑j
.text:1000E8DB                 push    offset unk_1008E5F0 ; Format
.text:1000E8C0                 call    sub_10003592
.text:1000E8C5                 mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:1000E8CC                 call    sub_10003592
.text:1000E8D1                 pop     ecx
.text:1000E8D2                 call    sub_10005567
.text:1000E8D7                 jmp     short loc_1000E8F4

xdoors_d:10094F88 ; char aFoundVirtualMa[]
xdoors_d:10094F88 aFoundVirtualMa db 'Found Virtual Machine,Install Cancel.',0
xdoors_d:10094F88                                      ; DATA XREF: InstallRT+33↑o
xdoors_d:10094F88                                      ; InstallSA+33↑o ...
```

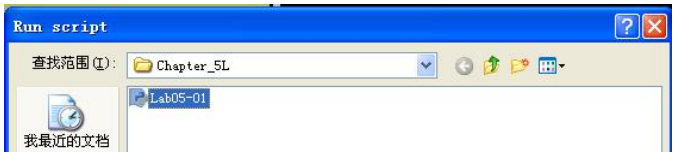
（18） 将你的光标跳转到 0x1001D988 处，你发现了什么？

G 快速跳转到 1001D988，看到一些像是随机的数据，没有可读性。

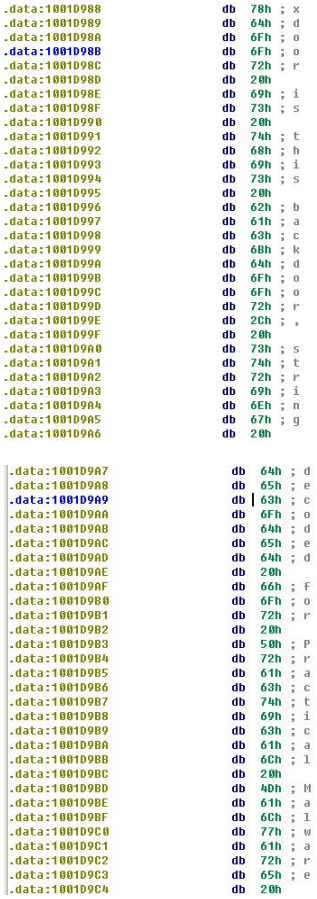


（19） 如果你安装了 IDA Python 插件(包括 IDA Pro 的商业版本的插件)，运行 Lab05-01.py，一个本书中随恶意代码提供的 IDA Pro Python 脚本，(确定光标是在 0x1001D988 处。)在你运行这个脚本后发生了什么？

File->Script File，选择 Lab05-01 Python 脚本运行。



运行后再次查看 0x1001D988 的内容，变成明文



```
.data:1001D9C5                db  41h ; A
.data:1001D9C6                db  6Eh ; n
.data:1001D9C7                db  61h ; a
.data:1001D9C8                db  6Ch ; l
.data:1001D9C9                db  79h ; y
.data:1001D9CA                db  73h ; s
.data:1001D9CB                db  69h ; i
.data:1001D9CC                db  73h ; s
.data:1001D9CD                db  20h
.data:1001D9CE                db  4Ch ; L
.data:1001D9CF                db  61h ; a
.data:1001D9D0                db  62h ; b
.data:1001D9D1                db  20h
.data:1001D9D2                db  3Ah ; :
.data:1001D9D3                db  29h ; )
.data:1001D9D4                db  31h ; 1
.data:1001D9D5                db  32h ; 2
.data:1001D9D6                db  33h ; 3
.data:1001D9D7                db  34h ; 4
```

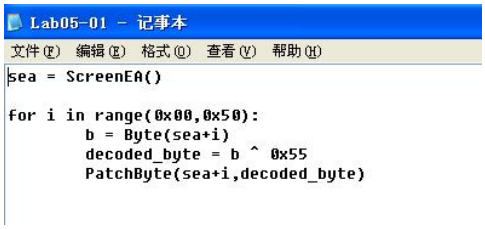
看到这段数据被反混淆得到一个字符串。

（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 ASCⅡI 字符串？

光标放在 0x1001D988 处，按下 A 键，将其变为一个可读的字符串：xdoor is this backdoor, string decoded for Practical Malware Analysis Lab:)1234。

```
.data:1001D988 aXdoorIsThisBac db 'xdoor is this backdoor, string decoded for Practical Malware Anal
.data:1001D988                 db 'ysis Lab :)1234',0
```

（21） 使用一个文本编辑器打开这个脚本。它是如何工作的？



sea = ScreenEA()用于获得光标当前位置，用作要解码数据的偏移值。

然后从 0 到 0x50 循环，调用 Byte 获得每个字节的值，将这些字节与 0x55 进行 XOR 异或操作。

最后将这些字节用 PatchByte 函数写回 IDA Pro 的显示中，但不修改原始文件。

（二） Yara

根据上面的分析，编写如下的 yara 规则：

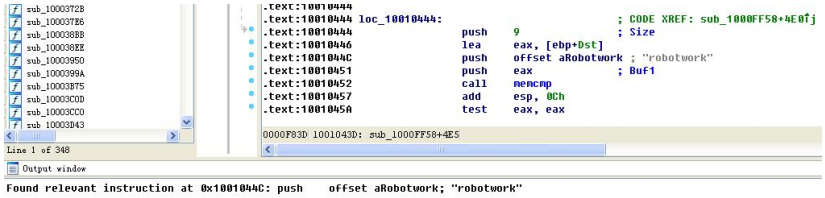
```
rule Detect_Lab05
{
    meta:
        description = "YARA rule to detect Lab05 malicious code"
    strings:

      $x1 = "Get DLL FileName Error,Update Failed" fullword ascii
      $x2 = "Move '%s' To '%s' Failed,Perhaps Other Process Updateing|Updated Same
```

```
Module"
        $x3 = "Process '%s' Not Found,Uninject Failed" fullword ascii
        $x4 = "Uninject '%s' From Process '%s' " fullword ascii
        $x5 = "error on get process info. " fullword ascii
        $x6 = "Process '%s' Not Found ,Inject Failed" fullword ascii
        $x7 = "Inject '%s' To Process '%s' Failed" fullword ascii
        $x8 = "Old Module Have Been Free,New ModuleName As Old,Will Take Effect Soon."
fullword ascii
        $x9 = "You Specify Service Name Not In Svchost\\netsvcs, must be one of
following:" fullword ascii
        $x10 = "Uninject '%s' From Process '%s' Failed, Module '%s' Not Found" fullword
ascii
        $x11 = "MayBe Inject Mode,You Only Need Close Master Process '%s' To Uninatll
This Mode" fullword ascii
        $x12 = "Resume '%s' To '%s' Failed,Update Failed" fullword ascii
        $x13 = "Move '%s' To '%s' Failed,Update Failed" fullword ascii
        $x14 = "Replace Old Service '%s' Failed,Install Cancle" fullword ascii
        $x15 = "Old Module Have Been Free,New ModuleName As Old,Will Take Effect Soon."
fullword ascii
        $x16 = "Repair Successfully,Need Reboot Machine!" fullword ascii
    condition:
//检查目标文件的前两个字节是否等于 0x5a4d，这是 Windows 可执行文件（PE 文件）的标
识符;检查目标文件中偏移为 0x3c 处的四个字节，看是否等于 0x00004550。在 PE 文件中，
这个值是 PE 头的标识，指示文件的格式;前面的条件中至少有 12 个条件为真才匹配。
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and
        1 of ($x*) and 12 of them
}
```

进行扫描，结果如下

（三） IDA Python 脚本

编写脚本解决问题 10 中查找代码中 robotwork 字符串的需求。




```
import idc
import idautils

# 目标注释
target_string = "robotwork"

# 获取函数的起始和结束地址
start_address = 0x1000FF58
end_address = 0x1001073F

# 初始化一个列表来存储包含特定注释的汇编语句
relevant_instructions = []

# 遍历指令，查找包含指定字符串的汇编语句
current_address = start_address

while current_address <= end_address:
    # 获取当前指令的反汇编文本
    disasm = idc.GetDisasm(current_address)

    # 检查是否包含指定字符串
    if target_string in disasm:
        relevant_instructions.append((current_address, disasm))

    # 移动到下一条指令
    current_address = idc.NextHead(current_address)

# 输出找到的相关汇编语句
for instruction in relevant_instructions:
    print("Found relevant instruction at 0x%X: %s" % (instruction[0], instruction[1]))
```

四、 实验结论及心得体会

1. 实验结论

本次实验通过使用 IDA Pro 工具，成功分析了 Lab05-01.dll 文件中的恶意代码。识别了 DLL 文件的入口点，即 DllMain 函数，以及重要的导入函数 gethostbyname，它在恶意代码中被多次调用。此外，还找到了用于执行命令的关键字符串"\cmd.exe /c"，以及用于检测虚拟机环境的反虚拟机技巧。在分析函数内的代码时，成功识别了局部变量、参数和全局变量，这有助于理解函数的工作原理和目的。此外，发现恶意代码调用了多个 Windows API 函数，包括网络和文件操作函数。确定了 Socket 的调用三个参数的含义，分别表示协议域、Socket 类型和协议，配置了一个 IPv4 的 TCP 连接。

2. 心得体会

本次实验加深了对逆向工程和恶意代码分析的理解。通过实际操作，学习了如何使用 IDA Pro 等工具 DLL 文件。熟悉了 IDA Pro 工具的基本功能，包括查找函数、查看导入表、分析字符串等。这些技能对于深入研究和理解恶意代码非常有帮助。分析恶意代码，了解了如何追踪函数调用关系、查找字符串、分析代码块等技术，对于理解代码的逻辑和功能非常重要。逆向工程和恶意代码分析是网络安全领域的重要技能，通过实际操作和学习，更深刻地理解了这些技术的应用和重要性。